# NativeSpeaker: Identifying Crypto Misuses in Android Native Code Libraries

Qing Wang, Juanru Li, Yuanyuan Zhang(✉), Hui Wang, Yikun Hu, Bodong Li, and Dawu Gu

Shanghai Jiao Tong University, Shanghai, China
yyjess@sjtu.edu.cn

**Abstract.** The use of native code (ARM binary code) libraries in Android apps greatly promotes the execution performance of frequently used algorithms. Nonetheless, it increases the complexity of app assessment since the binary code analysis is often sophisticated and time-consuming. As a result, many defects still exist in native code libraries and potentially threat the security of users. To assess the native code libraries, current researches mainly focus on the API invoking correctness and less dive into the details of code. Hence, flaws may hide in internal implementation when the analysis of API does not discover them effectively.

The assessment of native code requires a more detailed code comprehension process to pinpoint flaws. In response, we design and implement NativeSpeaker, an Android native code analysis system to assess native code libraries. NativeSpeaker provides not only the capability of recognizing certain pattern related to security flaws, but also the functionality of discovering and comparing native code libraries among a large-scale collection of apps from non-official Android markets. With the help of NativeSpeaker, we analyzed 20,353 dynamic libraries (.so) collected from 20,000 apps in non-official Android markets. Particularly, our assessment focuses on searching crypto misuse related insecure code pattern in those libraries. The analyzing results show even for those most frequently used (top 1%) native code libraries, one third of them contain at least one misuse. Furthermore, our observation indicates the misuse of crypto is often related to insecure data communication: about 25% most frequently used native code libraries suffer from this flaw. Our conducted analysis revealed the necessity of in-depth security assessment against popular native code libraries, and proved the effectiveness of the designed NativeSpeaker system.

## 1 Introduction

Android apps are typically written in Java. However, the limitations of Java such as memory management and performance drives many Android apps to contain

This work was partially supported by the Key Program of National Natural Science Foundation of China (Grants No. U1636217), the Major Project of the National Key Research Project (Grants No. 2016YFB0801200), and the Technology Project of Shanghai Science and Technology Commission under Grants No. 15511103002.

components implemented in native code. Android provides Native Development Kit (NDK) to support native development in C/C++, and the app supports a hybrid execution mode that allows a seamless switch between Java code and native code. In a hybrid execution, native code is largely compiled as the form of shared library (.so file) and its exported functions are invoked by Java code via a Java Native Interface (JNI). Since native code achieves better performance and flexible data manipulation, it is especially suitable for implementing data encoding/decoding (e.g., crypto transformation) and raw socket communication.

Although more efficient, native code can be more harmful compared to Java code. Despite the common memory corruption vulnerabilities, high level security flaws are also contained in native code especially some third-party libraries. Even though security assessment of native code libraries is essential, flaws are more difficult to be discovered. The audit of Android native code is sophisticated for two main reasons: First, the binary code is hard to be understood since the compilation process removes a large amount of symbol information from the source code. Without such symbol information the binary code contains little semantics and the comprehension of low-level disassembling code is also time-consuming. Second, on Android platform the lack of fine-grained dynamic analysis tools (e.g., code instrumentation) restricts analysts from collecting runtime data to supplement the code comprehension. Therefore, an improper designed logic in a native code library often requires an in-depth analysis to be excavated.

Among all security flaws, crypto misuses in Android apps ia a major security issue of Android app security [6,20]. Although the security community has proposed utilities to detect crypto misuse in Android apps, the designed technique is mainly effective when analyzing Java code of Android apps. Our observation in recent app development reveals that apps tend to use native code version of crypto implementations in shared libraries rather than that of Java code version to fulfil the crypto operations. The main consideration is that crypto in native code is efficient and is not easily analyzed by reverse engineers. To assess crypto misuse in native code of apps, existing native code analysis techniques [8,12,14,19,22] are generally not domain-specific and thus are less effective.

To further improve the status quo and address the crypto misuse analysis issue against native code of Android app, in this paper we propose a native code analysis to help identifying typical crypto misuse patterns in Android native code libraries practically. Our approach firstly utilizes several heuristics to identify third-party libraries and then locate crypto functions in their native code. By summarizing typical implementation features of crypto in native code, our approach is able to locate two common patterns of crypto functions in native code. After the locating of crypto functions, we further detect relevant crypto misuse through checking obsolete algorithms and incorrect parameters. In particular, we design and implement NativeSpeaker, an automated native code analysis tool for crypto misuse identification. NativeSpeaker is able to analyze common Android third-party libraries and find certain crypto misuses such as predicable key generation and incorrect parameters for crypto APIs. Our evaluation is based on a corpus of 20,000 Android apps, which contain 20,353 instances of native shared library files (.so). Our analysis demonstrated that the occurrence of crypto and crypto-like functions are very popular in those files, and 21 potential crypto misuses were reported by our analysis. Furthermore, a fine-grained pattern-matching assessment on 310 frequently used native code shared libraries was conducted to find insecure communication issue. The results showed that NativeSpeaker is effective to find complicated crypto misuse cases among a huge amount of dynamic libraries, and revealed that communication with broken encryption routine is common in many shared libraries.

The main contributions of this paper are the followings:

- We achieve a large-scale security assessment of apps in those non-official Android markets. We collected 20,000 popular Android apps and extracted all 20,353 native code libraries used. Then we deduplicate them using semantic similarity comparison to reduce the number of targets to be analyzed. This native code library dataset reflects the common features of how functions are used in native code of Android apps.
- We propose a practical and lightweight analysis approach to find crypto misuses in native code. The approach combines static taint analysis and natural language processing of function names to locate crypto functions. Then, the crypto misuses are then found by searching typical patterns summarized from empirical studies.
- We made use of NativeSpeaker to search a sophisticated insecure behavior–raw socket data communication without encryption. We find several flaws in real-world implementations of native code which lead to the broken of communication protection. Compared with previous studies, our system provides not only capability of large-scale assessment on native code libraries in a reasonable time, but also find internal implementation vulnerabilities caused by cryptographic misuse.

## 2 Excavating Semantic Information of Native Code Library

The usage rate of native code libraries in Android app is rapidly increasing [7]. And it also brings many challenges to Android native code analysis especially large-scale analysis. An important aspect often ignored by existing analyses is how to utilize intrinsic characteristics and semantic information (e.g., functionality of the library, feature of exported interfaces) of native code libraries. In this section, We firstly list typical encountered challenges of native code library analysis. Then, we illustrate common features in native code libraries that can be leveraged to help retrieve more semantic information through an empirical study of libraries in current Android apps.

### 2.1 Challenges of Native Code Analysis

To conduct effective and scalable security analysis against widely used Android native code libraries, several restrictions should be taken into account. Typically, Android app contains native code in the form of shared library (*.so* file). Java code and native code communicates with each other through Android provided JNI interface: functions in native code can be invoked from Java layer through JNI interfaces and vice versa. To develop native code libraries, the Android Native Development Kit (NDK), a companion tool to the Android SDK, is used to help developers build performance-critical portions of apps in native code. It provides headers and libraries that allow developers to build activities, handle user input, use hardware sensors, and access application resources by programming in C or C++. As a result, almost all Android native code libraries are written in C or C++ and are compiled using the NDK. To analyze them, binary code reverse engineering techniques such as disassembling and decompilation are necessary. However, since the inherent complexity of binary code analysis [17], understanding those libraries in native code form is not easy.

Moreover, native code libraries are often provided to achieve low latency or run computationally intensive applications, such as games or physics simulations, and to reuse existing C/C++ libraries. Thus most Android native code libraries are implemented by third-party developers to fulfil some universal algorithms (e.g., crypto algorithm). App developers often directly integrate an Android native code library and invoke its functions through exported interfaces without knowing the implementation details. Due to the lack of source code, security assessment of those libraries are often ignored. Although this is convenient for app developers, potential security flaws in Android native code libraries may be introduced to a wide variety of apps.

Although static code analysis is often harnessed to help assess the security of Android app on a grand scale, finding security flaws, especially sophisticated logic vulnerabilities related to high-level functionality (e.g., data protection), is generally restricted to Java code with rich semantics. Existing flaw detection approaches (e.g., crypto misuse detection) strongly rely on static patterns of code to find vulnerability. Native code, due to the lack of symbol information, does not contain enough static pattern and thus a simple pattern matching approach is inadequate to effectively locate its contained flaws.

Dynamic analysis can collect more runtime information of an app and complements static analysis. Ideally, fine-grained dynamic analysis combined with static analysis is expected to generate precise analysis results and find security flaws. However, due to the considerable analysis time, dynamic analysis system such as the one proposed by Afonso *et al.* [7] to obtain a comprehensive characterization of native code usage in real world applications are often not applicable to large-scale security analysis. Such issue also exists among other heavyweight program analysis techniques (e.g., symbolic execution). Since not only the amount of native code libraries but also the code size of each library have increased to a considerable scale, a more lightweight analysis should be introduced.

**Table 1.** Word indicators and their related functionalities

| Word Indicator | Occurrence | Functionality |
|---|---|---|
| xml | 11200 | file parsing |
| png | 10572 | picture processing |
| pthread | 6828 | thread controlling |
| curl | 6220 | network |
| ssl | 5260 | crypto |
| http | 3860 | network |
| crypto | 3553 | crypto |
| evp | 3546 | crypto |
| x509 | 4182 | crypto |
| mutex | 2859 | thread control |

### 2.2 Extracing Semantics in Native Code Library

To obtain an in-depth understanding of how native code libraries are used, the first step we conduct is an analysis based on the interface name of a library. As a shared library, Android native libraries usually export numbers of functions as interfaces, and the name of those interfaces are generally not obfuscated. Thus, we utilize those interfaces in exported table as an important source of information to classify libraries. We utilize a simple natural language processing approach to analyze those interface names: an N-gram algorithm [3,18] is used to extract the English word sequences in interface names. After splitting an interface name into different units as a sequence, we can deduce relevant functionalities of the interface according to specific word indicator. For instance, if an interface name contains the word "x509", it is possibly related to cryptographic certificate operating and can be considered as possessing the functionality of "crypto".

We collected 20,000 apps and extracted 20,353 native code libraries (details are illustrated in Sect. 4.1). A part of the analyzing results with related deduction rules are listed in Table 1. In further, we choose 180 frequently used samples to conduct a manual investigation. The following observations of native code libraries are summarized through this manual investigation:

**Code reuse:** Developers transplant existing open source C/C++ projects to Android platform. In our investigation, the portion of native libraries including code migrated form open source project reaches 41.1% (74 of 180). Most commonly used open source projects are *bspatch*, *base64encoder*, *stagefright_honeycomb*, and *tnet*. The reuse of existing open source code allows analysts to utilize code similarity comparison techniques [16] to obtain more semantic informations.

**Native API invoking:** The invoking of certain API indicates the specific behavior of the program. Unlike Java code, native code can conveniently invoke many low-level system API such as *fork*. This helps analyze the behaviors of thread/memory management, process controlling, and network communication. In our investigation, we found there are 25.5% (46 of 180) of the analyzed libraries invoke at least one API related to the mention behaviors. Through monitoring such API invoking, we can better understand the library.

**Crypto functions:** An important trend of app protection is that developers tend to implement security related function in native code instead of in Java code. Java code is easily decompiled and most of the function logic can be recovered even if the method-renaming obfuscation has been broadly used by many Android apps. In contrast with Java code, native code is more difficult to be comprehended. Thus many developers tend to hide the critical function in native code. However, to protect secret, standard crypto functions are often adopted. The domain knowledge of cryptography can be leveraged as a supplementary semantic information. If the crypto functions can be identified, the relevant semantics can hugely assist the understanding of the behavior of the app.

There are two ways for an interface to fulfil cryptographic function: one is to take advantage of the Java Cryptography Architecture (JCA) which provides cryptographic services and specifies the developers how to invoke Cryptographic APIs on Android platform. The other is to implement a cryptographic function in native code directly. For those two cases, our investigation indicates more than 28.8% (52 of 180) of native library included at least one crypto encryption function.

## 3 NativeSpeaker

In this section we describe the design and architecture the proposed NativeSpeaker native code security analysis system. The workflow of NativeSpeaker system is depicted in Fig. 1. It first extracts all native code shared libraries from apps, then dedpulicates native libraries from same code base via semantic similarity comparison techniques. After the deduplication, a dataset of native code libraries is built. Then, NativeSpeaker conducts both *Java cryptography architecture (JCA) interface analysis* and *bitwise operation analysis* to locate crypto functions in those libraries. If crypto functions are found in a native code shared library, they are further checked with a crypto algorithm analysis–checking the use of obsolete crypto algorithms and incorrect crypto parameters. With the entire analysis workflow, NativeSpeaker could help analyst pinpoint typical crypto misuses such as insecure encryption mode and non-random crypto key in native code.

In the following, we detailed the process of how NativeSpeaker analyzes native code and search crypto misuses.

### 3.1 Preprocessing of Native Library

**Obtaining ARM Binary Code.** Notice that different Android devices use different CPUs, which in turn support different instruction sets. To adapt multiple

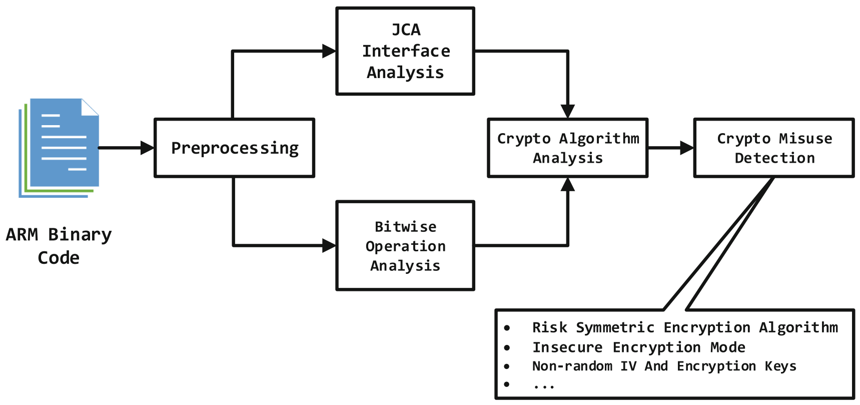
JCA Interface Analysis
ARM Binary Code
Preprocessing
Crypto Algorithm Analysis
Crypto Misuse Detection
Bitwise Operation Analysis
Risk Symmetric Encryption Algorithm
Insecure Encryption Mode
Non-random IV And Encryption Keys
...

**Fig. 1.** Workflow of NATIVESPEAKER

architectures of mobile devices, Android NDK supports a variety of Application Binary Interfaces (ABIs) such as *armeabi*, *armeabi-v7a*, *arm64-v8a*, *x86*, *x86_64*, *mips*, *mips64*, etc. As a result, a released APK usually contains multiple native code shared libraries with same functions. To simplify the analysis work, in our analysis only the most frequently used ARM version is analyzed if multiple libraries with same functions are extracted.

We mainly searched for ARM binary files in app's `/lib` and `/lib/armabi(XXX)` directories, which are the default directories for developers to store their native shared libraries. In addition, we found in those directories not all files are with the same file extension (.so). As a result, we further checked the file header to find those files started with ELF magic number (`7f454c46010101`). Thus even if some apps change a regular shared library's file extension to others such as *.xml* and *.dex* to hide it, our analysis would not miss it.

**Native Library Deduplication.** Among the 20,353 extracted native code files, many of them are the same (or adjacent versions) libraries integrated by different APKs. If duplicated libraries can be excluded, the amount of analysis could be reduced significantly.

However, it is not trivial to find duplicated libraries. We argue that using simple rules such as judging with file hashing is inadequate to deduplicate similar libraries. In general, two classes of similarity are considered:

(i) **Libraries compiled by different developers:** If two apps are developed by different developers, the integrated native code libraries are possibly compiled using different compilation kits but are from the same code base. In this case, libraries are slightly different and file hashing is not able to handle this similarity.
(ii) **Libraries of different versions:** Same libraries with adjacent code versions (e.g., ver 0.9 and 1.0) in our analysis are considered as similar ones. The product iteration often update its integrated native libraries, but the contents of the updated libraries are often similar to the old ones.

To cluster similar native code libraries efficient, we utilized an interface-based analysis to judge whether two libraries are similar. Given a native code library, four attributes can be considered to judge its provenance: **a** file hash, **b** file names, **c** author signature, and **d** interface names. The first three attributes of one particular native code library change frequently, but the names of exported interfaces are often consistent. Hence we make use of them to cluster similar libraries. First, we generate for each library files an *interface set*, which contains all function names extracted from its export table. Then, two sets from different libraries are compared if both sets contain at least 20 function names. We consider two files as the same library of different versions if a high overlap percentages (90+%) of their interface sets is found, and only choose the latest one as our analysis target.

In addition, even if two libraries are similar according to our analysis, we still tend to analyze them respectively if each library is integrated by more than 10 different apps. In this case, we believe that these frequently used libraries affect enough apps and should be meticulously checked.

## 3.2 Crypto Function Recognition

The problem of crypto function identification in binary programs of desktop platform has been studied previously for different motivations. But implementation of the cryptographic function in mobile platform is different from other platforms. In Android apps, crypto function can be implemented through two styles: Java style and Native style. To implement in the Java style, native code invokes crypto APIs in Java layer through JNI; to implement in the Native style, the crypto algorithms are directly developed using C/C++ and then compiled into native functions. In the following, we present the identification of crypto function implemented in each style, respectively.

**Java Style Crypto Identification.** JNI allows native code to interact with the Java code to perform actions such as calling Java methods and modifying Java fields. Developers can complete cryptographic functions by leveraging the Java Cryptography Architecture (JCA) which provides cryptographic services and specifies the developers how to invoke Cryptographic APIs on Android platform. The JCA uses a provider-based architecture and contains a set of APIs for various purposes, such as encryption, key generation and management, certificate validation, etc. We first use a concrete example to illustrate how native code invoke JCA API in Java layer. As Listing 1.1 shows, the entire sample involves the phase of initializing crypto key (Line 1–5), choosing crypto algorithm (Line 6–13), initializing IV, and executing encryption routine.

Regulated by the invoke convention of JNI, before executing each method, the following procedures are essentially employed: First, the code utilizes the *FindClass* method to search the class containing methods to be invoked. The, the *GetMethodID* is used to find the ID of specific method. Finally, a *CallObjectMethod* is invoked to conduct the concrete encryption (i.e., DES encryption).

```

...
KeyClass = env->FindClass("javax/crypto/spec/SecretKeySpec");
KeyInitMethodId = env->GetMethodID(KeyClass,
            "<init>",
            "([BLjava/lang/String;)V");
KeyObj = env->NewObject(KeyClass, KeyInitMethod, key);
CipherClass = env->FindClass("javax/crypto/Cipher");
CipherInstance = env->CallStaticByteMethod(CipherClass,
            "getInstance",
            "(Ljava/lang/String;)Ljavax/crypto/Cipher;",
            env->NewStringUTF("DES"));
DesInstance = env->CallStaticObjectMethod(CipherClass,
            CipherInstance,
            env->NewStringUTF("DES/CBC/PKCS5Padding"));
...
/*IV initialization here*/
...
DofinalMethod = env->GetMethodID(CipherClass,
            "doFinal",
            "([B)[B");
result = env->CallObjectMethod(DesInstance,
            DofinalMethod,
            msg);
```

**Listing 1.1.** Java crypto example

The identification of Java style crypto function in native code mainly relies on the string information of the JNI parameters. We can capture the JNI parameters involved and locate relevant JNI invoking. A total number of 230 frequently invoked JNI methods (see Appendix A) are monitored to collect such parameters. After obtaining the information, we further build a (*method*, *parameter*) tuple. For all tested functions in native code, we collect tuples from them and analyze them. If we find a (*FindClass*, *javax/crypto/Cipher*) tuple, the host function is expected to invoke crypto APIs in Java layer. Moreover, we can further analyze collected tuples to help recover the information of used crypto algorithms and operation modes. For instance, if a (*NewStringUTF*, *AES/CBC/PKCS5Padding*) tuple is found close to the (*FindClass*, *javax/crypto/Cipher*) tuple, it implies that *java.crypto.cipher* executes an AES-CBC encryption/decryption operation.

Although there are many crypto schemes such as *javax.crypto.Cipher*, *Bouncy Castle*, and *Spongy Castle* that support crypto operations in Java layer. Our analysis only observed the situation of native code utilizing *javax.crypto.Cipher*. Therefore, we only focus on the situation of using *javax.crypto.Cipher*. If any other crypto providers are involved, similar patterns can also be included.

**Native Style Crypto Identification.** Compared to Java style crypto code, native style crypto functions possess less features. They are generally implemented in C or C++ and are compiled to assemble code. The identification of such crypto functions is actually a procedure of understanding certain semantics in ARM binary code. In this case, there is no general standard of cryptographic cipher coding template. Recent researches have proposed multiple techniques on identifying crypto primitives. To meet our requirement, a technique to identify both symmetric cryptography and public key cryptography with an acceptable overhead is expected. We compared these approaches and choose the approach proposed by Caballero *et al.* [10], which utilizes a heuristic detection to locate potential crypto functions. The intuition of this approach is that the substantive characteristics of cryptographic and encoding methods, a high proportion of bitwise operations is necessary, and for ordinary methods, bitwise operation would hardly be used. This standard then leads to a efficient static analysis that could identify both symmetric and asymmetric crypto algorithms.

In detail, we re-implement Caballero approach through the following five steps: First, we statically disassemble the native code library to label every function. Second, each function is divided into several basic blocks and for each basic block, the number of contained instructions and that of bitwise instructions are counted. Third, the ratio of bitwise instructions to all instructions in one basic block is calculated. If the ratio exceeds a particular threshold (e.g., 55%), this basic block is considered as a potential crypto related block. Fourth, if a function contains more than one crypto related block, this function is labeled as a potential crypto/encoding function. Finally, we conduct a function-name heuristic filtering to determine whether a potential crypto/encoding function is actually a crypto function. Since most crypto functions are exported by the native libraries to provide particular functionalities, we believe a function without exporting information (i.e., interface name) is unnecessary to be analyzed. We directly remove those potential crypto/encoding functions without an exported function name. Then, we make use of words segmentation technique to handle function name as the following sample shows:

```
VP8Lencodestream => VP 8L encode stream
```

An N-gram algorithm [3,17] is employed to all function names collected. If the words set of one function contains names of mainstream crypto ciphers such as AES, DES, DESede, blowfish, RC4, RSA, etc., this function is labeled as a crypto function. To improve the efficiency, we optimize the adopted N-gram algorithm through carefully choosing the data set used. The words segmentation data set we used is a subset of Linguistic Data Consortium data set. Our data set contains 333,000 unigram words, and 250,000 bigram phrases.

Notice that using simple string matching or regular expression matching could accelerate the analysis, this however causes false positive. Take the function of *VP8Lencodestream* as an example, it contains a "des" substring but is actually not a crypto function.

### 3.3 Cryptographic Misuse Detection

cryptographic misuse can be very diverse and complex. In this paper, we only focus on crypto misuse of symmetric encryption algorithms. We refer to flaw model from the study of Shuai et al. [20] and mainly concern about two kinds of misuses: the **misuse of crypto algorithm** and the **misuse of crypto parameters**. The misuse of crypto algorithm include the case of using obsolete crypto algorithms such as DES and MD5. This kind of misuse often leads to brute-force attacks. The misuse of crypto parameters include the case of using non-random key material and the case of using improper mode. Using non-random key or IV directly leads to a weak or broken cryptosystem, while the improperly used mode such as ECB significantly weakens the security of adopted cryptosystem.

To detect crypto misuse in native code, we employ a series of analyzing strategies as follows:

**Non-random Key Material.** Using a non-random cryptographic key material to deduce crypto key, or directly using hard-coded cryptographic keys for encryption, is a severe and critical mistake of crypto engineering practice. However, this situation is still popular due to the reasons of ignorant developers or misunderstanding of cryptography. Also, using a non-random Initialization Vector (IV) with Cipher Block Chaining (CBC) Mode causes algorithms to be vulnerable to dictionary attacks. If an attacker knows the IV before he specifies the next plaintext, he can check his guess about plaintext of some blocks that was encrypted with the same key before. In order to find such misuse of key or IV in native cryptographic functions, we proposed a simple data dependency analysis approach. In List 3.3, the example demonstrates how developer uses a fixed string as key of the *DES_Encrypt_string* function. Through checking the function name (*DES_Encrypt_string*, an exported function in native library), the use of certain crypto algorithms has been located. Then, we follow a simple rule that all parameters of a symmetric crypto function should be dynamically generated. We conduct a simple intra-procedural data flow analysis only focus on the caller of the crypto function. If a parameter of crypto function is generated without involving of the caller function's parameters (i.e., random information from outside) or system APIs, a potential warning of key misuse is reported. Then we can conduct a manual verification to assure the misuse.

```
msg = JNIEnv::GetStringUTFChars(*env, msg_input, 0);
msg_len = strlen(msg);
DES_Encrypt_string(msg, msg_len + 1, "akazwlan", &output);
base64_encode(output, &base64_output, out_len);
result = JNIEnv::NewStringUTF(*env, (const char *)&base_output);
```

**Improper Encryption Mode.** The use of vulnerable modes such as Electronic Code Book (ECB) in symmetric encryption is common. For Java style crypto functions, we can obtain the encryption mode through analyzing its JNI parameters, searching certain string related to vulnerable mode (e.g., AES/ECB) and pinpoint typical misuses. However, this approach is not effective when analyzing native style crypto function if the implementation does not regulate the format encryption mode. We address this through a heuristic detection: we observe that most ECB encryptions are implemented within a loop to handle long messages. In this loop, the message parameter is directly handled by the encryption routine instead of firstly masked by the IV. Hence, we first identify the encryption routine with its name, and then check the caller function to search whether the encryption routine is invoked in a loop. If so, the parameter of the routine is checked with any related exclusive-or operation to find potential IV. The missing of IV implies a misuse of ECB mode.

**Obsolete Algorithms.** To find obsolete crypto algorithms, the major source of information is the function name. Notice that we can obtain the name of crypto algorithms from JNI parameters and the exported interfaces, which indicates for both Java style and Native style crypto functions the used obsolete algorithms can be searched. Although this method is straight-forward, it is effective to find typical misuse of crypto algorithm.

## 4 Evaluation

### 4.1 Dataset

To evaluate whether NativeSpeaker is able to analyze native code third-party libraries and find crypto misuses, we build an app dataset with a corpus of 20,000 popular Android apps downloaded from *myapp*, the largest non-official Android APP market. We unpacked apps and extract 20,353 ARM native code shared library files. We observed due to the strict regulation of Google Play market, many popular apps are not uploaded. Instead, users are often leaded to website of third-party non-official Android app markets to download them. Moreover, some apps are published to both Google Play market and non-official Android app markets, and the released versions for non-official Android app markets are usually different from that for Google Play market (e.g., add some functionalities not allowed by Google Play market). As a result, we choose to build the dataset through collecting apps from a non-official Android app market to cover more apps in use and find more potential flaws.

In our dataset, the chosen apps possess the following features: (1) Each app had been downloaded by at least 30,000 times. The top 12.5% (2,496 in 20,000) own more than one million users. (2) The category of these apps are various including shopping, gaming, news, traveling, social contacting, etc., Therefore, if any flaw is found in those apps, its influence is significant and it is expected to infect a huge amount of users.

### 4.2 Native Code Analysis

After unpacking the 20,000 apps and extracting the 20,353 native code shared libraries, we further disassembled those binaries to collect more information. We leveraged the *objdump* utility and the state-of-the-art disassembler IDA Pro (version 6.95) to analyze the collected native code. We found that among all shared libraries, there were 279 malformat files containing no export function information. Our manual analysis revealed that those files adopt native code packing protection. In addition, there are 13 files adopting anti-analysis protection and IDA Pro is not able to disassemble them. Since the code protection issues are outside the scope of this paper, we choose to ignored those failure cases.

**Table 2.** The 15 most popular Android native code libraries

| Library Name | Occurrence | Functional Description |
|---|---|---|
| liblocSDK6a.so | 2721 | Geographic Information System service |
| ibbspatch.so | 2017 | Incremental updating |
| libunity.so | 1205 | Game engine |
| libmono.so | 1203 | Game engine |
| libweibosdkcore.so | 1156 | Social networking services |
| libtpnsSecurity.so | 1064 | Security service |
| libtpnsWatchdog.so | 972 | Security service |
| libmain.so | 935 | Game engine |
| libgame.so | 882 | Game engine |
| libBugly.so | 847 | Crash information service |
| libcocklogic.so | 766 | Task restart service |
| libidentifyapp.so | 750 | Security service |
| libcasdkjni.so | 743 | In-app payment service |
| libgetuiext.so | 665 | Push service |
| libcocos2dcpp.so | 636 | Game engine |

For the rest files, we then conducted a library deduplication process. According to this analysis, 5,970 unique libraries were finally determined. For those library files, we build their profiles including interface information, disassembled code, and call graph through an automated analysis with IDAPython [2].

The deduplication of library significantly reduces the amount of analysis. As Table 2 shows, each of the 15 most popular third party native code libraries is frequently integrated by at least 600 different apps. In this situation, deduplicating those repeatedly used library files saves unnecessary expenses.

## 4.3 Cryptographic Algorithm Recognition

We run our analysis on a HP Z840 machine, with an Intel Xeon E5-2643 v3, 12-thread processor. We use twelve threads to run analysis task concurrently, the average time to analyze an so file is 4.75 s, and the cost of two function name filtering is 30 ms, which is within an acceptable range.

The function boundary and disassembled code was generated by IDA, even though identificating function boundary and resulting disassembled code with IDA Pro is not perfect, it is sufficient in our scenario.

*Java Crypto.* In Java cryptographic function recognition, we select Java class "javax.crypto.Cipher" as identification symbol. There are more than one way to implement Java cryptographic function in Java code, such as Bouncy Castle [1] and Spongy Castle [5], and these crypto libraries can be used in native code theoretically. But in our research, we didn't find any Java cryptographic method implemented without the "javax.crypto.Cipher" class. In this Java class, developers could realize cryptographic ciphers such as AES (CBC), AES (ECB), DES (CBC), DES (ECB), DESede (CBC) and DESede (ECB), all these ciphers could be recognized by our work.

In the experiment, we found a total of 47 libraries using JNI interface to invoke Java's cryptographic algorithms, there are 122 such cryptographic algorithms. The result is shown in Table 3. It is noteworthy that DES act as a cryptographic algorithm that obey best practice principles are still used widely, besides in the commonly used cryptos, blowfish, RC2, 3DES are all outdated cryptographic algorithms.

**Table 3.** Java cryptographic function occurrence

| Java cryptographic function | Encrypt mode | Occurrence |
|---|---|---|
| AES | CBC | 5 |
| AES | ECB | 10 |
| AES | CFB | 15 |
| AES | None | 30 |
| DES | ECB | 7 |
| DES | None | 12 |
| DESede | CBC | 4 |
| DESede | ECB | 14 |
| DESede | None | 2 |
| RSA | ECB | 23 |

Among these cryptographic algorithms, AES has the highest usage, but most of the scenarios that use AES are to encrypt a short string of information such as a string, so the encryption mode is not used. In addition to AES, DES and 3DES algorithm are frequently used as well. We also found 23 cases using RSA for encryption.

*Native Crypto.* In our experiment, the lowest value of the instruction number in Native Style Crypto Identification is 20, we select the threshold as 50% and get 100,218 encryption/encoding functions. After our first function name filtering, the remaining number of encryption/encoding functions with function names is 42,897, and it reduces to 13,642 after the word segmentation. The result is illustrated as Fig. 2.

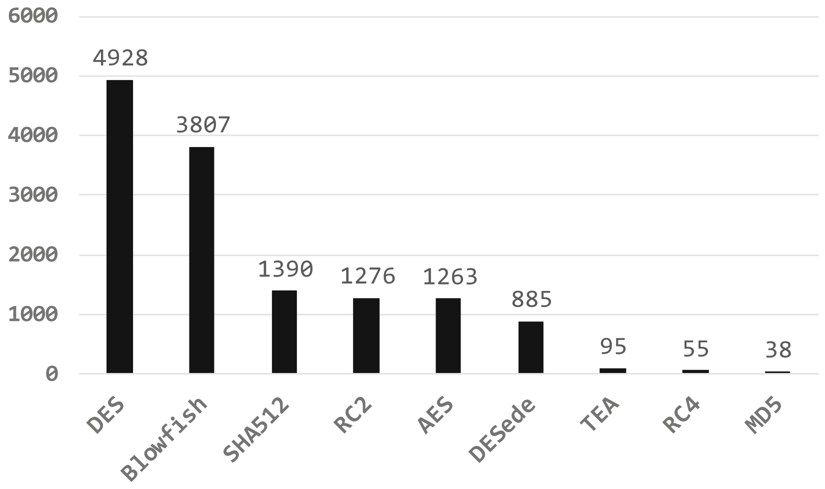

**Fig. 2.** Native crypto result

Obviously, the Java cryptographic functions usage is significantly less than the native cryptos. Reason for this may be the coding complexity for Java code in native programming environment is very high, and there are many mature cryptographic functions written in C language, such as OpenSSL project [4] (Table 4).

**Table 4.** Cryptographic misuse in top 60 libraries

| Function misuse | | Encryption mode misuse | Parameter misuse | | |
|---|---|---|---|---|---|
| DES | MD5 | ECB | Key | IV | Hard-coded Key |
| 3 | 2 | 8 | 16 | 11 | 4 |

## 4.4 Cryptographic Misuse Detection

In order to analyze the misuse of cryptographic algorithms implemented in native code, we performed manual analysis to 60 most frequently used shared libraries implementing cryptographic algorithms. 21 of the 60 libraries misused cryptographic algorithms, 3 of them used the obsolete DES algorithm, 16 of them adopted the insecure ECB mode for encryption, we also found 27 cases using predictable keys or IVs, 4 of them hard-coded the cryptographic key.

**Case Studies.** We describe a typical example which results in insecure communication in real world to illustrate the dangers of crypto misuse. In this example, native libraries implement cryptographic algorithms incorrectly by using non-random keys or IVs. We identify such libraries in three steps:

(i) We collect the shared libraries using communication-related APIs (e.g. socket, send, sendto) in our dataset.
(ii) We analyze whether the collected libraries use non-random keys or IVs when implementing cryptographic algorithms.
(iii) We further identify the shared libraries using such insecure cryptographic algorithms to encrypt communication traffic.

We found a total of 13 native libraries existing such problems. We attempted to decrypt the traffic sent out from these shared libraries, and successfully restored the encrypted traffic from eight native libraries, the traffic content included audio, video, program running information and so on. We couldn't trigger sockets in the remaining five cases, these libraries contained associated code but provided no relevant call interface. We think these code may be deprecated or remains to be further developed in the future, and does not affect the correctness of our identification scheme.

"*Libanychatcore.so*" is extracted from the *Anychat* SDK, it is used for transmitting audio and video, and the content is encrypted by AES. We find it hard-codes secret keys when analyzing its implementation of cryptographic algorithms, the hard-coded key is "*BaiRuiTech.Love*". This shared library is found to be used by a popular stock app named *DaZhiHui*, which has been downloaded more than 30 million times. We are able to decrypt the network traffic from *DaZhiHui* with the extracted hard-coded key.

"*Libgwallet.so*" is a shared library used for in-app payments in games released by *GLU Mobile*. It synchronizes data with the server, and the communication is encrypted by AES. In our analysis, we find it uses a hard-coded key "*3A046BB89F76AC7CBA488348FE64959C*" and a fixed IV "*Glu Mobile Games*" for encryption. This shared library is used by more than 10 game apps for synchronizing payment data with their servers. We conduct traffic analysis to these apps and decrypt their payment data successfully.

## 5 Related Work

– **Android Native Code:** The sandboxing mechanisms can be feasible and useful in restraining privileged API invoking from native code. NativeGuard [21] is a framework isolates native libraries into a non-privileged application, dangerous behavior of native code would be restricted by the sandbox mechanism. Afonso et al. [7] complement the sandboxing mechanisms and generate a native code sandboxing policy to block malicious behavior in realworld applications. While for security flaws like crypto misuses in native code, the sandboxing mechanisms are less effective.
  Virtual machine based dynamic analysis platform provide a feasible way to track information flow and implement dynamic taint analysis. Henderson et al. [14] proposed a virtual machine based binary analysis framework, it provides whole-system dynamic binary analytical ability, for Android platform, they include DroidScope [22] as an extension. DroidScope is a dynamic analysis platform based on the Android Emulator, it reconstructs both the OS-level

and Java-level semantics simultaneously and seamlessly, and enable dynamic instrumentation of both the Dalvik bytecode as well as native instructions. NDroid [19] tracks information flows cross the boundary between Java code and native code and the information flows within native codes.
All these dynamic analysis platform may incur 5 times overhead at least, meanwhile, these techniques are not domain-specific and thus are less effective for assessing crypto misuse in native code of apps.
- **Cryptographic Misuse in Android Applications:** A number of efforts have been made to investigate the cryptographic misuse problem in Android Java code. Shao et al. [20] build the cryptographic misuse vulnerability model in Android Java code, they conclude the main classes of cryptographic misuse are misuse of cryptographic algorithm, mismanagement of crypto keys and use inappropriate encryption mode. Egele et al. [11] made an empirical study of the cryptographic misuse. They adopt a light-weight static analysis approach to find cryptographic misuse in real word Android applications, but their work only targets Dalvik bytecode, therefor, applications that invoke misused cryptographic primitives from native code cannot be assessed out. Enck et al. [13] design and execute a horizontal study of Android applications, from a vulnerability perspective, they found that many developers fail to take necessary security precautions. Our analysis complements all these research efforts by performing an in-depth analysis focused on native code.
- **Third Party Library Detection:** Backes et al. [9] proposes a Android Java library detection technique that is resilient against common code obfuscations and capable of pinpointing the exact library version used in apps. Li et al. [15] collect a set of 1,113 libraries supporting common functionality and 240 libraries for advertisement from 1.5 million Android applications, they investigated several aspects of these libraries, including their popularity and their proportion in Android app code. Li et al. [16] utilizes the internal code dependencies of an Android app to detect and classify library candidates, their method is based on feature hashing and can better handle code obfuscation.

## A Appendix

## Monitored JNI Functions

| | | | |
|---|---|---|---|
| AllocObject | CallStaticBooleanMethod | GetDoubleArrayRegion | NewObjectA |
| CallBooleanMethod | CallStaticBooleanMethodA | GetDoubleField | NewObjectArray |
| CallBooleanMethodA | CallStaticBooleanMethodV | GetFieldID | NewObjectV |
| AllocObject | CallStaticBooleanMethod | GetDoubleArrayRegion | NewObjectA |
| CallBooleanMethod | CallStaticBooleanMethodA | GetDoubleField | NewObjectArray |
| CallBooleanMethodA | CallStaticBooleanMethodV | GetFieldID | NewObjectV |
| CallBooleanMethodV | CallStaticByteMethod | GetFloatArrayElements | NewShortArray |
| CallByteMethod | CallStaticByteMethodA | GetFloatArrayRegion | NewString |
| CallByteMethodA | CallStaticByteMethodV | GetFloatField | NewStringUTF |
| CallByteMethodV | CallStaticCharMethod | GetIntArrayElements | NewWeakGlobalRef |
| CallCharMethod | CallStaticCharMethodA | GetIntArrayRegion | PopLocalFrame |
| CallCharMethodA | CallStaticCharMethodV | GetIntField | PushLocalFrame |
| CallCharMethodV | CallStaticDoubleMethod | GetJavaVM | RegisterNatives |
| CallDoubleMethod | CallStaticDoubleMethodA | GetLongArrayElements | ReleaseBooleanArrayElements |
| CallDoubleMethodA | CallStaticDoubleMethodV | GetLongArrayRegion | ReleaseByteArrayElements |
| CallDoubleMethodV | CallStaticFloatMethod | GetLongField | ReleaseCharArrayElements |
| CallFloatMethod | CallStaticFloatMethodA | GetMethodArgs | ReleaseDoubleArrayElements |
| CallFloatMethodA | CallStaticFloatMethodV | GetMethodID | ReleaseFloatArrayElements |
| CallFloatMethodV | CallStaticIntMethod | GetObjectArrayElement | ReleaseIntArrayElements |
| CallIntMethod | CallStaticIntMethodA | GetObjectClass | ReleaseLongArrayElements |
| CallIntMethodA | CallStaticIntMethodV | GetObjectField | ReleasePrimitiveArrayCritical |
| CallIntMethodV | CallStaticLongMethod | GetPrimitiveArrayCritical | ReleaseShortArrayElements |
| CallLongMethod | CallStaticLongMethodA | GetShortArrayElements | ReleaseStringChars |
| CallLongMethodA | CallStaticLongMethodV | GetShortArrayRegion | ReleaseStringCritical |
| CallLongMethodV | CallStaticObjectMethod | GetShortField | ReleaseStringUTFChars |
| CallNonvirtualBooleanMethod | CallStaticObjectMethodA | GetStaticBooleanField | reserved1 |
| CallNonvirtualBooleanMethodA | CallStaticObjectMethodV | GetStaticByteField | reserved2 |
| CallNonvirtualBooleanMethodV | CallStaticShortMethod | GetStaticCharField | reserved3 |
| CallNonvirtualByteMethod | CallStaticShortMethodA | GetStaticDoubleField | SetBooleanArrayRegion |
| CallNonvirtualByteMethodA | CallStaticShortMethodV | GetStaticFieldID | SetBooleanField |
| CallNonvirtualByteMethodV | CallStaticVoidMethod | GetStaticFloatField | SetByteArrayRegion |
| CallNonvirtualCharMethod | CallStaticVoidMethodA | GetStaticIntField | SetByteField |
| CallNonvirtualCharMethodA | CallStaticVoidMethodV | GetStaticLongField | SetCharArrayRegion |
| CallNonvirtualCharMethodV | CallVoidMethod | GetStaticMethodID | SetCharField |
| CallNonvirtualDoubleMethod | CallVoidMethodA | GetStaticObjectField | SetDoubleArrayRegion |
| CallNonvirtualDoubleMethodA | CallVoidMethodV | GetStaticShortField | SetDoubleField |
| CallNonvirtualDoubleMethodV | DefineClass | GetStringChars | SetFloatArrayRegion |
| CallNonvirtualFloatMethod | DeleteGlobalRef | GetStringCritical | SetFloatField |
| CallNonvirtualFloatMethodA | DeleteLocalRef | GetStringLength | SetIntArrayRegion |
| CallNonvirtualFloatMethodV | DeleteWeakGlobalRef | GetStringRegion | SetIntField |
| CallNonvirtualIntMethod | EnsureLocalCapacity | GetStringUTFChars | SetLongArrayRegion |
| CallNonvirtualIntMethodA | ExceptionCheck | GetStringUTFLength | SetLongField |
| CallNonvirtualIntMethodV | ExceptionClear | GetStringUTFRegion | SetObjectArrayElement |
| CallNonvirtualLongMethod | ExceptionDescribe | GetSuperclass | SetObjectField |
| CallNonvirtualLongMethodA | ExceptionOccurred | GetVersion | SetShortArrayRegion |
| CallNonvirtualLongMethodV | FatalError | IsAssignableFrom | SetShortField |
| CallNonvirtualObjectMethod | FindClass | IsInstanceOf | SetStaticBooleanField |
| CallNonvirtualObjectMethodA | FromReflectedField | IsSameObject | SetStaticByteField |
| CallNonvirtualObjectMethodV | FromReflectedMethod | MonitorEnter | SetStaticCharField |
| CallNonvirtualShortMethod | GetArrayLength | MonitorExit | SetStaticDoubleField |
| CallNonvirtualShortMethodA | GetBooleanArrayElements | NewBooleanArray | SetStaticFloatField |
| CallNonvirtualShortMethodV | GetBooleanArrayRegion | NewByteArray | SetStaticIntField |
| CallNonvirtualVoidMethod | GetBooleanField | NewCharArray | SetStaticLongField |
| CallNonvirtualVoidMethodA | GetByteArrayElements | NewDirectByteBuffer | SetStaticObjectField |
| CallNonvirtualVoidMethodV | GetByteArrayRegion | NewDoubleArray | SetStaticShortField |
| CallObjectMethod | GetByteField | NewFloatArray | Throw |
| CallObjectMethodA | GetCharArrayElements | NewGlobalRef | ThrowNew |
| CallObjectMethodV | GetCharArrayRegion | NewIntArray | ToReflectedField |
| CallShortMethod | GetCharField | NewLocalRef | UnregisterNatives |
| CallShortMethodA | GetDirectBufferAddress | NewLongArray | |
| CallShortMethodV | GetDoubleArrayElements | NewObject | |

## References

1. Bouncy castle. https://www.bouncycastle.org/
2. Ida-python. https://github.com/idapython/src/
3. N-gram wiki. https://en.wikipedia.org/wiki/N-gram
4. openssl project. https://www.openssl.org/
5. Spongy castle. https://rtyley.github.io/spongycastle/
6. Acar, Y., Backes, M., Bugiel, S., Fahl, S., McDaniel, P., Smith, M.: SoK: lessons learned from android security research for appified software platforms. In: 2016 IEEE Symposium on Security and Privacy (SP), pp. 433–451. IEEE (2016)
7. Afonso, V.M., de Geus, P.L., Bianchi, A., Fratantonio, Y., Kruegel, C., Vigna, G., Doupé, A., Polino, M.: Going native: using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In: NDSS (2016)
8. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Acm Sigplan Not. **49**(6), 259–269 (2014)
9. Backes, M., Bugiel, S., Derr, E.: Reliable third-party library detection in android and its security applications. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 356–367. ACM (2016)
10. Caballero, J., Poosankam, P., Kreibich, C., Song, D.: Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, pp. 621–634. ACM (2009)
11. Egele, M., Brumley, D., Fratantonio, Y., Kruegel, C.: An empirical study of cryptographic misuse in android applications. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pp. 73–84. ACM (2013)
12. Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. Comput. Syst. (TOCS) **32**(2), 5 (2014)
13. Enck, W., Octeau, D., McDaniel, P.D., Chaudhuri, S.: A study of android application security. In: USENIX Security Symposium, vol. 2, p. 2 (2011)
14. Henderson, A., Prakash, A., Yan, L.K., Hu, X., Wang, X., Zhou, R., Yin, H.: Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, pp. 248–258. ACM (2014)
15. Li, L., Bissyandé, T.F., Klein, J., Le Traon, Y.: An investigation into the use of common libraries in android apps. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, pp. 403–414. IEEE (2016)
16. Li, M., Wang, W., Wang, P., Wang, S., Wu, D., Liu, J., Xue, R., Huo, W.: LibD: scalable and precise third-party library detection in android markets. In: Proceedings of the 39th International Conference on Software Engineering, pp. 335–346. IEEE Press (2017)
17. Meng, X., Miller, B.P.: Binary code is not easy. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, pp. 24–35. ACM (2016)

18. Mochihashi, D., Yamada, T., Ueda, N.: Bayesian unsupervised word segmentation with nested Pitman-Yor language modeling. In: Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: vol. 1, pp. 100–108. Association for Computational Linguistics (2009)
19. Qian, C., Luo, X., Shao, Y., Chan, A.T.: On tracking information flows through JNI in android applications. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 180–191. IEEE (2014)
20. Shuai, S., Guowei, D., Tao, G., Tianchang, Y., Chenjie, S.: Modelling analysis and auto-detection of cryptographic misuse in android applications. In: 2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing (DASC), pp. 75–80. IEEE (2014)
21. Sun, M., Tan, G.: NativeGuard: protecting android applications from third-party native libraries. In: Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks, pp. 165–176. ACM (2014)
22. Yan, L.-K., Yin, H.: DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic android malware analysis. In: USENIX Security Symposium, pp. 569–584 (2012)