A Comparative Study of Misapplied Crypto in Android and iOS Applications

Johannes Feichtner[1,2]
[1]*Institute of Applied Information Processing and Communications (IAIK), Graz University of Technology, Austria*
[2]*Secure Information Technology Center – Austria (A-SIT), Austria*
johannes.feichtner@iaik.tugraz.at

Keywords: Static Analysis, Slicing, Android, iOS, Cryptography, Application Security

Abstract: Many applications for Android and iOS process sensitive data and, therefore, rely on cryptographic APIs natively provided by the operating system. For this to be effective, essential rules need to be obeyed, as otherwise the attainable level of security would be weakened or entirely defeated. In this paper, we inspect the differences between Android and iOS concerning the proper usage of platform-specific APIs for cryptography. For both platforms, we present concrete strategies to detect critical mistakes and introduce a new framework for Android that excels in pinpointing the origin of problematic security attributes. Applied on real-world apps with cryptography, we find that out of 775 investigated apps that vendors distribute for both Android and iOS, 604 apps for iOS (78%) and 538 apps for Android (69%) suffer from at least one security misconception.

1 INTRODUCTION

In recent years, Android and iOS have emerged as the two leading operating systems in the smartphone industry. Many applications for these platforms perform security-critical tasks and require that sensitive data is processed reliably. Sadly, there is little information on how responsibly apps treat personal information, such as passwords and encryption keys. A wrong application of cryptography or security-critical functionality, however, may expose secrets to unrelated parties and, thereby, undermine the intended security level.

The research of security aspects on mobile platforms has received a lot of attention. A majority of publications in this field focus on the Android ecosystem where the openness of the platform promotes program inspection. This process of reverse engineering is predominantly driven by the motivation to check the existence and correct implementation of security mechanisms. Manually verifying how critical functionality has been realized can be challenging due to the rising complexity and size of today's programs. Automated solutions, on the other side, are often tailored to the inspection of particular parameters but fail to perform a conclusive identification and analysis of relevant program parts. This issue is aggravated by the heterogeneity of Android and iOS, which does not only cause distinct attack vectors but also prevents inspection tools from being re-used for both platforms.

A common method to protect sensitive information in Android and iOS apps is the use of system-provided APIs that expose cryptographic functionality. While these high-level interfaces reduce the burden of the developer to understand how cryptographic primitives work internally, it is still vital to use APIs with parameters that do not give a false sense of security. Therefore, several platform-independent rules have to be observed. Among them are for example, (1) the need to prevent the electronic code book (ECB) mode for block ciphers,(2) the need to avoid constant keys or credentials that are in the worst-case hard-coded into the app, or (3) the need to derive initialization vectors (IV) from a pseudo-random number generator.

While analyzing thousands of Android apps in 2013, researchers have defined six common types of mistakes in using cryptographic APIs and confirmed that issues were present in 88% of all inspected apps (Egele et al., 2013). Since then, approaches have been elaborated to better detect (Shao et al., 2014; Muslukhov et al., 2018) and mitigate (Ma et al., 2016) crypto API misuse in Android. Similar efforts have also been made regarding iOS apps (Feichtner et al., 2018; Li et al., 2014) where security problems have been uncovered in 82% of apps checked. As related work underlines that correct usage of system-provided APIs for crypto purposes is rather the exception than the norm, it stresses the need for further research that discloses reasons for the high frequency of misuse.

Problem. It is still unclear whether apps that are available for both Android and iOS suffer from similar implementation weaknesses. Available research of misapplied crypto in mobile apps usually does not cross the line between platforms. Thus, it is hard to argue whether API misuse is the result of a developer's lack of knowledge or ignorance, or more likely due to problematic design decisions in the system-provided API and its too complex documentation.

Moreover, most related work for Android focuses on highlighting the widespread of misapplied crypto rather than pinpointing the exact origin of problematic statements in code. While there is undoubtedly a need for tools to warn about these kinds of problems, so far they are unable to give clear advice to developers and analysts about the origin of rule-violating code.

The ability to directly compare execution traces of API invocations is important, since it is crucial to see why methods are invoked with cryptographically weak values. In addition, also with regards to the platform, execution traces can provide valuable insight into why some rules are significantly more violated than others.

Approach. To assess and compare the occurrence of misapplied crypto in Android and iOS apps, we pursue a three-stage process:

1. Streamline application analysis for both platforms by establishing a workflow and output format that ensures inspection results on Android and iOS are comparable. To achieve that, we designed and implemented a framework that bridges the previously defined gap in analyzing Android apps.
2. Based on a general set of crypto API rules not to violate, we identify detection strategies that hold on both platforms. Due to the different design of Android and iOS, individual constraints apply regarding parameters crypto API methods can take.
3. Using analysis frameworks that incorporate the predefined rules, we conduct an automated study of misapplied crypto on a manually compiled set of apps that include cryptography and that are officially distributed for both Android and iOS.

Results. We manually collected a set of 775 mobile apps that apparently originate from the same vendor and are distributed in the official stores of Android and iOS. All of them had at least 1,000 installations or ratings on each platform, and their use of cryptography seemed obvious, i.e., password managers or secure messengers. We found that 78% or 604 apps for iOS and 69% or 538 apps for Android violated at least one basic security rule. The analysis also showed strong indications that the prevalence of some issues is almost solely based on design decisions of the API.

Contribution. Our contributions are as follows:

1. We introduce a new framework to automatically dissect Android apps, identify common misuse of crypto APIs and exactly pinpoint the origin of wrongly chosen security-relevant attributes [1]. As security rules might change over time or by APIs being updated, we propose a modular design that allows for easy adaptability to future needs. The analysis output shows a precise data flow of parameters that are used in calls to crypto APIs. Apart from analysts, the resulting report in XML format can also easily be understood by developers, who can use it to fix found issues quickly.
2. We derive concrete detection strategies for general security rules and assess how they can be violated on each platform. Used in combination with analysis frameworks for Android and iOS, problems can reliably be identified based on given data flows and, referring to the objective of this paper, the results on both platforms are comparable.
3. We present the first comparative evaluation of misapplied crypto in Android and iOS apps. Applied on a carefully compiled set of 775 apps for iOS and their counterparts for Android, we assess the spread of common mistakes in apps for both platforms. We ask if developers know how to use the system-provided APIs correctly and check if apps that are considered to be secure on one platform, typically fulfill the same expectations on the other.

Outline. The remainder of this paper is organized as follows. In Section 2, we discuss related work. In Section 3, we explain the structure of Android and iOS apps, cryptography APIs, and the principle of Program Slicing. In Section 4, we present our modular solution for Android to automatically identify security-relevant attributes in apps by means of static slicing. Section 5 shows precise detection strategies that have been modeled as ruleset within our framework. In combination with an existing framework for analyzing iOS apps, in Section 6, we study violations of security rules in real-world apps. Section 7 concludes our work.

2 RELATED WORK

The analysis of cryptography-related problems on mobile platforms has attracted a lot of attention in the past years. In this section, we discuss several related research that is closely related to our work.

[1] The source code of the framework is available at https://github.com/IAIK/CryptoSlice

Misapplied Crypto APIs. Evaluation of reports in Common Vulnerabilities and Exposures (CVE) revealed that 83% of all findings related to cryptography in mobile applications were due to the wrong usage of crypto APIs (Lazar et al., 2014). Focusing on the usability of different crypto libraries, other studies (Acar et al., 2017; Nadi et al., 2016) conclude that, aside from too complex interfaces and insecure defaults, typical reasons for misuse are often to be found in poor documentation, missing code samples, and further features required by the library to work securely.

Although reducing the decision-space for the choice of insecure parameters and a simplification of APIs helps in preventing mistakes, a variety of potential weaknesses remain (Chatzikonstantinou et al., 2015). However, keeping security rules that match these issues up-to-date can be challenging as APIs evolve over time. A recent work, thus, concentrated on automatically aligning security rules to changes in API design (Paletov et al., 2018).

Detecting API Misuse. Among the first tools to automatically check Android applications for misapplied crypto APIs was CryptoLint (Egele et al., 2013). Based upon the reverse-engineering framework Androguard, it dervies a control-flow graph over all functions. Subsequently, static program slicing is employed to inspect the parameters passed to cryptographic operations. Tested with almost 11,000 Android applications, the authors found that 88% of them commit at least one security mistake. As the reasoning of their rules is valid for both Android and iOS, we re-implement them using our own detection strategies. Inspired by CryptoLint, the CMA analyzer pursued a similar objective (Shao et al., 2014). Besides rules for crypto APIs, the authors also proposed to consider further security-relevant rules. Newer case studies of crypto API misuse on Android confirm that even more than 5 years after the first study, the general issue is still widespread (Muslukhov et al., 2018; Ma et al., 2016).

In comparison with Android, only few related contributions target the iOS platform. Supported by the fact that Dalvik bytecode in Android applications can be decompiled to Java code, existing tools for static analysis are easily applicable. On iOS, solutions for binary analysis have to be aligned to the specifics of Objective-C and Swift, such as control-flow decisions at runtime and computing points-to sets. In a recent study, the impact of crypto misuse was assessed for a set of iOS apps (Feichtner et al., 2018). The authors tested 417 apps and found that 82% of them violated at least one security rule. Other works for iOS concerning APIs survey the usage of private APIs or pursue a source-to-sink analysis using static and dynamic methods (Deng et al., 2015; Li et al., 2014).

Static and Dynamic App Analysis. Publications concerning mobile app analysis usually involve either dynamic or static analysis. Dynamic approaches work by monitoring the live execution of an app. On Android, tools like TaintDroid or TaintART can analyze and detect privacy leakage in the current execution path (Enck et al., 2010; Sun et al., 2016). Nevertheless, they inherently miss code paths that are not visited at runtime. With the need to manually run each app, a dynamic approach is less suited for the objective of this work and will, thus, not be considered more in-depth.

Methods for static analysis, as an alternative, typically apply taint tracking on a reverse-engineered representation of Dalvik bytecode (Android) or ARMv8 64-bit code (iOS). On Android, with a source-like representation, the main challenge is to follow all execution paths as sound and precisely as possible. Fully-fledged frameworks, such as FlowDroid (Arzt et al., 2014) and IccTA (Li et al., 2015) tackle this challenge. Unfortunately, their general design prevents them from being specifically applied using security rules. Self-contained implementations and specific output formats make them challenging to extend and, regarding our purpose, do not contribute to making analysis results between Android and iOS comparable.

3 BACKGROUND

In this section, we highlight substantial differences of applications for Android and iOS with regards to reverse-engeering. Followed by that, we present the idea of program slicing and finally focus on the use of system-provided crypto APIs on both platforms.

3.1 Reverse-Engineering Apps

The heterogeneity of Android and iOS also extends to the file formats used for apps of the corresponding platform. In the following, we point out individual characteristics and describe how apps have to be transformed to be able to perform program inspection.

3.1.1 Android

Android apps are mostly developed in Java or Kotlin. During compilation, stack-based JVM bytecode is translated to register-based Dalvik bytecode that is later interpreted by the Dalvik Virtual Machine (DVM). By reusing and eliminating repetitive function signatures, code blocks, and string values, the Dalvik compiler manages to effectively reduce the uncompressed bytecode size. As a result, all parts are merged into a single executable file named *classes.dex*.

A *classes.dex* executable consists of multiple sections. Starting with a header, specifying the file type using a magic number, and the format version number, the subsequent sections reference all strings, type identifiers, class and method signatures, fields, and method identifiers using unique IDs. The actual program code is stored in a separate *data* section.

The process of reverse-engineering Android apps usually consists in converting the Dalvik file to Java's *.class* format. Using predefined rules and constraint solving mechanisms, tools, such as *dex2jar*[2] or *enjarify*[3], translate register-based Dalvik bytecode to stack-based JVM code. As this process is non-deterministic, it often leads to spurious or wrong code translations.

Alternatively, instead of decompiling Dalvik bytecode to Java, it can be represented as Smali code. Smali is a mnemonic language to represent Dalvik bytecode in a parseable format. As it keeps the semantics of code very close to the original, it is often a preferable choice over more intricate decompilation.

3.1.2 iOS

iOS executables are packaged in the Mach-O file format and enclose at least one 64-bit executable for an ARMv8 CPU. The file is split into three sections: *Header, Load Commands*, and *Data*. The first part identifies the file as Mach-O file and includes information about the target CPU architecture. Next, the file layout and designated memory location of segments are defined. The third section finally consists of different regions with program code and sections that are loaded into memory during runtime.

In contrast to Android where apps are provided in a reversible bytecode format, iOS applications are compiled to machine code that is tailored to a particular CPU architecture. Manually inspecting the disassembled code of an iOS binary can be challenging due to the complexity and size of today's apps. Automated solutions, as an alternative, have to be aligned to the characteristics of ARMv8 code for iOS. As apps for iOS are developed in runtime-oriented languages, such as Objective-C and Swift, most control-flow decisions are made during runtime. Instead of calling methods directly or through virtual method tables, it uses a dynamic dispatch function in the runtime library.

Recent research (Feichtner et al., 2018) proposed the following workflow to reverse-engineer iOS apps: Decompile the ARMv8 disassembly of an app into LLVM IR code[4] and then compute points-to sets for all pointers using a context-insensitive approach.

[2]https://github.com/pxb1988/dex2jar

[3]https://github.com/Storyyeller/enjarify

[4]https://llvm.org/docs/LangRef.html

3.2 Program Slicing

Static slicing can be used to determine all code statements of a program that may affect a value at a specified point of execution (*slicing criterion*). The resulting *program slices* cover all possible execution paths and allow conclusions to be drawn about the functionality of the program. In our Android framework, we adopt the algorithm of (Weiser, 1981) to create slices of Smali code and to find paths from the origin of a parameter to its use, e.g., in cryptographic functions.

Weiser presented a method that models the data flow within a function using equations. Relevant variables and statements are determined iteratively. The algorithm consists of two steps (Tip, 1995):

1. *Follow data dependencies:* This step is executed iteratively, if control dependencies are found.
2. *Follow control dependencies:* Includes relevant variables of control flow statements. The first step is repeated for affected variables.

3.3 Cryptography APIs

Mobile apps rely on cryptography to protect sensitive data. In the following, key aspects of the cryptographic APIs included in Android and iOS are highlighted.

3.3.1 Android

By including the Java Cryptographic Architecture (JCA), Android supports a well-established set of security APIs. The JCA is provider-based which means that the interfaces may be implemented by multiple cryptographic engines in the background. The actual providers and available algorithms vary based on the used version of Android.

Symmetric and asymmetric encryption schemes are made available to developers through the `Cipher` class[5]. To use a specific scheme, developers provide a transformation string as an argument to the `Cipher.getInstance()` method. The parameter encloses the algorithm name, cipher mode, and padding scheme to use within the returned `Cipher` object.

3.3.2 iOS

On iOS, the platform libraries *CommonCrypto* and *Security* expose APIs to perform security-related operations. The first library provides symmetric ciphers, hash functions, and a key derivation function (PBKDF2). The second library features functionality for asymmetric ciphers, certificate handling, and keychain access.

[5]https://developer.android.com/reference/javax/crypto/Cipher

4 PINPOINTING API MISUSE IN ANDROID APPS

Designed for the inspection of single aspects in Android apps, our framework features static analysis on definable slicing patterns in order to identify and highlight the improper usage of security-relevant functionality. The modular architecture of the Java-based framework offers the ability to automatically extract, disassemble and investigate programs. By applying static backtracking techniques, the control and data flow of relevant code segments is opened up and serves as input for further evaluation. For this purpose, our framework incorporates rules (see Section 5) to make reliable assumptions about the degree of security, common cryptography-related constructions are able to provide. Besides analyzing these constructions, it is determined whether they deviate from being employed correctly. The framework also supports the manual analysis of apps by delivering well-arranged graphs, representing the static slice of user-definable patterns.

In comparison with similar solutions, such as CMA (Shao et al., 2014) or CryptoLint (Egele et al., 2013), we can give clear advice about the origin of rule-violating code, rather than being only able to confirm its existence. In experiments, we also noticed that related work tend to flag all apps that violate any rules as insecure. While this is true in the strict sense, it disregards the practical impact of rule violations: apps might use crypto APIs also for other purposes than encryption. Without looking at the actual execution paths, it might go unnoticed if encryption is only employed for obfuscation. A similar case occurs if, e.g., libraries used in apps initialize variables holding encryption keys, salt values, or IVs with insecure default values but overwrite them before actual usage. As a consequence, our framework does not draw a binary conclusion on the (in)security of apps but provides reports with *potentially problematic* execution paths, as identified by the conditions in our security rules.

The overall workflow of our framework, as illustrated in Figure 1, can be summarized as follows:

1. **Preprocessing:** We translate the Dalvik bytecode from an Android app archive in *.apk* format to Smali code and parse it to an object representation.
2. **Static Slicing:** Based on the given security rules defining abstract slicing patterns, we derive concrete slicing criteria. We perform static backtracking, follow all possible execution paths, and organize these data flows in a graph representation.
3. **Security Rule Evaluation:** Using a breadth-first search, we browse individual execution paths and check whether they comply with our security rules.

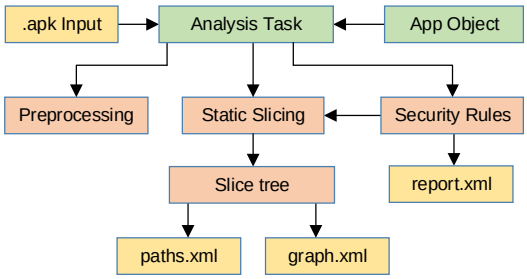

Figure 1: Analysis workflow

In the following, we provide a more detailed insight and explain the individual analysis steps.

4.1 Preprocessing

Although Android code is usually based on Java or Kotlin code, we do not try to decompile Dalvik bytecode back to Java. As pointed out by the creators of the Dare decompiler (Octeau et al., 2012), decompilation is only likely to succeed for 95% of the app classes. Instead, we disassemble Dalvik bytecode to Smali code, a register-based intermediate language. Therefore, the file *classes.dex* contained in each Android app archive is processed using the tool *baksmali*[6]. As a result, we obtain a list of files with code similar to Java, which we parse into an object representation that resembles all essential attributes and basic blocks.

4.2 Static Slicing

The ability to trace information in backward direction is a core component of our framework in order to isolate those parts of an app that are relevant regarding a specific slicing criterion. In the following, we present the implemented techniques for static slicing and highlight practical challenges.

4.2.1 Slicing Patterns

Slicing naturally depends on a slicing criterion referencing a specific line of program code. Considering our objective to track arbitrary data flows, a more generic representation is needed. Therefore, we propose so-called slicing patterns that conceptually describe an object to track in XML format.

Patterns include no references to particular program lines but comprise all necessary info to dynamically build slicing criteria. Using slicing patterns in our security rules, they become applicable for a large set of targets and lead to a multitude of slicing criteria.

[6] https://github.com/JesusFreke/smali

In the default behavior, slicing criteria are determined by searching for all `invoke` statements matching the given pattern. Therefore, all code lines of an application are scanned, looking for the provided method signature. For each match, the appendant program statement is considered as a starting point for slicing. Subsequently, the name of the register to track is located by associating the index of each occurring register with the given parameter (index) of interest. As a result, a set of suitable slicing criteria is delivered.

4.2.2 Backtracking

Backward slicing traverses all preceding code statements that have influenced a specific register at the initial slicing criterion. Starting at a particular statement, we follow all usages of a register back to the point where it is defined. This approach is commonly referred to as *Use-Definition* (or short use-def). The resulting slice models the data flows of all variables that affect the starting point.

For each criterion, first the register is identified which matches the defined parameter of interest. While backtracking the register, all involved program statements are recorded as slice nodes and added to a slice tree. Tracking ends when constants are assigned to the currently tracked register and slicing is no longer feasible. Likewise, the slicing process terminates when the tracked register is overwritten or track is lost, e.g., when referenced methods cannot be resolved.

Technically, our implementation works using a FIFO queue. The to-do list collects all registers, fields, return values, and arrays that are subject to tracking. Moreover, it holds a reference to all objects that have already been followed and excludes them from being re-processed. When requested by the slicer, the queue returns the next object to track, which includes the register to track and the location of the next opcode.

The implemented slicing algorithm is not contingent on previously generated program dependency graphs and, hence, supports fast and computationally inexpensive program analysis. Nevertheless, to leverage the potential of today's multi-core processors, the framework supports the analysis of multiple Android apps in parallel. Thereby, it is possible to inspect large amounts of apps in minimal time.

4.2.3 Slice Trees and Constants

The slicing process dynamically builds a tree with all encountered slice nodes for a specific slicing criterion. The top node is always the criterion, deduced from the pattern since it represents the root of all possible execution paths that can be modeled. Subjacent nodes stand for all code lines which are contained in the slice. In case there are multiple execution paths (e.g. if-else statement), a slice node might have links from multiple predecessor nodes. When code statements are iterated multiple times (e.g. for or while loop) cycles are induced between vertices. Each (intermediate) node involves a list of all predecessor nodes, including the originating register name and the register name, related to the current program statement.

A slice tree can comprise one or multiple leaf nodes whereas each describes either a constant or indicates an abruptly ended slicing process. Assuming that a constant value, such as an integer, an array, or a string, is copied into the tracked register, slicing may stop since the register value is redefined. For backward slicing this signifies that the tracking process has led to one or more values that affect the slicing criterion. Leaf nodes are also inserted in case slicing loses track. This happens, for instance, when registers are set as parameters in calls to unresolvable methods.

Since all disclosed path endpoints are invariant, we regard them as constants. Aside from containing information about values that are assigned to registers, constants also explain why paths end at certain points. This is achieved by retaining metadata from slicing. For example, each constant is assigned a category which clearly defines the type of the underlying value. Similarly, in case tracking stops abruptly, constants are put in place to describe the cause.

4.3 Security Rule Evaluation

Each of the implemented security rules first tries to find predefined method signatures and then, based on the found matches, verifies whether they correspond to the predefined security model. In case a rule is violated, an alert is issued, accompanied by details about the problematic statement. Thereby, the framework manages to evaluate the exact location of rule-violating code lines and writes information about the involved class, method and code to the console as well as to an XML report. As a consequence, the result is easily readable and can, moreover, be interpreted automatically by parsing the XML report. Evidently, to also consider the context in which security-related findings appear and to prevent false positives, studying linked execution paths can provide valuable insights.

The security rules are executed in consecutive order and independent from each other. The only shared feature is a reporting system, used to summarize the output of all evaluated checks. Each rule incorporates the entire workflow described in the previous sections. Due to the extensible approach, adding further security rules is supported by requiring only little effort.

5 CRYPTOGRAPHIC API MISUSE

In this section, we present detection strategies for a set of security rules we use to detect common security-critical implementation flaws in Android and iOS apps.

Similar to related research (Muslukhov et al., 2018; Feichtner et al., 2018), we investigate whether apps that use system-provided crypto APIs achieve a cryptographic notion of IND-CPA security. *Indistinguishability under a chosen plaintext attack* or IND-CPA states that attackers are unable to extract even a single bit of plaintext from a ciphertext within a certain amount of time. Encryption can be considered secure if it is IND-CPA secure (Egele et al., 2013). In practice, however, faulty implementations or wrongly-chosen parameters, e.g., a constant or hard-coded encryption key, thwart IND-CPA security. Therefore, six general rules were proposed to keep encryption secure. In the following, for each we derive concrete detection strategies that can immediately be applied for Android and iOS apps.

Rule 1: Do not use ECB mode for encryption. In ECB mode, data blocks are enciphered independently from each other and cause identical message blocks to be transformed into identical ciphertext blocks. Consequently, data patterns are not hidden and confidentiality may be compromised.

On Android, developers can request an instance of a particular cipher by passing a suitable *transformation* value as a parameter to `Cipher->getInstance()`. Typically, this value is composed of the algorithm, an operation mode, and the padding scheme to use. E.g., to request an instance providing AES in ECB mode with PKCS#5 padding, "*AES/ECB/PKCS5Padding*" has to be specified. While it is indispensable to declare an algorithm, explicitly setting mode and padding may be omitted. In that case, the underlying provider will implicitly assume ECB mode. Besides AES, this affects all symmetric block ciphers.

1. Determine all invocations of the method `Cipher->getInstance()` and for each occurrence, backtrack the first parameter, register `v1`, holding the transformation value.
2. Find all possible execution paths, where the endpoint resembles a transformation or specifies a symmetric block cipher, such as AES or DES.
3. For each selected path, verify whether it includes parts of a transformation value, e.g. */OFB/NoPadding*. If found, complete the algorithm name in the path endpoint with the determined mode and padding descriptor.
4. Raise an alert if the transformation value either explicitly declares ECB mode, or specifies only the algorithm name.

On iOS, by default CBC is preferred over ECB mode. ECB mode is only used when the developer explicitly specifies to use this mode of operation.

1. For each invocation of `CCCryptorCreate()`, `CCCrypt()`, `CCCryptorCreateWithMode()`, or backtrack the third parameter, `options`, specifying whether to use ECB or CBC mode.
2. Raise an alert if the any of the possible execution paths ends with a constant value of 2 which would indicate ECB mode (`kCCOptionECBMode`).

Rule 2: No non-random IVs for CBC encryption. Constant or predictable initialization vectors (IVs) lead to a deterministic and stateless encryption scheme, susceptible to chosen-plaintext attacks. If CBC mode is selected for encryption, developers should provide a non-random IV. If no IV is given at all, the cipher uses an all-zeros IV, which is at least as bad as a constant.

By analyzing byte arrays set as IVs, we learn if IVs are composed of static values or deduced from constants, e.g. strings. IVs can also be predictable when weak pseudo-random number generators (PRNG) are employed. Besides probing for specific indicators, we are naturally unable to make assumptions about the predictability of values. Similarly, non-static IVs cannot implicitly be assumed unpredictable.

On Android, to specify an IV for encryption, typically an `AlgorithmParameterSpec` is passed as argument to `Cipher->init()`. If it encapsulates an object of the type `IvParameterSpec`, an IV is manually defined rather than being generated randomly.

1. For all invocations of `Cipher->init()` with an `AlgorithmParameterSpec` object as 2nd argument, backtrack the value of this parameter.
2. Using the found list of constants, verify whether an object of the type `IvParameterSpec` is created by calling its constructor. Abort, if none is found.
3. From each available slicing path, extract the subpath that begins at the `iv` argument, passed to the constructor of the `IvParameterSpec` object.
4. Raise an alert if the `iv` parameter is derived from a statically defined byte array, a string (e.g. using `String->getBytes()`), or by calling the insecure `Random` API.

On iOS, we assert that an IV is evidently generated using a cryptographically secure PRNG.

1. For each invocation of `CCCryptorCreate()`, `CCCrypto()`, or `CCCryptorCreateWithMode()`, we backtrack the 6th parameter, specifying the IV.
2. Alert if a possible execution path does not call `CCRandomGenerateBytes()` in *CommonCrypto* or `SecRandomCopyBytes` in the *Security* library.

Rule 3: Do not use constant encryption keys. Keeping encryption keys secret is a vital requirement to prevent unrelated parties from accessing confidential data. Statically defined keys clearly violate this basic rule and render encryption useless.

On Android, `SecretKeySpec` can be used to specify a key. If derived from a constant, it must not be used for symmetric encryption. However, with public-key crypto, the encryption key is not a secret and can also be wrapped as a `SecretKeySpec` object. Thus, we distinguish between keys for symmetric and asymmetric encryption and also check the 2nd parameter of `SecretKeySpec`, which specifies the algorithm to use.

1. For all invocations of `SecretKeySpec->init()`, backtrack the first parameter, holding the key.
2. Verify for all contained constants if the `key` parameter is derived from a statically defined byte array, or a string (e.g. using `String->getBytes()`).
3. If at least one possible key has been found, also backtrack the 2nd parameter of the corresponding `SecretKeySpec` object and extract all data flows.
4. For each execution path, verify whether one of the following asymmetric encryption schemes is specified: *DHIES, ECIES, ElGamal, RSA*. If the algorithm is not a known public-key algorithm, we conclude that the statically defined key is used for symmetric encryption and raise an alert.

On iOS, we assure that any byte array specified as key does not exclusively consist of constant values.

1. For each invocation of `CCCryptorCreate()`, `CCCrypto()`, or `CCCryptorCreateWithMode()`, we backtrack the 4th parameter, holding the key.
2. For any path, we ensure that key elements originate from a non-constant or hard-coded source.

Rule 4: Do not use constant salts for PBE. A randomly chosen salt ensures that a password-based key is unique and slows down brute-force and dictionary attacks. Salts passed to key derivation functions (KDF), thus, must not exclusively depend on constant values.

On Android, parameters to use with KDFs can be declared using the `PBEKeySpec` API. Subsequently, a `SecretKeyFactory` instance transforms the password to an encryption key by invoking `generateSecret()`.

1. Find all invocations of `PBEKeySpec->init()` or `PBEParameterSpec->init()` and backtrack the parameter with the `salt` value.
2. Raise an alert if any execution path providing the `salt` parameter, is derived from a statically defined byte array, a string (e.g. using `String->getBytes()`), or by calling the insecure `Random` API.

On iOS, `CCKeyDerivationPBKDF()` must not be provided with an entirely constant salt value.

1. For all calls to `CCKeyDerivationPBKDF()`, backtrack the 4th parameter, specifying the `salt` value.
2. For any execution path, we ensure that byte arrays with the salt originate from a non-constant origin.

Rule 5: Do not use $<$ 1,000 iterations for PBE. A low iteration count significantly reduces the costs and computational complexity of table-based attacks on password-derived keys. We, thus, expect apps to use $\geq$ 1,000 rounds in KDFs, as proposed by RFC 8018[7].

On Android, the iteration count for PBE is declared using the `PBEKeySpec` or `PBEParameterSpec` API.

1. Find all invocations of `PBEKeySpec->init()` or `PBEParameterSpec->init()` and backtrack the parameter with the `iteration count` value.
2. Raise an alert if any execution path terminates at a constant integer whose value less than 1,000.

On iOS, the `rounds` parameter of the method `CCKeyDerivationPBKDF()` specifies the amount of iterations to use for key derivation.

1. For all calls to `CCKeyDerivationPBKDF()`, backtrack the 7th parameter with the `iteration` count.
2. For any execution path, we raise an alert if it does not end at a constant integer value $\geq$ 1,000.

Rule 6: Do not use static seeds for random-number generation. If a PRNG is seeded with a statically defined value, it will produce a deterministic output which is not suited for security-critical applications.

With Android 4.2 (API level 16), the default PRNG provider has been changed from Apache Harmony to the native `AndroidOpenSSL`. Before that, it was possible to override the internally designated seed with a custom value which, in case it was constant, caused the generation of deterministic output values.

1. For all invocations of `SecureRandom->init()`, backtrack the first parameter, holding the byte array with the seed. Likewise, for all calls to `SecureRandom->setSeed()` compute slices for the `seed` argument, which may consist of a byte array or eight bytes stored in a `long` integer value.
2. For all found execution paths, check if any supplies the `seed` parameter with constant input values.
3. An alert is raised if the parameter is derived from a statically defined byte array, a string (e.g. using `String->getBytes()`), or a 64-bits integer.

On iOS, the platform APIs do not support seeding the underlying PRNG. This misuse, thus, cannot occur.

[7] https://tools.ietf.org/html/rfc8018#section-4.2

6 EVALUATION

In the following study, we present the first comparative evaluation of misapplied crypto APIs in Android and iOS apps. Related research regarding crypto misuse never crossed the line between platforms. As it has already been demonstrated that security-critical implementation weaknesses are prevalent in both Android or iOS, we focus our analysis specifically on apps that are available for both platforms.

The goal of this evaluation is twofold. *First*, we ask if developers know how to use system-provided APIs correctly and give an impression of how often security misconceptions occur in popular apps. To prevent false positives or out-of-context findings, we manually study reported execution traces. *Second*, we ask how likely it is that apps which vendors distribute for both platforms, also violate the same security rules. Therefore, we compare the findings of mistakes in apps for Android with those in their iOS counterparts.

6.1 Method and Dataset

For the automated study of Android apps, we employ our framework, as presented in Section 4. For each inspected app, it outputs a report in XML format, lists found rule violations and associated execution paths. For iOS, we rely on an existing open-source framework[8], which pursues a very similar approach and emits HTML reports with execution traces for inspected security properties. We, however, align the security rules to the detection strategies presented in Section 5 to establish for both platforms comparable conditions to detect crypto API misuse.

6.1.1 Dataset

We manually compiled a set of apps where the use of cryptography seemed inevitable to provide specific functionality. Empirically, we found that this requirement affects at least apps for password management, secure messaging, document encryption, sensitive data exchange, and secure cloud storage.

While Google Play uniquely identifies apps by their package name, e.g., *com.example*, the iOS App Store features no comparable identifiers. We were, thus, looking for other descriptors suited to match apps that were provided for both platforms. We found that for the vast majority of multi-platform offered apps, the title and/or description text used in the stores are usually widely consistent over both platforms.

Consequently, we identified and downloaded 1,322 free apps from the iOS App Store, where we assumed the use of cryptography. Using the title and description text of each app, we were searching Google Play for an Android pendant and were successful for 976 apps. All of them had at least 1,000 installations or ratings as indicated by Google Play and the iOS App Store. With a version for Android and iOS each, in total, we have acquired $2 \times 976 = 1,952$ apps for analysis.

Table 1: Dataset of apps for evaluation.

	Count	[%]
Downloaded from iOS App Store	1,322	
Matching Android apps in Google Play	976	
iOS: No *CommonCrypto* calls	172	18%
Android / iOS: With crypto API calls	804	82%
iOS: App not decompilable	21	3%
Android: App archive corrupted	4	0.5%
Android / iOS: Out of memory	4	0.5%
Analyzable apps with crypto API usage	775	96%

After fetching iOS apps via iTunes, we used *Clutch*[9] on a jail-broken iPhone to decrypt them. By inspecting their library bindings, it became evident that only 82% or 804 iOS apps included calls to *CommonCrypto*. As the remaining set of 172 apps without calls to system-offered crypto APIs also provided functionality where the use of cryptography seemed reasonable, security might be missing or provided by third-party libraries, which our rules are not designed to cover.

6.1.2 False Positives and False Negatives

As the two frameworks do not only warn about the existence of problematic statements but can also pinpoint their origin, we leverage the data flow seen in execution paths to assess the soundness of issues found. By *manually* validating all obtained analysis reports, we prevent findings of false positives and ensure that all rule violations indeed occur in code that has real practical impact on the security of apps.

False positives or out-of-context results can occur, e.g., if encryption is only used for obfuscation and not for actually enciphering secret messages. Likewise, apps might include code where crypto routines are only initialized with insecure default attributes but never used. By manually examining the execution traces before acknowledging a rule violation, we minimize false positives that could have occurred, e.g., if a backtracked value was a parameter to a function that had never been invoked.

As our security rules evaluate method signatures of system-provided APIs, we can be confident that these calls are always found, even if apps obfuscate their program code. On Android and iOS, we, thereby, effectively avoid false negatives.

[8] https://github.com/IAIK/ios-analysis

[9] https://github.com/KJCracks/Clutch

Table 2: Violations of security rules in 775 apps for Android and iOS.

Rule (R)	Overall rule violations in apps Android		iOS		Rule violations in same apps on both platforms	
R1: Do not use ECB mode for encryption	598	(77%)	192	(25%)	172	(22%)
R2: No non-random IVs for CBC encryption	271	(35%)	494	(64%)	200	(26%)
R3: Do not use constant encryption keys	320	(41%)	455	(59%)	243	(31%)
R4: Do not use constant salts for PBE	112	(14%)	84	(11%)	49	(6%)
R5: Do not use $<$ 1,000 iterations for PBE	119	(15%)	145	(19%)	104	(13%)
R6: Do not use static seeds for PRNGs	25	(3%)	-		-	
Applications with $\geq$ 1 rule violations	538	(69%)	604	(78%)	404	(52%)
No rule violation	237	(31%)	171	(22%)	371	(48%)

6.2 Results

In total, we studied 976 apps where vendors provided a version for Android and iOS via the official app stores. As summarized in Table 1, we did not find references to *CommonCrypto* in 18% or 172 apps for iOS. Irrespective of whether they actually contain cryptography-related code, we excluded these apps from further analysis, since we knew a priori that our security rules would not detect any violations in them. Interestingly, for all remaining 804 iOS apps that use the *CommonCrypto* API, also all corresponding Android pendants included calls to methods in `java.security.*` or `java.crypto.*`. This strongly indicates that crypto is actually needed for the correct functionality of these apps, rather than being just included incidentally, e.g., via used third-party libraries.

The inspection of 24 iOS and 5 Android apps failed due to errors in processing the app archives. Of them, 21 apps for iOS could not be decompiled from ARMv8 to LLVM IR code due to missing mappings of instruction codes. Also, our Android framework was unable to process four apps, where the archive was damaged, despite repeated tries to re-download a functional version from Google Play. For another four apps, out of memory errors occurred during pointer analysis on iOS or register tracking on Android.

For 775 apps supplied to the iOS framework and our solution for Android, the analysis workflow terminated successfully. For each inspected application, we obtained a generated report that included the result of the performed security checks and for all rule violations, listings with problematic execution paths.

6.2.1 Violations of Security Rules

We found that 78% or 604 apps for iOS and 69% or 538 apps for Android commit at least one security-critical mistake. Among them, we identified 52% or 404 apps, where the Android and iOS versions of the same app commit at least one mistake on both platforms.

Table 2 lists our observations of violated security rules. We discuss the findings in more detail below.

Rule 1: Do not use ECB mode for encryption. On Android, we observed that 77% or 587 apps use instances of symmetric ciphers where the underlying mode of operation is ECB. Manually studying found execution path clarifies that block ciphers are mostly declared without explicitly specifying the mode and padding to use. This causes the underlying provider to apply ECB mode implicitly. Although the use of AES is predominant, at times we also noticed `Cipher` objects, specifying the nowadays weak DES algorithm.

CBC being the default mode on iOS, 25% or 192 apps explicitly declared to use ECB. In 22% or 172 apps, this mode was specified in the iOS version and also deployed in Android pendant of the same app.

Rule 2: No non-random IVs for CBC encryption. 35% or 271 apps on Android specified a static IV originating from hard-coded byte arrays or constant values. Manually verifying the results, we found that the majority derived a cryptographically secure IV using the API `SecureRandom`. Some Android apps relied on the `Random` API instead, which leads to predictable IVs.

On iOS, this was the most prominent problem: 64% or 494 apps used a constant IV. Mostly, no IV was declared at all, thus, implicitly causing an all zeros IV.

Rule 3: Do not use constant encryption keys. Hard-coded encryption keys were identified as an issue in 41% or 320 Android apps and 59% or 455 iOS apps. Regarding the prevalence in apps for both platforms, 31% or 243 apps used constant data as key material.

Table 3 highlights the provenience of key material. The according execution traces show that most apps which employ constant keys do so by deriving them from string values or by declaring byte arrays with constants. In practice, apps typically create distinct instances of `Cipher` (Android) and `CCCryptor` (iOS) objects for encryption and decryption. However, as they usually refer to the same constant key material,

Table 3: Origin of constant secrets used as key material.

# Violations on	Android	iOS
Constant string used as encryption key	238	305
Constant byte array as key	96	164
Hash value of constant string	9	36
iOS: Secret retrieved from *NSUserDefaults*	-	28
Applications violating rule 3	320	455

the total number of hard-coded secrets exceeds the number of apps with this issue.

Besides being entirely variable or static, from studying the execution paths, we learned that encryption keys can also be mixed, consisting of a statically defined key concatenated with a non-constant value. Interpreting these keys as constants would be inaccurate as we are unable to make assumptions about the entropy provided by the non-constant part.

Rule 4: Do not use constant salts for PBE. We identified that 14% or 112 Android apps and 11% or 84 inspected iOS apps passed constant salt values as input to key derivation functions. This effectively undermines the protection of password-based encryption against table-based attacks. On both platforms, most apps violating this rule declared a static byte array initialized with zero values. In 6% or 49 apps, this issue occurred in both versions of the same app.

Rule 5: Do not use $<$ 1,000 iterations for PBE. On Android, we found that 15% or 119 apps employ less than the minimally advised amount of 1000 iterations for Password-Based Encryption (PBE). Likewise, 19% or 145 apps for iOS and 13% or 104 apps on both platforms declared a too small value. Regarding the distribution of the iterations, most rule-violating apps specified a count of either 20, 50, 64, or 100. In execution traces of iOS apps without this issue, we could observe a significant prevalence of apps using `CCCalibratePBKDF()` API to dynamically derive an iteration count, rather than hard-coding a value.

Rule 6: Do not use static seeds for random-number generation. As the platform APIs on iOS do not offer a seedable PRNG, this rule can only be violated by Android apps. Changes in the underlying PRNG implementation have globally fixed this vulnerability for systems running Android 4.2 (API level 16) or newer. Due to this and only 3% or 25 Android apps declaring a constant seed for use with `SecureRandom`, the practical impact of this rule violation is limited.

6.3 Discussion

The study of 775 apps that were distributed for Android and iOS revealed that in 52% or 404 apps security-critical mistakes were present in both versions. The individual rule violations per platform have clearly shown a trend that some misuses happen more often on a specific platform. At the same time, it became obvious that the likelihood for some other mistakes is independent of a particular API design.

The most significant difference in rule violations across platforms was observed in the first rule targeting the use of ECB mode for encryption. The high number of 77% or 598 Android apps with this issue can be attributed to the fact that, unless a mode is specified, an implicit fallback to ECB occurs. In contrast, on iOS only 25% or 192 apps explicitly specified ECB instead of the default mode CBC. This circumstance strongly indicates that the prevalence of this problem on Android is caused by an insecure default setting.

Another notable difference between the two platforms has shown regarding the use of non-random IVs for CBC encryption. On Android, if no IV is specified, `Cipher` will automatically generate a strong IV that can be retrieved via `Cipher->getIV()`. As opposed to that, not declaring an IV on iOS will cause that a NULL (all zeros) IV is used for encryption. This again highlights the importance of secure default settings.

The most frequently violated issue in apps for both platforms affects constant encryption keys. Compared with other security violations, the `key` parameter always has to be provided manually. Sadly, both developers of Android and iOS apps, seem to be not aware of the radical consequences of hard-coding secrets.

Summarizing, we have seen that the origins of mistakes fall into two categories: firstly, those which are based on insecure default values in the corresponding API, e.g., implicit ECB mode on Android, or a NULL IV on iOS, and secondly, security problems that occur due to developers not carefully handling security-critical parameters, like encryption keys.

As a remedy, we propose a two-fold strategy:

1. **API Changes:** Unsafe default values in APIs, such as ECB mode on Android, should be replaced by secure alternatives. In occasions, where omitting arguments impairs security, e.g., a NULL value as IV on iOS, a cryptographically secure random value should be generated instead. Although such profound changes might break compatibility with existing code, they would require developers to improve their implementations and, thereby, minimize the prevalence of problematic code.
2. **Raising Awareness:** Modern IDEs for application development, such as Android Studio or Apple Xcode, feature sophisticated code inspection. Following our recipes presented to evaluate security-critical attributes (see Section 5), code analysis in IDEs should be extended to warn about harmful practices, such as hard-coded encryption keys.

7 CONCLUSION

In this paper, we studied misapplied crypto APIs in Android and iOS apps. By introducing an easily adaptable framework to track data flows throughout Android apps, we succeed in reliably pinpointing the origin of wrongly chosen security-relevant attributes. Therefore, we elaborated and implemented detection strategies to assess the cryptographic soundness of parameters that are used with encryption and key derivation APIs.

We presented the first comparative evaluation of crypto API misuse across platforms. Evaluated using a carefully selected set of 775 apps that were distributed for Android and iOS, we found security mistakes in 69% or 538 Android apps and 78% or 604 iOS apps. Unsafe default values in platform-provided APIs and missing developer awareness strongly contribute to this widespread of problems. Our results also highlight the need to compare concrete execution traces of API invocations to better understand the context of methods that are invoked with cryptographically weak values. Finally, our study underlines that misapplied crypto is still a severe issue in Android and iOS apps.

REFERENCES

Acar, Y., Backes, M., Fahl, S., Garfinkel, S. L., Kim, D., Mazurek, M. L., and Stransky, C. (2017). Comparing the Usability of Cryptographic APIs. In *IEEE Symposium on Security and Privacy – S&P*, pages 154–171. IEEE Computer Society.

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y. L., Octeau, D., and McDaniel, P. D. (2014). FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Programming Language Design and Implementation – PLDI*, pages 259–269. ACM.

Chatzikonstantinou, A., Ntantogian, C., Karopoulos, G., and Xenakis, C. (2015). Evaluation of Cryptography Usage in Android Applications. In *Bio-inspired Information and Communications Technologies – BICT*, pages 83–90. ICST/ACM.

Deng, Z., Saltaformaggio, B., Zhang, X., and Xu, D. (2015). iRiS: Vetting Private API Abuse in iOS Applications. In *Conference on Computer and Communications Security – CCS*, pages 44–56. ACM.

Egele, M., Brumley, D., Fratantonio, Y., and Kruegel, C. (2013). An Empirical Study of Cryptographic Misuse in Android Applications. In *Conference on Computer and Communications Security – CCS*, pages 73–84. ACM.

Enck, W., Gilbert, P., Chun, B., Cox, L. P., Jung, J., McDaniel, P. D., and Sheth, A. (2010). TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Symposium on Operating Systems Design and Implementation – OSDI*, pages 393–407. USENIX Association.

Feichtner, J., Missmann, D., and Spreitzer, R. (2018). Automated Binary Analysis on iOS: A Case Study on Cryptographic Misuse in iOS Applications. In *Security and Privacy in Wireless and Mobile Networks – WISEC*, pages 236–247. ACM.

Lazar, D., Chen, H., Wang, X., and Zeldovich, N. (2014). Why does cryptographic software fail?: a case study and open problems. In *Asia-Pacific Workshop on Systems, APSys'14, Beijing, China, June 25-26, 2014*, pages 7:1–7:7. ACM.

Li, L., Bartel, A., Bissyandé, T. F., Klein, J., Traon, Y. L., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., and McDaniel, P. D. (2015). IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *International Conference on Software Engineering – ICSE*, pages 280–291. IEEE Computer Society.

Li, Y., Zhang, Y., Li, J., and Gu, D. (2014). iCryptoTracer: Dynamic Analysis on Misuse of Cryptography Functions in iOS Applications. In *Network and System Security – NSS*, volume 8792 of *LNCS*, pages 349–362. Springer.

Ma, S., Lo, D., Li, T., and Deng, R. H. (2016). CDRep: Automatic Repair of Cryptographic Misuses in Android Applications. In *Asia Conference on Computer and Communications Security – AsiaCCS*, pages 711–722. ACM.

Muslukhov, I., Boshmaf, Y., and Beznosov, K. (2018). Source Attribution of Cryptographic API Misuse in Android Applications. In *Asia Conference on Computer and Communications Security – AsiaCCS*, pages 133–146. ACM.

Nadi, S., Krüger, S., Mezini, M., and Bodden, E. (2016). Jumping through hoops: why do Java developers struggle with cryptography APIs? In *International Conference on Software Engineering – ICSE*, pages 935–946. ACM.

Octeau, D., Jha, S., and McDaniel, P. D. (2012). Retargeting Android applications to Java bytecode. In *Foundations of Software Engineering – FSE*, page 6. ACM.

Paletov, R., Tsankov, P., Raychev, V., and Vechev, M. T. (2018). Inferring crypto API rules from code changes. In *Programming Language Design and Implementation – PLDI*, pages 450–464. ACM.

Shao, S., Dong, G., Guo, T., Yang, T., and Shi, C. (2014). Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications. In *Conference on Dependable, Autonomic and Secure Computing – DASC*, pages 75–80. IEEE Computer Society.

Sun, M., Wei, T., and Lui, J. C. S. (2016). TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime. In *Conference on Computer and Communications Security – CCS*, pages 331–342. ACM.

Tip, F. (1995). A Survey of Program Slicing Techniques. *J. Prog. Lang.*, 3.

Weiser, M. (1981). Program Slicing. In *International Conference on Software Engineering – ICSE*, pages 439–449. IEEE Computer Society.