iCryptoTracer: Dynamic Analysis on Misuse of Cryptography Functions in iOS Applications *

Yong Li, Yuanyuan Zhang, Juanru Li, and Dawu Gu

Dept. of Computer Science and Engineering
Shanghai Jiao Tong University
Shanghai, China
E-mail: yyjess at sjtu.edu.cn

Abstract. Cryptography is the common means to achieve strong data protection in mobile applications. However, cryptographic misuse is becoming one of the most common issues in development. Attackers usually make use of those flaws in implementation such as non-random key/IV to forge exploits and recover the valuable secrets. For the application developers who may lack knowledge of cryptography, it is urgent to provide an efficient and effective approach to assess whether the application can fulfill the security goal by the use of cryptographic functions. In this work, we design a cryptography diagnosis system *iCryptoTracer*. Combined with static and dynamic analyses, it traces the iOS application's usage of cryptographic APIs, extracts the trace log and judges whether the application complies with the generic cryptographic rules along with real-world implementation concerns. We test *iCryptoTracer* using real devices with various version of iOS. We diagnose 98 applications from Apple App Store and find that 64 of which contain various degrees of security flaws caused by cryptographic misuse. To provide the proof-of-concept, we launch ethical attacks on two applications respectively. The encrypted secret information can be easily revealed and the encryption keys can also be restored.

1 Introduction

Mobile devices such as smartphones and tablet computers are becoming the vessel of personal information such as contact list, physical location, social information and even banking service, online payment. As the popularity of such devices grows, the malicious software have the increasing impact on personal privacy. Current mobile OSes(mainly Android and iOS) use layered security strategies to endow the dominance to the end-users to control the access of the sensitive data. Aiming at providing high security assurance, iOS is designed with various security features. At system level, full-disk encryption, ASLR [1], sandboxing

* This work is supported by The National Key Technology R&D Program(2012BAH46B02), National Science and Technology Major Projects (Grant No.2012ZX03002011), and Technology Innovation Project of Shanghai Science and Technology Commission (No.13511504000).

profile and privilege assignment are adopted to fulfil access control policy [2]. At applications level, the Apple App Store scrutinizing on the applications also reduces the risk of malicious behaviors in the apps as a beneficial supplement. Besides for those built-in security features, third-party iOS developers resort to modern cryptographic algorithms to provide stronger protection on sensitive data.

It is possible that the emphasis on cryptographic techniques for protecting information mitigates the attention to the issue of cryptographic usage. The security of the primitives are provided by intellectual properties or industrial standards. There is a tendency to focus on problems that are mathematically interesting to the exclusion of implementing problems which must be solved in order to actually increase operational security. We've seen lots of security applications contradicting to some basic cryptography applying rules caused by developer's ignorance of general cryptographic usage guidelines, or sometimes the ambiguous documentation misleading to defective implementations. Both facts could result in software vulnerability or privacy leaks.

The well-known *Citibank* iOS application [3] and *Starbucks* application [4], for instance, storing the customer's privacy information such as payment passcode, bank account number, etc. The Verge has reported that Starbucks' iPhone application stores user passwords in plaintext. By connecting iOS device through iTunes, an attacker can easily retrieve the password and payment records. Therefore, it's crucial to evaluate the correctness of cryptographic usage inside the emerging third-party iOS applications.

As a contrast to the open-source Android system, iOS is a proprietary operating system and is relatively close. Developing a third-party security analysis extension for iOS system requires essential work and is difficult for lacking details of the operating system. Recent studies on iOS application mainly apply static analysis to detect security vulnerability such as privacy leak. Egele et al. proposed PiOS [5] based on static analysis using a control-flow graph to identify from where the sensitive data leaks. However, static analysis tends to be less accurate due to the dynamic messaging mechanism of iOS applications, which are primarily developed with Objective-C. Most iOS applications are heavily based on event-driven schemes. Simply analyzing an application with static analysis is not feasible because the dynamic events can not be predicted, the inputs can not be constructed either, parameters may be generated while executing, and the return value is unforeseeable. Such dynamic characteristic determines that many information can only be monitored accurately at runtime and in this situation dynamic analysis would be a better choice.

Dynamic analysis of iOS applications is facing lots of challenges. One challenge is that encryption is input-related, so that some data should be provided. iOS Applications are GUI-rich, and most of input areas are of *UITextField* component, and sometimes files should be provided as input, so manual work is inevitable during test. To study iOS kernel and Objective-C runtime to dynamically observe the application running in iOS, we have to resort to instrumentation

and API hooking techniques. To the best of our knowledge, no previous dynamic analysis on cryptographic usage has been proposed on mobile system so far.

An approach to diagnose the implementation code to assure the proper validity of cryptographic usage is in demand. We present *iCryptoTracer* to fulfill such purpose. As a cryptographic usage vetting system (we use *crypto-vetting* to indicate the whole diagnosis process in the following) based on static and dynamic analysis techniques, *iCryptoTracer* situates at Core OS layer gathering crucial information (we call it *crypto-trace* in this work) at iOS runtime that cannot be observed by static analysis, such as the sequence of API calls, arguments and return values. Afterwards, *iCryptoTracer* conducts analysis on the *crypto-trace* files according to some generic rules to diagnose the vulnerability of usage. We diagnose 98 typical security-oriented iOS applications and find out 64 of them contain various degrees of security flaw in cryptographic misuse aspect. We validate the effectiveness of *iCryptoTracer* diagnosis by successfully launching ethical attacks on two iOS applications as proof-of-concept, including a banking application and a password managing application.

The rest of the paper is organized as follows: Section 2 presents the preliminaries on the techniques we adopt to implement *iCryptoTracer*. Section 3 describes the design philosophy of our work. The implementation of *iCryptoTracer* is presented in Section 4. As well as the evaluation of our work in section 5, we also present typical ethical attacks against two applications that have been judged as weak implementation of cryptography by *iCryptoTracer*. Section 6 lists related work, Section 7 is about the limitation, and Section 8 concludes the paper.

2 Messaging in iOS

In this section, the basics of Objective-C runtime, *message swizzling* and *Common Crypto* libSystem will be briefly introduced.

2.1 Messaging

Objective-C is a strict superset of the C programming language that adds object-oriented features to the basic language. An innegligible characteristic of Objective-C is *messaging*, i.e. `objc_msgSend`. The invoking of object method in Objective-C is not direct but through virtual method tables(*vtables*), i.e., a *message* is sent to a receiver and these messages are handled by the dynamic dispatch function `objc_msgSend`. A message consists of the receiver of the message(`object` in Figure 1), method name(`method`), parameter names(`para name`$_i$) and their argument value(`arg`$_j$). The method name and parameter names are called *selector*. There is a one-one mapping between selectors and the code region of the method(see Figure 2), i.e. $selector_A$ mapping on IMP_a, $selector_B$ on IMP_b, and so on. Then, `objc_msgSend` will forward the *message* to the receiver to execute the code region.

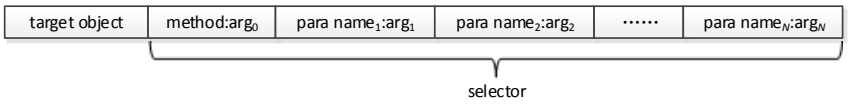

Fig. 1. A syntax of a *message*

2.2 *Method Swizzling*

As we discussed above, there exsits mapping between selector and its implementation. Objective-C provides a mechanism to allow developers to exchange or modify the mapping, which allows a selector to be redirected at runtime. This is called *method swizzling*. Consider $selector_A$ mapping IMP_a and $selector_B$ mapping IMP_b, we can exchange their mapping and the result is $selector_A$ mapping IMP_b and $selector_B$ mapping IMP_a(see Figure 3). Also, for a selector, we can modify its function by redirecting it to another implementation developed by ourselves(Figure 4). By utilizing modification, **API hook** is enabled, which is essential in *iCryptoTracer*'s tracing module(see Figure 5).

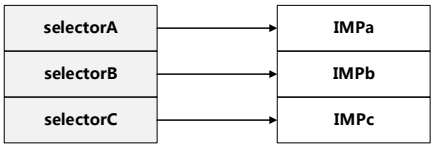

Fig. 2. Method Swizzling - Original mapping

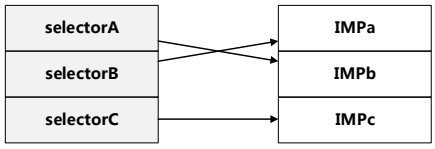

Fig. 3. Method Swizzling - Exchange

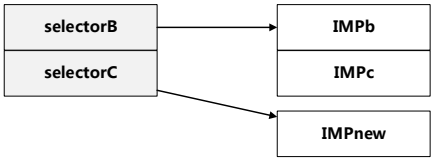

Fig. 4. Method Swizzling - Modification

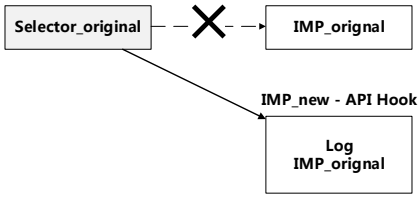

Fig. 5. Method Swizzling - Modification

2.3 *Common Crypto* Library in iOS Security Framework

Apple provides open source *Common Crypto* library for cryptographic usage. Being a part of *Security Framework* in iOS, the library consists of various kinds

of cryptography APIs, such as symmetric encryption, HMAC and digests, etc. By calling API from the lib, developers can encrypt target data with some algorithm. However, documentation going with the lib is not very specific, and explanation of cryptography is not provided. As a result, simply calling API from *Common Crypto library* does not necessarily mean data is well protected by encryption.

3 System Design

In this section, we elaborate the design of *iCryptoTracer*.

3.1 Overview of *iCryptoTracer*

iCryptoTracer diagnoses the cryptographic usage adopting static-dynamic combined analysis on a target security application. The structure of *iCryptoTracer*, as depicted in Figure 6, is designed to meet the requirement that the diagnosis of *iCryptoTracer* should be intelligible. *iCryptoTracer* monitors overall cryptographic functions in the application, meanwhile, it is vigilant against the threats to sensitive data, which is the purpose we design the system. The intuition behind is that a complete sensitive information protection scheme must be based on a series of proper cryptographic functions. In other word, as long as the sensitive information is not confined inside a closure area, i.e., encrypted packet or file using cryptographic functions, it is of insecure status and may be illegally acquired. Hence, it is able to reveal potential insecure data protection or sensitive data leakage through checking the improper using of cryptographic functions in an application.

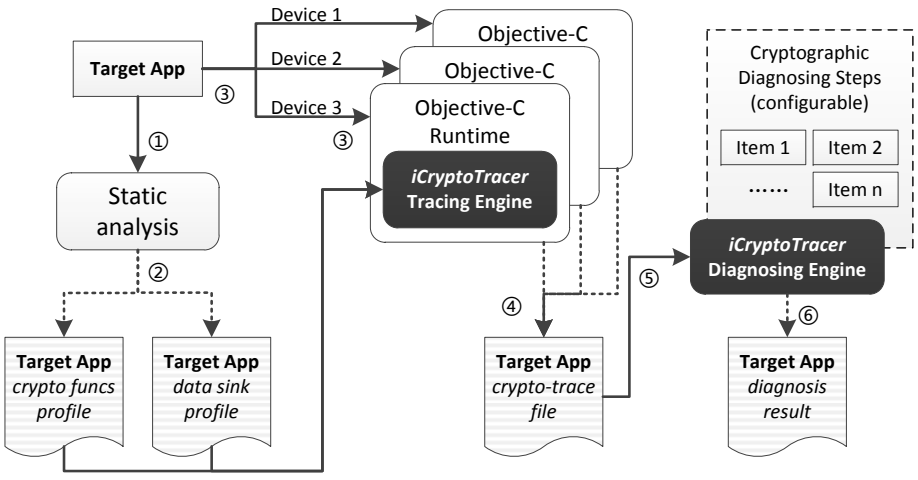

Fig. 6. Overview of *iCryptoTracer* architecture

To accurately evaluate the improper using of cryptographic functions, we mainly take account of the sensitive data when it is at stake such as sending through network or writing to a file and make the following assumptions: First, the security of the cryptographic primitives, which are the basic blocks in cryptography(e.g., AES block cipher, SHA-512 hash functions, etc.), are guaranteed by industrial standards and have been verified widely. Application should only use standard cryptographic primitives to protect the data. Second, the sensitive data of interests usually involves in cryptographic operations under specified cryptographic usage strategy(i.e., cryptographic protocols). Cryptographic usage strategy is the operating parameters or operating orders of cryptographic primitives on sensitive data. Even the primitives are solid, a proper usage strategy is also crucial for building a solid protection scheme.

Based on these assumptions, *iCryptoTracer* tries to locate two typical flaws: 1) the sensitive data, when sending through network or writing to a file, is not protected with any cryptographic functions. Sensitive information is the critical protege and should be under the protection of cryptographic operations. If there exists no cryptographic operation related to sensitive information, the protection must be insecure. 2) the sensitive data, when sending through network or writing to a file, is protected with cryptographic functions but with improper usage strategy.

Among various factors to concern in security context, *iCryptoTracer* takes the following three types of contents into account: 1) sensitive information to be protected, 2) cryptographic primitives, and 3) cryptographic usage strategy. However, for the dynamic analysis of iOS application, it is very difficult to deploy advanced information flow tracking techniques such as taint analysis. The approach adopted by *iCryptoTracer* is therefore a synthetic one. That is to say, our approach synthesizes information collected from separated spots during the runtime and diagnoses the problems. It first scans the target application to spot and put surveillance on all the possible *data sinks*. *Data sinks* are spots that are tightly related to several specific system APIs, such as network I/O and file I/O APIs. The information of data sink is gathered by static analysis on applications. Then, *iCryptoTracer* scans and records the cryptographic function APIs locations in the application, especially those that wraps those I/O as the diagnosis objectives for further use. After these two steps, *iCryptoTracer* generates the *cryptographic functions profile* and the *data sink profile*(step 1-2 in Figure 6). With the guide from *cryptographic functions profile* and *data sink profile*, *iCryptoTracer* monitors those API calls at runtime adopting *message swizzling* technique. Those related function calls are redirected to the *tracing engine* of *iCryptoTracer*, and all useful information including sequence of API calls, arguments, return values, etc. are logged as the *crypto-trace* file (step 3-4 in Figure 6). Finally, *iCryptoTracer* synthesizes information recorded in the *crypto-trace* file with its *diagnosis engine* and judges whether a cryptographic function usage flaw exists in the application.

3.2 Static Analysis

The iOS applications(in the form of *ipa* file) are downloaded from Apple Store, which are encrypted with device-dependent key. We first extract the binary of the applications by means of reverse engineering on the *ipa* file. To identify the crucial API invoking points, we filter the API calls concerning data operation, transmission and encryption and resolve the location into two files *crypto funcs profile* and *data sink profile*.

Static analysis helps us to locate the position of the APIs, which aids the system to narrow down the APIs to observe during the dynamic analysis phase. As we described in the previous section, static analysis is not capable of collecting sufficient runtime information, so more precise information on the target APIs in the *profiles* will be collected during runtime.

3.3 Log Tracing

The *messaging* realizes the redirection of the system calls to cryptographic APIs. As indicated in two *profiles* we acquired from static analysis, those specific calls are wrapped and extended with a logging function. It takes record of the relevant API information, such as method names, arguments, return values, etc. and stores them into a log file. A typical log entry is as below:

```
1  func:CCCrypt
2  algorithm:kCCAlgorithmAES128
3  dataIn:tN/m8LhVi5xRsjKWnvFvXPz6y5qPN0HZknNQHqiLs0Q=
4  dataOut:2088002061679122
5  dataOutAvailable:48
6  iv:!zmcbbmmyyana...
7  key:cxlbylvyfrever!!
8  op:kCCDecrypt
9  options:kCCOptionPKCS7Padding
10 returnValue:0
```

In this entry, *CCCrypt* is the name of the method the application called. The *algorithm* field reveals the information of the *kCCAlgorithmAES128* cryptographic algorithm invoked inside method *CCCrypt*. In this case, it is an *AES* encryption with 128-bit block. The fields of *iv* and *key* indicate the IV and encryption key in use, and the *options* field reveals the data padding method.

3.4 Trace Log Diagnosis

After the log tracing phase, the collected log is delivered to the *diagnosis engine*(Figure 6). In order to ensure a comprehensive analysis, we introduce cross-reference diagnosis to analyze the trace logs from different security context. We collect cryptographic logs for the same iOS application executed on various devices with different versions of operating system. The diversity we introduced

tends to bring multiple trace logs. Then the diagnosis engine could detect possible invariance from the synthesis of several logs, which usually indicates the non-randomness *iCryptoTracer* is designed to have a configurable *diagnosis engine*, the successive diagnosis exam items should be defined carefully. There are quite many generic cryptographic rules in security application developing [6] [7], such as the choice of encryption mode, randomness of IV and encryption keys, etc. Based on those rules and a full study of *Common Crypto* library, diagnosis will be conducted on the log following the exams as below:

Item 1: Using constant encryption keys The randomness of the encryption keys is mandatory. Intuitively, a constant key hard-coded can be easily observed, thus the resulting secrecy of encryption is not guaranted.

Item 2: Using a non-random IV for CBC encryption As elaborated in [7], the CBC-mode construction should always use a random IV. However, a common error is to use fixed (usually all zero) IV in real-world implementations.

Item 3: Using stateless encryption There are mainly two kinds of encryption interfaces in *Common Crypto* library: the stateless encryption APIs and the stateful ones. The stateless encryption is deterministic, that is, if the same message is encrypted twice with the same key, the identical ciphertext is returned. A deterministic encryption is not secure, because a distinguisher which distinguishes message streams with repeated messages can be created by detecting repeats in the encrypted message blocks [6].

4 Implementation

We implement the design of *iCryptoTracer* (see Figure 6) in a prototype that supports iOS version from 6.1.3 to 7.0.6. *iCryptoTracer* requires a jailbroken device as its runtime environment to inject a hooking dynamic library into the application during the execution, which redirects and logs target APIs. In the following, we present selected implementation details of *iCryptoTracer*.

4.1 Static Analysis

Common ipa files downloaded from App Store are encrypted so that they should be decrypted before the analysis stage. To decrypt the analyzed app, *iCryptoTracer* resorts to **Clutch** [8] to dump the code segment of an ipa file in memory and fix the encrypted ipa file as a workable executable on a jailbroken device. Then iCryptoTracer implements a script of IDA to statically analyze the binary code to find out the imported APIs that are concerned.

4.2 Tracing Engine

As the major part of dynamic analysis at Objective-C runtime, *tracing engine*(Figure 6) locates in between Core OS and Core Service layers. It is mainly based on *message swizzling* introduced in Section 2.

The main idea of *tracing engine* is to redirect the API calls and dynamically log information such as parameters based on the *profiles* at runtime. We realize this by utilizing *method swizzling* (Section 2.2), that is, replacing $Selector_{orignal}$'s original mapping implementation, $IMP_{original}$ to a new one, IMP_{new}, and the latter including log functions (added) and $IMP_{original}$, which enables hooking and logging, and makes sure the original method be executed (see Figure 5). Through this way, both hooking and logging are achieved at the same time.

Tracing engine has been implemented to monitor three types of APIs: 1) cryptographic functions, 2) file I/O APIs, and 3) network I/O APIs. The cryptographic functions are provided by *Common Crypto* library of the iOS. Any invoking of cryptographic functions in this library are completely recorded as part of the trace. The tracing of the data flow is obtained by hooking the file I/O and network communication related APIs. When the target application is running, and if any of those APIs are invoked, the related information such as parameters will be recorded as part of the trace.

4.3 Diagnosis Engine

Diagnosing engine is responsible for three tasks: 1) comprehending the *crypto-trace* logs, 2) detecting invariance in the synthesis of the logs, and 3) applying the exam rules and output diagnosis result.

Notice that the log obtained from *tracing engine* is actually incomprehensible byte stream from memory, so parsing on the trace file is required. *Diagnosis engine* contains a parsing module to resolve data according to API specifications, and data is encoded with base64 and saved in a SQLite database, which can be decoded and analyzed later in the following diagnosis steps. Then, entries in multiple cryptographic logs are synthesized and *Diagnosis engine* will extract each field of the same cryptographic operation in multiple logs. Finally, given the synthesized information, *Diagnosis engine* exams the security weakness according to the three exam items we proposed in Section 3. The *Diagnosing engine* is perceptible to determine which exam item the function breaks, so the target application that passes through all the exams will get a higher evaluation value. We define three security degree for diagnosed iOS applications, *critical*, *weak*, and *healthy*, see Table 1.

Critical We define that if any of the cryptography uses in this application does not pass one or more item exams, and this encryption operation is happening at one or more *data sinks* to be *critical*.

Weak We define that cryptography use which does not pass all item exams, but the encryption operation is not at any *data sink* to be *weak*.

Healthy We define the cryptography which pass all item exams to be *healthy*.

After the evaluation of the cryptographic use of a target iOS application, *iCryptoTracer* outputs a file called *diagnosis result* at the end of the diagnosis procedure. The overall evaluation result (*Critical*, *Weak*, or *Healthy*) is given, and the locations of suspicious encryption functions and their relevant *data sink*

Table 1. Applying the diagnosis based on the items

	Pass all the item exams	**Happen at data sink**
Healthy	YES	-
Weak	NO	NO
Critical	NO	YES

are included as well. Given the diagnosis result, not only the application users, but also the program developers can benefit from the detail introspection for next stage improvement on their works.

5 Evaluation

In this section, we analyze the effectiveness of *iCryptoTracer* by applying on the selected security-oriented applications downloaded from Apple's App Store. The results are carefully analyzed to measure the security of the target application. Finally, we give two specific cases on misuse of crypto-functions. An ethical attack is launched against the applications to prove the effectiveness of *iCryptoTracer* in locating the weakness of a security-oriented application.

5.1 Selection of Test Apps

To demonstrate the effectiveness of *iCryptoTracer*, 98 typical security-oriented applications from the official Apple App Store are chosen as the target applications, see Table 2. These applications contain privacy related information, such as online payment password, bank account number, SMS, confidential files, etc. These applications are relative to online bank and online payment, or personal passwords management, such as *Alipay* and *1password*. The former is the most popular online payment application and it has been downloaded for more than 10 million times from China, and the latter is a popular application that stores and manages nearly all the personal passwords and files in a centralized way.

5.2 Testing on Selected Apps

iCryptoTracer has been implemented on several testbeds that supports from iOS 6.1.3 to iOS 7.0.6 on various models of iPhone and iPad. 98 selected security applications are diagnosed through *iCryptoTracer*. The diagnosis procedure only requires manually installation of those target applications on testbed devices. The running of the diagnosis is fully automatic and silent at backstage. Each of the application receives 10 different forged inputs, e.g., 10 forged bank account numbers for online banking application, or 10 different files for file protection applications, in order to extend the analysis results.

According to the outputs, we have in Table 2 the number of the applications with defective implementations we have found during the diagnosis. The results

show that 64 out of 98 target applications have various security flaws. Moreover, 8 of 64 unhealthy applications are diagnosed as *critical*. As to *critical* applications, it require little effort to recover the secret message sent via *data sinks*. We give two ethical attacks in the following section. The overall diagnosis results are listed in Table 2.

Table 2. Diagnosis results of 98 applications

Category	Total	Healthy	Weak	Critical
online bank	28	8	20	0
mobile payment	22	6	16	0
account protection	16	8	5	3
file protection	32	12	15	5

5.3 Case Studies

As we can see from the evaluation section, there exist a lot of applications breaking encryption rules, and some of them can be easily attacked. We select two typical applications for further study.

Ethical attack on a banking application We use *iCryptoTracer* to evaluate a banking application (for security issue we hide the actual name of the application) that is used for querying user's account activities and performing e-trading. This application is diagnosed as *critical*, which means there exists a misuse of cryptographic function related to *data sink*. In detail, this application does not pass the examines on Exam item 2 and 3. It employs non-random key and an empty IV, see the first record in Table 3.

The misused function is an AES encryption with an empty IV and a fixed key, i.e. the same key is repeatedly used in different cryptographic contexts. We run the same banking application on a non-jail-break iOS device, and provide username and password as input. In a configured WLAN with a sniffer, we are able to eavesdrop the encrypted communication data between the app and server. By applying the encryption algorithm, fixed key and empty IV obtained from *iCryptoTracer*, we can decrypt the encrypted communication data, including username and password, which should be well protected by the app. Furthermore, we successfully use the decrypted username and password to log into the on-line bank.

Ethical attack on a password management application We perform the diagnosis on a password management application, which claims to be able to protect user's privacy through encryption and the encryption algorithm applied is of open standard. In order to use the application, users have to enter a password, which proves to be irrelevant to encryption but only for authorization. The

Table 3. Case Study

Encryption	**Key**	IV	**Option**	**Key Repetition**
AES 128	njwftwr,xjtxrft.		No Padding	YES
3DES	1234567890def13579ace	init vec	PKCS7Padding	YES

application can automatically encrypt user's privacy data, such as passwords and files, and save on the device. For the sake of convenience, the encrypted data can be exported for backup through iTunes. It's true that the application utilizes *3DES* for encryption, but from its tracing log obtained from *iCryptoTracer*, we find out that the application's encryption is of serious flaw, i.e. a simple constant key and a fixed IV are used in different encryption contexts. First, we encrypted test files with the application on a non-jailbreak iOS device, and then exported encrypted data to a computer with the help of iTunes. By applying the constant key and fixed IV (see 3) logged by crypto-trace file in *iCryptoTracer*, all test files are instantly decrypted.

6 Related Work

With the development of mobile system, it gains more and more popularity and attention from researchers, especially on Android and iOS. MoCFI [9] and CFR [10] are both designed to defend iOS applications from control flow attacks. They are of system's security, but have nothing to do with encryption analysis. Egele et al. presented an approach - PiOS [5], which is able to automatically create control flow graphs (CFG) from decrypted iOS binaries and then perform reachability analysis on CFGs to identify possible leaks of user's privacy data from device to third parties. The test results of PiOS demonstrate that a majority of iOS applications leak the device ID. Han et. al [11] also presented a way by massively examining security-sensitive APIs to detect potential access to sensitive resources that may cause privacy breach or security risks. Static analysis is a kind of reference and more proof is needed from dynamic analysis.

Due to the dynamic characteristic of Objective-C, runtime attack aimed at iOS system is invented [12] [13]. The attack results show that by dynamically loading methods, static analysis can be easily bypassed. As a complement for static analysis, dynamic analysis for mobile system has already been presented, such as *TaintDroid* [14]. However, Android and iOS are totally different mobile systems and the technique on Android can never be implemented on iOS due to its close-source. Szydlowski et al. presented a dynamic way to analyze iOS applications [15], whose method is to analyze iOS applications in a static way at first, and then set breakpoints at objc_msgSend methods while running. The most obvious limitation is there will be a lot of breakpoints during running process, which may crash the running application. Also, by setting breakpoints

at every objc_msgSend methods, the specific method is unknown, and there will be too much data obtained from registers and memory.

Bhargavan et al. [16] illustrate tools that can be used to verify the security of cryptographic protocol implementations, and Mitchell et al. present Murφ [17] which is able to detect vulnerabilities in cryptography and security protocols. Both of their work are achieved on PC platform. For cryptographic misuse analysis on mobile system, Egele et al. developed a light-weight static analysis approach to check for common flaws of encryption use for Android apps, CRYPTOLINT [7]. Its main idea is to use static program slicing to identify flows between cryptographic keys, initialization vectors and similar cryptographic material and the cryptographic operations. Anyway, it is inevitable that static analysis misses the data generated during runtime and sometimes error-prone, especially when the binary program can load function or methods in a dynamic way. Our work is similar to CRYPTOLINT, but we apply the analysis in a dynamic way on iOS and can be a better complement to static analysis. Though *iCryptoTracer*'s efficiency may not as good as CRYPTOLINT, it's more specific and accurate.

7 Limitation

iCryptoTracer diagnoses limited types of APIs and its information at the specific surveillance location such as *data sinks*, so it can not be categorized as a full-grained information flow analysis system. Moreover, for some developers, they may resort to a third-party cryptography library instead of Apple's *Common Crypto* libSystem, *iCryptoTracer* is not yet capable of analysing third-party security libraries. To achieve this, *iCryptoTracer* has to be equipped with cryptographic primitive identification and other advanced dynamic analysis techniques, which will be presented in our other works soon.

While *iCryptoTracer* can automatically analyze and verify the cryptography use of iOS applications, it still can not verify the security of protocol. During tracing process, in order to provide input data for GUI-rich applications, human interaction or manual work for input is required, which lead to lower efficiency than static analysis.

8 Conclusion

In this paper, we proposed a diagnosis system *iCryptoTracer* for security iOS applications to assess whether the fashion of cryptographic usage leads to a proper notion of security. The diagnosis process is a staged procedure combining static and dynamic analysis techniques. For static analysis, we can efficiently locate the methods to observe later on during iOS runtime. Dynamic analysis helps to collect method call information that cannot deduce at static analysis stage.

Designed as an automatic diagnosis system, *iCryptoTracer* works silently at backstage monitoring the running of the target applications. In the end, *iCryptoTracer* outputs the diagnosis result by given rules and steps. A target application

can be considered as *healthy* only when it passes all the item exams. Any *weak* or *critical* application will be diagnosed with a detail result including the defectively implemented APIs and their corresponding *data sinks* locations.

We have diagnosed 98 security iOS applications and found 65.3% of which are suffering from various degree of vulnerability from defective implementation of misuse. Further study on the misuse leads to two ethical attacks on two chosen applications, a banking application and a password management application. We successfully recovered the personal information encrypted and sent via network.

References

1. Esser, S.: Antid0te 2.0 - ASLR in iOS. In: Hack In the Box(HITB). (2011)
2. Apple: iOS Security Guide (2014)
3. Staff, P.: Citibank admits security flaw in its iPhone app. http://nypost.com/2010/07/26/citibank-admits-security-flaw-in-its-iphone-app/
4. Hollister, S.: Starbucks admits its iPhone app stores unencrypted user passwords. http://www.theverge.com/2014/1/15/5313648/starbucks-admits-ios-app-stored-passwords-in-plain-text
5. Egele, M., Kruegel, C., Kirda, E., Vigna, G.: PiOS: Detecting Privacy Leaks in iOS Applications. In: Proceedings of the Network and Distributed System Security Symposium (NDSS). (Feburary 2011)
6. Bellare, M., Rogaway, P.: Introduction to Modern Cryptography. http://cseweb.ucsd.edu/ mihir/cse207/classnotes.html
7. 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 and Communications Security. CCS '13, New York, NY, USA, ACM (2013) 73–84
8. Cracks, K.J.: Fast iOS executable dumper. https://github.com/KJCracks/Clutch
9. Davi, L., Dmitrienko, A., Egele, M., Fischer, T., Holz, T., Hund, R., Stefan: MoCFI: A framework to mitigate control-flow attacks on smartphones. In: Symposium on Network and Distributed System Security (NDSS). (Feburary 2012)
10. Pewny, J., Holz, T.: Control-flow Restrictor: Compiler-based CFI for iOS. In: Proceedings of the 29th Annual Computer Security Applications Conference. ACSAC '13, New York, NY, USA, ACM (2013) 309–318
11. Han, J., Yan, Q., Gao, D., Zhou, J., Deng, R.H.: Comparing Mobile Privacy Protection through Cross-Platform Applications. In: Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA (Feburary 2013)
12. Han, J., Kywe, S.M., Yan, Q., Bao, F., Deng, R., Gao, D., Li, Y., Zhou, J.: Launching Generic Attacks on iOS with Approved Third-Party Applications. In: Proceedings of the 11th International Conference on Applied Cryptography and Network Security (ACNS 2013). Volume 7954., Banff, Alberta, Canada, Springer Berlin Heidelberg (June 2013) 272–289 Lecture Notes in Computer Science.
13. Wang, T., Lu, K., Lu, L., Chung, S., Lee, W.: Jekyll on iOS: When Benign Apps Become Evil. (2013)
14. Enck, W., Gilbert, P., 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. In: Proceedings of the 9th USENIX Conference on Operating

Systems Design and Implementation. OSDI'10, Berkeley, CA, USA, USENIX Association (2010) 1–6
15. Szydlowski, M., Egele, M., Kruegel, C., Vigna, G.: Challenges for Dynamic Analysis of iOS Applications. In Camenisch, J., Kesdogan, D., eds.: Open Problems in Network Security. Volume 7039 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 65–77
16. Bhargavan, K., Fournet, C., Corin, R., Zalinescu, E.: Cryptographically verified implementations for TLS. In: Proceedings of the 15th ACM conference on Computer and communications security, ACM (2008) 459–468
17. Mitchell, J.C., Mitchell, M., Stern, U.: Automated Analysis of Cryptographic Protocols Using Murphi, IEEE Computer Society Press (1997) 141–151