DMIA: A MALWARE DETECTION SYSTEM ON IOS PLATFORM

Hongliang Liang, Yilun Xie and Yan Song

Beijing University of Posts and Telecommunications, Beijing, China
{hliang, xieyilun, yansong}@bupt.edu.cn

ABSTRACT

iOS is a popular operating system on Apple's smartphones, and recent security events have shown the possibility of stealing the users' privacy in iOS without being detected, such as XcodeGhost. So, we present the design and implementation of a malware vetting system, called DMIA. DMIA first collects runtime information of an app and then distinguish between malicious and normal apps by a novel machine learning model. We evaluated DMIA with 1000 apps from the official App Store. The results of experiments show that DMIA is effective in detecting malwares aimed to steal privacy.

KEYWORDS

iOS, Malware Detection, Dynamic Analysis, Machine Learning

1. INTRODUCTION

Apple iOS is one of the most popular and advanced operating systems for mobile devices on the market. By the end of January 2015, Apple had sold one billion iOS devices [1]. Apple exposes some APIs that can access to users' private data. This arises the privacy and security concerns. Because, for example, accessing to the users' location, can be used to track users across applications. If apps upload user's privacy without notifying users, we can regard these apps as malware. As the same, according to iOS developer license agreement [2], if an app use Private API, it is likely to be malware. Because Private APIs are functions in iOS frameworks reserved only for internal uses in built-in applications. They provide access to various device resources and sensitive information. After all, iOS apps face two threats: *abuse of security-critical Private APIs* and *stealing* (uploading without notifying the user) *privacy data in devices*.

To prevent third-party applications from performing malicious activities, Apple does review each app submission. And any violations of the App Store Review guidelines lead to rejection. It is generally believed that App Review is quite effective. However, recent work [3,4] shows that by constructing the names of Private APIs at runtime, it is possible to invoke Private APIs in third-party applications and still be able to pass the vetting process. Besides, there are several automated binary analysis systems [5, 6, 7, 8] proposed by security researchers to analyse iOS applications. However, the static analysis method in [5] could not resolve API names composed at runtime because of the runtime future and dynamic binding mechanism of Objective-C.

Natarajan Meghanathan et al. (Eds) : NETCOM, NCS, WiMoNe, GRAPH-HOC, SPM, CSEIT - 2016
pp. 63– 72, 2016. © CS & IT-CSCP 2016 DOI : 10.5121/csit.2016.61506

Dynamic approaches in [6, 7, 8] suffer from incomplete code coverage, thus would fail to detect uses of private APIs if malicious application authors place the invocations in complicated triggering conditions. And they could not find the private data uploading.

To improve the situation, we present DMIA in this paper. DMIA puts a monitor layer between system and application to catch the behaviours of an app, without the deficiencies (could not resolve API names composed at runtime) of static analysis caused by iOS runtime. We use 150 popular apps from App Store to train our classification model and it's equivalent to build a whitelist of the app behaviour. In summary, DMIA can solve some problems both in static and dynamic analysis tools. Monitor layer compensates the lack of static analysis which can't resolve API names composed at runtime. Machine learning model improve the problems of dynamic analysis which has high rate of false negatives due to the incomplete coverage of paths.

The main contributions of our paper are as follows:

(1) We insert a monitor layer between iOS system and applications to access applications' sensitive behaviours and network data. The layer can be regarded as a novel and effective dynamic binary instrumentation tool on iOS.

(2) We train a classification model of malicious behaviours based on machine learning method, which can distinguish malicious and normal applications.

The rest of the paper is organized as follows. We present the design of DMIA in section 2 and describe the implementation in section 3. Then we evaluate DMIA in section 4 and compare with related work in Section 5. Section 6 concludes the paper.

2. DESIGN

2.1. System Architecture

The general architecture of DMIA is depicted in Figure 1. DMIA consists of two parts: (1) The monitor layer between applications and the iOS system, (2) The classification model of malicious behaviours.

2.2. Monitor Layer between Applications and iOS System

The monitor layer is consisted of original network monitor, privacy function monitor, and special private APIs monitor.

Several tools like Wireshark can capture the network packages, but it's hard to handle with the issues of data encryption, packet loss, etc. Original network monitor of DMIA get original network data by hooking network functions. It outcomes the deficiencies of Wireshark and lessen the impact of encryption, through preset-value inspection which we will present in 2.3.

As we present before, one goal of DMIA is to detect malware by deciding whether it has uploaded private data or not. iOS will notice users to authorize privacy rights only at the first time to access it. Once a user has authorized it, he will not know when the app accesses his private data. So monitoring privacy function is important in DMIA. We hook those sensitive public APIs,

such as CLLocationManager which is provided by CoreLocation framework to get the user location, AddressBook Framework for access to directories and so on to monitor the privacy related behaviour.

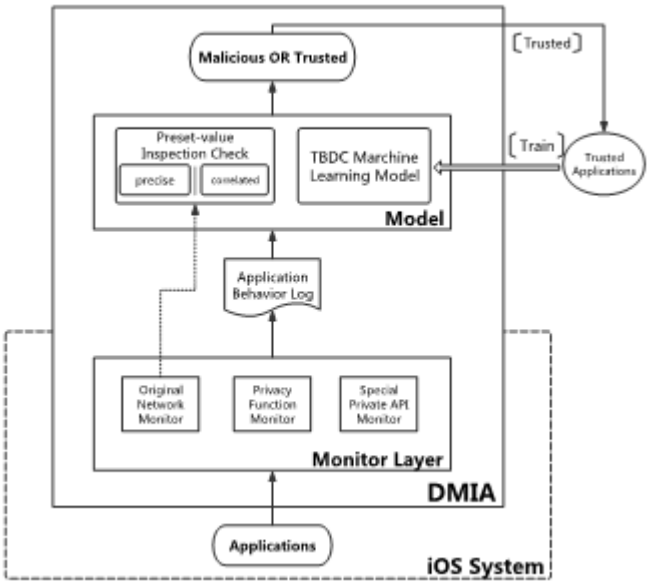

Figure 1. The architecture of DMIA

Private APIs are those functions in iOS frameworks reserved only for internal uses by built-in applications. They can be used to access to various device resources and sensitive information. To monitor private API abuse, we hook these related APIs which are selected according to the head files of private framework and related work about private API abuse, and our iOS development experience.

2.3. Malicious Behaviour Classification Technology Based on the Preset-Value Check

In order to detect whether an app will upload the user privacy, we designed the preset-value mechanism which consists of two kinds: ***precise*** and ***correlated***. We first describe the precise one. We forge privacy data in mobile phone, such as Reminder with specific text and address list with special phone number. We then collect mock privacy data and create a sensitive library based on it. Next, we match it with those network data obtained by the monitor layer. A full match indicates that the application is uploading privacy information illegally. Obviously, the ***precise*** method will fail if malware encrypt network data. To solve this problem, we introduce the ***correlated*** mechanism. Its main idea is the correlation detection. We also collect those mock privacy data to create the sensitive library, but we detect the relevance of them instead of a perfect match. By changing the content of the sensitive library regularly, DMIA monitors whether the communication data will change with it accordingly. If the correlation is greater than a threshold (0.6 is used in the paper), we think that the app uploads the privacy data.

2.4. Malicious Activity Classification Model Based on Machine Learning

Feature selection is a crucial step for machine learning. A reasonable feature will directly outperform the accuracy of most machine learning classifiers, despite some powerful models (e.g.

Long Short-Term Memory [9]) don't have to construct features manually. In section *Feature Vectors*, we discuss the feature selection strategy. In section *TBDC*, we propose our threshold-based dynamic classification model (TBDC).

Feature Vectors

Frequency of Sensitive APIs: malwares usually ask for more permissions than needed, and use them to obtain and upload sensitive information frequently. For example, a video app usually doesn't need to know who are in the user's contacts. Accordingly, the frequency of using sensitive APIs will be different between malicious and benign apps. Thus, we use the frequency of sensitive APIs as one feature.

In details, let f^{1i} denote the occurrence frequency of the *i*th API, where the superscript 1 means it is the first feature and $i \in [1,2,\dots,L]$ where *L* denoting the size of APIs that are being monitored, and it is computed as:

$$c_i = \begin{cases} c_i + 1 & if\ S_j = c_i \\ c_i & if\ S_j \neq c_i \end{cases} \quad (1 \leq j \leq |S|, 1 \leq i \leq L) \tag{1}$$

$$f^{1i} = c_i / |S| \tag{2}$$

Where S denotes the API sequence of an app, and is extracted from the system logs. |S| is the length of S. c_i is the occurrence number of the *i*th API in sequence S, which is initialized to zero.

Frequency of TBDC: The amount of sensitive API in iOS is very huge (In our experiment, we totally monitor 140 APIs). Intuitively, one app will just call a part of the APIs in their life cycle instead of all APIs (we prove that the conjecture is correct by our experiments). This phenomenon will lead to sparse feature vectors, which may increase the difficulty of model's training.

Consider this condition, we group the sensitive APIs into a much smaller set, which is based on their functions (e.g. Network, AddressBook). Assume the size of grouped API is L_g, we replace the API scope size L with L_g in equation (1) to compute the TBDC frequency. We define the feature vector obtained in this step as f^2.

Uncontrollable Behaviour Extraction: Frequency based features have enough power to represent the characteristics of different kinds of apps, but have limitations on one-class apps. For example, network-related APIs are used in video apps more often than in other apps, no matter they are malicious or benign.

In order to overcome this shortage, it is much important to know whether the behaviours of an app are under the user's control. We call the behaviours without requesting user's permissions as uncontrollable behaviours. For example, if there is a user interaction event (e.g. click button) just before a network request, the behaviour is regarded under the user's control (controllable behaviour).

In summary, a behaviour (e.g. URL request, address request) is defined as controllable behaviour only when it is just after a user interaction event. In the opposite, we define it as uncontrollable

behaviour. We adopt the frequency of uncontrollable behaviours to generate the feature vector. In more detail, let $Behavior_{all}$ be all behaviours, which we are interested in, occurred in the API sequence. Alike, let $Behavior_{uncon}$ be those uncontrollable behaviours. Then, the feature vector f^3 is computed as $f^3 = Behavior_{uncon} / Behavior_{all}$.

Threshold-Base Dynamic Classification Model

Generally, before machine learning classifier get good performance, it needs a lot of data to training. Because, except the over fitting problem, more good data always lead to better performance, at least not worse than before. But when the training dataset is not enough for the classifier to learn necessary attributes, it just become a shot in the dark.

From this, we propose Threshold-Based Dynamic Classification Model (TBDC), which can own good result even when training dataset is small. The essential idea is, first, we train a regression model with the small dataset. Then, we compute a threshold of becoming malware according to the output of the regression model with the initial dataset as input. Finally, make the test samples' feature vectors as inputs, we can get the outputs of the learned regression model. If the output fall outside the threshold range, we classify it into malicious, otherwise, benign. And if there are new training samples, we will retrain the regression model to adjust the parameters, and generate new threshold dynamically, which makes TBDC have the ability to classify in a more fine-grain way.

In more detail, Let f be a vector consisting of all the feature vectors $[f^1,f^2,\ldots,f^m]$, where m is the amount of features (e.g. m = 3 in our feature space). Let $M \in \mathbb{R}^{n\times\Sigma_j^m|f^j|}$ be a matrix consisting of all samples' feature vector, where n quantifies the number of input samples, which is also treated as input matrix. For example, M_i is the *i*th row of M, which donates the *i*th sample's feature vector. Then, the output vector y and thresholds are computed as:

$$y = g(M) \tag{3}$$

$$[threshold_{min}, threshold_{max}] = h(y) \tag{4}$$

Where g is the regression function, h is a algorithm, which is used to compute the thresholds.

3. IMPLEMENTATION

In this paper, the monitor Layer runs between the jail-broken iOS system and applications. We use *Tai Chi tool* to jailbreak iOS 8.3 and *MobileSubstrate* to insert hooks at system level. We develop a dylib by iOS OpenDev and program with logo language. We debug and test our dylib on iPod Touch 5. Our preset-value inspection and TBDC model are developed by python.

3.1. Arrange Privacy Related Functions

In order to obtain more accurate and comprehensive information of privacy behaviour, we search Apple API documents based on all options in iOS system - Settings - Privacy. In the end, we collect 216 related functions. Then remove duplicate functions based on action and implement 89 hooks of key functions. Besides, we have also collect APIs about device information, such as

[UIDevice identifier For Vendor] (it can be used to get device UUID) and so on, total 15 functions. At last, to vet private API abuse, we export private API headers of iOS 8.3 SDK by class-dump. According to our development experience and function names, we sort out 31 privacy related functions. Then controlling of the existing research on private API abuse such as iRiS [10], we ultimately determine 36 private API function related privacy and hook them. At this point we have completed the work of arranging privacy related functions (140 totally).

3.2. Insert Monitor Layer

Hook the 140 functions sorted out by section 3.1. Program tweak by logo language and the program consist of 11 parts. NetworkHook, AddressBookHook, EKEventStoreHook, Calendar and Reminder, PhotoHook, MicHook, CameraHook, HealthHook, HomeKitHook, CLLocationHook, PrivateApiHook, OtherHook. Under the premise of keeping the original function of method, we append behaviour record to them. Thus, we can record the parameters, return values and call time into the system log according to the prescribed format. Finally, we compile the code into BehaviorMonitor.dylib and load it into iPod touch 5.

3.3. Implementation of TBDC

First, we give each API an index to map the text name into vector space. For example, *initWithRequest:delegate* is the first sensitive API that we monitored, thus, we index it with integer 1. Next, we extract app's API call sequence from the system log, and record it with API index. For example, a simple network request is achieved with *initWithRequest:delegate:startImmediately:* and *connection:WillSendRequest:redirectResponse* after it. So, we transpose this sequence into 2 11, where 2 and 11 are the index of the two APIs respectively. Then we construct the feature vectors as we discussed in section 2.4.1.

As for the regression function, we tried Support Vector Regression (SVR), which is based on Support Vector Machine (SVM), and Multilayer Perceptron (MLP). To get the threshold range, we simply set the minimum of benign samples' outputs as the minimum, and the maximum of benign samples' outputs as the maximum.

4. EVALUATION

In order to judge whether DMIA is effective and efficient in detecting malware, we have carried out massive experiments. Further more, for the two threats (abuse of security-critical Private APIs and stealing privacy data in devices) focused by DMIA, we expound them respectively in the end of evaluation as case studies.

We evaluate DMIA with 1000 applications from App Store. There are 24 categories in total: Children, education, shopping, photo & video, efficiency, food, live, fitness, journey, music, sport, business, news, tools, entertainment, social contact, newspapers and periodicals, finance, reference, navigation, medical treatment, books, weather and commodities guide. We download them from iTunes and install them in iPod Touch5, which is iOS 8.0 version. We run and capture every app's behaviour by Monitor Layer.

In the experiments, we collect 606132 pieces of text messages (over 64MB, size in total), which record behaviours of these apps. In these messages, about 430 thousand pieces (71%) are related

to network API, about 48 thousand pieces (8%) are related to location API, and 30 thousand pieces (5%) are related to photo and camera API. 97 thousand pieces (16%) are related to all the rest APIs.

We make a statistic of that whether one app in particular category use one privacy related API or not. We assume that API calls of each normal app in particular category are similar. So, if there are a handful of apps (less than % 3) in particular category using a privacy related API, we think it suspicious. Following, we take Location API and AddressBook API for example. For the frequency of using Location API, navigation class applications are the highest (100%) and weather (98%), social (85%), food (81%), finance class applications and efficiency class applications are the lowest (5%). The APIs related to address book, are used by 76 of the 100 apps in social contact classification, in contrast, by 2 of the 100 apps in weather classification. So we think these two apps suspicious and analyse them carefully. Interestingly, they are not real weather-class apps. They just mark themselves as weather category when applying for app review. Among them, *pp assistant for phone* uploads users' privacy data without notification obviously and it is regarded as a malware. We review its *detail page* and *comments* on iTunes and find that its details screenshot is a game rather than a weather or assistant picture. 1177 of its 1283 comments are puzzling sentences and generated by robot obviously.

Case Studies

This paper focuses on the two kinds of threats in iOS system. Abuse of security-critical Private APIs and stealing (uploading without notifying the user) privacy data in devices. Here, we take i4Tools and Youmi SDK [11] for example to explain how DMIA resist the two threats and demonstrate the effectiveness of DMIA again.

i4Tools. In the lot-sizing tests, DMIA find the features of i4Tools are far away from normal value, which means it may be a malware. So we analyse its text carefully, which has 5600 lines. 4512 lines (80.5%) of them are related to network, 128 lines refer to geographical location information, 124 lines are private API. Especially, 84 of the 124 lines are about LSApplication class. So we know i4Tools break the iPhone developer agreement. What is more, according to preset-value detection, we find it still request network at a fixed time when screen interaction events don't happen. The correlation value between getting and uploading privacy is 0.85. It is greater than our threshold of 0.6. So it uploads data without permission or knowledge of the user. In a conclusion, it is a malware.

Youmi SDK. For apps using advertising app SDK, the proportion of malware in is much higher than others. Especially, almost all of the app texts containing youmi.com are judged to be abnormal by DMIA. So we suspect that the issue is in Youmi SDK. We download Youmi SDK from its official website and program a demo app according to its instructions. Then we test this demo with DMIA. And we get a total of 3221 lines information, of which 153 line involving private API. But it had unauthorized network transmission only when starting the app, and the requests at the rest of the time are all normal. So we can only say it violates the Apple's user agreement and abuse the user privacy data.

5. RELATED WORK

The work related to DMIA can be classified into two categories: (1) Privacy Leak Detection on iOS (2) Machine Learning Model.

5.1. Privacy Leak Detection on iOS

SecLab's PiOS [5] can detecting privacy leak of app. It creates hierarchical structure of class from binary file and build CFG. Analysing data stream to judge that weather privacy information transform from origin to leak point. This is a kind of static analysis. There are several shortcomings of PiOS such high false positive.

To overcome it, Peter Gilbert introduce some other ideas in 2011.6. They create AppInspector [12], a dynamic analysis tool. It can obtain application behaviour by analysing system call. Then summer out wither application access to privacy information or not.

Martin Szydlowski discussed the challenge on dynamic analysis of iOS app and developed a prototype. It can rack sensitive API calls by breakpoint debug and gets app UI model automatically [6]. 2012, Joorabchi and Mesbah implemented iCrawler. It can view the app UI and generate a model containing different UI state. This tool has accelerated the process of iOS app reverse [8]. Although the achieved coverage of their navigation technique looks promising when applied on a few open-source apps, it does not support simulation of any advanced gestures or external events. Moreover, the technique used by iCrawler is only applicable to standard UI elements, and, most notably, iCrawler has not been designed to perform privacy analysis.

Andreas Kurtz introduces DIOS, which is an iOS privacy leak analysis model based on dynamic API call sequence [7]. DIOS mainly includes three parts: Backend is used as the central data storage, worker is data interactive link between backend and iOS device and client is used for analysing the behaviour of the iOS App. DIOS can monitor privacy data access by hooking iOS API function. But the access to private data is not the same as privacy leak. And compared to the static analysis dynamic analysis has high false negative rate and low speed. In contrast, DMIA not only hook a greater variety of private functions, but also monitor the Private APIs and the network. And based on application behaviour, it can determine that it is normal access or privacy theft by preset-value inspection and TBDC model.

5.2. Machine Learning Model

Resent work by Gorla et al. [13] try to use app descriptions and sensitive APIs to check app behaviour in Android platform. They cluster apps that have analogical behaviours into one category and selected the most used APIs the feature of that category. An app will be classified depending on whether it's APIs is accord with the category's, which it belongs, APIs. But they do not construct any features with the sensitive APIs.

DroidADDMiner [14] is a machine learning model based on FlowDroid [15]. It adopts data flow analysis of sensitive APIs to capture the semantics information of malware. But it relies on big training dataset to get good performance.

Sundarkumar et al. [16] tried to use API information to characterize Android malware. They use text mining and topic modelling, combined with machine learning classifier, to detect malwares. But due to the shortage of static analysis, which their works mainly based on, there are false negative and false positive problems.

Some other systems [17, 18] also use static analysis to get API information, as part of their features. Their features also contain other information (e.g. permissions, package information) which they think crucial. But they all suffer at static analysis shortage and big training dataset.

6. CONCLUSION

In this paper, we present the design and implementation of a malware vetting system, called DMIA. It first collects application behaviour information and original network data via its monitor layer. The monitor layer can be considered as a novel dynamic binary instrumentation tool on iOS. Then, DMIA captures violations of stealing users' privacy by the novel *machine learning model*. Finally, our experiments with 1000 applications show that DMIA is powerful in detecting malwares. In our future work, we aim to provide DMIA as a usual app without requiring users to jailbreak their devices. Users can detect their apps by DMIA and upload the results to our server. We hope optimize our training set by this crowdsourcing technique and make DMIA more powerful.

REFERENCES

[1] Gize BusinessInsider. Apple has shipped 1 billion ios devices. http://www.businessinsider.com/apple-ships-one-billion-ios-devices-2015-1.

[2] Apple. ios developer program license agreement. http://www.thephoneappcompany.com/ios_program_standard_agreement_20130610.pdf..

[3] J. Han, S. M. Kywe, Q. Yan, F. Bao, R. Deng,D. Gao, Y. Li, and J. Zhou. Launching generic attacks on ios with approved third-party applications. In Applied Cryptography and Network Security, pages 272-289. Springer, 2013.

[4] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee. Jekyll on ios: When benign apps become evil. In Usenix Security, volume 13, 2013.

[5] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios: Detecting privacy leaks in ios applications. In NDSS, 2011.

[6] M. Szydlowski, M. Egele, C. Kruegel, and G. Vigna. Challenges for dynamic analysis of ios applications. In Open Problems in Network Security, pages 65-77. Springer, 2012.

[7] A. Kurtz, A. Weinlein, C. Settgast, and F. Freiling. Dios: Dynamic privacy analysis of ios applications. Technical Report CS-2014-03, Department of Computer Science, June 2014.

[8] M. E. Joorabchi and A. Mesbah. Reverse engineering ios mobile applications. In Reverse Engineering (WCRE), 2012 19th Working Conference on, pages 177-186. IEEE, 2012.

[9] Hochreiter S, Schmidhuber J. Long short-term memory[J]. Neural computation, 1997, 9(8): 1735-1780.

[10] Zhui Deng, Brendan Saltaformaggio, Xiangyu Zhang, Dongyan Xu: iRiS: Vetting Private API Abuse in iOS Applications. ACM Conference on Computer and Communications Security 2015: 44-56

[11] https://www.theiphonewiki.com/wiki/Malware_for_iOS#Youmi_Ad_SDK_.28October_2015.29

[12] Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung. Automating Privacy Testing of Smartphone Applications.

[13] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, Checking app behavior against app descriptions, in 36th International Conference on Software Engineering, ICSE' 14, Hyderabad, India - May 31 - June 07, 2014, 2014, pp. 1025-1035.

[14] Yongfeng Li, Tong Shen, Xin Sun, Xuerui Pan, Bing Mao: Detection, Classification and Characterization of Android Malware Using API Data Dependency. SecureComm 2015: 23-40

[15] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: Proceedings of the 35th ACM SIGPLAN Conference on.

[16] G. G. Sundarkumar, V. Ravi, I. Nwogu, and V. Govindaraju, Malware detection via API calls, topic models and machine learning, in IEEE International Conference on Automation Science and Engineering, CASE 2015, Gothenburg, Sweden, August 24-28, 2015, 2015, pp. 1212-1217.

[17] P. P. K. Chan and W.-K. Song, Static detection of Android malware by using permissions and API calls, in 2014 International Conference on Machine Learning and Cybernetics, Lanzhou, China, July 13-16, 2014, 2014, pp. 82-87.

[18] A. Sharma and S. K. Dash, Mining API Calls and Permissions for Android Malware Detection, in Cryptology and Network Security-13th International Conference, CANS 2014, Heraklion, Crete, Greece, October 22-24, 2014. Proceedings, 2014, pp. 191-205.