Plagiarizing Smartphone Applications: Attack Strategies and Defense Techniques

Rahul Potharaju, Andrew Newell, Cristina Nita-Rotaru, and Xiangyu Zhang

Department of Computer Science, Purdue University, IN, USA
{rpothara,newella,crisn,xyzhang}@cs.purdue.edu

Abstract. In this paper, we show how an attacker can launch malware onto a large number of smartphone users by plagiarizing Android applications and by using elements of social engineering to increase infection rate. Our analysis of a dataset of 158,000 smartphone applications *meta-information* indicates that 29.4% of the applications are more likely to be plagiarized. We propose three detection schemes that rely on syntactic fingerprinting to detect plagiarized applications under different levels of obfuscation used by the attacker. Our analysis of 7,600 smartphone application *binaries* shows that our schemes detect all instances of plagiarism from a set of real-world malware incidents with 0.5% false positives and scale to millions of applications using only commodity servers.

1 Introduction

Smartphone applications repositories have been growing at a high rate with support from hundreds of thousands of developers. AppStore [1] contains more than half-a-million applications, and Android Market [2] has just crossed the two hundred thousand mark. The two repositories use different procedures for accepting an application. Apple's AppStore accepts only applications that have been tested for potential vulnerabilities by Apple's test engineers. Android Market accepts applications without subjecting them to any code review or inspection.

The approach of being open allows the Android Market to make applications immediately available to users. However, this also makes it an easy target for attacks where plagiarized applications are used by an attacker as means to launch malware or gain personal profits. First, an attacker can easily reverse engineer applications using existing tools. Second, an attacker can easily manipulate any arbitrary application from the market and re-pack it under his name. Third, the attacker can leverage the centralized nature of the market, dashboard features that make applications immediately available to users, and social engineering (*e.g.*, using catchy titles) to push malicious applications to a large number of victims. Thus, an attacker can easily download a popular application, insert malicious code into the application, and resubmit the malicious version back into the market without being detected. We refer to this class of actions as *plagiarism* and to the modified application as a *plagiarized* application.

While several plagiarizing incidents [3, 4] targeting applications from the Android Market have been reported, there are currently no fool-proof preventative

G. Barthe, B. Livshits, and R. Scandariato (Eds.): ESSoS 2012, LNCS 7159, pp. 106–120, 2012.

mechanisms in place to detect plagiarism in open markets. Signature-based malware detection tools such as Lookout Security [5], Norton Mobile Security [6] and BitDefender Mobile Security [7] detect applications that contain malware. However, they do not detect plagiarized applications that use legitimate permissions, users will still be infected before an attack signature is learned, and the number of infected users can be large due to the centralized nature of markets. Information leakage detection techniques based on taint analysis [8], access control policies [9], and kernel modifications [10] protect against stealing critical user information. However, such schemes require significant user input in order to achieve high accuracy. In addition, most of these techniques work on the client-side and often demand heavy resources that lead to battery drain.

Contributions: In this paper we propose a solution that detects plagiarized applications and prevents their acceptance into the market. As a result, our approach raises the bar for the attackers, forcing them to create original applications to host their malware. Our solution is designed to be applied on the market side and is complementary to client-side defense techniques such as malware detection and information leakage prevention. Our contributions are:

– We analyze the *meta-information* of 158,000 applications from the Android Market and find that 29.4% of the applications are more likely to be plagiarized because of the permission rights they provide to an attacker. We also found that an attacker can use category, total number of downloads, and published weekday to increase the first-day number of downloads for the plagiarized application.
– We propose three schemes *Symbol-Coverage*, *AST-Distance*, and *AST-Coverage* that rely on symbol tables and method-level Abstract Syntactic Tree (AST) fingerprints to detect plagiarized applications under different levels of obfuscation used by the attacker. Our analysis of 7,600 smartphone application *binaries* shows that our schemes can detect all instances of plagiarism from a set of real-world malware incidents while having only a 0.5% false positive rate.
– We show that our detection schemes scale to millions of applications using commodity servers, i.e. it takes 2-8 seconds to reverse-engineer and fingerprint an application and 0.8-1.4 seconds to retrieve plagiarized versions of an application.

The rest of the paper is organized as follows. Section 2 describes our system and threat model. Section 3 presents more details about how an attacker can plagiarize applications. Section 4 gives an overview of our defense solution, and Section 5 presents its evaluation. Section 6 discusses related work and Section 7 concludes our paper.

2 System and Threat Model

In this section we first give more details about the application submission process in the Android Market. Then, we describe the resources and mechanisms available to an attacker to plagiarize applications.

2.1 Android Development Process

Android [11] is an open source software stack for mobile devices that includes an operating system, an application framework, and core applications. The operating system relies on a kernel derived from the Linux kernel. The application framework consists of the Dalvik Virtual Machine [12] that runs .dex files. Applications are written in Java using the Android SDK [13], compiled into .dex (Dalvik Executable), and packaged into .apk (Android package) archives for installation.

To submit an application, a developer must have a publisher account obtained by paying a nominal one-time fee of $25.00 USD and by having a valid Gmail account. He can then upload the application, optionally setting the price for a paid application. Each binary is accompanied by *meta-information* which includes: name, rating, date updated, version, category, number of installs, size, price, etc. The Android Market requires that all applications are digitally signed, with the public key made available as a *digital certificate*. As an optional step, developers can obfuscate their binaries through a tool called ProGuard [14] which removes any debugging information and renames the identifiers (*e.g.*, class, method, variable names) while maintaining the same functionality.

2.2 Threat Model

The attacker collects sensitive information stored on smartphone devices or obtains monetary profit by exploiting users. Examples of sensitive information include usage information, IMEI numbers, and GPS location. Ways to obtain monetary profit include redirecting ad-revenue or forcing smartphones to call a toll number that is owned by the attacker. We assume that the attacker can obtain a developer account for the Android Market without being traced by the market administrators.

We consider attacks that exploit the popularity and permissions already available in existing applications. Thus, an attacker chooses an existing application that already has permissions that can be exploited, modifies it according to his needs, and uploads the modified version to an open market. The modified version not only has the same functionality as the original application but also includes malicious code to collect sensitive information or obtain monetary profit.

Android Market requires developers to use digital certificates to attest their identity. However, it does not require a trusted *certificate authority* (CA) to sign the certificates. Thus, digital certificates will not prevent an attacker from plagiarizing an application. Establishing trust of developers through CAs would hinder the openness of the markets. If CAs were to enforce high requirements to deter malicious developers then many legitimate developers will also be excluded from being trusted by the CA.

2.3 Obfuscation Model

We assume that an attacker can apply the following obfuscation techniques:

– **Level-1:** Symbol table is obfuscated such that methods, classes, variables, and other identifiers are all changed. The tool ProGuard provided by Google allows only Level-1 obfuscation. To the best of our knowledge, this is the only kind of obfuscation that is being applied in the real-world for mobile applications.
– **Level-2:** α random methods with no functionality are added. This level of obfuscation has not been seen yet in real smartphone application repositories, we nonetheless consider it as attackers can leverage it without substantial efforts.

While more advanced obfuscations have been proposed in the research community [15, 16], their applicability to mobile applications remains unknown due to the specific byte-code format and the tight resource and energy constraints.

3 Plagiarizing Applications

The goal of an attacker plagiarizing an application is to take advantage of its popularity and collect sensitive information or obtain monetary profit. We first describe the attack payloads that the attacker can embed inside the plagiarized version of the application and then describe strategies that an attacker can leverage to increase the overall infection count.

3.1 Plagiarism Mechanisms and Payload

An attacker resorting to plagiarism first downloads an application and obtains the .dex files of the application. The attacker then uses one of the two approaches: (1) *direct byte-code insertion*, (2) *assisted byte-code insertion*. In *direct byte-code insertion*, the attacker writes his own bytecode into the application byte-code and re-packages it into an APK package. This approach usually requires heavy expertise in writing Dalvik specific byte-code but has the advantage that it can evade detection of certain static analysis tools that rely on application-level source code heuristics. In *assisted byte-code insertion*, the attacker first writes his malicious code as part of a stub application and compiles it using the Android SDK. The attacker reverse engineers his own APK to obtain the .dex files and extracts relevant portions of the byte-code for insertion into the original application and then re-packages it into a separate APK package. Possible payloads for the plagiarized application include:

– **Privacy Exploitation:** If the original application requests for permissions to obtain the GPS coordinates of the user (ACCESS_FINE_LOCATION) or to read the user's contact data (READ_CONTACTS), the attacker can insert a code snippet that obtains this information and sends it to back to the attacker.
– **Monetary Exploitation:** If the original application requests for permissions to send SMS messages (SEND_SMS) or to allow the application to initiate a phone call without going through the *Dialer* user interface that forces users to confirm the call being placed (CALL_PHONE).

Table 1. Attack payload collected from a dataset of 158,000 applications. A 29.4% of the applications are susceptible to at least one payload listed in the table.

Permission	Permission	# of App.	Possible Attack Payload
INTERNET	ACCESS_COARSE_LOCATION	28,759	retrieve location through WiFi
INTERNET	ACCESS_FINE_LOCATION	27,258	retrieve location through GPS
INTERNET	READ_CONTACTS	11,870	retrieve user's contacts
INTERNET	CAMERA	6,936	record/retrieve images from camera
ANY	SEND_SMS	7,652	send SMS messages
ANY	CALL_PHONE	8,074	place phone calls
ANY	BRICK	11	permanently disable the device
ANY	INSTALL_PACKAGES	430	install arbitrary packages

To gain insights into how many applications are vulnerable to attacks we examined the *meta-information* (descriptive information about an application) of 158,000 applications that we collected from the Android Market. We count the number of applications from our dataset that request permissions that can be exploited for various types of attacks. The results presented in Table 1 show that many applications require permissions that can be leveraged by attackers. For instance, an attacker interested in sending SMS from legitimate phones can choose applications to plagiarize out of a set of 7,652. We estimate that 29.4% of the applications from our dataset are susceptible to at least one attack payload listed in Table 1. Our findings are consistent with results in [17] which showed that many Android applications violate the principle of least privilege and request more permissions than needed.

3.2 Improving Infection Count

An attacker can increase the infection count of a plagiarized application by carefully choosing what applications to plagiarize and what day of the week to perform the attack. A good strategy for an attacker is to target applications that can rapidly become popular the first day. The optimal strategy we found from our dataset of applications is to plagiarize an *Arcade & Action* game that has more than 250,000 downloads and release this plagiarized application on Sunday.

We extract from our dataset: *(i) the category of the application*, *(ii) the number of downloads the first day the application was submitted*, *(iii) the current download count*, and *(iv) the day of the week published*. From our dataset of 158,000 applications, a subset of 36,000 applications have sufficient information to extract these four pieces of information. We cannot obtain exact download counts each day due to the way Android Market reports download counts in ranges, so we simply assume the average of the upper and lower bound to be the number of downloads (100-500 downloads is interpreted as 300 downloads).

We divide the 36 categories on the Android Market into 4 groups: {*Personal, Games, Media, Leisure*}. Figure 1(a) shows a TreeMap [18] where each category is represented by a rectangle with a size corresponding to number of applications and color corresponding to number of first day downloads. Clearly, some categories have a much higher number of initial first day downloads compared to other categories, e.g., the category *Arcade & Actions* achieves nearly double

the downloads of the next highest category, *Casual Gaming*. Also, *Games* has high first day download counts compared to the other groups. An attacker can choose specific categories to achieve higher initial downloads since the category choice affects initial download count.

We found that applications which have a high download count went viral on their first day. Figure 1(b) plots the average download count on the first day for each download range listed currently in the market. The applications that have reached greater than 250,000 downloads are the most popular applications, receiving four times the initial download count as opposed to the next highest bracket of 50,000-250,000 downloads.

An attacker can also try to choose the day of the week to load the application to increase the probability that the plagiarized application will become viral on its first day. Figure 1(c) shows the average download count on the first day of an application given the day of the week it was uploaded to the market. We have between 3,996 and 5,750 samples for each day of the week, so we can confidently conclude that an attacker can expect higher first day download counts by roughly 20% by selecting an appropriate day to execute the attack (e.g., weekends).

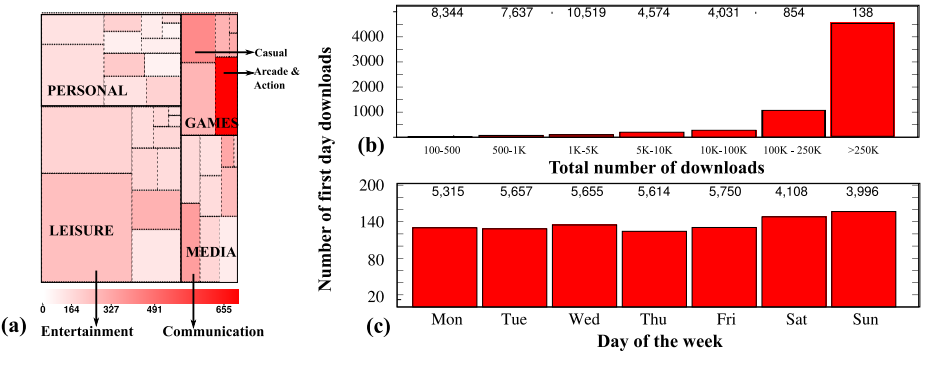


Fig. 1. Choosing an application based on: (a) **category:** the area of each rectangle maps to the number of applications and the color corresponds to first day download counts, (b) **total downloads:** number of applications is on top of each bar, (c) **application publish weekday:** number of applications is on top of each bar

4 Detecting Plagiarized Applications

We first describe the extraction of symbol tables and application fingerprints from binary code. We then describe how we use the extracted information to detect potentially obfuscated plagiarized applications with three schemes *Symbol-Coverage*, *AST-Distance*, and *AST-Coverage*. See Appendix for more detailed descriptions of the algorithms presented in this section.

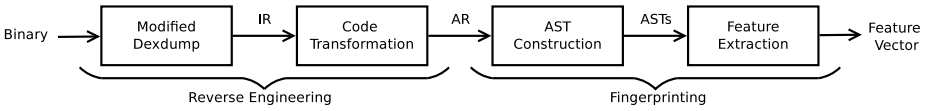


Fig. 2. Reverse Engineering and Fingerprinting Procedure

4.1 Reverse Engineering and Fingerprinting Android Applications

Figure 2 shows how we obtain application fingerprints by starting with an application binary. We first reverse engineer it to obtain an Abstract Representation (AR) of the corresponding source code. Then we use the AR to construct method-level ASTs and extract feature vectors that represent the fingerprints.

Reverse Engineering. Android has a built-in open source disassembler called *dexdump*. Given a .dex file, *dexdump* creates a dump file (IR code) of all the classes and methods. We modified five functions: *dumpClass*, *dumpMethod*, *dumpCode*, *dumpSField* and *dumpIField*[1] to capture relevant method-related information. *Code Transformation* is then used to obtain an AR based on pre-defined rules for each statement type. A rule is a regular expression that captures the various parts of the statement (such as method name, variable names, number of arguments etc.). Once this information is captured, all variables are named alike (e.g. x,y,z are all replaced with ARG). The advantage of using AR is that even if the code is obfuscated through means such as variable re-naming, the syntactic structure is preserved and any fingerprint generated out of this transformed code will be the same for both obfuscated and non-obfuscated code.

Consider an excerpt from the IR representation of a larger method shown in Figure 3(a). The example shows a method invocation that starts by defining the type of invocation (*virtual*, *static*, *direct*, *super* or *interface*), then the registers that are to be checked for the arguments, and finally the signature of the method that is being invoked. We fingerprint the two most common types of invocations: *invoke-direct*, which is used to invoke an instance method that is by nature non-overridable (either a *private* method or a *constructor*) and *invoke-virtual*, which is used to invoke a normal virtual method (a method that is not *static* or *final*, and is not a *constructor*). For instance, the first invocation is made to the *init* method of the super class *android/app/Activity* and the second to the *findViewById* method of the Fuzzer class from our sample application. The *Code Transformation* module rewrites this set of statements as seen in Figure 3(b).

Extracting Fingerprints. Once we have an abstract representation of the IR code, we perform *AST Construction* which generates a special type of Abstract Syntax Tree called the *method-level Abstract Syntax Tree*, for each method in the byte-code. A method-level AST captures the following information: (i) Number of arguments that the current method accepts, (ii) Other methods invoked by

[1] While our tool is home-brewed and not ready for a production usage yet, we do acknowledge the presence of several other promising tools [19, 20] that came out recently.

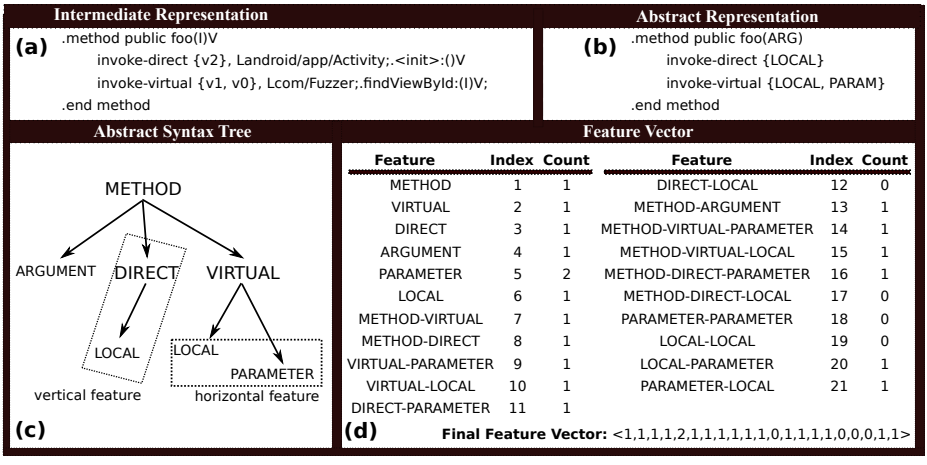


Fig. 3. Example of Fingerprinting an Applications

the current method with the invocation type, direct or virtual. Other syntactic artifacts inside a method, such as assignment and conditional statements, are precluded. For instance, consider the AST given in Figure 3(c). The root node is always the METHOD label and subsequent nodes ARGUMENT, DIRECT, and VIRTUAL denote the method has an argument and two function invocations, one with the direct type and the other the virtual type. The children of the second-level nodes then, are the registers that these methods are utilizing.

We then use the ASTs to extract a feature vector that represents the application fingerprint. We adapt the structural feature extraction method in [21] to work with the method-level ASTs that we constructed. For a given AST, we record two types of patterns of structural information: *(l,m)-leaf* and *n-path*. A *(l,m)-leaf* is a pair of leaf nodes having a common parent and is used to capture the *horizontal paths* in an AST. In Figure 3(d), {*LOCAL-PARAMETER*} is one such *horizontal path* in the AST. Note that an AST does not need to have any *(l,m)-leaf* pairs. This can happen when the method does not have any arguments or does not contain other methods that have arguments. An *n-path* is a directed path of n nodes, *i.e.*, a sequence of n nodes in which any two consecutive nodes are connected by a directed edge in the tree. *DIRECT-LOCAL* is one such *vertical path* in the AST. Other *vertical paths* are: {*METHOD, METHOD-ARGUMENT, ...* }. A special case is *1-path* which contains only one node.

The feature vector in our case is the occurrence count of all the *horizontal* and *vertical* paths extracted from the AST. To derive this for a given AST, we first allocate a vector filled with 0's and compute all *(l,m)-leaf paths* and *n-paths*. For each individual path, we get the path's identifier from a global lookup table that holds all possible paths. This identifier is used to determine which of the dimensions in the vector needs to be incremented. A feature vector for an entire application can be generated in the same way by calculating the feature vector over a graph that is a forest where each tree represents a method of the application. Figure 3 shows an example of this procedure.

In terms of performance, feature extraction is at most quadratic with respect to the total number of nodes, n. For vertical feature extraction, the paths of an AST are limited to a constant length, three, so the number of total paths to traverse is $O(n)$. For horizontal feature extraction, the number of pairs to iterate over and count in the worst case, a single DIRECT or VIRTUAL node has $O(n)$ children (this case is highly unlikely), corresponds to $O(n^2)$ pairs.

4.2 Detection Techniques

We design three defense schemes: Symbol-Coverage for non-obfuscated applications, AST-Distance for applications with Level-1 obfuscation(observed in the real-world), and AST-Coverage for applications with Level-2 obfuscation (not observed in real-world to the best of our knowledge but possible).

Symbol-Coverage. If an attacker does not obfuscate, the symbol table information is available from the application byte-code. We consider the coverage of symbol table information for every application $A_1, A_2, ..., A_n$ by an uploaded application A. If some application A_i is covered highly by A, then we consider A a plagiarized version of A_i. The coverage of an application A_i by A is computed as the number of classes and methods in A_i that also exist in A divided by the total number of classes and methods in A_i. We only consider methods as matching if they belong to classes that match. The application A_i with the highest coverage is reported if the coverage exceeds some threshold.

AST-Distance. If an attacker obfuscates symbol table information (Level-1), we use a defense based on feature vectors derived from method-level ASTs (see Section 4.1). We use Euclidean distance due to its high accuracy in preliminary results where we tested various distance metrics. Let A_i be the application with a feature vector that has the smallest distance to the feature vector of the application A, then, A_i is reported if this distance is smaller than some threshold.

AST-Coverage. In our final algorithm, we aim to accurately detect plagiarism where applications were obfuscated with Level-2 obfuscation. We combine the AST based feature vectors with the coverage approach. Specifically, once the ASTs of each method of the applications in the market $A_1, A_2, ..., A_n$ and the uploaded application A are transformed into feature vectors, feature vectors of each method of $A_1, A_2, ..., A_n$ that are close to a feature vector of A are marked as covered. The maximally covered application A_i is reported if the coverage is greater than some threshold value.

5 Defense Evaluation

We evaluate the detection accuracy of our schemes using a dataset of 7,600 application binaries that we refer to as the Pseudo-Market.

5.1 Real-World Plagiarism Detection

We analyzed 13 instances[2] of the HongTouTou [22] malware which plagiarized highly-popular legitimate applications and relied on social engineering. The

[2] We thank Tim Strazzere of Lookout Security for sharing the malware samples with us.

Table 2. Real-World Plagiarisms: List of plagiarized instances of legitimate applications that have occurred in the Android Market. We show the coverage of AST-Coverage with (+) and without (-) the legitimate application in the market.

Application		Characteristics		AST-Coverage	
Legitimate Title	**Malware Title**	**Price**	**Downloads**	+	-
yxPlayer	Flash Player	Free	≥250,000	1.000	0.100
Steamy Window	Screen Mist	Free	≥250,000	1.000	0.118
Hello Kitty LWP Lite	HelloKitty Livewallpaper	Free	≥250,000	1.000	0.053
Wave Live Wallpaper	Wave Livewallpaper	Free	50,000-250,000	1.000	0.077
AndroMax	Multi-Keyboard Shortcuts	Free	50,000-250,000	1.000	0.100
Shamrock Live Wallpaper	Clover Wallpapers	Free	50,000-250,000	1.000	0.053
City at Night	NightCity	$0.99	50,000-250,000	1.000	0.077
Hi-Hiker Pro	Hiker	Free	50,000-250,000	1.000	0.100
Dandelion Livewallpaper	TAT-LWP-Mod-Dandelion	Free	10,000-50,000	1.000	0.006
Robo Defense	Robo_Defense	$1.88	1,000-5,000	1.000	0.105
Sense Live Wallpaper Pro	Beautiful Live Wallpaper	$1.88	1,000-5,000	1.000	0.333
Yo Handcar: Off the Rails	yohandcar	Free	1,000-5,000	0.992	0.182
Roller Rev 99	Crazy Roller Coaster	$2.99	100-500	1.000	0.182
Stickers Off	Miniv	Free	100-500	1.000	0.100
Snow Flurry Live Wallpaper	LiveWinter	$0.99	100-500	1.000	0.043

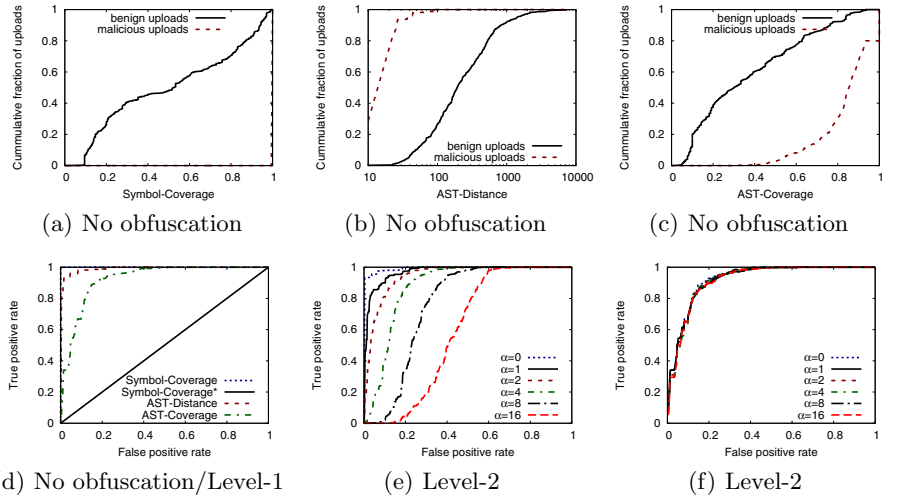
malware sends the device's IMEI and IMSI numbers to a remote host [22] and receives a set of search engine URLs and keywords that are then used to emulate keyword searches and clicks committing various types of click-fraud.

Table 2 presents results for the AST-Coverage algorithm for HongTouTou instances of plagiarism when compared with applications from our Pseudo-Market. We found that all but one application resulted in full AST-Coverage. A full coverage means that all the methods in an original application are covered by the given plagiarized version. The only exception was "*yohandcar*" (malware) which covered 99.2% of the methods of "*Yo Handcar: Off the Rails*" (goodware). We manually verified the reverse-engineered code of both applications to confirm that the malware was indeed mimicking the functionality of the legitimate application. We also found that this malicious instance of HongTouTou not only adds new methods to the original version, but also changes some existing methods in the original version, leading to the slight mismatch.

To show that our detection does not wrongly accuse applications of plagiarism, we removed the legitimate versions of the malware samples from our Pseudo Market and re-ran the AST-Coverage detection. As seen in Table 2, it correctly reported a low coverage of the malware samples with the other applications.

5.2 Accuracy

We perform 500 benign and 500 plagiarized uploads. For a benign upload, we select a random application, remove it from the Pseudo-Market, and then upload it back. For a plagiarized upload, we select a random application, insert malicious code into it, and upload the application back to the Pseudo-Market. Both the original and the plagiarized versions are in the Pseudo-Market. We model insertion of malicious code by selecting a random method from all the methods of all applications, and inserting it into the application. By not inserting specific



(a) No obfuscation (b) No obfuscation (c) No obfuscation

(d) No obfuscation/Level-1 (e) Level-2 (f) Level-2

Fig. 4. (a) CDF of largest coverage of Symbol-Coverage; (b) CDF of smallest distance of AST-Distance; (c) CDF of largest coverage of AST-Coverage; (d) ROC of the accuracy for all schemes; Symbol-Coverage changes from no obfuscation (case w/o asterisk) to Level-1 obfuscation (case with asterisk); (e) ROC of accuracy AST-Distance; (f) ROC of accuracy AST-Coverage

malicious code, but using random code, the results approximate the performance in the presence of arbitrary types of malicious code.

We show CDFs of the coverage value for Symbol-Coverage in Figure 4(a), the plagiarized uploads are not obfuscated. Symbol-Coverage distinguishes all plagiarized applications from correct applications since all malicious uploads have 1.0 coverage and no benign uploads had 1.0 coverage. Thus, there are no false positives in this case.

Figures 4(b) and 4(c) show the CDF distance and coverage for the AST-Distance and AST-Coverage, when the plagiarized uploads are not obfuscated. The two schemes cannot perfectly distinguish benign uploads from plagiarized uploads based on a single threshold value. In both cases, there is some fraction of benign uploads that overlaps with plagiarized uploads. For AST-Distance, the plagiarized upload distance to the original application is quite small (note the log-scale of the x-axis) in relation to the closest distance of some benign uploads, but there is a portion of benign applications that are close to some other benign application in terms of AST-Distance. For AST-Coverage, the plagiarized uploads have methods where the AST fingerprint is identical to many methods of various applications due to shared libraries, and the methods are not always matched to the methods of the original application that was plagiarized.

Each of our detection techniques can distinguish, with high accuracy, a malicious versus benign upload given the coverage or distance of the uploaded application to the next closest application in the market. The exact accuracy for each is shown in the ROC curve of Figure 4(d) which is created by plotting TPR and

FPR for each possible threshold value of Figures 4(a), 4(b), and 4(c). Although AST-Distance and AST-Coverage have lower accuracy when no obfuscation is used, these two schemes are more resilient to obfuscated uploads.

5.3 Obfuscation Resilience

We now evaluate the robustness of our schemes to Level-1 (replace method and class names with random names) and Level-2 (add α random methods) obfuscation. While more advanced obfuscations have been proposed in the research community [15, 16], their applicability to mobile applications remains unknown due to the specific byte-code format and the very tight resource constraints.

Figure 4(d) shows results for no obfuscation and Level-1 obfuscation. Symbol-Coverage's performance degrades substantially when Level-1 obfuscation is used, with an accuracy equivalent to guessing instead of a perfect accuracy. This is because the algorithm relies completely on values inside the symbol table which are obfuscated under Level-1 obfuscation. The accuracy of AST-Distance and AST-Coverage remain the same since they do not use any symbol table information. AST-Distance is the most effective under Level-1 obfuscation.

Figures 4(e) and 4(f) show the results of AST-Distance and AST-Coverage under Level-2 obfuscation. We increase α to show that the accuracy of AST-Distance degrades significantly while the accuracy of AST-Coverage does not change significantly. No threshold exists for AST-Distance that can distinguish between the distance of random methods added and the distance between a legitimate application and all other applications in the Pseudo-Market. AST-Coverage is robust to this problem because it relies on coverage of each application in the Pseudo-Market instead of distance to each application, so a larger size does not affect coverage of other applications as much.

5.4 Computational Feasibility

We evaluate the execution time of our schemes on a Quad-Core AMD Opteron(tm) Processor 2380 machine of 2.50 GHz with 16 GB of RAM that represents a typical commodity server used in data centers such as Amazon EC2. We use synthetic datasets with sizes of 250K, 500K, 750K, 1000K, 1250K and 1500K applications, created by randomly selected applications from our dataset of the real 7,600 binaries. For the Symbol-Coverage algorithm we store symbols in a B+ tree of a MySQL database. For AST-Distance and AST-Coverage we utilize BDD-trees [23] which is a data structure for performing k-nearest neighbor searches on high-dimensional vectors. Figure 5 shows that our schemes scales to millions of applications. The AST-Distance and AST-Coverage differ in performance because the AST-Coverage must search over many more vectors.

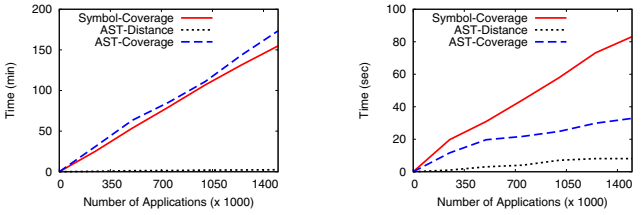


Fig. 5. (a) Preparation time for inserting and indexing the symbols inside the MySQL database, (b) Query time to fetch all symbols from the database. Notice that after indexing, the query time is less than 0.3 seconds per symbol

6 Related Work

Our work builds upon work on clone detection [24, 25] that determines the existence of duplicated code fragments in large enterprise source code bases. Deckard [24], a state-of-the-art tree-based approach, extracts characteristics vectors from parse trees by counting *q-level* binary subtree patterns. Nguyen et al. [21] improve upon this approach by efficiently capturing more structural characteristics. Compared to these techniques, ours handles Dalvik byte-code. Our feature extraction method is also different.

Our work is also related to algorithms comparing program versions, such as a program and its obfuscated version [26–30]. These algorithms perform program differencing at various levels: control flow graph level [26, 28], procedure level [27], and statement level [29, 30]. These approaches require source-code, whereas our approach works on byte-code. They are far more computationally demanding than our AST based algorithms that are effectively accurate.

From a client-side defense perspective, there has been significant work in the area of program analysis and access control to protect users against malicious applications. Enck *et al.* [8] describe a framework to detect potentially malicious applications based on permissions requested by Android applications. Nauman *et al.* [9] propose Apex, a policy enforcement framework for Android that allows a user to selectively grant permissions to applications as well as impose constraints on the usage of resources. These solutions must run on the resource-constrained mobile devices, and they rely on appropriate configuration by the user.

7 Conclusion

In this paper we focused on attacks that plagiarize popular smartphone applications to collect sensitive information or obtain monetary profit. We analyze the *meta-information* of 158,000 applications from the Android Market and find that 29.4% of the applications are more likely to be plagiarized. We proposed three schemes that rely on method-level AST fingerprints to detect plagiarized applications under different levels of obfuscation used by the attacker. Our analysis of 7,600 smartphone application *binaries* shows that our schemes detect all

instances of plagiarism from a set of real-world malware incidents with 0.5% false positives and scale to millions of applications using only commodity servers.

References

1. Kerris, N., Neumayr, T.: Apple App Store Downloads Top Two Billion (2009)
2. Chu, E.: Android Market: A User-driven Content Distribution System (2008)
3. Animal Rights Protesters use Mobile Means for their Message, http://goo.gl/An7Rp
4. Warning on Possible Android Mobile Trojans, http://goo.gl/A8Ow9
5. Lookout Anti-Virus, https://www.mylookout.com/
6. Norton Mobile Security, http://us.norton.com/mobile-security/
7. Bitdefender Mobile Security, http://m.bitdefender.com/
8. Enck, W., Gilbert, P., Chun, B., Cox, L., Jung, J., McDaniel, P., Sheth, A.: TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: OSDI (2010)
9. Nauman, M., Khan, S., Zhang, X.: Apex: Extending Android Permission Model with user-defined runtime constraints. In: ICCS (2010)
10. Jakobsson, M., Johansson, K.: Retroactive detection of malware with applications to mobile platforms. In: HotSec (2010)
11. Google Android, http://code.google.com/android
12. Dalvik Virtual Machine, http://www.dalvikvm.com
13. Google Android SDK, http://developer.android.com/sdk/
14. Lafortune, E., et al.: ProGuard (2004), http://proguard.sourceforge.net
15. Linn, C., Debray, S.K.: Obfuscation of executable code to improve resistance to static disassembly. In: CCS (2003)
16. Collberg, C.S., Thomborson, C.D.: Watermarking, Tamper-Proofing, and Obfuscation-Tools for Software Protection. In: IEEE TSE (2002)
17. Felt, A., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. Technical Report UCB/EECS-2011-48, University of California, Berkeley, Tech. Rep. (2011)
18. Shneiderman, B.: Treemaps for space-constrained visualization of hierarchies. In: ACM TOG (1998)
19. de-Dexer, http://dedexer.sourceforge.net
20. dex2jar, http://code.google.com/p/dex2jar/
21. Nguyen, H., Nguyen, T., Pham, N., Al-Kofahi, J., Nguyen, T.: Accurate and Efficient Structural Characteristic Feature Extraction for Clone Detection. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 440–455. Springer, Heidelberg (2009)
22. Lookout Security Blog, http://goo.gl/q9sI8
23. Arya, S., Mount, D., Netanyahu, N., Silverman, R., Wu, A.: An optimal algorithm for approximate nearest neighbor search in fixed dimensions. JACM (1998)
24. Jiang, L., Misherghi, G., Su, Z., Glondu, S.: Deckard: Scalable and accurate tree-based detection of code clones. In: ICSE. IEEE Computer Society (2007)
25. Li, Z., Lu, S., Myagmar, S., Zhou, Y.: CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. In: IEEE TSE (2006)
26. Apiwattanapong, T., Orso, A., Harrold, M.: A Differencing Algorithm for Object-Oriented Programs. In: ASE (2004)
27. Jackson, D., Ladd, D.: Semantic Diff: A Tool for Summarizing the Effects of Modifications. In: ICSM (1994)
28. Laski, J., Szermer, W.: Identification of Program Modifications and its Applications to Software Maintenance. In: ICSM (1992)

29. Aiken, A., et al.: Moss: System for detecting software plagiarism, http://www.cs.berkeley.edu/aiken/moss.html
30. Komondoor, R., Horwitz, S.: Semantics-Preserving Procedure Extraction. In: POPL (2000)

Appendix: Feature Extraction and Detection Techniques

Algorithm 1 shows our feature extraction alogorithm which is performed on a forest of abstract syntax trees (as shown in Figure 3c). The pairs of children of VIRTUAL and DIRECT nodes are used to update the horizontal features, and a depth first traversal is used to find all paths to update vertical features.

Algorithms 2, 3, and 4 show the details of our detection techniques. Symbol Coverage, Algorithm 2, compares the coverage by the submitted application with each application existing in the repository. AST Distance, Algorithm 3, compares the feature vector distance of a submitted application with every application in the repository. AST Coverage, Algorithm 4, compares the coverage of methods by the submitted application with each application in the repository, and methods are compared by the distance between their feature vectors. In each algorithm, the closest matched algorithm in the repository is compared with a threshold to determine whether the new application is a plagiarized version.

Algorithm 1. Feature Extraction

let G be a forest of the method ASTs
for each each node v in G **do**
  traverse($v, \{v\}$)
for each VIRTUAL or DIRECT node v in G **do**
  for each each child pairs (u, w) of node v **do**
    update_horizontal_feature($\{u, w\}$)

procedure traverse(v, p) **do**
  update_vertical_feature(p)
  for each child u of node v **do**
    traverse($u, p + \{u\}$)
end procedure

Algorithm 2. Symbol-Coverage:

Initialize numbers $c_1, c_2, ..., c_n, t_1, t_2, ..., t_n$ to zero
for all $i \in \{1, 2, ..., n\}$ **do**
  $shared\_classes = Classes[A] \cap Classes[A_i]$
  $t_i := len(Classes[A_i])$
  $c_i := len(shared\_classes)$
  if $len(shared\_classes) > 0$ **then**
    for all $x \in shared\_classes$ **do**
      $shared\_methods = Methods[A][x] \cap Methods[A_i][x]$
      $t_i := t_i + len(Methods[A_i][x])$
      $c_i := c_i + len(shared\_methods)$
$j :=$ argmax$(\frac{c_i}{t_i})$
$p := \frac{c_j}{t_j}$
if $p >$ Threshold **then**
  Alarm(A_j)

Algorithm 3. AST-Distance

$\mathbf{x} :=$ Extract_AST_Feature_Vector(A)
for all $i \in \{1, 2, ..., n\}$ **do**
  $\mathbf{y} :=$ Extract_AST_Feature_Vector(A_i)
  $d_i := ||\mathbf{x} - \mathbf{y}||$
$j :=$ argmin(d_i)
$d := d_j$
if $d <$ Threshold **then**
  Alarm(A_j)

Algorithm 4. AST-Coverage

Initialize set Z to be empty
for all $i \in 1, 2, ..., n$ **do**
  for all $y \in$ Extract_Methods(A_i) **do**
    $\mathbf{y} :=$ Extract_AST_Feature_Vector(y)
    $Z := Z \cup \mathbf{y}$
for all $x \in$ Extract_Methods(A) **do**
  $\mathbf{x} :=$ Extract_AST_Feature_Vector(x)
  $Y :=$ Nearest_Neighbors($k, Z, \mathbf{x}$)
  for all $\mathbf{y} \in Y$ **do**
    $A_i :=$ Get_Application($\mathbf{y}$)
    Update_Coverage_Information($A_i, \mathbf{y}$)
for all $i \in 1, 2, ..., n$ **do**
  $c_i := Count\_Covered\_Methods(A_i)$
  $t_i := Number\_Of\_Methods(A_i)$
$j :=$ argmax$(\frac{c_i}{t_i})$
$p := \frac{c_j}{t_j}$
if $p >$ Threshold **then**
  Alarm(A_j)