

Received December 27, 2017, accepted February 3, 2018, date of publication February 9, 2018, date of current version March 15, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2804321

CodeTracker: A Lightweight Approach to Track and Protect Authorization Codes in SMS Messages

JINKU LI¹, YANGTIAN YE¹, YAJIN ZHOU^{2,3}, AND JIANFENG MA¹

¹School of Cyber Engineering, Xidian University, Xi'an 710071, China

²Cyber Security Research Institute, Zhejiang University, Hangzhou 310027, China

³College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China

Corresponding author: Jinku Li (jkli@xidian.edu.cn)

This work was supported in part by the Natural Science Basic Research Plan in Shaanxi Province of China under Grant 2015JM6351, in part by the Key Program of the National Natural Science Foundation of China under Grant U1405255, and in part by the Shaanxi Science and Technology Coordination and Innovation Project under Grant 2016TZC-G-6-3.

ABSTRACT Short message service (SMS) authorization codes play an important role in the application ecosystem, as a number of transactions (e.g., personal identification and online banking) require users to provide a code for authorization purposes. However, authorization codes in SMS messages can be stolen and forwarded by attackers, which introduces serious security concerns. In this paper, we propose CodeTracker, a lightweight approach to track and protect SMS authorization codes. Specifically, we leverage the taint tracking technique to mark the authorization code with taint tags at the origin of the incoming SMS messages (taint sources), and then, we propagate the tags in the system. To this end, we modify the related array structure, array operations, string operations, inter-process communication mechanism, and file operations for secondary storage of SMS authorization codes to ensure that the taint tags cannot be removed. When the authorization code is sent out via either SMS messages or network connections (taint sinks), we extract the taint tag of the data and enforce pre-defined security policies to prevent the code from being leaked. We have developed a prototype of CodeTracker on Android's ART virtual machine and used 1, 218 SMS-stealing Android malware samples to evaluate the system. The evaluation results show that CodeTracker can effectively track and protect SMS authorization codes with a small performance overhead (<2% on average).

INDEX TERMS Data privacy, tags, Android, short message service (SMS) authorization codes.

I. INTRODUCTION

Smartphones are widely used in our daily life. Increasingly more users leverage smartphones for online transactions, bank transfers and other operations. Simultaneously, increasingly more websites and applications (apps for short) leverage codes delivered via short message service (SMS) messages to authorize users. We call this type of code an authorization code in this paper. For instance, an SMS authorization code can be required when users log into a banking application or reset their passwords. Leveraging SMS codes for authorization is convenient; however, it may present security concerns. If the code is stolen by attackers, it can cause financial losses to users.

On the other hand, SMS-stealing malware is emerging [1], [2]. A research report from the Qihoo 360 company [3] revealed that 6.1% of mobile malware is

stealing information. Among these information-stealing malware samples, 67.4% of them are targeting SMS messages. A research paper [4] noted that among the 49 malware families, 27 of them are harvesting user information, including user accounts and short messages. To this end, there is an urgent need to protect the SMS authorization codes in smartphones.

Before Android version 4.4 (KitKat) [5], malicious apps could intercept SMS messages to retrieve authorization codes and then block the SMS broadcasting stealthily without informing users. However, starting with Android version 4.4, the SMS mechanism has been changed. Malicious apps are unable to block SMS broadcasting, and the system SMS app will get the SMS messages. However, malicious apps can still steal SMS messages by registering a broadcast receiver that listens to certain system events or requesting

the `READ_SMS` permission to retrieve SMS messages from the database.

We noted that a number of systems have been proposed to protect SMS authorization codes. For instance, TISSA [6] can provide null or bogus values instead of real data, which avoids data leakage (including SMS authorization codes). However, TISSA is currently implemented on legacy Android's Dalvik runtime and not the newly designed ART runtime. SecureSMS [7] is another system used to protect SMS messages by changing the Android framework. In particular, when an SMS message arrives, SecureSMS searches the message text. If a pre-defined keyword is found in the message, it adjusts the apps' receiving sequence of text messages in the system so that the default SMS app can get the text message first. Then, it stops the SMS broadcasting to prevent malicious apps from getting the message. This system works but may cause compatibility issues in some benign apps that rely on received text messages. In addition, starting with Android version 4.4, the SMS broadcasting mechanism has been changed, and the new unordered broadcasting cannot be blocked.

From another perspective, because SMS authorization codes are a type of sensitive data in smartphones, they can be protected with the well-known taint tracking technique. TaintDroid [8] is such a system for real-time privacy monitoring that can be used to protect authorization codes. However, TaintDroid is implemented on the Dalvik virtual machine under Android version 4.4 and has not been applicable for the newly introduced ART runtime since Android 4.4. TaintART [9] implements a practical multi-level information-flow tracking system on Android's ART virtual machine and can be used to track and protect private data. However, its extensibility is an issue because the bit length of a register (32 bits) for taint indication is limited. ARTist [10] is a system in Android that tracks private data by instrumenting apps using a customized dex2oat tool. ARTist is an excellent system; however, it only works for intra-application tracking and lacks the inter-application tracking that is necessary for SMS authorization code protection.

In this paper, we propose CodeTracker, a lightweight approach to track and protect SMS authorization codes in Android SMS messages. Specifically, CodeTracker adds taint tags to mark the authorization code at the very beginning of the incoming SMS messages, and it modifies the related array structure, array operations, string operations, IPC (Inter-Process Communication) mechanism, and file operations for the secondary storage of SMS authorization codes to ensure that the tags cannot be removed. Finally, when the authorization code is sent out (via either SMS or the network), CodeTracker extracts the tag of the data and checks with pre-defined security policies. By doing so, it prevents authorization codes from being stolen by attackers.

We have developed a prototype of CodeTracker on an Android system with the ART runtime, and we collected 1,218 state-of-the-art SMS-stealing Android malware samples to evaluate the system. The evaluation results show that

CodeTracker can track and protect SMS authorization codes from being stolen. We further analyzed the remote server addresses where the stolen authorization codes are forwarded to, and we found that 87.66% of them are located in China.¹ This may be due to the popularity of third-party app stores in China with less regulation. The evaluation of the performance overhead shows that CodeTracker incurs a small overhead (<2% on average).

In summary, this paper makes the following contributions:

- We propose a lightweight approach with data-flow tracking to protect SMS authorization codes in Android smartphones, called CodeTracker.
- We have implemented a prototype of CodeTracker in the Android ART runtime. CodeTracker adds taint tags to the SMS authorization code at the very beginning of the incoming SMS messages and ensures that the tags cannot be removed when propagating through the system. When the authorization code is sent out, CodeTracker protects the code by enforcing pre-defined security policies.
- We have evaluated our system with a collection of 1,218 malware samples. The evaluation results demonstrate the effectiveness and low performance overhead of our system.

The remainder of the paper is structured as follows. We first introduce the necessary background information in Section II. We then present the motivation examples and provide a case study of existing SMS-stealing malware samples in Section III. We illustrate the design, implementation, and evaluation of our system in Section IV, Section V, and Section VI, respectively. We discuss the limitations and potential improvements to our system in Section VII, and the related work is presented in Section VIII. Finally, we conclude the paper in Section IX.

II. BACKGROUND

In this section, we will briefly introduce the key concepts of the Android SMS system, as well as the Android runtime environment, to provide necessary background information for our proposed approach.

A. ANDROID SMS SYSTEM

In Android, when receiving a text message, the system sends the message from the RIL (Radio Layer Interface) layer to the framework layer. The framework layer then packs the text message into an SMS PDU and sends a broadcast indicating the receiving of an SMS message. All apps with the `RECEIVE_SMS` permission will receive the broadcast along with the SMS message if they have registered the `SMS_RECEIVED_ACTION` action.

Before Android version 4.4, SMS broadcasting was ordered, and apps with higher priority (declared by apps in the manifest file) could access SMS messages first and then

¹To engage the community, we will release the destination addresses where the samples are forwarding stolen SMS authorization codes to.

discard the messages, which makes apps with low priority unreachable to the SMS messages. This mechanism has been abused by malware to intercept SMS messages [4]. In addition, if a malicious app has the permissions (READ_SMS or WRITE_SMS) to directly operate on the SMS database, it could monitor the database continuously. Once an SMS authorization code is received, it could steal the code and then delete it.

Starting with Android version 4.4, the SMS system has been changed. When the system receives a text message, the framework layer encapsulates the text message into an SMS PDU and sends it with two types of broadcasting. One type is ordered broadcasting, i.e., SMS_DELIVER_ACTION, in which only the default SMS app can receive it. In other words, only the default SMS app has the permission to delete and insert the text messages to the SMS database. The other type is unordered broadcasting, i.e., SMS_RECEIVED_ACTION, in which the broadcasting cannot be interrupted, and all apps can receive SMS messages by registering the broadcasting. Due to this difference, malicious apps cannot intercept and delete the received SMS messages, but they still can steal and forward the SMS messages to remote servers.

B. ANDROID RUNTIME ENVIRONMENT

On an Android system, each app is running inside a separated runtime environment and has its own unique running environment. This runtime environment was called the Dalvik runtime in old Android versions and is called the ART runtime in Android versions 5.0 and above.

Dalvik is a register-based virtual machine that will translate a dex file into an odex file with the `dexopt` command and then execute it. To further improve the performance of Android, Google introduced a new Android runtime, i.e., ART (Android Runtime) [11], which adopts the AOT (ahead of time) mechanism. When an Android app is being installed, the ART virtual machine leverages the dex2oat tool to transform the app's dex file into an oat file, which actually compiles the bytecode into native machine code. When the app is running, the machine code will be directly executed, which greatly improves the performance.

The transition from the the Dalvik to ART runtime leads to several challenges to the taint tracking system. For instance, TaintDroid [8] is implemented in Dalvik, which stores the taint tags by applying extra space adjacent to the variables in the stack of the Dalvik virtual machine. In the ART runtime, some of the parameters are stored directly in registers. To support taint tracking in the ART runtime, the method of storing taint tags should be changed accordingly. This is only one challenge, and we will illustrate how to implement taint tracking on the ART runtime in Section V.

III. MOTIVATING EXAMPLES

In this section, we present the results of an initial study of real SMS-stealing malware samples to better motivate our work.

A. SMS-STEALING MALWARE SAMPLES

Most SMS-stealing malware induces users to install it with phishing web links. Some malware masquerades as online banking login interfaces to obtain a user's account information and password and then monitor SMS messages to steal the SMS authorization code. In the following, we will describe several well-known malware families that steal SMS messages.

1) MazarBOT

MazarBot [12] requests a number of permissions, including SEND_SMS, WRITE_SMS, INTERNET, RECEIVE_SMS, and READ_SMS, which are relevant to the network and SMS. Then, it is injected into the infected user's browser application and obtains the user's bank account and password. MaZarBOT also registers the SMS receiving broadcast receiver (`Android.provider.Telephony.SMS_RECEIVED`). When receiving an SMS message, MazarBOT first determines whether the message is a special instruction. If not, it will put the content of the message and the address in an Intent; then, the Intent will be submitted to an `IntentService`. In the `IntentService`, it packs the SMS message and SMS address into a JSON object, and it sends the message content to a special remote server, i.e., `http://pc35h1ptpcwqezgs.onion`, through an HTTP request. After obtaining the user's bank account and password and the SMS authorization code, it can steal the assets in the user's account, e.g., their money.

2) ANDROID.BANKBOT

Android.BankBot [13] is a malicious app that is remotely controlled with a C&C server. When a user opens his bank application, Android.BankBot loads a disguised banking page to cover up the original page. When the user logs into their bank account, the attacker can intercept the user's bank account number, password, and SMS authorization code to bypass the two-factor authentication. This malware family can forge a variety of bank login interfaces and constantly update these fake bank pages through a remote server.

B. INITIAL STUDY OF THE SMS-STEALING MALWARE SAMPLES

1) PERMISSION ANALYSIS

To systematically understand the malicious behaviors of SMS-stealing malware, we have collected 1,218 malicious samples from three websites, i.e., VirusTotal [14], VirusShare [15], and Contagio Mobile [16]. We found that most samples request the permissions related to SMS and the network. Among these 1,218 collected malware samples, 1,015 of them (83.33%) request RECEIVE_SMS permission, 799 (65.60%) of them request READ_SMS permission, 1,020 samples (83.74%) request SEND_SMS permission, and 1,005 samples (82.51%) request INTERNET permission.

2) TWO METHODS TO OBTAIN SMS AUTHORIZATION CODES

We then decompiled these samples to further understand the methods used to steal the SMS authorization code. We found that these samples usually leverage two different methods to steal SMS messages. One is through the SMS broadcast receiver, and the other is through the SMS database monitoring mechanism. Among the 1,218 samples that we collected, 429 of them steal messages through the SMS broadcast receiver, 120 of them steal messages by monitoring the SMS database, and 336 of them steal messages using both mechanisms.

a: SMS BROADCAST RECEIVER

As mentioned in Section II-A, before Android version 4.4, a malicious app could obtain text messages ahead of other apps by setting its priority to a higher value and then it blocks the SMS broadcast. This could prevent other apps (the system SMS app for example) from receiving the SMS messages. In Android versions 4.4 and above, a malicious app can still receive text messages via registering a broadcast receiver, but it cannot block the message broadcast.

b: MONITORING SMS DATABASE

Malicious apps with `READ_SMS` permission are able to monitor the SMS database. When a new SMS message is received and inserted into the SMS database, the monitor will be informed.

3) TWO METHODS TO SEND OUT SMS AUTHORIZATION CODES

We further installed the samples on an LG Nexus 5 phone and captured the running log information. The results show that there are two main methods that the samples use to send out the stolen SMS messages: through SMS messages and through the network interface.

a: SMS FORWARDING

If a malicious app has the `SEND_SMS` permission, it is easy for the app to forward the stolen authorization code through another SMS message by invoking the well-defined system APIs, e.g., `sendTextMessage()`.

b: NETWORK FORWARDING

If a malicious app has the `INTERNET` permission, it can forward the authorization code through the network interface. The possible channels include emails, HTTP requests, and direct TCP/UDP sockets.

IV. SYSTEM DESIGN

A. THREAT MODEL

In this work, the SMS authorization code is the user's private data that need to be protected. Third-party apps installed on the system are not trusted either because they are malicious or vulnerable. These apps can steal SMS authorization codes in smartphones and forward the codes to a remote server. Similar to other works, we trust the underlying

Android framework and the operating system. While the physical security of the devices (including the smartphones and the SIM cards) is out of the scope of this work.

B. OVERALL DESIGN

The goal of our work is to track and protect authorization codes in SMS messages. To achieve this, there are several challenges that need to be addressed. First, we must determine whether a text message contains an authorization code and then mark it with the taint tag. Second, the SMS authorization code could be processed in many locations, e.g., it might be copied or passed to a new variable or be saved to the SMS database. The taint tags need to be reversed and propagated in these scenarios. Third, we need to determine the correct place to enforce pre-defined security policies to ensure that the SMS authorization code cannot be stolen.

To overcome these challenges, we propose a lightweight approach to track and protect SMS authorization codes, called `CodeTracker`. The overall design of `CodeTracker` is shown in Figure 1.

As marked in digital numbers in Figure 1, there are 15 possible steps in the processing of SMS authorization codes. These steps are described as follows: 1, Android receives an SMS message; 2-3, according to the pre-defined rules, the SMS message is marked as a potential SMS authorization code by adding a `t_p` tag; 4, an SMS message with the `t_p` tag is sent to the default system SMS app and the third-party apps that register as an SMS broadcast receiver; 5, the system SMS app inserts the SMS content into the SMS database and adds the `t_p` tag as an extra extended attribute to the SMS database file; 6, a third-party app fetches an SMS message from the SMS database, and the content of the message is marked with the `t_p` tag, which is obtained from the extra extended attribute of the SMS database file; 7, a database taint tag, i.e., `t_d`, which represents that the data are read from the SMS database, is added to the SMS message; 8-9, it determines whether the message contains an authorization code; if so, it adds a `t_a` tag to the SMS message; 10, a third-party app obtains the SMS authorization code data from the SMS database, which contains three taint tags, i.e., `t_a`, `t_d`, and `t_p`. These tags are denoted as `t_a|d|p`; 11-12, the SMS messages are sent through the SMS interface or the network interface; 13-15, it extracts the tags of the data to be sent out and then processes them according to the pre-defined security policies.

C. IDENTIFY THE SMS AUTHORIZATION CODE

To identify an SMS authorization code and then apply the taint tag, our system has to determine whether an SMS message contains an authorization code. First, we need to decide when to identify the authorization code. Note that the Android SMS system mainly obtains SMS messages via SMS broadcasting or by reading from the SMS database. Therefore, we only need to determine whether an SMS message contains an authorization code before the SMS broadcasting and after the message is fetched from the SMS database.

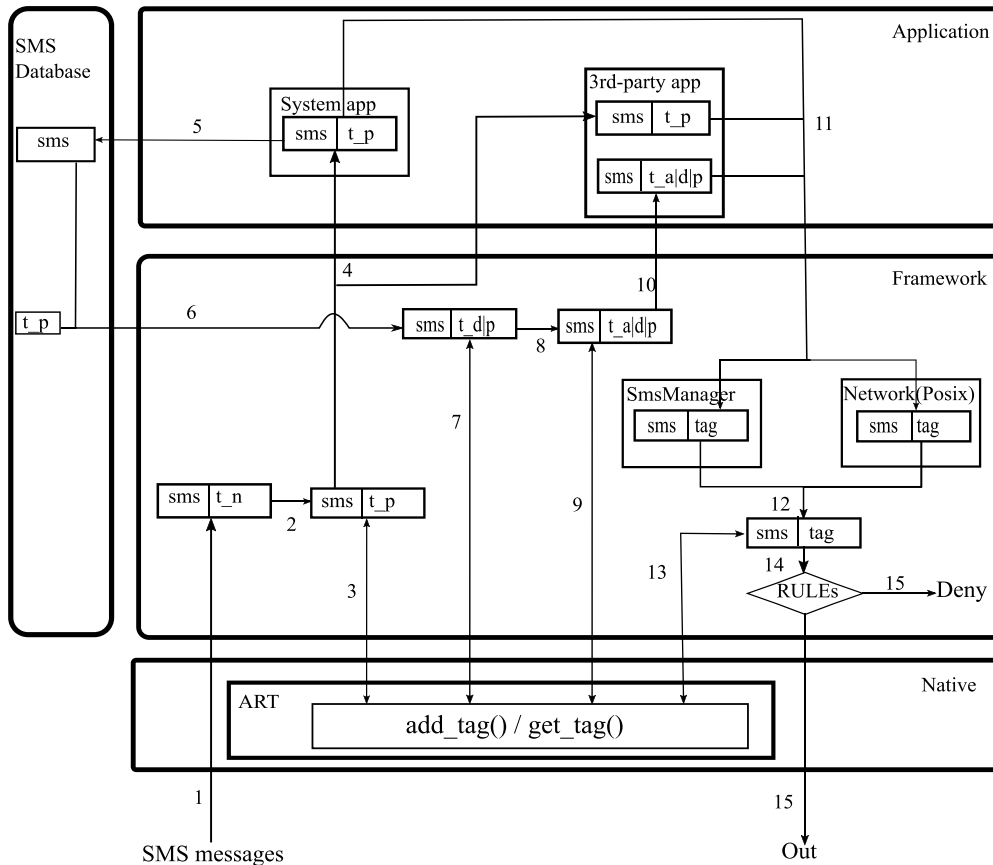


FIGURE 1. The overall design of CodeTracker (*t_n* represents no tag; *t_p* represents a potential tag; *t_{dlp}* represents a database tag and a potential tag; *t_{a/dlp}* represents an authorization code tag, a database tag and a potential tag).

However, because the framework layer of Android will not have decoded the message content before the SMS broadcasting, it is difficult for us to recognize the authorization code by searching the content of the message. Therefore, we leverage the sender address of the SMS message to determine whether the message possibly contains an authorization code; if so, we mark it as a potential SMS authorization code. We maintain a list of sender addresses of SMS authorization codes, and we treat all the SMS messages that originate from these addresses as messages potentially containing SMS authorization codes. After the SMS message can be read from the SMS database, we search the content of the message to obtain the string pattern of the authorization code to determine whether the message contains an authorization code.

After identifying an SMS message that contains an authorization code (or potentially contains such a code), we mark and track the message by adding a tag (or taint tag) to it (the marked message is called a taint source). It is important to note that if we add tags to all the variables in the system, it can better track the data, but the memory overhead will become a concern. We observe that an SMS message is generally stored in a character or byte array; therefore, we only need to add tags in character and byte arrays. In addition, we add one tag for each array to reduce the memory overhead.

The taint tags are defined and applied in the Android native layer; this is transparent to the application and Android framework. We operate on the tags at the application and frame layers only by calling the self-defined API methods in the native layer through JNI. To prevent malicious apps from modifying or deleting taint tags, we only define the methods to add and get tags in the native layer. Moreover, to save space and maintain compatibility, we mark the messages in the SMS database with the same taint tag in the extra extended attributes of the database file.

D. PROPAGATE TAINT TAGS

Ensuring that the taint tags cannot be removed during the internal processing of the system is a challenge. Because the SMS message is saved in an array that is created in the heap, the taint tag will not be removed during general operations, e.g., function calls. However, in the processing of multiple cases, the Android system can lose a tag carried by an array. These cases include (1) IPC, (2) string operations, (3) single element processing in an array, and (4) the secondary storage of the data.

For case (1), when the data are sent between processes, the processing data will be packed into a Parcel object, which might lead to the removal of the taint tag. This is indeed the

case when an SMS message is sent from the framework to the application layer via SMS broadcasting. Therefore, we need to modify the structure of the `Parcel` class. Specifically, when the array data are packed into a `Parcel` object, we extract the taint tag of the array and save it into the `Parcel` object. Consequently, when the data are unpacked from the `Parcel` object, we add the tag stored in the `Parcel` object to the corresponding array object.

For case (2), because many string operations will create a new array by invoking the methods in the native layer, corresponding changes must be made. Specifically, we extract the taint tag of the source array and then add it to the new target array.

For case (3), to save memory, we add a taint tag for an array but not for each element in the array. This could lead to a loss of tags when the data elements in an array are traversed and might be assigned to a new array. Some malicious apps often encrypt and modify text messages byte by byte, resulting in the loss of taint tags. Therefore, we need to instrument the compiler for the ART runtime and the interpreter. In particular, when it executes the instruction of fetching array elements, we save the tag of the array in the current thread instance. Later, when it executes the instruction of storing array elements, we get the tag from the current thread instance and add it to the target array.

For case (4), when a user saves the tainted data into a file, the taint tag in the data could be lost. To prevent this from occurring, we save the taint tag in the file's extra extended attribute. When the data are read from the file, we restore the tag back to the data.

E. ENFORCE SECURITY POLICIES

To prevent the SMS authorization code from being stolen, we enforce the corresponding policies at the endpoints where the code could be sent out (these endpoints are called taint sinks). Our initial study of the malware samples shows that a stolen authorization code can be sent out through SMS messages or the network interface (Section III-B). For the first case, we modify the SMS manager to detect whether an app sends an SMS message, we extract the tag of the data and enforce the security rules. For instance, we could allow or prevent the sending, warn the user, record the target address, etc. For the second case, there are different ways in the Android application layer to send out the data through the network interface. However, all of these ways will eventually call the native layer method through `Posix`. Therefore, we can extract the taint tag of the data and enforce similar security policies at this endpoint.

V. IMPLEMENTATION

We have implemented a prototype of `CodeTracker`. Our system consists of three components: **authorization code identification module, taint tag propagation module, and security policy enforcement module**. In this section, we will illustrate the implementation details of each module.

A. AUTHORIZATION CODE IDENTIFICATION

In the following, we will present how the taint tags are defined and how the authorization code is identified. We also present the way that we apply the taint tags to the authorization code.

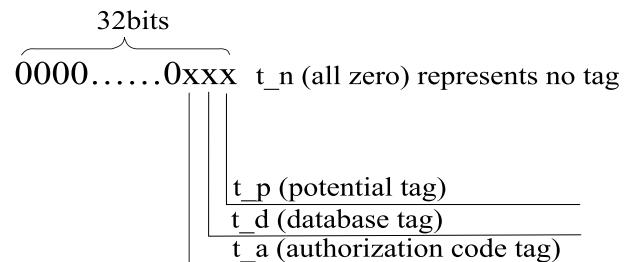


FIGURE 2. The definition of a taint tag.

1) THE DEFINITION OF A TAINT TAG

In our system, each byte or character array contains a taint tag, as defined in Figure 2. A taint tag is a 32-bit integer data, and each bit has a specific meaning. We only define the meaning of the lower three bits, and we leave the remaining bits for future extension. If the data contain a taint tag, the corresponding bit of the tag of the data will be set to 1; otherwise, it will be set to 0. To apply taint tags to the SMS authorization code, we define several different tags. We define the `t_n` tag as `0x00000000`, which represents no tag, and data containing the `t_n` tag denotes that the data contain no taint tags and thus are general data. We define the `t_p` tag as `0x00000001`, which represents the potential tag, and data containing the `t_p` tag are data possibly containing an authorization code. We define the `t_d` tag as `0x00000002`, which represents the database tag, and data containing the `t_d` tag are data that are fetched from the SMS database. We define the `t_a` tag as `0x00000004`, which represents the authorization code tag, and data containing the `t_a` tag are data that contain an authorization code. Note that it is possible for data to contain more than one tag. For instance, some data might contain the `t_d` and `t_a` tags (`t_a|d` for short), which denotes that the data are fetched from the SMS database and contain an authorization code.

2) THE STORAGE OF THE TAINT TAG

The purpose of our system is to protect SMS authorization codes. Notice that an SMS authorization code is stored in a string or a character/byte array; therefore, we only need to add a taint tag to the character and byte arrays (because a string belongs to a special character array). To save memory, there is no need to store tags for other types of data. To this end, we modify the method of the `Array` class in Android. Because the taint tag is 32-bit (i.e., 4 bytes) integer data, the memory space occupied by an `Array` object will increase by 4 bytes. Thus, we modify the `ComputeArraySize()` method of the `Array` class. Specifically, `ComputeArraySize()` obtains the new result by increasing the original calculation result by 4. Each character array or byte array will spend an extra 4 bytes to store its taint tags.

We store the taint tag at the end of an array. Note that the content of an `Array` object is stored in a variable array, called `elements_`. To get the taint tag, the tag's offset in the array must be calculated. We define a static method to calculate the actual memory address of the taint tag; this is easy to do with the array's starting address, the type of the array content and the length of the array as parameters. We could not access the taint tag using the array's subscript, and the taint tag is transparent to the app.

3) THE OPERATIONS OF THE TAINT TAG

In the native layer, we define operations to add or get tags for different types of data objects. These types include byte array, character array, and string. Then, we register these methods as internal methods in the virtual machine. We only define the operations to add or get the taint tags, not operations to delete or update the tags. This is to prevent the app from abusing these operations to remove the tag. We have internal ways that cannot be observed by apps to update or remove the tags.

4) IDENTIFYING AUTHORIZATION CODES BEFORE SMS BROADCASTING

As mentioned in Section IV-C, because the framework layer of Android does not decode the message content before SMS broadcasting, it is difficult for us to determine whether a message contains an authorization code by searching the content of the message. We leverage the source address of the SMS message to determine whether the message possibly contains an authorization code. If it might, we mark the message with a `t_p` tag, which represents a potential tag. To this end, we need to collect the phone numbers that send out SMS authorization codes. We observe that such phone numbers are very different in different countries and regions. For example, in China, the phone numbers that send SMS authorization codes usually start with 95 or 106 or are special numbers. Thus, we maintain a list of source addresses of SMS authorization codes, which currently includes, for example, 106*, 95???, 12306, 10086, 10000, and 10010. Among the list, '106*' stands for phone numbers starting with 106, and the length of such numbers is not fixed but is no longer than 20 bits. Most corporations send SMS authorization codes to their users with '106*'. '95???' stands for phones numbers starting with 95 and have a length of five bits. In China, almost all the banks utilize '95???' phone numbers to send SMS authorization codes. We think that SMS authorization codes mainly come from the addresses in the maintained list, and we can certainly extend this list later.

5) IDENTIFYING AUTHORIZATION CODES AFTER GETTING FROM THE SMS DATABASE

When getting an SMS message from the SMS database, Android will return a `Cursor` object using the `query()` method, and then, it gets the message content with the `getString()` method of the `Cursor` object. After getting the message content, we will add a `t_d` tag (i.e., 0x00000002)

to the string of the message, which represents that the string is fetched from the SMS database. Then, we determine whether the string contains an authorization code by searching the content. Specifically, we search for "authorization" or "password" key words. If either key word is found, we conduct a subsequent search; if the string contains four or more digits or a sub-string of four or more digits or English characters, then we think that the string contains an authorization code and add a `t_a` tag (i.e., 0x00000004) to the message data. As a result, when an SMS message is fetched from the SMS database, the general message will carry a tag as `t_d` or `t_d|p`. It is possible for a message from the SMS database to contain a `t_p` tag; as for the SMS database, all the records in the database share the same tag set that is stored in the extra extended attribute of the database file. If the SMS database contains one record with the potential tag, then all the records fetched from the database will carry such a tag. The goal of such a design is to reduce storage requirements and ensure the compatibility of the database operations. Thus, for the messages fetched from the SMS database, we need to make a further determination. If such a message contains an authorization code, then we mark it with a new tag, i.e., `t_a`. Otherwise, if it is a general message, then we take no further action for its tag.

B. TAINT TAG PROPAGATION

In the following, we will illustrate how the taint tags are propagated in the system.

In Android, array objects are stored in the heap. In addition, the taint tags of the array object might be lost when the object is copied, moved or saved into a file. In our system, we made changes to the string methods, compiler, `Parcel` class for inter-process communication, and methods related to file operations to propagate the taint tag.

1) MODIFICATION TO THE STRING METHODS

In Java, the `String` class contains a private array that contains the actual data. Assigning elements from an old array to a new target array is usually done by the `System.arraycopyTUnChecked()` method in the native layer. Specifically, the `System.arraycopyTUnChecked()` method takes the source array and the target array as parameters. To maintain the taint tag, when a `String` object is copied, we first obtain the source array's taint tag, and then, we add it to the target array, as shown in Figure 3.

2) INSTRUMENTATION TO THE COMPILER

In Android, many apps encode the data in text messages before they send it through the Internet, similar to the URL coding process. The encoding operations might modify a single array element of the source array and copy it to a new array, which can lead to a loss of the taint tag. Therefore, we need to instrument the compiler of the ART virtual machine. In particular, we first append a new field (i.e., `tag_container`) to the `Thread` class to store the taint tag carried with the thread; then, we modify the process of

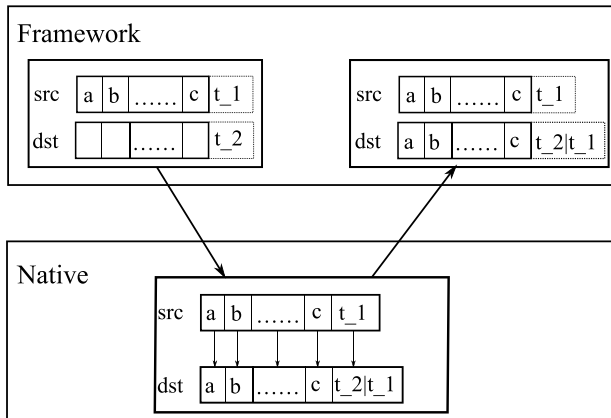


FIGURE 3. Modification to the string methods (*src* represents the source array, and *dst* represents the target array).

array assignment. When this process copies an element of the source array to the target array, we extract the taint tag from the source array and save it into the current thread's *tag_container* field. Later, we get the taint tag from the *tag_container* field of the current thread and add it to the target array.

Array access operations will be converted into *AGET* and *APUT* operations in the dex bytecode and eventually translated into the corresponding machine code by the compiler. To maintain the taint tag, we instrument the compiler for byte- and character-related *AGET* and *APUT* operations. We take the byte-related operations as an example to describe our approach. Note that the corresponding operations for byte are *AGET-BYTE* and *APUT-BYTE*.

When the system takes out an element from a byte array with an *AGET-BYTE* operation, it gets the taint tag of the source array and the taint tag from the *tag_container* field of the current thread. Then, we get the result of the bitwise OR computation for these two tags. Finally, we store the result as a new tag to the *tag_container* field of the current thread. On the other hand, when the system assigns an element to a byte array with an *APUT-BYTE* operation, we first get the taint tag from the *tag_container* field of the current thread and the taint tag of the target array. Then, we get the result of the bitwise OR computation of the two tags. Finally, we store the result as a new tag in the target array. By doing so, we ensure that the taint tag will not be lost when handling a single element of the array.

To better understand the instrumentation to the compiler in Android by our approach, we show the assembly code of the new *AGET-BYTE* operation in Figure 4 and the assembly code of the new *APUT-BYTE* operation in Figure 5.

As shown in Figure 4, the original compiler of the system produces six lines of assembly code for *AGET-BYTE*, i.e., lines 01-03 and 12-14. First, it loads the array length to the *r3* register (line 01); then, it stores the array subscript into memory (lines 02-03). Before loading the array element to the *r5* register (line 14), it judges if the array is out of boundary (line 12); if so, it jumps to an exception handling (line 13).

```

01 ldr r3, [r5, #8]           #Load array length to r3 register
02 movs r2, #1               #Store array subscript to r2 register
03 str r2, [sp, #12]         #Store r2 register to memory
04* movs r0, #1              #Store size of type BYTE to r0
05* mul r12, r3, r0          #Calculate size of the array
06* adds r12, r12, #12       #Calculate offset of the tag
07* adds.w r12, r12, r5      #Calculate memory location of tag
08* ldr.w r1, [r12, #0]      #Load the tag to r1 register
09* ldr.w r12, [r9, #44]     #Load tag of current thread to r12
10* orr.w r1, r1, r12        #Get new tag with a bitwise OR
11* str.w r1, [r9, #44]     #Store new tag to current thread
12 cmp r3, #1                #Is the array out of boundary?
13 bls +14                   #If so, jump to exception
14 ldrsb.w r5, [r5, #13]    #Load array element to r5

```

FIGURE 4. The assembly code of the new *AGET-BYTE* operation.

```

01* ldr.w r2, [r9, #44]      #Load tag of current thread to r2
02* mov r7, r0               #Store array address to r0
03* ldr r3, [r7, #8]        #Load array length to r3
04* movs r0, #1              #Store size of type BYTE to r0
05* mul r12, r3, r0          #Calculate size of the array
06* adds r12, r12, #12       #Calculate offset of the tag
07* adds.w r12, r12, r7     #Calculate memory location of tag
08* ldr.w r1, [r12, #0]     #Load the tag to r1 register
09* orr r2, r1               #Get new tag with a bitwise OR
10* str.w r2, [r12, #0]     #Store new tag in target array
11 ldr r2, [r7, #8]         #Load array length to R2 register
12 cmp r2, #1                #Is the array out of boundary?
13 bls +22                   #If so, jump to exception
14 strb r5, [r7, #13]       #Store r5 register into memory

```

FIGURE 5. The assembly code of the new *APUT-BYTE* operation.

To maintain the taint tag, we instrument the compiler to add 8 instructions (lines 04-11 marked by *). Specifically, it first calculates the memory location of the taint tag of the array (lines 04-07) and loads the tag to the *r1* register (line 08); then, it loads the taint tag saved in the current thread to the *r12* register (line 09) and calculates the new tag by a bitwise OR operation for these two tags (line 10). Finally, it saves the new tag to the current thread for later use in *APUT-BYTE* (line 11).

As shown in Figure 5, the original compiler of the system produces four lines of assembly code for *APUT-BYTE*, i.e., lines 11-14. First, it loads the array length to the *r2* register (line 11), and then, it judges if the array is out of boundary (line 12). If so, it jumps to an exception handling (line 13); otherwise, it stores the value in *r5* into the target array (line 14). To maintain the taint tag, we instrument the compiler to add 10 instructions (lines 01-10 marked by *). Specifically, it first loads the taint tag of the current thread to the *r2* register (line 01). Then, it calculates the memory location of the taint tag of the array (lines 02-07) and loads the tag to the *r1* register (line 08). Finally, it calculates the new tag with a bitwise OR operation for these two tags (line 09) and saves the new tag to the target array (line 10).

Parcel Class for IPC When a text message arrives, Android encapsulates it as a PDU and then broadcasts it to all apps that register as an SMS broadcast receiver. The PDU data are transmitted to the apps via the IPC of `Binder`. The source process `com.Android.phone` packs the data to be sent to other processes into a `Parcel` object and broadcasts them, and the target process unpacks the `Parcel` object after receiving the data. In the original processing, if the source data contain a taint tag, it will be lost when the data are packed. Therefore, to maintain the tag, we append a taint tag field to the `Parcel` class in the framework and native layers to store the taint tag of the packed data contained in the `Parcel` objects. When the array data are packed into a `Parcel` object, the taint tag of the array is extracted and stored into the `Parcel` object; then, it is updated to the corresponding `Parcel` object in the native layer. When unpacking a `Parcel` object in another process to obtain a character or byte array, we add the taint tag saved in the `Parcel` object.

3) FILE OPERATIONS

When an SMS message is stored into the SMS database file, the taint tag carried by the text message will be lost. In Android, because the file operations are performed by calling the native layer methods through the `Posix` class, which is in the framework layer, we modify the operations associated with the `Posix` class for this purpose. Specifically, when a byte array is stored in a file, we first extract the taint tag of the array, and then, we add it to the extra extended attribute of the target file. Later, when we read the data from a file, we first get the taint tag from the file's extra extended attribute and then add it to the corresponding byte array.

C. SECURITY POLICY ENFORCEMENT

The malicious apps could forward the stolen SMS authorization code through `SmsManager` or network interface (taint sinks). Therefore, to catch such behaviors, we need to modify the corresponding interfaces in Android's framework layer and apply corresponding security policies. When it forwards the message through the `SmsManager`, we extract the tag of the data to be sent. When it forwards the data through the network interface, it could be in several ways, e.g., by email, with HTTP request, and with TCP/UDP sockets. However, in any way, the network data will eventually be submitted to the system call of the kernel, which is performed through the `Posix` class. Therefore, we could detect and protect the SMS authorization data by monitoring the network-related operations in the `Posix` class.

1) SECURITY POLICIES

If we get a taint tag from a byte or character array, we may possibly get several values. Among these values, `0x00000000` (i.e., `t_n`) represents that the data do not contain any taint tags; `0x00000001` (i.e., `t_p`) represents that the data potentially contain an authorization code and that the data are directly obtained through SMS broadcasting; `0x00000002` (i.e., `t_d`) and `0x00000003` (i.e., `t_d|p`) represent that the

data are fetched from the SMS database; and `0x00000007` (i.e., `t_a|d|p`) represents that the data are fetched from the SMS database and contain an authorization code. When the value is `0x00000001` or `0x00000007`, we manipulate the data according to our pre-defined rules (e.g., prohibit sending, warn the user, or send a bogus value). It is important to note that if an app sends out data with a tag of `0x00000001` (i.e., `t_p`), we think that it is a dangerous operation. This is because the data are directly obtained through SMS broadcasting, and then, the app is attempting to send it out. This is a malicious action, as a benign app always fetches an SMS message from the SMS database and then sends it out.

VI. EVALUATION

In this section, we evaluate the effectiveness as well as the performance overhead of our prototype system.

A. EFFECTIVENESS

As mentioned in Section III-A, we have collected 1,218 SMS-stealing malware samples from three websites to evaluate our system. The submission times of these samples are from October 2016 to July 2017.

After decompiling these samples, we found that many samples present very similar behaviors. Most of the samples use some confusing or misleading icons and names to fool users, e.g., the bank's name. There are also samples that entice users to install them through the implantation of popular games or pornographic content. Some of the samples are bank-related samples, which may have serious consequences. For instance, these samples are disguised as the user interface of the original bank apps and steal the user's bank account and password as well as the user's SMS authorization code.

We ran our prototype to track and protect the SMS authorization code on an LG Nexus 5 smartphone. To evaluate the effectiveness of our system, we ran each sample every time and sent real authorization code messages through some popular shopping and banking websites to the smartphone. The prototype printed log information about the authorization codes. With this log information, we then retrieved the evaluation results.

The evaluation results show that among the 1,218 malicious samples, 885 of them obtained the messages, and the remaining 333 samples failed to obtain them. Of the 885 samples that obtained messages, 429 of them obtained messages through SMS broadcasting, 120 of them obtained messages by reading the SMS database, and 336 of them obtained messages both through the SMS broadcasting and by reading the SMS database. Concerning subsequent actions, in the 885 samples that obtained messages, 312 of them did not forward the messages, 152 of them forwarded the messages, although we did not get any taint tags for the data, and 421 of them forwarded the messages, but we did obtain the authorization code tag. Among the 421 samples that forwarded messages with taint tags, 212 of them transmitted the data through `SmsManager`, 108 of them transmitted the data through the network interface, and 101 of them transmitted the data

through both the `SmsManager` and the network interface. The distribution map of the 1, 218 SMS-stealing malware samples is shown in Figure 6.

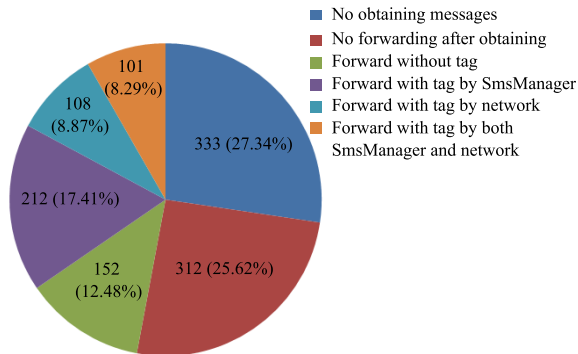


FIGURE 6. The distribution map of 1, 218 SMS-stealing malware samples.

Among the 333 samples that failed to retrieve SMS messages, we found that the main reason for their failure was the expiration of their software license or software errors. For the 312 samples that did not forward the messages after obtaining them, we found that the system did not print any log information about SMS forwarding. We believe that this is reasonable because these samples may need certain conditions to be satisfied to trigger the behavior. How to automatically trigger this malicious behavior remains an ongoing research problem. Among the 152 samples that forwarded messages but where we did not get any taint tags, we checked the log information and found that these samples did not forward the messages successfully. The main reasons for this are that these samples failed to connect to a remote server before sending the authorization code data, the mailbox's password had been changed when sending through email, or the license expired.

In summary, our system effectively captured the taint tags for the 421 samples that actually forwarded the SMS authorization code and blocked such attempts.

In the log information produced by the 1, 218 samples, we also collected 1, 311 target IP addresses and 294 target phone numbers for the stolen authorization codes. We found that the top three target addresses are located in mainland China, USA, and Hong Kong, and 87.66% (1, 407/1, 605) of the target addresses are located in mainland China. The distribution map of target addresses for SMS-stealing samples is shown in Figure 7. We will release the addresses that we collected to the community to improve the detection of such malware.

B. PERFORMANCE OVERHEAD

To measure the performance overhead introduced by `CodeTracker`, we have performed several microbenchmarks, i.e., a compiler microbenchmark, a Java microbenchmark, and an IPC microbenchmark. The evaluation is conducted on an LG Nexus 5 phone with our system.

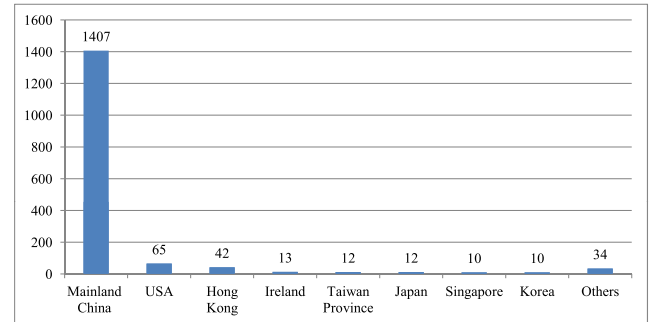


FIGURE 7. The distribution map of target addresses for SMS-stealing samples.

1) COMPILER MICROBENCHMARK

Because `CodeTracker` instruments the ART compiler and inserts the taint logic at compilation time, we measure the size of the oat files and the total compilation time. For the compiler microbenchmark, we select the ten most popular apps in Google play as our evaluation datasets. The ten apps are FreeVPN, GoogleTranslate, Youtube, Instagram, Lantern, Taobao, Twitter, Alipay, GoogleGame, and Tumblr. We compile each app every time with the original compiler and our instrumented compiler and record the size of the produced oat file and the time of compilation. We conduct the benchmarks ten times for each app and calculate the average. Table 1 shows the evaluation results of the compiler microbenchmark. On average, `CodeTracker` introduces an approximate 0.07% overhead with respect to the size of oat files and an approximate 1.79% overhead with respect to the compilation time. Compared to TaintART [9], `CodeTracker` has a better performance as TaintART incurs about 19.9% overhead with respect to the compilation time.

2) JAVA MICROBENCHMARK

Because the Java microbenchmark can accurately reflect the runtime overhead introduced by `CodeTracker`, we choose `CaffeineMark 3.0` [17] to evaluate our system. Our evaluation includes 6 items, i.e., Sieves core, Loop score, Logic score, String score, Float score, and Method score. The results are shown in Figure 8. Among the results, the maximum overhead introduced by `CodeTracker` is 6.92% (String score), and the minimum loss is 0.01% (Sieve score). The average performance overhead incurred by `CodeTracker` with `CaffeineMark` is 1.33%, which is much better than TaintART [9] that introduces about 14%. Our approach is thus a lightweight solution.

3) IPC MICROBENCHMARK

To perform the evaluation of the IPC microbenchmark on `CodeTracker`, we have developed a pair of client/server apps that communicate through `Binder` in Android. Specifically, the client records the time (t_0) before sending a message to the server. The server will also send a message back to the

TABLE 1. Performance overhead with the compiler microbenchmark introduced by CodeTracker.

App Name	Oat File Size (bytes)		Overhead	Compilation Time (s)		Overhead
	Original	CodeTracker		Original	CodeTracker	
FreeVPN	7,815,600	7,819,696	0.05%	1.161	1.18	1.64%
GoogleTranslate	15,946,160	15,962,544	0.10%	2.071	2.102	1.50%
Youtube	43,483,568	43,504,048	0.05%	4.788	4.824	0.75%
Instagram	25,268,656	25,285,040	0.06%	2.961	3.097	4.59%
WhatsApp	24,605,104	24,629,680	0.10%	2.865	2.935	1.58%
Taobao	29,188,528	29,204,912	0.06%	3.096	3.145	1.58%
Twitter	53,821,872	53,858,736	0.07%	6.829	6.885	0.82%
Alipay	102,699,440	102,769,072	0.07%	9.447	9.494	0.50%
GoogleGame	15,688,112	15,692,208	0.03%	1.814	1.890	4.19%
Tumblr	47,669,680	47,722,928	0.11%	5.239	5.276	0.71%
Average:			0.07%			1.79%

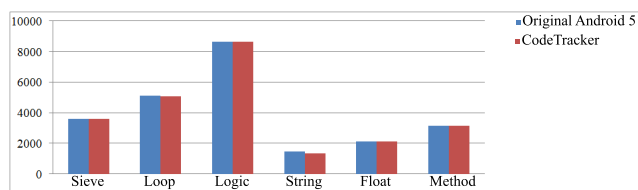


FIGURE 8. Performance overhead with CaffeineMark introduced by CodeTracker.

TABLE 2. Performance overhead with the IPC microbenchmark introduced by CodeTracker.

Microbenchmark Name	Original	CodeTracker	Overhead
Execution Time	4,462ms	4,502ms	0.90 %
Memory (client)	153,580KB	154,590KB	0.66 %
Memory (server)	153,780KB	155,580KB	1.17 %

client after receiving the client’s message. Then, the client records the time (t_{-1}) at which it receives a message from the server. Therefore, ‘ t_1-t_0 ’ represents the elapsed time. We repeat the test ten thousand times and record the execution times. We also calculate the memory usage for the client and server apps during their communication. Table 2 shows the test results. The overhead of the IPC execution time is 0.90%, and the memory usage overheads for the client and server are 0.66% and 1.17%, respectively. In contrast, the overheads introduced by TaintART [9] are about 4.35% for the IPC execution time and 4% for the memory usage, which are both higher than CodeTracker.

VII. DISCUSSION

In this section, we discuss some possible limitations of CodeTracker and potential future work.

A. LIMITATIONS

Many privacy tracking systems [8]–[10] cannot trace implicit data flows [18]; they can only track explicit data flows. Some anti-TaintDroid techniques are mentioned in [19], such as removing taint tags through pixels,

the alphabet, etc. so that they can escape from the tracking of TaintDroid [8].

```

1 public String countToXTrick(String in) {
2     String out = "";
3     for (int i = 0; i < in.length(); i++) {
4         int y = 0;
5         for (int j = 0; j < (int) in.charAt(i); j++) {
6             y = y + 1;
7         }
8         out = out + (char) y;
9     }
10    return out;
11 }

```

FIGURE 9. A code snippet of anti-TaintDroid.

For example, Figure 9 shows a code snippet of anti-TaintDroid [19]. Specifically, the `countToXTrick()` method gets each character in the string variable `in` and assigns it to an integer variable `y` (lines 4-7); then, it appends `y` as a character to the string variable `out` (line 8), which is ultimately returned by the method (line 10). In Figure 9, there is an implicit data flow when it assigns each single element of the source string variable `in` to the variable `y` and then appends `y` to the destination string variable `out`, which overcomes the taint tracking of TaintDroid, as the taint tag of `in` will be lost in the processing. CodeTracker can successfully trace such implicit data flow in Figure 9 with our instrumented `AGET` and `APUT` operations, as described in Section V-B. Thus, CodeTracker can effectively handle certain cases of implicit flows. However, if a malicious app transmits one byte of data every time, then our tracking will not be successful. Fortunately, in our decompilation and testing of samples of malicious apps, no such operations are found.

Additionally, some malicious apps could directly modify the taint tag of the data by invoking the methods in its built-in .so libraries in the native layer and then send out the data. As a result, the malicious app will escape the tracking and protection provided by CodeTracker. To address this problem, we propose a conservative approach. Specifically, when an app gets the content of an authorization code message via the SMS broadcasting or from the SMS database, we save the

PID of the app to a specified local file. Later, before sending the data through the network in the native layer, we determine whether the sending app's PID is in the list of the specified local file; if so, we block the transmission. Although this may cause false positives, it works in most cases. In addition, we can leverage a white list of trusted apps to reduce the false positive rate. Nevertheless, we leave this as one of our future directions.

B. FUTURE WORK

First, CodeTracker is designed for the protection of SMS authorization codes, not for the protection of general text messages. However, in our decompilation process, we found that many malware apps steal general messages. Therefore, in the future, we can easily extend CodeTracker into a prototype system to protect all text messages by applying the taint tags to SMS messages and changing the security policies accordingly. Second, CodeTracker requires changes to the underlying framework; it cannot be transparently supported as a user-level solution. We may leverage a similar technique to ARTist [10] to propose a pure user-level solution in the future.

VIII. RELATED WORK

A. PROTECTION OF SMS MESSAGES

A variety of systems have been designed to prevent SMS messages from being leaked in smartphones. For example, SecureSMS [20] and other similar systems [21]–[23] leverage cryptographic algorithms to encrypt the SMS messages for confidentiality, integrity and authentication services, which is a different goal compared to CodeTracker. SecureSMS [7] attempts to protect SMS messages by adjusting the apps' receiving sequence of text messages in the system so that the default SMS app can get the text message first. Then, it blocks the SMS broadcasting to prevent malicious apps from getting the message. However, SecureSMS only works in Android versions prior to 4.4. Other systems [24]–[27] have also been proposed to prevent phishing messages. Specifically, these systems search the content of SMS messages to find URLs that might link to malicious apps for installation and then block users' dangerous operations. In contrast to these apps, CodeTracker aims to provide protection for authorization codes in SMS messages.

B. STATIC AND DYNAMIC ANALYSIS SYSTEMS

To understand the possibility of privacy leakages, a number of information-flow analysis systems have been proposed by researchers. These systems can be classified into two main types. One type includes static analysis systems that perform analysis on the disassembled codes of apps, including FlowDroid [28], ComDroid [29], AmanDroid [30], DroidForce [31], CHEX [32], etc. However, the limitation of static analysis systems is that they cannot detect runtime information disclosure. Therefore, dynamic analysis systems have

been proposed to track the information flows at runtime in apps. For example, TaintDroid [8] and several extended systems (including DataChest [33], NDroid [34], DroidBox [35], etc.) can enforce taint tracking for real-time privacy monitoring on legacy Dalvik (but not ART) runtime in Android. TaintART [9] and ARTist [10] are two dynamic systems designed for the newly introduced ART runtime in Android and can be leveraged to track and protect sensitive data (including authorization codes) in smartphones. However, as mentioned in Section I, TaintART suffers from the issue of extensibility, and ARTist does not perform well for inter-application tracking, which impede their use for tracking and protecting SMS authorization codes.

Several other dynamic analysis systems targeting malware have also been proposed [36]–[38]. For example, DroidRanger [37] includes a permission-based behavioral footprinting scheme to detect new samples of known Android malware families; it applies a heuristics-based filtering scheme to identify certain inherent behaviors of unknown malicious families. Malton [38] provides a comprehensive view of malware behaviors by conducting multi-layer monitoring and information-flow tracking, as well as efficient path exploration. In contrast, CodeTracker has a different goal of providing tracking for and protection of authorization codes in SMS messages.

C. CONFINEMENT OF SMARTPHONE APPS

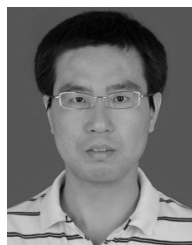
A number of systems have been implemented to limit apps' access to sensitive data. For example, Kirin [39] confines apps by preventing third-party apps from accessing private data. FlaskDroid [40] achieves this goal by hooking Android system services. AppCage [41] leverages two complimentary user-level sandboxes to interpose and regulate an app's access to sensitive APIs. To prevent potential privacy leakage, Aurasium [42], AppGuard [43], TISSA [6], and RetroSkeleton [44] have been proposed to enforce fine-grained access control on sensitive data. All these systems may be able to be leveraged to provide protection for sensitive data (including SMS authorization codes) on legacy runtimes (i.e., Dalvik) in Android, but not on the ART runtime. In contrast, CodeTracker works well on Android's ART runtime and can provide protection as well as tracking for authorization codes in SMS messages.

IX. CONCLUSION

In this paper, we design a dynamic lightweight approach for tracking and protecting authorization codes in Android, called CodeTracker. Specifically, we leverage the taint tracking technique and mark authorization codes with taint tags at the origin of the incoming SMS messages and propagate the tags through the system. Then, we apply security policies at the endpoints where the tainted authorization code is being sent out. The evaluation results on real malware samples demonstrate the effectiveness of our system, and the introduced performance overhead is low ($< 2\%$ on average).

REFERENCES

- [1] *We Steal SMS: An Insight Into Android. KorBanker Operations*. Accessed: Dec. 26, 2017. [Online]. Available: <https://www.fireeye.com/blog/threat-research/2014/09/we-steal-sms-an-insight-into-android-korbanker-operations.html>
- [2] *SophosLabs Report Explores Mobile Security Threat Trends, Reveals Explosive Growth in Android Malware*. Accessed: Dec. 26, 2017. [Online]. Available: <https://news.sophos.com/en-us/2014/02/24/sophoslabs-report-explores-mobile-security-threat-trends-reveals-explosive-growth-in-android-malware/>
- [3] *Special Report on Android Malware in 2016*. Accessed: Dec. 26, 2017. [Online]. Available: <http://zt.360.cn/1101061855.php?dtid=1101061451&did=490301065>
- [4] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy*, May 2012, pp. 95–109.
- [5] *Android 4.4*. Accessed: Dec. 26, 2017. [Online]. Available: <https://developer.android.google.cn/about/versions/kitkat.html>
- [6] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, "Taming information-stealing smartphone applications (on Android)," in *Proc. 4th Int. Conf. Trust Trustworthy Comput.*, 2011, pp. 93–107.
- [7] D. Kim and J. Ryou, "SecureSMS: prevention of SMS interception on Android platform," in *Proc. 8th Int. Conf. Ubiquitous Inf. Manage. Commun.*, 2014, p. 32.
- [8] W. Enck et al., "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, p. 5, Jun. 2014.
- [9] M. Sun, T. Wei, and J. C. S. Lui, "TaintART: A practical multi-level information-flow tracking system for Android runtime," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 331–342.
- [10] M. Backes, S. Bugiel, O. Schranz, P. von Styp-Rekowsky, and S. Weisgerber, "ARTist: The Android runtime instrumentation and security toolkit," in *Proc. IEEE Eur. Symp. Secur. Privacy*, Apr. 2017, pp. 481–495.
- [11] *GoogleIO 2014*. Accessed: Dec. 26, 2017. [Online]. Available: <https://www.google.com/events/io/io14/videos/b750c8da-aebe-c311-b297-00155d5066d7>
- [12] *MazarBOT*. Accessed: Dec. 26, 2017. [Online]. Available: <https://www.tripwire.com/state-of-security/featured/mazarbot-android-malware-distributed-via-sms-spoofing-campaign/>
- [13] *bankbot*. Accessed: Dec. 26, 2017. [Online]. Available: <https://github.com/bemre/bankbot-mazain>
- [14] *VirusTotal*. Accessed: Dec. 26, 2017. [Online]. Available: <https://www.virustotal.com/>
- [15] *VirusShare*. Accessed: Dec. 26, 2017. [Online]. Available: <https://virusshare.com/>
- [16] *Contagio Mobile*. Accessed: Dec. 26, 2017. [Online]. Available: <https://contagiomimidump.blogspot.com/>
- [17] Pendragon Software Corporation. *CaffeineMark 3.0*. Accessed: Dec. 26, 2017. [Online]. Available: <http://www.benchmarkhq.ru/cm30/>
- [18] G. S. Babil, O. Mehani, R. Boreli, and M.-A. Kaafar, "On the effectiveness of dynamic taint analysis for protecting against private information leaks on Android-based devices," in *Proc. Int. Conf. Secur. Cryptogr.*, 2015, pp. 1–8.
- [19] *gsbabil AntiTaintDroid*. Accessed: Dec. 26, 2017. [Online]. Available: <https://github.com/gsbabil/AntiTaintDroid>
- [20] N. Saxena and N. S. Chaudhari, "SecureSMS: A secure SMS protocol for vas and other applications," *J. Syst. Softw.*, vol. 90, pp. 138–150, Apr. 2014.
- [21] A. De Santis, A. Castiglione, G. Cattaneo, M. Cembalo, F. Petagna, and U. F. Petrillo, "An extensible framework for efficient secure SMS," in *Proc. Int. Conf. Complex, Intell. Softw. Intensive Syst.*, 2010, pp. 843–850.
- [22] H. Harb, H. Farahat, and M. Ezz, "SecureSMSPay: Secure SMS mobile payment model," in *Proc. Int. Conf. Anti-Counterfeiting, Secur. Identificat.*, 2008, pp. 11–17.
- [23] G. C. C. F. Pereira et al., "SMScrypto: A lightweight cryptographic framework for secure SMS transmission," *J. Syst. Softw.*, vol. 86, no. 3, pp. 698–706, 2013.
- [24] *LinkScanning*. Accessed: Dec. 26, 2017. [Online]. Available: <https://play.google.com/store/apps/details?id=com.directionsoft.linkscan&feature>
- [25] *S-GUARD*. Accessed: Dec. 26, 2017. [Online]. Available: <https://play.google.com/store/apps/details?id=kr.co.seworks.sguard>
- [26] *AntiSmishing*. Accessed: Dec. 26, 2017. [Online]. Available: <https://play.google.com/store/apps/details?id=com.nprotect.antismiting>
- [27] *T-GUARD*. Accessed: Dec. 26, 2017. [Online]. Available: http://www.tstore.co.kr/userpoc/game/viewProduct.omp?_top=DP000504&dpCatNo=DP04003&insDpCatNo=DP04003&insProdId=0000329718&prodGrdCd=PD004401&stPrePageNm=DP04003&stActionPositionNm=06&stDisplayOrder=1
- [28] S. Arzt et al., "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, 2014, pp. 259–269.
- [29] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proc. Int. Conf. Mobile Syst., Appl., Services*, 2011, pp. 239–252.
- [30] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 1329–1341.
- [31] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden, "DroidForce: Enforcing complex, data-centric, system-wide policies in Android," in *Proc. 9th Int. Conf. Availability, Rel. Secur.*, 2014, pp. 40–49.
- [32] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: Statically vetting Android apps for component hijacking vulnerabilities," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 229–240.
- [33] Y. Zhou, K. Singh, and X. Jiang, "Owner-centric protection of unstructured data on smartphones," in *Proc. 7th Int. Conf. Trust Trustworthy Comput.*, 2014, pp. 55–73.
- [34] C. Qian, X. Luo, Y. Shao, and A. T. Chan, "On tracking information flows through JNI in Android applications," in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2014, pp. 180–191.
- [35] *DroidBox*. Accessed: Dec. 26, 2017. [Online]. Available: <https://github.com/pjlantz/droidbox>
- [36] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: Scalable and accurate zero-day Android malware detection," in *Proc. Int. Conf. Mobile Syst., Appl., Services*, 2012, pp. 281–294.
- [37] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets," in *Proc. Annu. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 50–52.
- [38] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu, "Malton: Towards on-device non-invasive mobile malware analysis for art," in *Proc. USENIX Secur. Symp.*, 2017, pp. 1–19.
- [39] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proc. 16th ACM Conf. Comput. Commun. Secur.*, 2009, pp. 235–245.
- [40] S. Bugiel, S. Heuser, and A. R. Sadeghi, "Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies," in *Proc. USENIX Secur. Symp.*, 2013, pp. 131–146.
- [41] Y. Zhou, K. Patel, L. Wu, Z. Wang, and X. Jiang, "Hybrid user-level sandboxing of third-party Android apps," in *Proc. ACM Symp. Inf. Comput. Commun. Secur.*, 2015, pp. 19–30.
- [42] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical policy enforcement for Android applications," in *Proc. USENIX Secur. Symp.*, 2012, pp. 1–14.
- [43] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. Styp-Rekowsky, "Appguard—enforcing user requirements on Android apps," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2013, pp. 543–548.
- [44] B. Davis and H. Chen, "Retroskeleton: Retrofitting Android apps," in *Proc. Int. Conf. Mobile Syst., Appl., Services*, 2013, pp. 181–192.



JINKU LI received the B.S., M.S., and Ph.D. degrees in computer science from Xi'an Jiaotong University, Xi'an, China, in 1998, 2001, and 2005, respectively. From 2009 to 2011, he was a Post-Doctoral Fellow with the Department of Computer Science, North Carolina State University, Raleigh, NC, USA. He is currently an Associate Professor with the School of Cyber Engineering, Xidian University, Xi'an. His research focuses on system and mobile security.



YANGTIAN YE received the B.S. degree in computer science from Hangzhou Dianzi University, Hangzhou, China, in 2015. He is currently pursuing the M.S. degree with the School of Cyber Engineering, Xidian University, Xi'an, China. His research focuses on mobile security.



YAJIN ZHOU received the Ph.D. degree in computer science from North Carolina State University, Raleigh, NC, USA. He is currently a ZJU 100 Young Professor with the Cyber Security Research Institute and the College of Computer Science and Technology, Zhejiang University, China. His research mainly focuses on smartphone and system security, i.e., identifying real-world threats and building practical solutions.



JIANFENG MA received the Ph.D. degree in computer software and communications engineering from Xidian University, Xi'an, China, in 1995. From 1999 to 2001, he was with the Nanyang Technological University of Singapore, as a Research Fellow. He is currently a Professor with the School of Cyber Engineering, Xidian University. His current research interests focus on information and network security.

...