Towards Privacy-Preserving Malware Detection Systems for Android

Helei Cui*†, Yajin Zhou‡, Cong Wang*†, Qi Li§, and Kui Ren‡
*Department of Computer Science, City University of Hong Kong, Hong Kong, China
†City University of Hong Kong Shenzhen Research Institute, Shenzhen, China
‡College of Computer Science and Technology, Zhejiang University, Hangzhou, China
§Graduate School at Shenzhen, Tsinghua University, Shenzhen, China
cuihelei@outlook.com, {yajin_zhou, kuiren}@zju.edu.cn, congwang@cityu.edu.hk, qi.li@sz.tsinghua.edu.cn

***Abstract*—Android is the primary target for mobile malware. To protect users, phone vendors (e.g., Samsung and Huawei) usually leverage third-party security service providers (e.g., VirusTotal and Qihoo 360) to detect malicious apps in app stores and collect apps' runtime behaviors on users' phones to further spot malware missed in the previous step. However, this practice could cause privacy concerns to phone vendors, users and security service providers. Specifically, phone vendors do not want to share apps (including the paid ones) with security service providers, while the latter do not want to share the malware signatures with the former. Moreover, users do not want to expose apps' runtime behaviors to third parties. These concerns would cause a real dilemma for each involved party. In this paper, we propose a privacy-preserving malware detection system for Android, in which the privacy (or assets) of phone vendors, users, and security service providers are protected. It detects malicious apps in phone vendor's app stores and on users' phones, without directly sharing apps, apps' runtime behaviors, and malware signatures to other parties. We implement a prototype system called PPMDroid and apply several optimizations to save bandwidth and speed up the process. Extensive evaluation results with real malware samples demonstrate the effectiveness and efficiency of our system.**

***Index Terms*—Android, privacy preserving, malware detection.**

I. Introduction

Android is the world's most popular mobile platform. It remains relatively stable at 85% of the worldwide smartphone volume in 2018 [1]. Such success comes from the existence of the application (abbr. app) ecosystem. For instance, the Google Play is the most popular app store available on a large number of Android devices. Besides that, other phone vendors are also keen to establish their own app stores, e.g., Samsung Galaxy Apps and Huawei App Store, mainly for marketing purpose or due to the fact that the Google Play is not available in some regions or countries.

The popularity of Android also has attracted the attention of malware developers. They try to cheat users to download and install malicious apps. What's worse, once published to those app stores, a malware would infect a large number of users in a short period of time. For instance, the recent report showed that more than 50 apps on Google Play were found infected with the Judy malware in May 2017, with nearly 36 million downloads [2]. After infecting users, the malware could "steal users' private information, turn the devices into a remotely controlled botnet, or even cause financial loss to users" [3].

In light of these threats, different approaches have been proposed by both industry and academia. These approaches roughly fall into two categories. The first category consists of app source code analysis systems [4]–[6] based on pre-defined signatures or file similarity to statically detect malware. For instance, the DroidRanger [4] detects malware based on the static behavior footprints. While static malware detection is efficient, it is hard to deal with obfuscated malware samples. This motivates the second category on dynamic detection systems [7]–[9], which are deployed in different contexts but the essence is the same. They monitor and track apps' behaviors running in controlled environments, usually emulators or devices with customized ROMs. One of the representative systems is the Bouncer service [7] introduced by Google. It automatically runs the newly submitted apps, observes their behaviors in a short time period, and removes the malicious ones from the app store.

Despite efforts being made, developing an effective malware detection system to protect users is still challenging for most of the phone vendors, due to the quickly evolving capabilities of malware to evade detection and the lack of expertise in the area of malware detection. Because of this, phone vendors usually cooperate with third-party security service providers [10]. It is known that the accuracy of the static detection system relies on the completeness of signatures or features of malware families. And the effectiveness of the dynamic detection system highly depends on the known malicious runtime behaviors. It has been demonstrated that existing systems could be bypassed [11] due to the incompleteness of monitored behaviors. For instance, previous report showed that the Google Bouncer is leveraging virtual phones (emulators) to run the app and monitor its behaviors. However, due to the limited running time of each app and the differences in behaviors between the virtual phones and real devices, malware could evade analysis via the sandbox (or emulator) detection technique [12]. Thus, this motivates us to leverage *apps' running behaviors on real users' phones* to improve the detection capability [8].

However, such practice could cause privacy concerns to phone vendors, security service providers, and users, and each of them faces a real dilemma. For security service providers to

vet the apps in phone vendors' app stores, these apps (including the paid ones[1]) have to be shared with them. This violates the interest of phone vendors and developers because these apps (especially the paid ones), including underlying code and resource files, would be exposed completely[2], which is actually relinquishing vendors' control over the app's ecosystem. An alternative way for the security service providers is to share the malware signatures, so as to perform the scanning on the side of phone vendors. This protects the apps but leaks the signatures of malware families, which are the most valuable asset of security service providers. From the users' perspective, collecting apps' runtime behaviors on their devices could leak sensitive information and compromise their privacy.

In this paper, we propose a privacy-preserving Android malware detection system. It supports the operations that are needed in existing static and dynamic malware detection systems, while at the same time satisfies the privacy needs of the three involved parties. First, for the static detection, our proposed method leverages a set of features extracted from apps to be scanned rather than the apps themselves to detect the malicious ones. The extracted features are hashed via a cryptographic hash function (e.g., SHA256) and then shared with third-party service providers. Since only features (in hash values) are provided and the vetting process is performed on the side of security service providers, it does not reveal both the code of apps (that is unfeasibly recovered from the shared hash values) and the signatures of malware families (that are always kept private). Second, for the dynamic detection, our method collects apps' runtime behaviors on users' real phones, and search inside a *local* signature database (in *encrypted* form) of malware families from security service providers. During this process, apps' runtime behaviors are not leaked to other parties, and the encrypted malware signature database is never decrypted and thus protected.

It's worth noting that the detection methods of our system are mainly proposed by prior studies [4], [6] and they are not our contributions. However, we adopt them in a privacy-preserving way. To the best of our knowledge, our work is the first privacy-preserving Android malware detection system. In summary, this paper makes the following contributions:

- We analyze the current practice of Android malware detection that involves different parties and shed light on the privacy concerns.
- We propose a privacy-preserving Android malware detection system. It supports detection methods of existing malware detection systems, while at the same time improves the overall privacy of all involved parties.
- We implement a prototype called PPMDroid. The evaluation with real malware samples and formal security analysis demonstrate its accuracy, effectiveness, and security strength.

[1]The fact that the paid apps could be malware may contradict someone's intuition, but it actually happened [13].

[2]One may argue that apps may be eventually exposed since security service providers can use crawlers to download the apps. However, this crawling behavior just demonstrates the value of these apps, otherwise they will not invest resources to actively retrieve these apps. Moreover, the app stores usually deploy anti-crawling technique to prevent this behavior.

II. Background

A. Android Apps and Malware

The security boundary of the Android system is the UID (user ID) based sandbox, in which the apps cannot interact with other ones or perform sensitive operations by default. To access resources, apps have to request the necessary permissions in the app manifest file called AndroidManifest.xml.

The malware also needs to request corresponding permissions to execute malicious payloads, e.g., sending SMS to a premium-rate number. In practice, we can leverage the `SEND_SMS` permission to narrow down apps with this permission and then use particular strings to detect the particular malware samples. Note that to evade detection, malware authors could obfuscate strings in this app. In this case, more features, e.g., API call sequences and dynamic runtime behaviors, can be used to improve the detection accuracy.

B. Preliminary

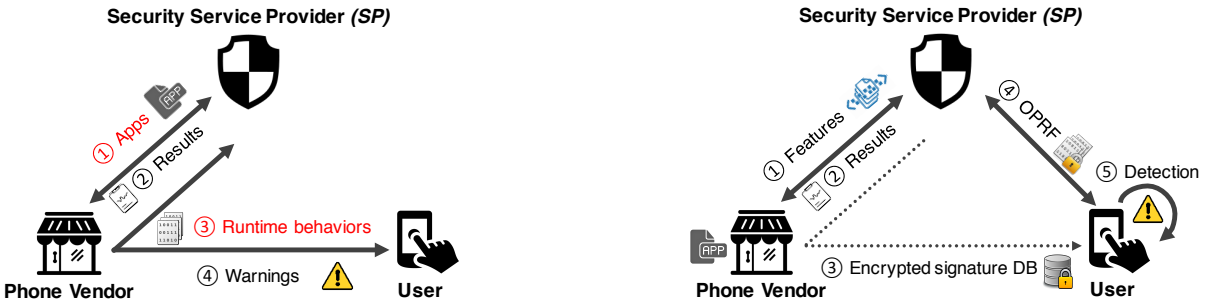
Oblivious Pseudorandom Function (OPRF) An OPRF protocol [14] enables two parties, say $\mathcal{A}$ holding an input x and $\mathcal{B}$ holding a secret key K, to jointly and securely compute a pseudorandom function (PRF) $F(K, x)$ (i.e., $\{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^*$). This two-party computation protocol is operated obliviously in the sense that $\mathcal{A}$ only learns the output value, while $\mathcal{B}$ learns nothing from the interaction. Here, we use a simple OPRF instantiation: $F(K, x) = (H_1(x))^K$, where H_1 is a hash function onto $\mathbb{G} \setminus \{1\}$ ($\mathbb{G}$ is a group of prime order p), and K is randomly selected in $\mathbb{Z}_p^*$. In particular, $\mathcal{A}$ picks a random r in $\mathbb{Z}_p^*$ and sends $a = (H_1(x))^r$ to $\mathcal{B}$. Then $\mathcal{B}$ sends back $b = a^K$. Lastly, $\mathcal{A}$ can obtain $(H_1(x))^K$ via $b^{1/r}$.

III. Problem Statement

A. Overview

Fig. 1 illustrates the flow of Android malware detection before (Fig. 1-(a)) and after (Fig. 1-(b)) adopting our proposed schema. In Fig. 1-(a), to detect malicious apps in phone vendors' app stores, apps have to be shared with the security service providers (abbr. *SP*) since they have more complete malware signatures (①). Then the static detection results are reported to phone vendors (②), and those malicious apps are removed accordingly. To further detect stealthy malware families that may be missed in the previous step, apps' runtime behaviors on users' phones could be collected and analyzed dynamically (③). The result is sent to phone for taking proper actions (④). However, sharing apps (for static detection) and their runtime behaviors (for dynamic detection) with third-parties violates phone vendors' interest and users' privacy.

We present a privacy-preserving way for Android malware detection shown in Fig. 1-(b) to protect apps, malware signature database, and users' privacy. First, to protect the apps' code from being leaked during the static detection procedure, features (in hash values) instead of apps are allowed to be shared with *SP*s (①). Note that the features, even being hashed, could still leak some information such as permissions, invoked functions, and API call sequences. But using these



(a) Without privacy-preserving schema.

(b) With privacy-preserving schema.

Fig. 1. The flow of malware detection before and after adopting our proposed privacy-preserving schema.

limited features cannot directly reverse-engineer the code of the queried app, because the one-way property ensures that it is infeasible to generate those benign yet unique algorithms and resource files from hash values. Later, based on the returned results (②), malicious apps will be removed from the app stores as previous. Second, to protect the malware signatures of *SP*s in our dynamic detection scheme, the signature database is encrypted with private keys only known by themselves (③). In the meanwhile, to protect users' privacy, the captured runtime behaviors will not be sent to the other parties in the cleartext. Users obtain authorized tokens from the *SP* through an OPRF protocol (④) and the dynamic detection is carried on locally, by comparing the tokens derived from the behaviors with the encrypted signature database (⑤).

B. Threat Model and Assumptions

Consistent with prior work in Android malware detection, our design goal is to detect malicious apps inside app stores and on users' phones. Phone vendors have the incentive to do this to protect their users, e.g., for the better marketing purpose. They usually cooperate with *SP*s and leverage their malware signatures for better detection. However, phone vendors should not be able to recover the well-collected malware signatures (because these are the most valuable asset of *SP*s.) From another perspective, vendors are not willing to directly share the apps with *SP*s.

Moreover, in the scenario of detecting malware on users' phones, *SP*s faithfully build the encrypted malware signature database for identifying malicious behaviors and help users to generate valid tokens. But they should not be aware of the runtime behaviors. In addition, we consider that an adversary, pretending to be a valid user, should not be able to decrypt the encrypted signature database.

IV. Static Malware Detection

A. Design Rationale

To detect and remove malware in app stores, phone vendors could cooperate with *SP*s and leverage their malware signatures. However, a dilemma is that vendors do not want to share the apps and *SP*s do not want to share the malware signatures.

In this section, we will demonstrate that two commonly used static detection techniques could be used in a privacy-preserving way. The first detection technique is inspired by DroidRanger [4]. In brief, we leverage the requested permissions and the extracted behavioral footprints, which are semantic-rich information of each app, to detect malware. Second, we compare the similarity of apps to be scanned with existing malware samples. If we find a similar pair of an app and a malware sample, then it's most likely the app is also malicious. In particular, we use the technique proposed in FSquaDRA [6] to calculate the file similarity of two apps.

Our design goal is to allow the *SP*s perform malware detection without holding the apps. Thus, the above-mentioned features, including the permissions, behavioral footprints, and file hashes are necessarily minimal leakage about the apps. The *SP*s cannot (easily) recover the original value thanks to the one-way property of the selected cryptographic hash functions. Even a *SP* may guess the original values using the brute-force attack, the cost of the process and the value of the recovered features do not deserve this attempt. Remember the code of the app is not shared and cannot be recovered at all. The entire process flow is illustrated in Fig. 2.

B. Permission Filter

Android apps have to request permissions during installation (before Android 6.0) or at runtime (since Android 6.0) to better control access to system-wide resources. Depending on the malicious payloads of the malware family, the malicious app has the corresponding essential permissions. For instance, in order to send SMS to the premium-rate number in the background, the malicious app has to request the `SEND_SMS` permission. And to launch the `Exploid` root exploit, the `CHANGE_WIFI_STATE` is needed. As pointed by many prior studies (e.g., [3], [4]), these essential permissions succinctly summarize the wrongdoings of an app and thus can be used to filter out unnecessary malware families to be checked.

Discussion Our permission filter is used for quickly narrowing down a small number of malware family candidates. Thus, the permission information of the queried apps are shared with the *SP*. This does not appear to be harmful because even a user could be aware of such information when installing (or using) an app. We are aware that our system can be augmented by existing advanced primitives (e.g., private set intersection [15]) to achieve the functionality with better privacy protection. However, such integration inevitably incurs extra overhead that would degrade the overall system efficiency.

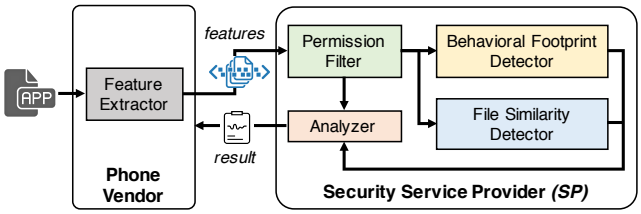


Fig. 2. An overview of our static detection approach: A phone vendor extracts features of apps in its app store and shares these features with the *SP*. The *SP* uses the permissions to quickly filter out unrelated malware families and then leverages two detectors, i.e., the behavior footprint detector and the file similarity detector, to detect whether the app is malicious. After that, the detection results are returned back to the phone vendor.

C. Behavioral Footprint Detector

We then perform a deeper detection based on an app's behaviors. Followed by [4], we use *behavioral footprints* to denote multiple dimensions of malware behaviors. In the current prototype, we use the following categories of behavioral footprints, which can be extended if needed.

- Constant strings: Constant strings in apps can represent some kinds of apps' behaviors. One such example is the destination number of SMS message of the FakePlayer malware family.
- Component names, package names and method signatures: These types of information represents the code syntax of an app, and can also be used to detect malware. For instance, the malicious component name of the DroidKungFu1 malware family is "`com.google.ssearch`" [3].
- API call sequences: API calls can represent the semantics of an app. The app could receive an SMS message through the broadcast receiver and then discard it using the $abort()$ API call. This type of semantic information can also be used to detect malware.

During the detection, the phone vendor extracts the above behavioral footprints of apps and sends the corresponding hash values to the *SP*. To check if the app matches a malware family $MF_i = \{bf_1, \cdots, bf_n\}$, the *SP* uses the signatures (i.e., the hash value of bf_i) to perform string matching with the hash values of the app' footprints $AF_q = \{bf_1, \cdots, bf_m\}$. Different from [4], we involve a weight for each type of behavior in the malware family. Particularly, each behavior footprint in a malware family is now associated with a weight value, i.e., $MF_i = \{(bf_1, \delta_1), \cdots, (bf_n, \delta_n)\}$. If a footprint bf_i in MF_i also belongs to AF_q (i.e., $bf_i \in MF_i \cap AF_q$), then we will add its weight δ_i to a score. Eventually, if the score is larger than a pre-defined threshold, then we consider the app is malicious.

Improvement via Bloom Filter Here, we can apply the *Bloom filter* [16], a space-efficient probabilistic data structure for high-speed set membership tests, to further reduce the bandwidth cost of exchanging the hash values of apps' behavioral footprints. Specifically, the phone vendor builds the Bloom filter BF (i.e., a m-bit array representing a Set of at most n items) at local and sends it (rather than the exact hash values of those behavioral footprints) to the *SP* for checking the behavior footprints and calculating the final score.

By doing so, our system gains two extra benefits. First, from the perspective of bandwidth efficiency, the size of the encoded Bloom filter is far less than the original behavioral footprints (or even the hash values). Second, from the perspective of data privacy, the encoded Bloom filter can reduce, to some extent, the information leakage. Because the *SP* can easily test whether a known item is included, but can hardly recover the exactly encoded items due to the huge behavior space.

D. File Similarity Detector

We now illustrate the way of using file similarity to detect malware. The intuition is that the malware authors could use automatic tools to produce a batch of malware samples, each with minor changes. If we find an app is very similar to a malware sample, then the app is most likely a malicious one.

There are multiple ways to detect the file similarity of two apps. In our prototype, we use the method proposed by the FSquaDRA system [6]. This system compares the file similarity based on the Jaccard similarity of file hash values. In particular, the Jaccard similarity coefficient to evaluate the similarity score is as follows: $jScore(FH_1, FH_2) = \frac{|FH_1 \cap FH_2|}{|FH_1 \cup FH_2|}$, where FH_1 and FH_2 are the sets of file hashes of a queried app and a malware sample. If the score exceeds a similarity threshold, then the app is probably malicious as the sample. As this pair-wise similarity measure could become costly when facing a large dataset, we will further explore other optimizations to speed up this procedure, e.g., adopting the Locality-Sensitive Hashing (LSH) techniques [17].

V. Dynamic Malware Detection

In this section, we elaborate our secure dynamic malware detection design, operated between users' phones and *SP*s.

A. Design Rationale

Unlike the static detection that has known malware samples as templates, our dynamic approach relies on apps' runtime behaviors on users' phones to detect stealthy malware that may be missed in the previous step of static detection.

To achieve this, one option is to collect and send apps' runtime behaviors to a *SP*. Then the *SP* compares the collected behaviors with the behaviors (or signatures) of malware families. However, this violates users' privacy since the apps' behavior may reveal users' private information. Another option is to deploy the malware signatures on the phone, and the comparison is carried on locally. This protects users' privacy, but puts the malware signature database at risk, since this database could be obtained by every user (including the competitors of the *SP*s).

To address this dilemma, we resort to the searchable symmetric encryption (SSE) technique [18], which allows a party to store data (e.g., the malware signature database) at another party in a private manner and later supports keyword search while maintaining privacy. However, directly using SSE does not solve the problem that users want to obtain query tokens from the *SP* while hiding their runtime behaviors. To this end, we add a crucial ingredient to our design, i.e., using OPRF [14]

Algorithm 1 Build Encrypted Signature Database

Input: The private keys of *SP*: $\mathbf{K} = (K_1, K_2)$; the malicious behavior set: $W = \{(w_1, id_1), \cdots, (w_n, id_n)\}$, where w_i is a behavior (string) and id_i is its malware family ID.
Output: Encrypted signature database $\mathcal{D}$.

```
Initialize a hash table D;
for i ← 1 to n do
    s_i ← H_1(w_i), where H_1 : {0,1}* → G;
    t_1 ← (s_i)^{K_1} = F(K_1, w_i);
    t_2 ← (s_i)^{K_2} = F(K_2, w_i);
    D.put(t_1, Enc(t_2, id_i));
end for
```

to jointly compute the valid query tokens without revealing the private keys of the *SP* and the queried behaviors of the user's phone. Then, the detection can be performed on the user's phone, by comparing the authorized tokens derived from the behaviors with the encrypted signature database, and the result will be kept locally.

B. Setup by the SP

First of all, a *SP* should build an *encrypted* malware signature database to compare with the app's runtime behaviors, without leaking the signatures. We apply one of the latest SSE constructions [18], which is implemented by using a generic hash table. There are multiple runtime behaviors that could be leveraged for malware detection, e.g., the behavior to send SMS in the background to some premium-rate numbers [3], executing special system calls [19], sensitive functions executed or access malicious command-and-control (C&C) servers [4], [9]. These behaviors could be expressed in the format of strings. For instance, the behavior of mounting system partition as read-write using the *sys_mount* system call could be denoted as "`syscall:sys_mount`". In the current prototype, we leverage the URLs (in strings) accessed by the app at runtime to detect whether it is malicious. Other behaviors could be supported similarly. In particular, the malware signature database contains a pair of encrypted key (malicious URLs) and value (the malware family ID). After finding a match inside the database, the ID will be obtained, and the corresponding warning information is showed to users.

Algorithm 1 illustrates the detailed construction of the encrypted signature database. We use w to denote the runtime behaviors in general (e.g., malicious URLs in our prototype) and id to denote the malware family ID. The objective is to transform the behavior set (expressed in strings) $W = \{(w_1, id_1), \cdots, (w_n, id_n)\}$ to a set of encrypted key-value pairs so that they can be stored in a hash table. For each behavior w_i, the *SP* first computes the signature s_i via the hash function $H_1 : \{0,1\}^* \to \mathbb{G}$. Then a token pair (t_1, t_2) are generated from s_i: $t_1 \leftarrow (s_i)^{K_1}$ and $t_2 \leftarrow (s_i)^{K_2}$, where K_1 and K_2 are private keys used for PRF $F(k, x) = (H_1(x))^k$. Next, the key-value pair $(t_1, Enc(t_2, id_i))$ will be inserted into a hash table $\mathcal{D}$, where Enc is symmetric encryption algorithm and id_i is the corresponding malware family ID. Accordingly, the actual description of malware family is stored separately on the user's phone and can be read by its ID (id_i).

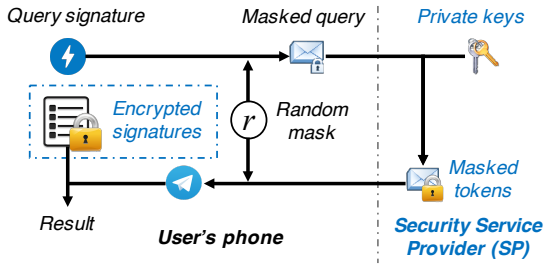


Fig. 3. An illustration of the query flow.

Also, the description information could be retrieved from the *SP*'s remote server after finding a match. After this step, the encrypted malware signature database $\mathcal{D}$ will be distributed to the users' phones for later dynamic detection.

C. Detection on the User's Phone

From a high-level point of view, the detection process is checking the collected runtime behaviors on users' phones with the entries in an encrypted signature database. However, for privacy protection, the collected runtime information should never be leaked to other parties, including the phone vendor and the *SP*. To this end, users' devices generate tokens via an OPRF protocol with the help of the *SP* and then tokens are searched over the encrypted signature database at the local device. Once a match is found, users will be notified of the corresponding malware family information for further action, or the app will be automatically removed. We follow the similar practice adopted by the Google Play Protect [20].

Fig. 3 shows the query flow, and Algorithm 2 illustrates the detailed operations. For each obtained app's behavior w_q, the user first computes the signature s_q via the same hash function H_1. Then it computes the masked signature x via $(s_q)^r$, where r is a random value used for hiding the user's query from the *SP*. Upon receiving x, the *SP* generates the masked token pair (y_1, y_2) with its private keys K_1 and K_2: $y_1 \leftarrow (x)^{K_1}$ and $y_2 \leftarrow (x)^{K_2}$, and returns them back to the user. Next, the user obtains the authorized token pair (t_1, t_2) by unmasking (y_1, y_2) with the random value r. At last, if t_1 matches a key in the $\mathcal{D}$, then the corresponding malware family ID id_q would be decrypted via $Dec(t_2, \mathcal{D}.get(t_1))$. Otherwise, the queried behavior w_q does not refer to a known malware family.

Remark In above design, the user's phone needs to interact with the *SP* to obtain the token. This operation may cause a delay of the main functionality of the app. To mitigate this, we could run the detection process in another thread that would not block the main execution of the app.

VI. Security Analysis

In this section, we provide formal security analysis to rigorously justify the security strength of our system design. First, in the static malware detection, phone vendor is not required to send the apps to the *SP*. The only thing necessary for our proposed detection methods is a set of features rather than the apps. Since only features are provided and the vetting process is performed on the side of *SP*s, it does not reveal both the code of apps and the signatures of malware families. We are

Algorithm 2 Detect Over Encrypted Signature Database

Input: The private keys of *SP*: $\mathbf{K} = (K_1, K_2)$; the encrypted signature database: $\mathcal{D}$; and a queried runtime behavior: w_q, which is collected on the user's phone.

Output: Malware family ID id_q or $\perp$.

User's phone: // *mask the query signature*

$\overline{s_q \leftarrow H_1(w_q)}$, where $H_1 : \{0,1\}^* \rightarrow \mathbb{G}$;

Pick a random $r \xleftarrow{\$} \mathbb{Z}_p^*$;

$x \leftarrow (s_q)^r$;

Send x to the *SP* via a secure channel;

SP: // *generate the query tokens with its private keys*

$y_1 \leftarrow (x)^{K_1}$; $y_2 \leftarrow (x)^{K_2}$;

Send back (y_1, y_2);

User's phone: // *unmask the query tokens and detect*

$\overline{t_1 \leftarrow (y_1)^{1/r}} = (s_q)^{r \cdot K_1 \cdot 1/r} = (s_q)^{K_1}$;

$t_2 \leftarrow (y_2)^{1/r} = (s_q)^{r \cdot K_2 \cdot 1/r} = (s_q)^{K_2}$;

if $\mathcal{D}.get(t_1) == null$ **then**

  Return $\perp$, i.e., w_q is not a malicious behavior;

else

  Return a malware family ID $id_q \leftarrow Dec(t_2, \mathcal{D}.get(t_1))$;

end if

aware that these features, even being hashed, could still leak some information about an app, such as permissions, invoked functions, and API call sequences. Nevertheless, knowing this information cannot directly reverse-engineer the code of the queried app, because the one-way property ensures that it is infeasible to generate those benign yet unique algorithms and resource files from hash values. Thus, apps of the phone vendor are always protected in our proposed design.

Second, in our dynamic detection scheme, the user's runtime behaviors are never leaked to other parties and the final results are never left the user's phone. We follow the security notion of SSE [18], and our dynamic detection scheme achieves security against adaptive chosen-keyword attacks (CKA2) under quantifiable leakage profiles. That is, three leakage functions are defined for the view of the encrypted signature database, the query pattern, and the access pattern, where the query pattern indicates the equality of queried signatures and the access pattern includes the results of queries:

- Since the encrypted signature database $\mathcal{D}$ is kept on the user's phone, the phone knows its capacity and size, which are captured in the leakage function $\mathcal{L}_1$, defined as follows: $\mathcal{L}_1(\mathcal{D}) = (n, (|u|, |v|))$, where n is the number of entries in $\mathcal{D}$, $|u|$ and $|v|$ are the bit lengths of encrypted key-value pairs.
- During the detection procedure, the phone sees the repeated tokens, accessed key-value pairs, and matched malware family IDs ids, which are captured in the query pattern $\mathcal{L}_2$, defined as follows: $\mathcal{L}_2(\{w_i\}_{1 \le i \le q}) = (\mathbf{N}_{q \times q})$, where $\mathbf{N}_{q \times q}$ is a symmetric binary matrix such that for $1 \le i, j \le q$, the element in the i-th row and j-th column is set to 1 if $w_i = w_j$, and 0 otherwise.
- Moreover, the phone also sees the detection results if there are matches for the query tokens $\{w_i\}_{1 \le i \le q}$, which are captured in the access pattern, defined as follows: $\mathcal{L}_3(\{w_i\}_{1 \le i \le q}) = (\{(u,v)_i, id_i\}_{1 \le i \le q})$, where $(u,v)_i$ is accessed key-value pair and id_i is its malware family ID.

Within a polynomial number of adaptive queries, the phone only learns the information defined in leakage functions, no other information about the content of the encrypted signature database. Now, we follow the security framework of [18] and give the simulation-based security definition:

***Definition* 1.** *Given our dynamic detection scheme* Π *with stateful leakage functions* ($\mathcal{L}_1$, $\mathcal{L}_2$, $\mathcal{L}_3$)*, and a probabilistic polynomial time (PPT) adversary* $\mathcal{A}$ *and a PPT simulator* $\mathcal{S}$*, we define the probabilistic games* $\mathbf{Real}_{\Pi,\mathcal{A}}(\lambda)$ *and* $\mathbf{Ideal}_{\Pi,\mathcal{A},\mathcal{S}}(\lambda)$ *as follows:*

$\mathbf{Real}_{\Pi,\mathcal{A}}(\lambda)$: *a challenger* $\mathcal{C}$ *generates private keys* $\mathbf{K} = \{K_1, K_2\}$*. Then* $\mathcal{A}$ *selects a behavior set* $W = \{(w_1, id_1), \cdots, (w_n, id_n)\}$ *and asks* $\mathcal{C}$ *to build the encrypted signature database* $\mathcal{D}$ *via Algorithm 1. Then* $\mathcal{A}$ *adaptively conducts a polynomial number of secure queries with the tokens* $\mathbf{t} = \{(t_1, t_2), \cdots\}$ *generated from* $\mathcal{C}$*. Finally,* $\mathcal{A}$ *returns a bit "1" as the game's output if the detection results are all correct and consistent; otherwise, "0".*

$\mathbf{Ideal}_{\Pi,\mathcal{A},\mathcal{S}}(\lambda)$: $\mathcal{A}$ *selects* W*, and* $\mathcal{S}$ *generates* $\widetilde{\mathcal{D}}$ *based on* $\mathcal{L}_1$*. Then* $\mathcal{A}$ *adaptively conducts a polynomial number of queries. From* $\mathcal{L}_2$ *and* $\mathcal{L}_3$ *of each query,* $\mathcal{S}$ *generates the corresponding* $\widetilde{\mathbf{t}}$*, which are processed over* $\widetilde{\mathcal{D}}$*. Finally,* $\mathcal{A}$ *returns a bit "1" as the game's output if the simulated results are all correct and consistent; otherwise, "0".*

Our proposed scheme Π *is* $(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3)$*-secure against adaptive chosen-keyword attacks (CKA2) if for all PPT adversaries* $\mathcal{A}$*, there exists a PPT simulator* $\mathcal{S}$ *such that* $Pr[\mathbf{Real}_{\Pi,\mathcal{A}}(\lambda) = 1] - Pr[\mathbf{Ideal}_{\Pi,\mathcal{A},\mathcal{S}}(\lambda) = 1] \le negl(\lambda)$*, where* $negl(\lambda)$ *is a negligible function in* λ*.*

***Theorem* 1.** *Our dynamic detection scheme* Π *is* $(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3)$*-secure against adaptive chosen-keyword attacks in the random oracle model if* (Enc, Dec) *is semantically secure, and* F *is secure PRF.*

Proof. Based on $\mathcal{L}_1$, the simulator $\mathcal{S}$ can build a randomized database $\widetilde{\mathcal{D}}$ with the same size as the real encrypted signature database $\mathcal{D}$, containing n key-value pairs. Each simulated key-value pair $(\widetilde{u}, \widetilde{v})$ are random strings with the same lengths as the real one (u, v). Due to the semantic security of symmetric encryption and the pseudorandomness of secure PRF, $\widetilde{\mathcal{D}}$ is computationally indistinguishable from $\mathcal{D}$.

When the first query w_1 is sent, $\mathcal{S}$ randomly generates two strings as the simulated tokens $(\widetilde{t_1}, \widetilde{t_2})$. After that, a random oracle $\mathcal{O}_1$ is operated in the way of $\widetilde{u} = \mathcal{O}_1(\widetilde{t_1})$ to select $(\widetilde{u}, \widetilde{v})$. Then the other random oracle $\mathcal{O}_2$ is operated to get the result (i.e., malware family ID) $id = \mathcal{O}_2(\widetilde{t_2}||\widetilde{v})$. Note that id is exactly the same as the real result indicated in $\mathcal{L}_3$. And the due to the pseudorandomness of secure PRF, $(\widetilde{t_1}, \widetilde{t_2})$ are also computationally indistinguishable from (t_1, t_2). In the subsequent queries, the repeated tokens are recorded by $\mathcal{L}_2$, so $\mathcal{S}$ can directly use the previously simulated ones. Otherwise, $\mathcal{S}$ generates tokens via $\mathcal{O}_1$ and $\mathcal{O}_2$ in the same way as mentioned above. And the results derived from $\widetilde{\mathcal{D}}$ are also identical to the real results. Therefore, $\mathcal{A}$ cannot differentiate the simulated tokens and results from the real tokens and results. □

TABLE I
THE RESULTS OF OPTIMIZATION VIA BLOOM FILTER.

Method	Avg. Bandwidth (KB)	Avg. Detect Time (ms)	Avg. Reconstruct Time (ms)
Baseline	38.89	0.23	**1.06**
Bloom filter	**1.33**	**0.14**	1.82

VII. EXPERIMENTAL EVALUATION

A. Experimental Setup

The static detection functionalities at the *SP* and the searching inside the encrypted malware signature database on the Android device are implemented in Java. Particularly, the static detection is conducted on a Microsoft Azure instance "Standard D8s v3" with 8 vCPU @ 2.3 GHz and 32 GiB of RAM in Linux (Ubuntu Server 16.04 LTS). And the client evaluation is performed on a Huawei Mate 9 Android phone. Regarding the cryptographic primitives, we use Java Cryptography Architecture (JCA) to realize the cryptographic hash function via SHA256, and JPBC library[3] with type "d159" setting to generate group elements and involved hash functions. In addition, we import the Bloom filter[4] for performance optimization. In our experiment, we use the malware samples released by the Android Malware Genome Project [3]. It has 49 malware families with 1,260 samples in total.

B. Evaluation

Accuracy Comparison with Original Malware Detection Recall that we leverage existing malware detection methods and apply them in a privacy-persevering way. To evaluate the effectiveness of our system, we need to answer the following question: *whether our system can achieve the same accuracy as the original malware detection designs?*

To this end, for static malware detection, we apply the same permission filter and feature sets for 49 malware families. It turns out that our system can detect all the samples that were detected by the original system. For the detection based on file similarity, we use 1,260 samples as our sampled dataset, and randomly select 1 to 1,000 apps as our query set. We want to compare the results of similarity comparison of the FSquaDRA system and our system. Note that we set the similarity threshold to 0.85. We then calculate the recall ration, i.e., the fraction of the relevant items that can be successfully retrieved ($\frac{true\_positives}{true\_positives+false\_negatives}$). Here the $true\_positives$ means the samples can be detected by both systems, and the $false\_negatives$ means the samples can be detected by FSquaDRA but missed by ours. In our evaluation, the recall ration achieves 100%, which means our system has the same detection accuracy as FSquaDRA.

Effectiveness of Optimization via Bloom Filter In our prototype, we leverage Bloom filter to optimize the bandwidth consumption when sharing feature sets between phone vendors and *SP*s (see Section IV-C).

[3]JPBC Library: http://gas.dia.unisa.it/projects/jpbc/index.html
[4]Bloom Filter in Java: https://github.com/MagnusS/Java-BloomFilter

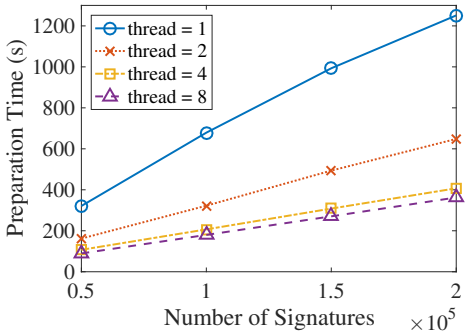


Fig. 4. Evaluation of encrypted signature database preparation.

TABLE II
TIME COST OF OPERATIONS IN DYNAMIC DETECTION.

User's phone (ms)			*SP* (ms)
Mask	Unmask	Query+Dec	Gen Token
36.2	69.1	1.67	5.87

Table I shows the comparison results. The baseline method is to send the hashes of extracted behavioral footprints directly, where each hash value in our experiment is 32 bytes in SHA256. On average, the size of extracted footprint hashes of each app is about 38.89 KB, which is almost 30 times larger than the size of encoded Bloom filter. Specifically, we set the false positive rate ϵ to 0.001, create a Bloom filter for each queried app with the optimal parameters (i.e., the size $m = \lceil -\frac{n \cdot \ln \epsilon}{(\ln 2)^2} \rceil$ and the number of hash functions $k = \lceil -\frac{\ln \epsilon}{\ln 2} \rceil$ [16] are determined based on the actual number of hash values), and then serialize the encoded filter into a binary file. Before performing the detection at the *SP*, it needs a little more time to reconstruct (i.e., deserialization) the filter. Nevertheless, the average detection speed is still very fast, say less than 1 ms for each malware family. Note that applying Bloom filter keeps the same accuracy as the original one.

Efficiency of Dynamic Detection In our prototype of the dynamic malware detection, the *SP* needs to prepare the encrypted signature database $\mathcal{D}$ periodically. Fig. 4 demonstrates the setup time of preparing 50,000 to 200,000 signatures (i.e., malicious URLs with malware family IDs). When preparing these encrypted key-value pairs, we use multiple threads with the thread number from 1 to 8. This number could be increased to further accelerate this process. We note that this is a one-time cost for each batch of signatures, and is inevitable for the purpose of security.

In our test, the size of the encrypted malicious signature database with $100,000$ entries (derived from URLs) is around 1.9 MB, which is acceptable to be stored on users' phones.

Besides that, we also test the major cryptographic operations required during the detection on a real Android phone, as shown in Table II. We see that the $mask$ and $unmask$ are quite efficient, but still relatively slower than the query and decryption due to the underlying exponentiation operations. Note that the server (*SP*) can perform the token generation

very fast, about 6 ms for each signature. In addition, for the bandwidth cost in the OPRF protocol, each queried signature only requires three group elements in $\mathbb{G}$, i.e., x and (y_1, y_2), where each element is 40 byte in our setting. As a result, the OPRF protocol does not introduce much overhead in terms of both computation resources and bandwidth consumption.

VIII. Related Work

Android Malware Detection In the literature, many static Android malware detection systems have been proposed over the past years. For example, DroidRanger [4] uses permissions to filter out unrelated apps and then leverages behavior footprint to detect new samples of known malware families. FSquaDRA [6] and DroidEagle [21] leverage file similarity to detect malware. Our proposed privacy-preserving schema can be applied to these systems. For instance, our prototype uses the detection methods proposed by the DroidRanger and FSquaDRA, but in a privacy-preserving way. From another perspective, dynamic malware detection systems leverage system call sequences [19], runtime information [8], accessed URLs [9], etc. However, these systems do not consider the privacy concerns in sharing runtime behaviors. Our system takes similar methods but protects users' privacy.

Searchable Encryption We propose to use searchable encryption (SE) techniques to achieve privacy-preserving dynamic malware detection on users' phones. In principle, most SE schemes surveyed in [22] are applicable to build encrypted yet queryable signature database as used in our system. These SE schemes (just to name a few) focus on improving locality and throughput [18], supporting boolean queries [14] and similarity queries [17], as well as some specific usage scenarios, like image denoising [23]. To the best of our knowledge, we are the first that apply SE to the mobile malware detection scenario. Considering the mobile context, we customize the efficient scheme proposed by Cash et al. [18], and further incorporate an OPRF protocol to enable users to perform secure detection on their local devices, without revealing the private inputs to other parties, i.e., the private keys of the *SP* and the queried signatures of the users.

IX. Conclusion

In this paper, we presented a design towards privacy-preserving Android malware detection systems. We first shed light on the privacy concerns of existing static and dynamic malware detection systems, involving phone vendors, security service providers, and users. Then we proposed a privacy-preserving schema to improve the overall privacy of all involved parties and implemented a prototype system called PPMDroid. Moreover, we provided rigorous security analysis to justify the security strength of the proposed system, and the evaluation with real malware samples demonstrated its effectiveness and efficiency. As a future direction, we will explore how to efficiently support other stateful information like API call graph.

Acknowledgment

This work was supported in part by the Research Grants Council of Hong Kong under Grant CityU 11276816, Grant CityU 11212717, and Grant CityU C1008-16G, in part by the Innovation and Technology Commission of Hong Kong under ITF Project ITS/168/17, in part by the National Natural Science Foundation of China under Grant 61572412, 61572278, and 61772236, and in part by a Microsoft Azure Grant.

References

[1] IDC, "Worldwide smartphone volumes will remain down in 2018 before returning to growth in 2019 and beyond, according to idc," https://www.idc.com/getdoc.jsp?containerId=prUS43856818, 2018.

[2] C. McGoogan, "Millions of android devices infected with malware in popular judy game," http://www.telegraph.co.uk/technology/2017/05/31/millions-android-devices-infected-malware-popular-judy-game/, 2017.

[3] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proc. of IEEE S&P*, 2012.

[4] 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. of NDSS*, 2012.

[5] E. Mariconti, L. Onwuzurike, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models," in *Proc. of NDSS*, 2017.

[6] Y. Zhauniarovich, O. Gadyatskaya, B. Crispo, F. La Spina, and E. Moser, "Fsquadra: fast detection of repackaged applications," in *Proc. of DBSec*, 2014.

[7] http://googlemobile.blogspot.hk/2012/02/android-and-security.html, 2017.

[8] M. Sun, M. Zheng, J. C. Lui, and X. Jiang, "Design and implementation of an android host-based intrusion prevention system," in *Proc. of ACSAC*, 2014.

[9] C. Zuo and Z. Lin, "SMARTGEN: Exposing Server URLs of Mobile Apps With Selective Symbolic Execution," in *Proc. of WWW*, 2017.

[10] "Huawei's emui 5.0 ups the ante on mobile security," http://www.marketwired.com/press-release/huaweis-emui-50-ups-the-ante-on-mobile-security-2175259.htm, 2017.

[11] "Researchers find methods for bypassing googles bouncer android security," https://threatpost.com/researchers-find-methods-bypassing-googles-bouncer-android-security-060412/76643/, 2017.

[12] T. Vidas and N. Christin, "Evading android runtime analysis via sandbox detection," in *Proc. of ACM ASIACCS*, 2014.

[13] "Top-five paid app on google play was an antivirus scam," https://www.cnet.com/news/top-five-paid-app-on-google-play-was-an-antivirus-scam/, 2017.

[14] S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Outsourced symmetric private information retrieval," in *Proc. of ACM CCS*, 2013.

[15] C. Dong, L. Chen, and Z. Wen, "When private set intersection meets big data: an efficient and scalable protocol," in *Proc. of ACM CCS*, 2013.

[16] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[17] X. Yuan, H. Cui, X. Wang, and C. Wang, "Enabling privacy-assured similarity retrieval over millions of encrypted records," in *Proc. of ESORICS*, 2015.

[18] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation." in *Proc. of NDSS*, 2014.

[19] I. Gasparis, Z. Qian, C. Song, and S. V. Krishnamurthy, "Detecting Android Root Exploits by Learning from Root Providers," in *Proc. of USENIX Security*, 2017.

[20] "Help protect against harmful apps with google play protect," https://support.google.com/accounts/answer/2812853?hl=en, 2017.

[21] M. Sun, M. Li, and J. C. Lui, "Droideagle: Seamless detection of visually similar android apps," in *Proc. of ACM WiSec*, 2015.

[22] C. Bösch, P. Hartel, W. Jonker, and A. Peter, "A survey of provably secure searchable encryption," *ACM Computing Surveys (CSUR)*, vol. 47, no. 2, p. 18, 2015.

[23] Y. Zheng, H. Cui, C. Wang, and J. Zhou, "Privacy-preserving image denoising from external cloud databases," *IEEE TIFS*, vol. 12, no. 6, pp. 1285–1298, 2017.