Taming Information-Stealing Smartphone Applications (on Android)

Yajin Zhou[1], Xinwen Zhang[2], Xuxian Jiang[1], and Vincent W. Freeh[1]

[1] Department of Computer Science, NC State University
yajin_zhou@ncsu.edu, {jiang,vin}@cs.ncsu.edu
[2] Huawei America Research Center
xinwen.zhang@huawei.com

Abstract. Smartphones have been becoming ubiquitous and mobile users are increasingly relying on them to store and handle personal information. However, recent studies also reveal the disturbing fact that users' personal information is put at risk by (rogue) smartphone applications. Existing solutions exhibit limitations in their capabilities in taming these privacy-violating smartphone applications. In this paper, we argue for the need of a new *privacy mode* in smartphones. The privacy mode can empower users to flexibly control in a fine-grained manner what kinds of personal information will be accessible to an application. Also, the granted access can be dynamically adjusted at runtime in a fine-grained manner to better suit a user's needs in various scenarios (e.g., in a different time or location). We have developed a system called TISSA that implements such a privacy mode on Android. The evaluation with more than a dozen of information-leaking Android applications demonstrates its effectiveness and practicality. Furthermore, our evaluation shows that TISSA introduces negligible performance overhead.

Keywords: smartphone applications, Android, privacy mode.

1 Introduction

Mobile phones are increasingly ubiquitous. According to a recent Gartner report [2], in the third quarter of 2010, worldwide mobile phone sales to end users totaled 417 million units, a 35 percent increase from the third quarter of 2009. Among the variety of phones, smartphones in particular received incredible adoption. This trend is further propelled with the wide availability of feature-rich applications that can be downloaded and run on smartphones. For example, Google provides *Android Market* [1] that contains a large collection of Android applications (or apps for short). It is important to note that these app stores or marketplaces contain not only vendor-provided programs, but also third-party apps. For example, Android Market had an increase from about 15,000 third-party apps in November 2009 to about 150,000 in November 2010.

Given the increased sophistication, features, and convenience of these smartphones, users are increasingly relying on them to store and process personal

J.M. McCune et al. (Eds.): TRUST 2011, LNCS 6740, pp. 93–107, 2011.

information. For example, inside the phone, we can find phone call log with information about placed and received calls, an address book that connects to the user's friends or family members, browsing history about visited URLs, as well as cached emails and photos taken with the built-in camera. As these are all private information, a natural concern is the safety of these data.

Unfortunately, recent studies [9,8,13,3] reveal that there are malicious apps that can be uploaded to the app stores and successfully advertised to users for installation on their smartphones. These malicious apps will leak private information without user authorization. For example, TaintDroid [9] shows that among 30 popular third-party Android apps, there are 68 instances of potential misuse of users' private information. In light of these privacy-violating threats, there is an imperative need to tame these information-stealing smartphone apps.

To fulfill the need, Android requires explicit permissions in an app so that the user is aware of the information or access rights that will be needed to run the app. By showing these permissions to the end user, Android delegates the task to the user for approval when the app is being installed. However, this permission mechanism is too coarse-grained for two main reasons. First, the Android permission mechanism requires that a user has to grant *all* the requested permissions of the app if he wants to use it. Otherwise the app cannot be installed. Second, if a user has granted the requested permissions to an app, there is no mechanism in place to later re-adjust the permission(s) or constrain the runtime app behavior.

To effectively protect user private information from malicious smartphone apps, in this paper, we argue for the need of a new *privacy mode* in smartphones. The privacy mode can be used to lock down (or fine tune) an app's access to various private information stored in the phone. More specifically, if a user wants to install an untrusted third-party app, he can control the app's access in a fine-grained manner to specify what types of private information (e.g., device ID, contracts, call log, and locations) are accessible to the app. Further, the user can flexibly (re)adjust at runtime the previously granted access (e.g., at install time).

As a demonstration, we have implemented a system called TISSA that implements such a privacy mode in Android. Our development experience indicates that though the privacy mode support requires modifying the Android framework, the modification however is minor with changes in less than 1K lines of code (LOC). We also have evaluated TISSA with more than a dozen of Android apps that are known to leak a variety of private information. Our results show that TISSA can effectively mediate their accesses and protect private information from being divulged. Also, the privacy setting for each app is re-adjustable at runtime without affecting its functionality.

The rest of this paper is organized as follows: Section 2 describes our system design for the privacy mode support in Android. Section 3 presents its detailed prototype. Section 4 presents evaluation results with a dozen of information-stealing Android apps as well as its performance overhead. Section 5 discusses the limitations of our approach and suggests future improvement. Finally, Section 6 describes related work, and Section 7 summarizes our conclusions.

2 Design of TISSA

2.1 Design Requirements and Threat Model

Design Requirements. Our goal is to effectively and efficiently prevent private information leakage by untrusted smartphone apps. Accordingly, to support the new privacy mode in smartphones, we follow several design requirements to balance privacy protection, system performance, user experience, and application compatibility.

Lightweight Protection: Smartphones are usually resource constrained, especially on CPU, memory, and energy. Therefore, it naturally requires that our security mechanism should be memory- and energy-efficient. Also, the performance overhead of the solution should not affect user experience.

Application Transparency: The privacy mode should also maintain the compatibility of existing Android apps. Accordingly, we may not change APIs currently provided by the default Android framework. Also, from the usability perspective, it is not a good design to partially grant permissions at install time or later revoke permissions at runtime. This is because when a permission is taken away or does not exist, the app may suddenly stop or even crash. As a result it could interfere with the normal execution of the app and hurt the user experience.

Small Footprint: Built on top of existing Android framework including its security mechanisms, the privacy mode support should minimize the changes necessary to the Android framework.

Threat and Trust Model. As our purpose is to prevent private information from being leaked by untrusted apps, we assume user downloaded third-party apps as untrusted. Note that although our scheme can be equally applicable to pre-installed apps on the device, we assume they are benign from privacy protection perspective and will not release private data of the device without authorizations. These pre-installed apps include those from device manufacturers or network operators.

In addition, we also trust the underlying OS kernel, system services, and the Android framework (including the Dalvik VM). Aiming to have a minimal modification to the Android code base, our system is designed to build on top of existing Android security mechanisms, which include the primitive sandbox functions and permission enforcement of Android [11,16]. Naturally, we assume that an untrusted app cannot access system resources (e.g., filesystem) or other apps' private data directly. Instead, they can only be accessed through the normal APIs provided by various content providers or service components in the Android framework.

2.2 System Design

In a nutshell, TISSA provides the desired privacy mode on Android by developing an extra permission specification and enforcement layer on the top of

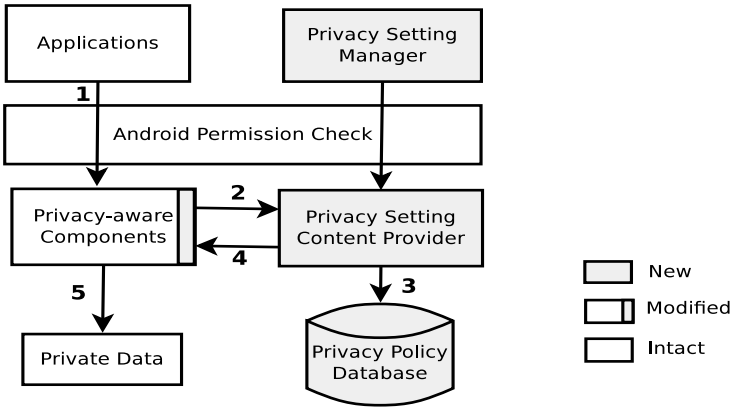


Fig. 1. The TISSA Architecture

existing Android permissions. As shown in Figure 1, TISSA consists of three main components. The first one is the *privacy setting content provider*, which is a privileged component to manage the privacy settings for untrusted apps. In the meantime, it also provides an API that can be used to query the current privacy setting for an installed app. If we consider TISSA as a reference monitor, the privacy setting content provider is the Policy Decision Point (PDP). The second component is the *privacy setting manager*, which is a privileged app that a mobile user can use to manage or update the privacy settings for installed apps. Therefore, it acts as the Policy Administration Point (PAP) in TISSA. The third component is *privacy-aware components*, including those content providers or services that are enhanced in a privacy-aware manner to regulate the access to a variety of user's personal information, including contacts, call log, locations, device identity. These privacy-aware components are designed to cooperate with the first component. In particular, once they receive requests from an app to access private data they manage, they will query the privacy settings, and response to the requests according to the current privacy settings for the app. In other words, they function as the Policy Enforcement Points (PEPs) in TISSA. As a result, there is only one instance of PDP and PAP while multiple PEPs exist and they are integrated with individual content providers or services in the Android framework.

To further elaborate how TISSA works, when an app tries to read a piece of private data, it sends a reading request (arrow 1 in Figure 1) to the corresponding content provider. The content provider is aware of the privacy requirement. Instead of serving this request directly, it holds the request and makes a query first to the privacy setting content provider (arrow 2) to check the current privacy settings for the app (regarding the particular reading operation). The privacy setting content provider in turn queries its internal policy database (arrow 3) that stores user specifications on privacy settings of all untrusted apps, and returns the query result back to the content provider (arrow 4). If this reading operation is permitted (stored in the policy database), the content serves the access request

and returns normal results to the app (arrow 5). This may include querying its internal database managed by the content provider.

However, if the reading operation is not permitted, the privacy setting may indicate possible ways to handle it. In our current prototype, we support three options: empty, anonymized, and bogus. The *empty* option simply returns an empty result to the requesting app, indicating "non-presence" of the requested information. The *anonymized* option instead provides an anonymized version from the original (personal) information, which still allows the app to proceed but without necessarily leaking user information. The *bogus* option on the other hand provides a fake result of the requested information. Note these three options may be further specialized for different types of personal information and be interpreted differently for different apps. Accordingly, the apps will likely behave differently based on the returned results. As a result, mobile users need to be aware of the differences from the same app under different privacy settings and exercise different levels of trust.

Through TISSA, we currently provide three levels of granularity for privacy policy specifications. In the first level, a policy defines whether a particular app can be completely trusted. If yes, it is given all requested accesses through the normal Android permission mechanisms. If not, TISSA provides the second level of policy specification, where one particular setting can be specified for each type of personal information the mobile user wants to protect. Note that it is possible that one app may access one type of personal information for its legitimate functionalities, but should be denied to access other types of information. For example, the *Yellow Pages* app (Section 4) may be normally allowed to access the current location but the access to phone identity or contacts should be prevented. Also, for the non-trusted access to other personal information, the third level of policy specification specifies the above empty, anonymized, and bogus options to meet different needs. For example, for a call log backup app, we do not want to give the plain-text access to the call log content, but an anonymized version. For a *Coupon app* (Section 4), we can simply return a bogus phone identity.

3 Implementation

We have implemented a proof-of-concept TISSA system based on Android version 2.1-update1 and successfully run it on Google Nexus One. In our current prototype, we choose to protect four types of personal information: phone identity, location, contacts, and call log. Our system has a small footprint and is extensible to add the support of other personal information. In the following, we explain in more details about the system implementation.

3.1 Privacy Setting Content Provider

The privacy setting content provider is tasked to manage a local SQLite database that contains the current privacy settings for untrusted apps on the phone. It also provides an interface through which a privacy-aware component (e.g., a

location manager) can query the current privacy settings for an untrusted app. More specifically, in our current prototype, the privacy-aware component will provide as the input the package name of the requesting app and the type of private information it is trying to acquire. Once received, the privacy setting content provider will use the package name to query the current settings from the database. The query result will be an app-specific privacy setting regarding the type of information being requested.

There are some design alternatives regarding the default privacy settings. For example, in a restrictive approach, any untrusted app will not be given the access to any personal information. That is, we simply apply the empty or even bogus options for all types of personal information that may be requested by the app – even though the user approves all requested permissions when the app is being installed. On the contrary, a permissive approach may fall back to the current Android permission model and determine the app's access based on the granted permissions. To provide the compatibility and transparency to current apps, our current prototype uses the permissive approach.

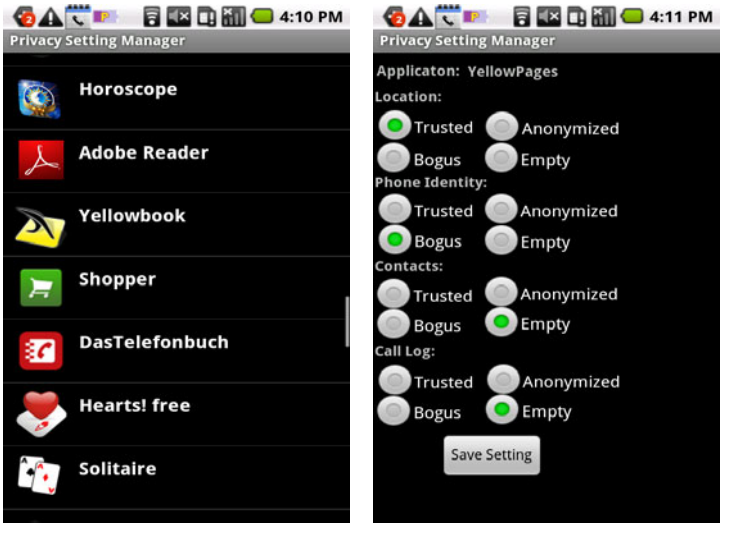
Beside the API interface used to query the privacy settings for a given app, there exists another API interface (in the privacy setting content provider) through which the privacy setting manager can use to initialize or adjust privacy settings in the policy database. Similarly, the input of this interface includes the package name of a target app and its privacy settings. The output will be the confirmation to the requested initialization or adjustment.

Because our policy database stores actual privacy settings for untrusted apps, the security itself is critical. In our prototype, we leverage the existing Android sandbox and permission mechanisms to protect its integrity. Specifically, the database file is a private data file of the privacy setting content provider (thus with the same UID in filesystem). Other apps have different UIDs and will be denied to access it directly. To further restrict the update capability of the database via corresponding API, we declare a dedicated Android permission (in the privacy setting content provider) with the protection level of `signature`. This means that the permission will only be granted to apps signed with the same certificate as the privacy setting content provider. In our prototype, the privacy setting manager is the only one that is signed by the same certificate.

In total there were 330 LOC in the privacy setting content provider implementation.

3.2 Privacy Setting Manager

The privacy setting manager is a standalone Android app that is signed with the same certificate as the privacy setting content provider. As mentioned earlier, by doing so, the manager will be given the exclusive access to the privacy setting database. In addition, the manager provides the visual user interface and allows the user to specify the privacy settings for untrusted apps. We stress that the privacy setting here is orthogonal to the permissions that the user has granted at the app install time; that is, the privacy setting manager provides a separate setting for the privacy mode.



(a) A list of installed apps (b) The privacy settings for the *YellowPages* app

Fig. 2. The Privacy Setting Manager

In particular, the manager app includes two activity components. The default one is `PrivacySettingManagerActivity`, which when activated displays a list of installed apps. The phone user can then browser the list and click an app icon, which starts another activity called `AppPrivacySettingActivity` and passes the app's package name. When the new activity is created, it queries the privacy setting content provider for the current privacy settings and displays the results to the user. It also has radio buttons to let user customize or adjust the current settings. Any change of the settings will be immediately updated to the policy database via the privacy setting content provider. In Figure 2, we show the screenshots of the manager app, which show the list of installed app (Figure 2(a)) and the privacy setting for the *Yellow Pages* app (Figure 2(b)).

In total there were 452 LOC in the privacy setting manager implementation.

3.3 Privacy-Aware Components

The third component in our prototype is those privacy-aware Android components, including the contacts content provider, the location manager, and the telephony manager. These Android components are enhanced for the privacy mode support.

Contacts and Call Logs: Figure 3 shows the flowchart for the contact information that is being accessed. Specifically, when an app makes a query to the contacts content provider, the request is received by the content resolver, which checks whether the app has the permission (based on existing Android permissions). If not, a security exception is thrown to the app and the access stops. Otherwise, it dispatches the request to the contacts content provider, which in the privacy

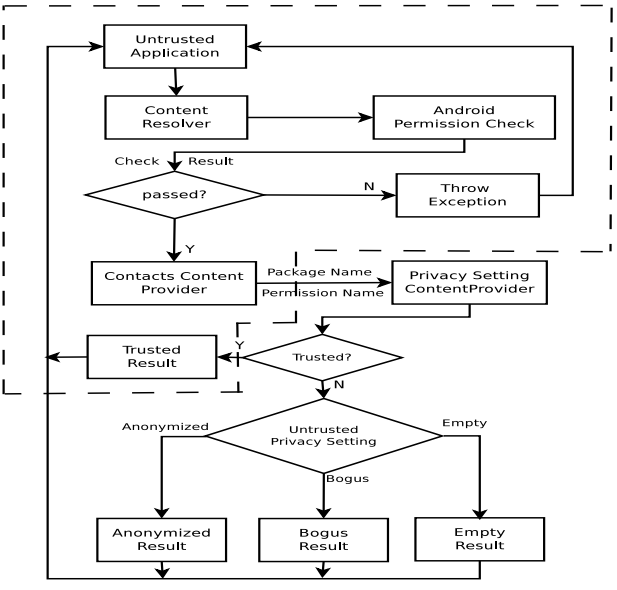


Fig. 3. Protecting Contacts in TISSA

Table 1. Detailed Breakdown of Android Modification in TISSA

Component Name	LOC
Privacy Setting Provider	330
Application Management Program	452
LocationManager & LocationManagerService	97
TelephonyManager	61
ContactsProvider (Contacts&Call Log)	48
Total	988

mode in turn queries the privacy setting content provider to check if the requesting app can access the contacts data. If the app is allowed, the contacts content provider then queries its own database and returns back authentic contacts to the app. The dotted line in Figure 3 encloses those components that also exist in the original Android. The rest components (outside the dotted area) show the additional components we added for the privacy mode support in Android.

From the privacy setting database, if the app is not trusted to access the requested data, the contacts content provider will respond differently: an *empty* setting returns an empty contact record; an *anonymized* settings returns an anonymized version of the contact records; and an *bogus* setting simply returns fake contact information. By doing so, we can protect the authentic contacts information from being leaked by this untrusted app. In the meantime, as the app is given different results, the mobile user should expect different app behavior and exercise different levels of trust when interacting with the app.

Phone Identity: Mobile phones have unique device identifier. For example, the IMEI and MEID numbers are unique identities of GSM and CDMA phones, respectively. An Android app can use the functions provided by telephony service to obtain these numbers. As a result, we hook these functions and return a device ID based on the current privacy setting for the requesting app. For example, in the case of the *Yellow Pages* app (Figure 2(b)), we simply return a bogus phone identity number.

Location: There are several location content providers in Android to provide either coarse-grained or fine-grained information (e.g., based on different devices such as GPS and Wifi). However, for an app to obtain the location, there are two main ways: First, it can obtain the information from the related devices through the `LocationManager`; Second, an app can also register a listener to receive location updates from these devices. When there is a location change, the registered listener will receive the updated location information. Accordingly, in our current prototype, we hook related functions to intercept incoming requests. To handle each request, we query the privacy setting for the initiating apps and then respond accordingly. Using the same *Yellow Pages* app example, our current setting (Figure 2(b)) returns the authentic location information.

Our TISSA development experience indicates that the privacy mode support in Android involves no more than 1K LOC implementation. As a result, we believe our approach has a small footprint – satisfying our third design requirement (Section 2). The detailed breakdown of revised components in Android is shown in Table 1.

4 Evaluation

We use a number of Android apps as well as standard benchmarks to test the effectiveness and performance impact of our system. Some of the selected Android apps have been known to leak private information [9]. Our testing platform is Google Nexus One that runs the Android version 2.1-update1 enhanced with the TISSA privacy mode.

4.1 Effectiveness

To evaluate the TISSA effectiveness in preventing private information leakage by untrusted apps, we choose 24 free apps from the Android Market at the end of November 2010. (The complete list of tested apps is in Table 2.) Among these 24 apps, 13 of them (marked with † in the table) are known to be leaking private information as reported from the TaintDroid system [9]. The remaining 11 apps are randomly selected and downloaded from the Android Market. Most of these apps require the permissions for locations (`ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION`) and phone identity (`READ_PHONE_STATE`). Also, one third of them require the access of contacts and call log (`READ_CONTACTS`).

In our experiments, we divide these apps into two sets. The first set has the 13 apps that are known to be leaking private information and the second set has the

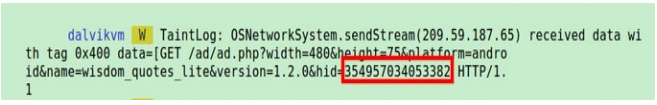
Table 2. Example Apps for TISSA Effectiveness Evaluation

Third-party Apps (24 in total)	Location	Phone Identity	Contacts	Call Log
The Weather Channel†; Movies†; Horoscope†; Layar†; Coupons†; Trapster†; Alchemy; Paper Toss; Disney Puzzle; Find It; (10)	×	×		
Wertago†; Yellow Pages†; DasTelefonbuch†; RingTones†; Knocking†; (5)	×	×	×	×
Wisdom Quotes Lite†; Classic Simon Free; Wordfeud FREE; Moron Test:Section 1; Bubble Burst Free; (5)		×		
Astrid Tasks†; (1)	×			
CallLog; Last Call Widget; Contact Analyzer; (3)		×	×	×

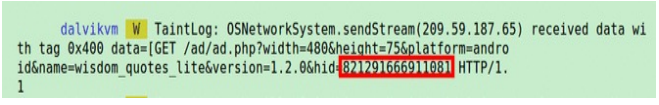
remaining 11 apps. While both sets are used to verify the transparency of TISSA to run apps, we use the first set to evaluate the effectiveness of TISSA. In the second set, we use TISSA to capture suspicious access requests from these apps and then verify possible information leakage with the TaintDroid system. Before running the experiments, we turn on restrictive privacy policy as the default one, i.e., not trusting any of these apps in accessing any personal information. If a particular app requires the user registration, we will simply create a user account and then log in with the app.

Our results with the first set of information-leaking apps show that TISSA is able to effectively capture the access requests from these untrusted apps to private data and prevent them from being leaked. As a demonstration, we show in Figure 4 two experiments with the same app named *Wisdom Quotes Lite*: one without the TISSA protection (Figure 4(a)) and another with TISSA protection (Figure 4(b)). Note that this app has declared the permission to access the phone identity, but this information will be leaked to a remote data server. The evidence is collected by TaintDroid and shown in Figure 4(a). In particular, it shows that the app is sending out the IMEI number to a remote server. From the log, we can see that the data sent out is tainted with the taint tag 0x400, which indicates the IMEI source. The destination IP address is `xxx.59.187.65`, a server that belongs to the company that developed this app. The leaked IMEI number (`354957034053382`) of our test phone is in the query string of `HTTP GET` request to the remote server. After confirming this leakage, we use the privacy setting manager to adjust the phone identity privacy for this app to be bogus. Then we run this app again. This time we can find (Figure 4(b)) that although the app is sending an IMEI number out, the value sent out is the bogus one, instead of the real IMEI number – this is consistent with our privacy setting for this app.

As another example, an app named *Horoscope* requested the permission of reading current locations. However this app's functionality does not justify such need. In our experiments, we find that phone location information is leaked by this app (confirmed with TaintDroid and the leaked information was sent to a remote server `xxx.109.247.8`). Then we use the privacy setting manager to adjust the location privacy to be empty, which means the location service neither



(a) The leaked IMEI number without TISSA



(b) The leaked IMEI number with TISSA

Fig. 4. Experimenting with an Android App *Wisdom Quotes Lite*

returns the last known location nor updates the new location to this app. We then re-run this app and do not find the true location information leaked.

In the experiments with the second set of apps, we use the default restrictive privacy policy and enable the bogus option for the phone identity. Interestingly, we observe that among these 11 apps, seven of them accessed the phone identity information. We then re-run them in TaintDroid and our results successfully confirm the information leakage among all these 11 apps. From the same set, we also confirmed that two of them leak location information. As a result, among the two sets of total 24 apps, 14 of them leak location information and 13 of them send out the IMEI number. Six of them leak both location and IMEI information. With TISSA, users can select the proper privacy setting and effectively prevent these private information from being leaked.

We point out that although no app in our study leaks the contacts and call log information, they can be similarly protected. Specifically, we can simply adjust the privacy setting of an app which has the `READ_CONTACTS` permission to be *empty*, then any query from the app for contacts will be returned with an empty contact list. Also, our experiments with these 24 apps indicate that they can smoothly run and no security exceptions have been thrown to affect the functionalities of these apps, which satisfies our second design requirement (Section 2).

4.2 Performance

To measure the performance overhead, we use a JAVA benchmark – CaffeineMark 3.0 – to gauge Android app performance and the resulting scores from the benchmark are shown in Figure 5. Note that the benchmark contains a set of programs to measure the app runtime (in particular the Dalvik VM) overhead. From the figure, we can see that there is no observable performance overhead, which is expected as our system only made minor changes to the framework and the Dalvik VM itself was not affected by our system.

In summary, our measurement results on performance indicate that TISSA is lightweight in successfully supporting the privacy mode on Android, which meets our first design requirement.

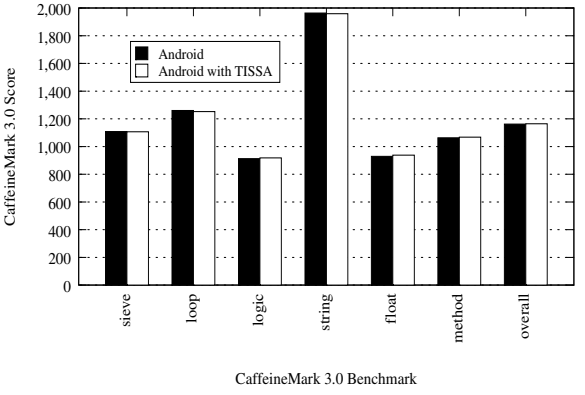


Fig. 5. The Result of CaffeineMark 3.0 Benchmark

5 Discussion

We observe that giving bogus private information could be problematic for some apps. For example the *Weather Channel* app needs to access the device location information to provide the weather forecast of a particular area. If the user chooses the bogus option for its access to this information, it will certainly not work as intended (even though the program does not crash and still functions). One possible enhancement is to provide accurate, but low-fidelity data. With the same app example, it can work equally well with an approximate location. In fact, a simple approximation option could return a random location within a radius (say 10 or 100 miles) of the current location. This indicates that our system can be readily extended to support much finer-grained access than currently demonstrated in this paper. Also, for efficiency, TISSA only uses one single privacy setting for one type of private information. This can be further extended to provide finer access to user data. An example will be to release some personal information, an option between empty and anonymous.

From another perspective, current application development can be improved to provide more contextual information to better help users to make the decision. Though the risk is still on the mobile user side, a privacy-aware app developer can offer alternative mechanisms to obtain private information, such as by providing a UI with necessary context information to justify the need for the access and asking the user to provide the required input (e..g, a ZIP code).

We also note that TISSA is not a panacea for privacy protection. For an app that backs up user contact data to a remote server, the user usually grants the access to the app. However, there is no assurance that the data will not be transferred to elsewhere or be stolen. Our solution does provide an *anonymized* option for lightweight protection. Specifically, when an app reads the contacts data, based on the anonymized setting, the privacy-aware component can return an encrypted version with a key specified by the user or derived from a password.

Therefore, even if data are leaked during transfer or at the server side, the cleartext information is not released. However, due to the resource constraints on mobile phones, the encryption for anonymization cannot be sufficiently strong as desired.

With TISSA enabled, a mobile user can specify privacy settings for installed apps at anytime. Furthermore, the privacy setting can be integrated with the application installer of Android, i.e., the privacy setting manager can be activated by the installer whenever a new app is installed. However, we note that TISSA fundamentally does not provide mechanisms or solutions to help user understand and reason about permission requirements for individual apps. To further improve the usability of TISSA, several approaches can explored. For example, as our future research directions, we can include additional contextual or guiding information to facilitate the privacy setting. Also, we can provide privacy-setting templates to reduce the burden on mobile users. This can be done by applying certain heuristics with the consideration of a device's location [5] or an app's category.

6 Related Work

Privacy issues on smartphones have recently attracted considerable attention. A number of research efforts have been conducted to identify private data leaking by mobile apps. TaintDroid [9] and PiOS [8] are two representatives that target Android and iOS platforms, respectively. Specifically, TaintDroid [9] uses dynamic taint analysis to track privacy data flow on Android. It first taints data from privacy information sources, and then integrates four granularities of taint propagation (i.e., variable-, method-, message-, and file-levels). When the data leaves the system via a network interface, TaintDroid raises an alert (Figure 4) with log information about the taint source, the app that sends out the tainted data, and the destination. PiOS [8] instead applies static analysis on iOS apps to detect possible privacy leak. In particular, it first constructs the control flow graph of an iOS app, and then aims to find whether there is an execution path from the nodes that access privacy source to the nodes of network operations. If such path exists, it considers a potential information leak.

From another perspective, Avik [6] presents a formal language to describe Android apps and uses it to formally reason about data flow properties. An automatic security certification tool called ScanDroid [12] has been developed to extract security specifications from manifests that accompany Android apps, and then check whether data flows through those apps are consistent with those specifications. Note that these systems are helpful to confirm whether a particular app actually leaks out private information. However, they are not designed to protect user private information from being leaked, which is the main goal of our system.

On the defensive side, Kirin [10] is a framework for Android which extracts the permissions from an app's manifest file at install time, and checks whether these permissions are breaking certain security rules. The security rules are manually defined to detect undesired combination of Android permissions, which may be

insecure and misused by malicious programs. As a result, Kirin aims to enforce security check at install time, while our system provides a lightweight runtime protection (after an app is installed).

As with our research, Apex [14] utilizes the Android permission model to constrain runtime app behavior. However, there are three key differences. First, they share different design goals. The main objective of Apex is to restrict the usage of phone resources (e.g., when and how many MMS messages can be sent within a day), instead of the exclusive focus on personal information leakage prevention in our system. Second, by design, Apex limits itself to current available permissions but allows for selectively granting permissions at install time or possibly revoking some permissions at runtime. In comparison, our privacy settings is completely orthogonal to current Android permissions. Third, as mentioned earlier, one undesirable consequence of Apex in only partially granting requested permissions is that it may throw security exceptions to a running app, causing it terminate unexpectedly, which violates our second design requirement and is not user-friendly.

Saint [15] is another recent work that allows an app developer to provide a security policy that will be enforced (by the enhanced Android framework) to regulate install-time Android permission assignment and their run-time use. Security-by-Contract [7] similarly allows an app to be bound to a contract, which describes the upper-bound of what the app can do. Notice that these security policies or contracts need to be provided during application development, *not* by mobile users. As a result, it is unclear on how they can be applied by mobile users to customize the privacy protection. MockDroid [4] allows users to mock the access from an untrusted app to particular resources at runtime (by reporting either empty or unavailable). The mocked access can be naturally integrated into TISSA as another privacy setting option on smartphones.

7 Conclusion

In this paper, we argue for the need of a privacy mode in existing smartphones. The need comes from the disturbing facts that (rogue) smartphone apps will intentionally leak users' private information. As a solution, we present the design, implementation and evaluation of TISSA, a privacy-mode implementation in Android. TISSA empowers mobile users the fine-grained control and runtime re-adjustment capability to specify what kinds of user information can be accessible (and in what way) to untrusted apps. Orthogonal to current Android permissions, these privacy settings can be adjusted without necessarily affecting the normal functionalities of apps. Our experiments demonstrate its effectiveness and practicality. The performance measurements show that our system has a low performance overhead.

References

1. Android Market, `http://www.android.com/market/`
2. Gartner November Report, `http://www.gartner.com/it/page.jsp?id=1466313`

3. IPhone and Android Apps Breach Privacy, http://online.wsj.com/article/SB10001424052748704694004576020083703574602.html
4. Beresford, A.R., Rice, A., Skehin, N., Sohan, R.: MockDroid: Trading Privacy for Application Functionality on Smartphones. In: 12th Workshop on Mobile Computing Systems and Applications (2011)
5. Bernheim Brush, A.J., Krumm, J., Scott, J.: Exploring End User Preferences for Location Obfuscation, Location-Based Services, and the Value of Location. In: 12th ACM International Conference on Ubiquitous Computing (2010)
6. Chaudhuri, A.: Language-Based Security on Android. In: 4th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (2009)
7. Desmet, L., Joosen, W., Massacci, F., Philippaerts, P., Piessens, F., Siahaan, I., Vanoverberghe, D.: Security by Contract on the.NET Platform. Information Security Technical Report 13(1), 25–32 (2008)
8. Egele, M., Kruegel, C., Kirda, E., Vigna, G.: PiOS: Detecting Privacy Leaks in iOS Applications. In: 18th Annual Network and Distributed System Security Symposium (2011)
9. Enck, W., Gilbert, P., Chun, B.-G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In: 9th USENIX Symposium on Operating Systems Design and Implementation (2010)
10. Enck, W., Ongtang, M., McDaniel, P.: On Lightweight Mobile Phone Application Certification. In: 16th ACM Conference on Computer and Communications Security (2009)
11. Enck, W., Ongtang, M., McDaniel, P.: Understanding Android Security. IEEE Security & Privacy 7(1), 50–57 (2009)
12. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: SCanDroid: Automated Security Certification of Android Applications (2009), http://www.cs.umd.edu/~avik/papers/scandroidascaa.pdf
13. Mahaffey, K., Hering, J.: App Attack: Surviving the Explosive Growth of Mobile Apps (2010)
14. Nauman, M., Khan, S., Zhang, X.: Apex: Extending Android Permission Model and Enforcement with User-Defined Runtime Constraints. In: 5th ACM Symposium on Information, Computer and Communications Security (2010)
15. Ongtang, M., McLaughlin, S.E., Enck, W., McDaniel, P.D.: Semantically Rich Application-Centric Security in Android. In: 25th Annual Computer Security Applications Conference (2009)
16. Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., Dolev, S., Glezer, C.: Google Android: A Comprehensive Security Assessment. IEEE Security & Privacy 8(2), 35–44 (2010)