SweetDroid: Toward a Context-Sensitive Privacy Policy Enforcement Framework for Android OS

Xin Chen*  
DiDi Labs  
Mountain View, CA  
xinchen@didichuxing.com

Heqing Huang  
IBM TJ Watson  
Yorktown Heights, NY  
hhuang@us.ibm.com

Sencun Zhu  
Penn State University  
University Park, PA  
szhu@cse.psu.edu

Qing Li  
Symantec Inc.  
Mountain View, CA  
qing_li@symantec.com

Quanlong Guan  
Jinan University  
Guangzhou, China  
gql@jnu.edu.cn

ABSTRACT

Android privacy control is an important but difficult problem to solve. Previously, there was much research effort either focusing on extending the Android permission model with better policies or modifying the Android framework for fine-grained access control. In this work, we take an integral approach by designing and implementing SweetDroid, a calling-context-sensitive privacy policy enforcement framework. SweetDroid combines automated policy generation with automated policy enforcement. The automatically generated policies in SweetDroid are based on the calling contexts of privacy sensitive APIs; hence, SweetDroid is able to tell whether a particular API (e.g., getLastKnownLocation) under a certain execution path is leaking private information. The policy enforcement in SweetDroid is also fine-grained – it is at the individual API level, not at the permission level. We implement and evaluate the system based on thousands of Android apps, including those from a third-party market and malicious apps from VirusTotal. Our experiment results show that SweetDroid can successfully distinguish and enforce different privacy policies based on calling contexts, and the current design is both developer hassle-free and user transparent. SweetDroid is also efficient because it only introduces small storage and computational overhead.

CCS CONCEPTS

• Networks → Mobile and wireless security;

KEYWORDS

Android OS, permission, privacy policy, access control

1 INTRODUCTION

Mobile devices and applications have become an essential part of our life, and they keep on shaping our daily life. Meanwhile, they store and process a large amount of users’ sensitive information, including device serial numbers, locations, private messages, contact lists, browsing history, and etc. This raises many security and privacy concerns. Based on the report from Mobile-Sandboxing [10], user-privacy threatening malware families are prominent, which contribute to 50% of all malware families in 2015. To remedy against privacy violations arising from mobile applications, the development of effective approaches to protecting user privacy is extremely important. Here effectiveness means the proposed techniques should not only work correctly and achieve the desired security properties, but also be developer- and user-friendly.

Past research efforts on enhancing user privacy in mobile devices mostly focused on permission model extension and enforcement [1–3, 7–9, 11, 17, 19, 20, 27, 28]. Those proposed solutions provided users with more flexible and fine-grained policy enforcement than the conventional Android’s permission system, which is notorious for its all-or-nothing permission authorization during an app installation phase [3, 19]. For example, a number of solutions [1, 3, 8, 9, 19, 27, 28] enable users to turn off the permissions granted to the installed applications or intervene in their access to sensitive resources based on user-defined security policies. In a similar fashion, Android 6.0 introduces a new permission model that does not request users to grant any permissions during app installation time, but asks for a permission with a pop-up window when an app needs it at the first time [14]. This is very similar to iOS permission model. The new Android permission model also provides users a graphical user interface (GUI) to enable/disable the permissions at any time.

However, most of the existing solutions still apply permissions to a whole app and hence fail to prevent privacy violations. They do not consider or distinguish the contexts of sensitive data access during the runtime. For example, users may grant a weather forecast app the permission to access the location data because the app needs to retrieve the weather forecast information based on the current location. However, it is unclear whether the app will access the same sensitive data in other (suspicious) contexts. To ensure the contextual integrity in accessing sensitive data, a simple solution is to force users to make a decision per sensitive data request. Nevertheless, there are two major weaknesses in practice.

*The majority of this work is finished in Penn State University.
First, it may cause dialog fatigue since users have to permit every sensitive data request. Second, users may not be capable of making right decisions [25, 30]. Although there are clear examples of using sensitive data for app functionalities, there are also less obvious cases where even privacy-aware users have no clue. For example, app developers could use device IDs, phone numbers or Google accounts as unique identifiers of devices or users for realizing app functionalities, and such code-level information is not available to end users when they make decisions [5]. In a nutshell, distinguishing the context of the access to sensitive data and reasoning about the use of sensitive data under the context are two critical keys to enhancing user privacy yet retaining usability for mobile applications.

In this paper, we aim at addressing these challenges by proposing a calling-context-sensitive privacy policy enforcement framework, named SweetDroid. Extending the existing Android framework, SweetDroid generates fine-grained privacy policies targeting installed apps on Android devices, and enforces these privacy policies at a calling-context level in application runtime to effectively enhance user privacy yet retain the app’s usability. Our approach achieves this fine-grained context-aware privacy enforcement without application repackaging or re-signing, which makes our approach practical for deployment.

Overall, this paper makes three main contributions:

- **Fine-grained policy enforcement via dynamic app patching (hassle-free to developers)** We propose a novel enforcement framework to enhance user privacy protection and yet retain application usability. To our best knowledge, our framework is the first to enforce fine-grained privacy control at the information-flow level. Our framework is able to distinguish sensitive data requests arising at different call sites and respond differently based on privacy policies.

- **Leveraging security analysis tools for automated policy generation (transparent to end users)** Different to conventional policy enforcement schemes, our framework automatically generates privacy policies based on security analysis rather than forcing users to specify policies. This not only largely releases users from the difficulty in making privacy policies but also optimizes the effectiveness of policies to achieve the sweet point balancing privacy protection and app usability.

- **Implementation and evaluation** We implement a prototype of SweetDroid and evaluate it with apps from a third-party market and known privacy-invasive apps. Our evaluation results demonstrate that SweetDroid can effectively distinguish different sensitive data requests within an app and respond appropriately to enhance user privacy yet retain app usability.

**Road map:** The remainder of the paper continues as follows: Section 2 provides an illustrative example that motivates our design, followed by discussions about the attack and trust models, as well as the our design goals. Section 3 elaborates on the whole architecture and design of our SweetDroid. Section 4 performs a security analysis of our SweetDroid system. A case study is shown in Section 5, followed by detailed evaluation result in Section 6. Section 7 reviews related works and Section 8 concludes this paper.

```
1 public class AppActivity extends Activity {  
2     private UserProfile mUserProfile = null;
3     
4     private void getUserProfile() {
5         TelephonyManager tm = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
6         String imei = tm.getDeviceId();
7         if (imei != null) {
8             // Get users' profile from the App server
9             mUserProfile = fetchUserProfile(imei);
10         }
11     }
12     
13     public class LibraryClass extends Service{
14         public void create() {
15             String imei = tm.getDeviceId();
16             sendImei();
17         }
18     }
19 
20     public class LibraryClass extends Service{
21         public void create() {
22             String imei = tm.getDeviceId();
23             sendImei();
24         }
25     }
```

**Listing 1:** An Example Android Application

## 2 MOTIVATION AND DESIGN GOALS

### 2.1 Motivation

This subsection first describes our motivation by presenting an example, and then explains the threat model and the trust model, followed by the design goals and challenges. List 1 gives two code snippets extracted from a real-world Android application. The first snippet of code implements an activity of this application (AppActivity). It contains a private method getUserProfile, which fetches a user’s profile from the application server by sending the user’s IMEI (returned by getDeviceID) and then puts the profile data into a private field mUserProfile (line 9). The purpose is to frictionlessly authenticate a user \(^1\) and fetch the user’s app data remotely. This kind of IMEI usage is common in Android applications where the fetched app data contain app record, game saves, remaining trial days, etc. It can avoid local data loss caused accidentally (e.g., reset system for rescue) or intentionally (e.g., renew a trial app). The AppActivity uses UserProfile to interact with users and accomplish its function; hence, impeding the IMEI retrieval could disable the application function or harm user experience.

The second snippet of code, as a part of a third-party library, implements a public method sendImei to send out users’ IMEI to a remote server in an Android service (line 19). It invokes the method when the service is created (line 23). The method is used by the third-party library to harvest users’ sensitive information, and this kind of sensitive information does not serve any application function.

\(^1\) It actually authenticates a device rather than a user since a user may have multiple devices.
To avoid similar unnecessary sensitive information leak, we should enforce the principle of least privilege. In the above example, it is clear that the IMEI sharing in the first case should be approved whereas the second sharing case should be considered as a privacy violation and hence prevented. Clearly, to realize the principle of least privilege here, two basic requirements are imposed: (1) correctly understand the calling context where each sensitive data access occurs, and (2) correctly control each sensitive data access. We call any privacy policy enforcement system meeting these two requirements context-sensitive. The current Android permission system (up to Android 6.0), as well as most of the existing policy enforcement approaches, are not context-sensitive. In other words, they are agnostic to the exact calling contexts within an app process, and their permission enforcement applies equally to all code (including Java and Native code) executing under the app’s assigned UID.

### 2.2 Security Model

**Threat Model.** We assume the following threat model in this work. The attacker can make a malicious app with arbitrary Dalvik bytecode. The attacker’s goal is to steal users’ private data through sensitive APIs and networking (such as the Internet and SMS). The malicious app can be a repackaged version of a legitimate app but for malicious purposes. We assume that the malicious app has full control over its process and memory address space. Moreover, the app’s code and thus its behavior may be self-modified at run-time through techniques like native code or Java reflection.

**Trust Model.** We assume that the Android OS is trusted, including the Linux kernel and the Android framework. That is, we assume that the malicious app cannot compromise the integrity of the kernel or the Android framework. We assume that the attacker has no way of circumventing the security mechanism of the Android platform or exploiting system vulnerabilities to gain excessive privileges. Our system deploys the enforcement policy based on such a trust base.

### 2.3 Design Goals and Requirements

**SweetDroid** is designed to be context sensitive. Specifically, it should meet the following performance requirements.

- **Effectiveness.** SweetDroid should be able to enforce privacy policies on sensitive data requests under different calling contexts, despite the fact that current Android permission system grants sensitive permissions to an entire app.
- **Usability.** Since ordinary users are not experts and lack knowledge in code-level contexts, to be user friendly, the learning and generation of calling contexts and their associated privacy policies must be automatic. Moreover, the run-time enforcement should also be automatic and transparent (i.e., invisible) to users. Ideally, normal users are not required to be aware of any privacy risk when installing apps or any protection that will be in place – they simply enjoy the functionalities of the apps they like and the system will take care of everything. In this sense, here usability is different from the traditional meaning of providing a nice GUI for users, but making the protection system transparent and intervention-free to end users. While this ideal situation sounds hard (or even impossible) to reach in practice, the system should require only minimal and simple user intervention if needed.
- **Security.** The privacy policy enforcement should be resistant to attacks targeting it.

**Scope of Research:** From the above one can see that context-sensitive access control is a very attractive security property for Android, but it may also sound too ambitious or even impossible. Indeed, there are technical challenges, as will be elaborated later. However, as the research community on Android security has been working ceaselessly to tackle the challenges (hence many sophisticated analysis tools have become available), we believe we can leverage the research community’s joint efforts to achieve our ultimate goal in the future. In this work, we will first design a framework for achieving our goal, and then propose some technical approaches for realizing this framework.

---

![Figure 1: The design of SweetDroid framework. Here the white-background components (boxes) are from the existing Android system, and the grey-background components are newly introduced by SweetDroid.](image-url)
There are mainly three technical phases in SweetDroid. We briefly explain each phase, and introduce more details in the followings subsections. The first phase is an offline process to automatically generate a privacy context file for each app. Privacy context is a privacy policy regarding sensitive API invocations of the app and will be referred to at runtime for sensitive data access control. For a given third-party app, we use a component called privacy analyzer, which is an offline analysis tool, to analyze and generate its privacy context. From the design point of view, the privacy analyzer is highly replaceable. That is, any static or dynamic analysis tool that can evaluate the trustworthiness of sensitive API invocations is suitable. With this offline automatic policy generation phase, end users are released from the challenges and inconvenience in making privacy polices for apps.

The second phase is called application installation mediation. It is a lightweight process that statically analyzes and "patches" the app’s sensitive APIs with privacy contexts generated in the first phase, so that appropriate privacy policies can be enforced on these sensitive APIs at runtime.

The third phase is privacy policy enforcement, which, as a reference monitor, interacts with an app at runtime to enforce privacy policies. Essentially, the patched sensitive API invocations are all redirected here to ask for the access based on existing privacy polices. This phase involves three technical components: Privacy Mediator, Policy Manager App and Policy Manager Service.

### 3.2 Phase I: Privacy Context Generation

The Privacy Analyzer in SweetDroid is a component to analyze a given third-party app, evaluate how risky each sensitive API invocation is and generate the corresponding privacy context for the privacy policy enforcement phase.

**Privacy Context.** We define an app’s privacy context as the collection of the app’s API invocations for accessing user’s sensitive information, their calling contexts in the app’s bytecode, and their trustworthiness based on security analysis. Here, the calling context refers to the context where a sensitive API invocation is called. To be concrete, it is an $N$-depth call stack when the sensitive API function is called.

An app’s privacy context file, in JSON format, records the app’s sensitive API invocations, their calling contexts including the classes and methods where they are invoked, how trustworthy each sensitive API invocation is, and recommended rules for privacy policy enforcement. Listing 2 shows an example of a partial privacy context file generated in our implementation (where the depth of calling context, $N$, is set to 1). The file contains a JSON object, which has a privacy_context property. The privacy_context property consists of a list of items, where every item contains a sensitive API invocation (i.e., call property), the type of the related sensitive information (i.e., type property), the signature of its calling method plus the zero-based index of its occurrence in its calling methods (i.e., context property) (note the same sensitive API may be invoked multiple times in one method), its analysis result (i.e., analysis property), and the recommended enforcement rule for the invocation (i.e., recommended_rule property). The analysis result of the first sensitive API invocation here is "Used by advertisement libraries", which means the invocation is used by advertisement libraries, and the recommended rule is "mock". The analysis result of the second API invocation is "Used to fulfill app functions", which suggests that the invocation is used for app functions, and thus the recommended rule is "allow".

These analysis results are generated based on Privacy Analyzer, and it has two major benefits. First, it largely releases ordinary users from the challenges in making app privacy polices. Prior privacy enforcement schemes highly rely on a security administrator role to make up privacy polices, which could be impractical to most end users in Android world. Second, leveraging state-of-the-art security analysis tools significantly optimizes privacy polices to achieve the sweet point balancing privacy protection and app usability.

Privacy Analyzer is customizable and replaceable. That is, any static or dynamic analysis tool that evaluates the trustworthiness of sensitive API invocations could be extended to fulfill the task, such as [5, 25, 26, 30]. In the implementation of our prototype, we use DroidJust [5] as the privacy analyzer, and the depth of calling context of analyzed sensitive API invocations is set to $N = 1$. DroidJust provides a fully automated approach to justifying an app’s sensitive information transmission. It is based on the observation that the

```
Listing 2: Example Privacy Context File
```

```json
{
  "privacy_context": [
    {
      "type": "Phone_Info",
      "call": "android.telephony.TelephonyManager: java.lang.String getDeviceId()"
    },
    {
      "type": "Location",
      "call": "android.location.LocationManager: android.location.Location getLastKnownLocation()"
    }
  ]
}
```
functionality of a mobile app (except some background services) is commonly experienced by users during their interactions with the app through some sensible phone states (SPS). SPSs are the phone output events that can be directly sensed by phone users, e.g., display, sound, vibration and light. If there is no difference at all in user’s experience whether one shares his private data or not, it means such sharing is not necessary and should be prevented. Specifically, if an app sends out sensitive information \(f_o\) only without receiving any information \(f_i\) from network, such an \(f_o\) cannot lead to any SPS and is hence considered unintended. On the other hand, if an \(f_i\) exists, it is not necessarily used to fulfill an app’s functionality; that is, it may not lead to any SPS (directly or indirectly). If the \(f_i\) actually leads to some SPS, it is sensible. If an \(f_o\) is linked to a sensible \(f_i\), it is intended. The above perspective allows one to automate the detection of intended and unintended information flows.

Besides DroidJust, other static or dynamic analysis approaches such as [25, 29, 30] can also be employed to provide various qualitative and quantitative metrics for evaluating the trustworthiness of sensitive API invocations and generate default privacy policies for them automatically. Specifically, [25] provides an effective approach to judge if a privacy release is legitimate based on statistical reasoning throughout an app’s dynamic analysis. [29] introduces a static program analysis approach that extracts the context of security-sensitive behaviors to assist app analysis in differentiating between malicious and benign behaviors. [30] uses a semi-automatic approach to judge if a privacy release point is legitimate based on the sequence of GUI manipulations that lead to the privacy release. Eventually, these approaches can be employed to generate default privacy policies regarding the access to sensitive API invocations in an app.

Privacy Analyzer classifies the output of the above analysis tools (i.e., sensitive API invocations justified or not) into three categories. The first category applies to the sensitive API invocations that are used for fulfilling app functions. The default enforcement rule is “allow”. The second category applies to the sensitive API invocations that are used by advertisement libraries. The default enforcement rule is “mock” if applicable or “deny” otherwise. The last category applies to the sensitive API invocations that are neither used for app functions nor by advertisement libraries. The default enforcement rule is “deny”.

We note that while our default rules are intuitively suitable, some end users may want to change the recommended enforcement rules (e.g., change “mock” to “allow” to permit tracking and targeted advertisements). For convenience, in our system we also provide a user interface through a Policy Manager App, which allows advanced users to adjust default enforcement rules at any time. The change of rules (e.g., “mock” to “allow” for advertisement) can be generic (i.e., applicable to all apps) or specific (i.e., applicable to an individual app).

The output of this phase is a privacy context file for each app. We can host these files in a dedicated server (or servers) for public sharing and retrieval. It would be ideal if Android app markets can add to each app page its privacy context file, along with other types of meta data (e.g., permissions, privacy policy, etc.) so that the privacy context file can be downloaded with the app at once.

### 3.3 Phase II: Application Installation Mediation

When users install an app, our system mediates the app installation process through “patching.” For instance, when a user downloads a third-party app from an app market or the Internet for installation, the APK file will be first passed to the PackageInstaller, which is the default application for Android to interactively install a normal package (step (a) in Figure 1). Once the user confirms the installation request, the PackageInstaller will call the PackageManagerService (a system service) through an Install-AppProgress activity (step (b)). Then, the PackageManagerService verifies the app, keeps the metadata and calls the Installation Mediator (IM) (step (c)).

Just before the app is optimized and transformed into the Android runtime oat format for later execution, the IM scans the app’s bytecode to find all the sensitive API invocations and their calling contexts. Note that although this simple scanning cannot track information flows or identify privacy leakage, it is able to identify the calling contexts of all the sensitive API invocations. A calling context includes the class and the method where each sensitive API invocation occurs and the zero-based index of its occurrence in that method. By matching calling contexts, our system will be able to apply the privacy context generated offline in Phase I to enforce privacy protection online in Phase III.

The Installation Mediator (IM) then rewrites each sensitive API of this app with an extra parameter, whose value is the calling context information, to uniquely identify the API. Note that the uniqueness is guaranteed since every JAVA method in an APK has its unique signature and the numbered index further orders the sensitive API invocations in the same JAVA method. For example, the getDeviceId() API (as in line 4 of Listing 2) will be rewritten to getDeviceId(context) where context is its calling context (i.e., the string in line 5 of Listing 2). The purpose is to distinguish the same type of sensitive API invocations in different calling contexts so that every sensitive API invocation can be uniquely identified at run-time. Later on, our Privacy Mediator will use such unique information for accurate runtime privacy policy enforcement.

For security reasons, in the patching process, we further encrypt the extra parameter (i.e., the calling context string) by a randomly-generated secret key. This will prevent an attacker from preparing a sophisticated APK pre-injected with proper context strings in an attempt to bypass our privacy enforcement scheme. For example, if an attacker intends to have a malicious sensitive API invocation (e.g., getDeviceId()) to bypass the privacy policy enforcement, he may find the calling context of a benign getDeviceId() invocation and rewrite the malicious getDeviceId() with getDeviceId(benignContext). This attack would allow the malicious API invocation to impersonate a benign one in our system. Adding encryption during the patching phase simply prevents this attack because the attacker cannot guess encryptedBenignContext correctly before the patching and supplying benignContext in plaintext will fail to pass the check of Privacy Mediator at runtime. Specifically, the IM applies the same secret key throughout the entire app (i.e., encrypting all the extra parameters) and sends the key associated with the app’s package name to the Policy Manager Service (PoMS).
After that, the PackageManagerService in Android will take over to optimize the modified bytecode by calling the native installd daemon and then notify the PackageInstaller with installation success or failure information (step 4). The output of this installation is a modified oat file, which is usually located at /data/​​dalvik-cache/ in the Android internal storage (step 5). Note that both PackageManagerService and PackageInstaller are from the Android framework, and we do not need to modify them.

We call this process of introducing the encrypted context information as an extra parameter in app’s bytecode as dynamic API patching. We implement the dynamic API patching on top of the open-sourced Apktool [6]. Different from conventional repackaging techniques, the integrated patching process in SweetDroid does not require third-party app repackaging and re-signing because the patching occurs after the PackageManagerService verifies the signature of the app. Hence, the whole API patching process is transparent to the user. Besides, the authorship of the app remains the same because of the shared UID feature of Android [15], and hence it is hassle-free during app updates [13].

3.4 Phase III: Privacy Policy Management and Enforcement

**Privacy Policy Management.** Once the app is successfully installed, the Policy Manager App (PMA) will download the aforementioned privacy context file for this app from a remote server (or app stores) and pass this file to Policy Manager Service (PoMS). PoMS is a system service to store and parse privacy context files and generate default privacy policy rules accordingly. It also provides APIs, which are accessible to PMA only, for managing privacy policies.

Specifically, PoMS stores the contents of privacy context files in a SQLite database. In addition, PoMS provides PMA with APIs to manage privacy policies, and in turn PMA provides a user-friendly interface for end users to view, comprehend and modify app’s privacy policies if needed. For example, users can easily turn on or off the location information access for advertisement providers in an app. We will show a case study in Section 5. Note that PMA, as an interface in an application layer, does not store any privacy policy. All operations, including privacy policy read and write, go through PoMS.

**Privacy Policy Enforcement.** During the execution of this app (i.e., when this oat file is running), whenever a patched sensitive API is invoked, the Privacy Mediator (PM) will handle the invocation by the means of hooking those overloading methods. It queries the PoMS for the policy rule for this API invocation by providing the app’s package name, the signature of the sensitive API and the encrypted calling context (the last parameter in the sensitive API invocation). The PoMS decrypts the encrypted calling context with the corresponding secret key previously received from the IM in the installation phase. Without knowing the encryption key, an attacker cannot generate a valid calling context. If the decryption is successful, it will return the corresponding policy rule. If none is matched, the PoMS returns “deny” by default. If the action is “deny”, the mediated API will return a null value. If the action is “mock”, the API will return manipulated data. Table 1 shows the types of protected sensitive data and the supported enforcement rules based on our current engineering efforts.

<table>
<thead>
<tr>
<th>Sensitive Data Types</th>
<th>Supported Enforcement Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phone Information</td>
<td>Allow, Mock, Deny</td>
</tr>
<tr>
<td>Location</td>
<td>Allow, Mock, Deny</td>
</tr>
<tr>
<td>SMS</td>
<td>Allow, Deny</td>
</tr>
<tr>
<td>Contacts</td>
<td>Allow, Deny</td>
</tr>
<tr>
<td>Call Logs</td>
<td>Allow, Deny</td>
</tr>
<tr>
<td>History &amp; Bookmarks</td>
<td>Allow, Deny</td>
</tr>
<tr>
<td>Accounts</td>
<td>Allow, Deny</td>
</tr>
</tbody>
</table>

Table 1: Protected Sensitive Data and Supported Enforcement Rules

4 SECURITY ANALYSIS

In this section, we discuss the security of SweetDroid. We focus on analyzing whether SweetDroid introduces new security problems or whether malicious attacks can evade SweetDroid. The study of other types of privacy leakage (e.g., caused by exploiting Android’s or JVM’s vulnerability) is out of scope. Specifically, we consider the following attack interfaces.

**Java Code.** SweetDroid relies on replacing app’s sensitive API invocations and adding sensitive API wrappers in Android framework to enforce privacy policies. SweetDroid implements these two components in Android framework, and hence they are not under the control of the app’s UID.

In an attempt to bypass SweetDroid, a malicious app may try to use Java reflection or dynamic class loading to invoke original sensitive APIs instead of rewritten ones. For example, a malicious app could download a Jar from a remote server, load a class in the Jar dynamically and invoke its method to access user’s sensitive information. However, the kind of attack can be easily detected because the Installation Mediator has statically replaced all sensitive API invocations in the app’s bytecode and the untouched sensitive API invocations must be from either Java reflection or dynamic class loading. To defeat the attack, we force original sensitive APIs to deny the data access if the caller UID belongs to a third-party app (since we do not rewrite any system app). To support a legitimate app to invoke sensitive APIs via Java reflection or dynamic class loading, we could add a privacy policy option for each app in PMA so that when users enable this option for a third-party app, this app is allowed to access sensitive data through original sensitive APIs.

Moreover, a malicious app could use Java reflection or dynamic class loading to invoke our rewritten sensitive APIs with a fake calling context parameter that matches an “allow” privacy policy rule. However, the malicious app cannot acquire the right calling context due to two reasons. First, the Installation Mediator encrypts the calling context with a randomly generated secret key and shares the secret key only with PoMS. That means that even if the malicious app acquires a legitimate calling context of a sensitive API invocation, it cannot produce a valid encrypted version. Second, the malicious app may try to look for a legitimate calling context directly from bytecode. For instance, it could read its APK file under /data/app/ to look for the legitimate calling context in runtime.
Note that the app does have the read access to its APK file in storage. However, the APK file is the original one, which was untouched by Installation Mediator. The file containing the rewritten APIs is the oat file under /data/dalvik-cache/ but the app has no access to read the file unless the app escalates to root privilege.

Native Code. In general, employing native code cannot bypass our privacy policy enforcement because SWEETDROID intercepts neither IPC nor system calls, but enforces privacy policies inside the Android framework. The malicious app may employ native code to tamper with the associated memory address space, but this cannot circumvent SWEETDROID for a privilege escalation.

We notice that more advanced native code based attacks may manage to read and scan the memory at runtime to retrieve an encrypted, benign calling context and then replay it for a malicious sensitive API invocation. As a result, although a malicious invocation might have been detected and labeled as non-justifiable by the Privacy Analyzer offline, at runtime it would still be allowed. While theoretically this attack is possible to a determined attacker, we have not seen any real-world attack cases based on this and the cost of implementing such a sophisticated attack could be too high. At this moment, we do not exactly know how to implement this attack, but probably lots of manual work is needed and it is unlikely for this attack to be fully automated. After all, even though an attacker may strive hard to evade SWEETDROID for his own malicious app, this does not prevent SWEETDROID from serving as a viable solution to defeat unintended privacy leakage caused by other apps. In our future work, we will further investigate on this attack.

Red Pill. SWEETDROID is not designed to be invisible to an untrusted app. The untrusted app can use Java reflection to deduce that the app’s code has been rewritten in runtime. Thus, a malicious app could hide its misbehavior and refuse to function in such a hardened environment. While it could harm usability of an app for end users, the malfunction cannot lead to a privilege escalation.

5 A CASE STUDY

In this section, we present a case study to illustrate the usage of SWEETDROID on a popular weather forecast app from the Anzhi app market (a leading Android app market in China). We can observe that SWEETDROID can distinguish the different calling contexts of sensitive information requests and thus enhance user privacy.

We start the case study by installing the weather forecast application in a Nexus 5 device. Figure 3 (in Appendix A) shows the requested permissions when the app is installed. It requests sensitive permissions to access users’ phone and location information. From user’s point of view, it is reasonable to grant the location information access since the app can provide weather forecast tailored to the user’s location. However, a user may have no clue on why the app needs to access phone information at the current stage.

Suppose the user knows nothing or little about privacy risk, so she simply grants the permission request. SWEETDROID installs the weather forecast app through our API patching process. Once the app is successfully installed, the Policy Manager App (PMA) will download the privacy context for this application from our server (its privacy context is generated by DroidJust [5]), and pass the privacy context file to the underlying Policy Manager Service (PoMS).

Figure 4 illustrates a simple user interface of PMA for displaying and modifying privacy policies for installed applications. We note that all user interfaces shown here are for research demonstration purposes, and in practice, for transparency to end users, we do not expect ordinary users to interact with them unless they are expert users who want to change rules. On the other hand, these user interfaces can certainly be improved in various aspects, but this is currently out of our scope.

The top left toggle in Figure 4 is used to switch between different apps. This figure particularly shows the privacy policy for the weather forecast app and the title is the app’s package name. The following is a list of sensitive information requests, each representing a sensitive information request at a specific calling context.

Figure 5 shows an example of a location information request. It contains the signature of the sensitive API, the signature of the API’s calling context, the analysis result imported from the privacy context file, and the current rule being applied to the sensitive information request. In the example, the location information request is used to fulfill app functions (based on the analysis result) and the default rule is “allow”.

Figure 6 shows another location information request, which is used by an advertisement library based on the offline analysis result. The default rule for the sensitive information request by advertisement libraries is “mock”. Figure 7 shows that the advertisement library also requests the device’s IMEI information. Note that although this case study has shown the case of privacy leakage by an ad library, SWEETDROID handles also malicious logic of the apps (our evaluation in the next section will show that a good percentage of apps leak sensitive information through their own logic without being justified).

Besides viewing the sensitive information requests in the PMA, the user can modify the policy rule for a sensitive information request at any time. Figure 8 shows an example of a popup menu after a long click on the rule that a user wants to modify. SWEETDROID provides three options including “allow”, “mock”, and “deny”. “allow” returns genuine sensitive information, “mock” returns fake sensitive information, and “deny” simply returns a null value. The “RECOMMENDED” flag shows the default rule based on the analysis result. After the modification, the rule takes effect immediately.

6 EVALUATION

We evaluated SWEETDROID on a collection of Android applications to show that (1) applications can be installed successfully and the added code does not impede the original functionality of applications, and (2) privacy policies can be generated automatically for a large percentage of applications. We conducted a broad evaluation, using 3760 apps from Anzhi and 2759 malicious apps from VirusTotal [16]. Note that we do not evaluate the effectiveness of runtime privacy policy enforcement here since the system design has clearly shown that sensitive API invocations are hardened by the dynamic API patching process and thus their privacy policies are always enforced at runtime.
<table>
<thead>
<tr>
<th>Source</th>
<th># of Apps</th>
<th>Unjustified Leak by App</th>
<th>Leak by Ads</th>
<th>Either</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anzhi</td>
<td>3760</td>
<td>488(13.0%)</td>
<td>1004(26.7%)</td>
<td>1254(33.4%)</td>
</tr>
<tr>
<td>VirusTotal</td>
<td>2759</td>
<td>887(32.1%)</td>
<td>199(7.2%)</td>
<td>989(35.8%)</td>
</tr>
</tbody>
</table>

Table 2: Privacy Leakage Analysis

<table>
<thead>
<tr>
<th>Source</th>
<th>Phone Info</th>
<th>Location</th>
<th>Messages</th>
<th>Contacts</th>
<th>Call Logs</th>
<th>Accounts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anzhi</td>
<td>1222</td>
<td>576</td>
<td>3</td>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>VirusTotal</td>
<td>719</td>
<td>79</td>
<td>457</td>
<td>102</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 3: Leaked Sensitive Information

6.1 Rewriting Evaluation

We first performed an evaluation to determine how many Android applications were successfully rewritten and installed by Sweat-Droid. Table 4 shows the success rate of our rewritten process.

<table>
<thead>
<tr>
<th>Source</th>
<th># of App</th>
<th>Success Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anzhi</td>
<td>3760</td>
<td>98.6%(3709)</td>
</tr>
<tr>
<td>VirusTotal</td>
<td>2759</td>
<td>95.9%(2646)</td>
</tr>
</tbody>
</table>

Table 4: Patching Evaluation Results

We have over 95% success rate in application rewriting. The failures in rewriting arbitrary applications are due to errors in apktool when disassembling APK files (e.g., error in decoding app resource files, error in opening zip files and invalid magic number in decoding). We are trying to work on improving apktool to achieve a higher success rate.

6.2 Size Overhead

We also evaluated the app size increase due to Sweat-Droid’s rewriting process, as shown in Figure 2. On average, Sweat-Droid increases the application size by only 11.6 KB, which is a very small overhead for the majority of applications.

![Figure 2: Application Size Increase After Repackaging](image)

6.3 Privacy Policy Generation

We employed DroidJust [5] to automatically analyze privacy leakage and generate privacy policies for those Android applications. Table 2 shows the results, where the third column shows the number of apps that contain unintended sensitive information leak in their application logic, the fourth column shows the number of apps that contain sensitive information leak in their advertisement logic, and the last column shows the number of apps. The results demonstrate that more than 30% apps from both Anzhi and VirusTotal have privacy leakage issues, which exposes a serious privacy threat.

We note that these two app datasets have a significant difference in the leaking areas. Particularly, most privacy-invasive apps from Anzhi leak users’ sensitive information through advertisement libraries rather than application logic, whereas nearly all privacy-invasive malicious apps (887 of 989) from VirusTotal leak users’ sensitive information directly in their application logic.

There are two important observations. The first observation is that advertisement libraries are the major cause of users’ sensitive information leakage since most advertisement libraries send out users’ sensitive information such as IMEI and location for location-based or targeted advertising. The second observation is that current anti-virus vendors determine a privacy-invasive app as malware or not based on the fact if an app leaks user privacy through their application logic rather than through advertisement logic. Table 3 shows a breakdown of the leaked sensitive information for those apps. On the one hand, we can see that the privacy-invasive apps from Anzhi mostly leak users’ phone information (e.g., IMEI, IMSI, phone number, etc.) and locations but rarely leak other kinds of information such as messages and contacts. It validates our previous observation that the major cause of the leaks in this dataset is for advertising purpose. On the other hand, we can see that the privacy-invasive apps from VirusTotal not only leak phone and location information but also frequently leak messages and contacts. Based on our study, we found that those malware leak contacts and messages for malicious attacks. For example, harvesting user contacts is used for message or email spam, and a malware stealthily calling premium numbers usually intercepts and forwards the subscription messages to its malicious server.

The above analysis results indicate that a fine-grained access control mechanism like SweetDroid is highly desirable for strong privacy protection of Android app users. SweetDroid handles not
only the privacy leakage issue caused by third-party libraries, but also malicious logic of the apps.

6.4 Performance Evaluation

We employed two most popular Android benchmark applications from the official market and applied SweetDroid to them in order to check the performance overhead that SweetDroid introduces to a real Android device. From Table 5, we can see that the benchmark scores are largely unaffected by SweetDroid. Note that higher scores mean better performance.

Because SweetDroid introduces the most overhead when an application performs sensitive API invocations, we further evaluate an artificial app’s runtime delay caused by SweetDroid when it invokes sensitive APIs. The artificial app calls each sensitive API invocations for 1000 times. Results in Table 6 show that SweetDroid introduces an overhead of 13-15% in both cases. We believe that the incurred overhead is acceptable since the averaging delay for each sensitive API invocation is less than 0.3 ms and such a short time interval will not affect user experience.

7 DISCUSSIONS

In this section, we discuss two limitations of our system.

SweetDroid currently leverages the existing program analysis tools [5, 12, 25, 29, 30] to generate privacy contexts automatically. Hence, it inherits the limitations of these tools, that is, their imperfection in analyzing programs and understanding the intention of information flows. In our future work we will try to improve the accuracy and efficiency of automatic information flow-based privacy policy generation for large-scale app analysis.

In our framework, we define calling context as an N-depth call stack when the sensitive API function is called. In our implementation, however, we set N = 1 for simplicity. As a result, if multiple sensitive information flows (if any) reach the same sensitive API through the same calling context (i.e., the same class and method call where the sensitive API is located) and go to the same network sink, we will not be able to differentiate among them. As long as one is justified, all of them are justified. One potential concern is that whether an attacker may exploit this limitation to get a free ride for his malicious flows, that is, causing false negatives for our system. While this is theoretically possible, in practice this attack is not likely to happen for the following reasons. First, if the attacker is a malicious app developer, he only needs to obtain the sensitive information once through a single justifiable flow. In other words, he will have no motive to introduce additional flows to steal the same sensitive information. Note that if a free-riding flow goes to a different network sink, it will be considered as a different flow and hence need to be justified separately. In this case, free-riding does not help evade detection. Second, while third-party libraries may invoke sensitive APIs directly in their own code, they cannot invoke the APIs in the calling context of the host app. On the other hand, setting N = 1 will not introduce false positives to our system (i.e., justifiable flows become unjustifiable). Due to the above reasons (simplicity without hurting security), we consider N = 1 as the best choice.

8 RELATED WORK

We categorize the previous work on privacy protection and enhancement on smartphones into the following categories based on their primary mechanisms.

Permission Model Extension and Enforcement. Most past research works enhance user privacy by either modifying Android source code [3, 7, 19, 20, 31] or rewriting app code [2, 8, 9, 11, 17, 28] to extend and enforce Android permission model. Apex allows users to selectively grant permissions to applications and restrict the usage of resources [19]. Introducing a privacy mode, TISSA empowers users to flexibly control what kinds of personal information are accessible to an application [31]. CRePe enforces fine-grained permission policies based on the contextual information of the mobile device such as time, location and user interaction [7]. Saint provides a security infrastructure that governs install-time permission assignment and enforce runtime application-centric security policies [20]. MockDroid modifies Android framework to allow users to mock an app’s access to a resource [3].

I-ARM-Droid rewrites app’s bytecode to interpose on the invocations of sensitive API methods in order to enforce desired security policies. [9]. RetroSkeleton is an app rewriting framework that supports retrofitting of app’s behaviors by statically and dynamically inception of method invocations [8]. AppGuard rewrites and repackages an app on the phone to mediate security-relevant methods [2]. Aurasium enforces user-defined policies by rewriting an app and low-level libc.so [28]. [17] provides fine-grained permissions on resource accessing by supporting parameterized permissions. AppFence enables users to feed shadow data to apps in place of data that users want to keep private and block data exfiltration [11]. Additionally, Boxify uses full-fledged app sandboxing

<table>
<thead>
<tr>
<th>Benchmark App</th>
<th>Without SweetDroid</th>
<th>With SweetDroid</th>
</tr>
</thead>
<tbody>
<tr>
<td>AnTuTu Benchmark (5.7.1)</td>
<td>35621 Pts</td>
<td>35527 Pts</td>
</tr>
<tr>
<td>Geekbench 3 (3.3.2)</td>
<td>786 Pts</td>
<td>780 Pts</td>
</tr>
</tbody>
</table>

Table 5: Performance on Benchmark Applications

<table>
<thead>
<tr>
<th>1000 API invocations</th>
<th>Without SweetDroid</th>
<th>With SweetDroid</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get Device ID</td>
<td>1786 ms</td>
<td>2022 ms</td>
<td>13.2%</td>
</tr>
<tr>
<td>Get Last Location</td>
<td>278 ms</td>
<td>318 ms</td>
<td>14.4%</td>
</tr>
</tbody>
</table>

Table 6: Performance on API invocations
to enforce security and privacy policies without modifying Android source code or app code [1].

**Mandatory Access Control.** Another way to enhance access control is to use Security Enhanced Linux (SELinux)[24], the most prominent mandatory access control solution, on Android. Several research works support mandatory access control on both Android’s middle and kernel layers [4, 23].

However, nearly all existing approaches in these two categories heavily rely on user-defined security or privacy policies to prevent privacy violations, which is not realistic for normal users since they lack professional knowledge on security. Besides, the contextual information used in the previous policies is too coarse-grained and/or heuristic-oriented that could interfere with user-desired functionality [11]. In contrast, the generated privacy policies by our framework distinguish sensitive API invocations in an app and handle them differently according to analysis results. It achieves the sweet point to enhance user privacy while retaining app functionality.

**Permission Separation for Libraries.** There are a few research works focusing on restricting permissions for one component [26] or a third-party library [18, 21, 22]. Compac distributes a narrowed set of permissions into every component and hence enforces access control at component level [26]. AdSplit extends Android to allow an application and its advertising libraries to run as separate processes (under separate user-ids) in order to separate permission sets for different domains [22]. PEDAL uses a novel machine learning classifier to detect ad libraries and rewrite app’s bytecode for privilege de-escalation [18]. AdDroid separates privileged advertising functionality from host applications, allowing apps to show advertisements without requesting privacy-sensitive permissions [21]. Though these approaches can restrict access control at component or library level to eliminate unnecessary sensitive data access (e.g., for advertisement), they cannot handle overly-curious or privacy-invasive apps because there is no clear boundary between the privacy-violating part and the user-defined function part.

9 CONCLUSION

In this work, we proposed and implemented SweetDroid, a calling-context-sensitive privacy policy enforcement framework. SweetDroid integrates automated policy generation and automated policy enforcement. By considering calling contexts, SweetDroid is able to tell if a particular data sensitive API (e.g., getKnownLoc()) under certain execution path is leaking private information. We designed a developer hassle-free scheme based on dynamic patching to rewrite the applications at runtime. The system also handles the policy configuration and enforcement automatically, so it is transparent to the end users who may know nothing about privacy. The experiment result demonstrates that our system can effectively patch various apps, including apps from a third party market and malicious apps from VirusTotal. It only incurs small storage and computational overhead.

ACKNOWLEDGMENTS

The work of Zhu was supported through NSF CNS-1618684. The work of Quanlong Guan was supported by NSFC (61602210), the Science and Technology Planning Project of Guangdong Province, China (2014A040401027, 2015A030401043), the Fundamental Research Funds for the Central Universities (21617408).
A FIGURES

Figure 3: Permission request
Figure 4: Policy manager app
Figure 5: Location req. by app

Figure 6: Location req. by lib
Figure 7: IMEI request by lib
Figure 8: Modify current rule
REFERENCES


