Computer Science and Information Systems 15(1):111–137 https://doi.org/10.2298/CSIS170728047C

Prevention of Cross-update Privacy Leaks on Android

Beumjin Cho[1], Sangho Lee[2], Meng Xu[2], Sangwoo Ji[1], Taesoo Kim[2], and Jong Kim[1]

[1] Department of Computer Science and Engineering, Pohang University of Science and Technology (POSTECH), Pohang, Republic of Korea
beumjincho, sangwooji, jkim@postech.ac.kr
[2] School of Computer Science, Georgia Institute of Technology, Atlanta, GA, USA
sangho, meng.xu, taesoo@gatech.edu

Abstract. Updating applications is an important mechanism to enhance their availability, functionality, and security. However, without careful considerations, application updates can bring other security problems. In this paper, we consider a novel attack that exploits application updates on Android: a cross-update privacy-leak attack called COUPLE. The COUPLE attack allows an application to secretly leak sensitive data through the cross-update interaction between its old and new versions; each version only has permissions and logic for either data collection or transmission to evade detection. We implement a runtime security system, BREAKUP, that prevents cross-update sensitive data transactions by tracking permission-use histories of individual applications. Evaluation results show that BREAKUP's time overhead is below 5%. We further show the feasibility of the COUPLE attack by analyzing the versions of 2,009 applications (28,682 APKs).

Keywords: Android, Privacy, Information flow, Permission.

1. Introduction

Android has become one of the most valuable attack targets due to its popularity. The global market share of Android is over 80% [28], while 99% of mobile malware targets Android [46]. Most mobile malware aims to steal a huge amount of sensitive user data kept in Android devices, e.g., emails, contacts, and photos. Therefore, we should develop effective solutions to protect such private data from attackers.

To prevent various attacks, Android adopts a permission-based mechanism that restricts the capabilities of an application pursuant to the application specification and a user's approval. The security mechanism specifies whether an application can access sensitive user data (e.g., contacts, SMS, or call logs) or system resources (e.g., files, network sockets, or microphones). Users who worry about their security and privacy would not install an application that unnecessarily demands critical permissions [25] or use security extensions, such as Kirin [19], to enforce additional security policies at installation.

However, restricting the capabilities of individual applications still has security problems because a malicious application can indirectly escalate its privilege by using two kinds of application-level privilege-escalation attacks: *confused deputy* [17, 27, 60] and *collusion attacks* [49, 37]. In a confused deputy attack, a malicious application exploits the public interfaces of a vulnerable application to use the permissions granted to the vulnerable application. In a collusion attack, multiple malicious applications collude to combine their restricted permissions (e.g., they can access either private data or system resources)

to leak privileged data and resources. The permission-based security mechanism cannot detect either attack because each application has restricted permissions. Thus, some researchers propose detection systems that restrict inter-component communication (ICC) between different applications according to their permissions and security policies [24, 18, 55, 10].

To overcome the limitations of the permission-based mechanism, many researchers propose *information-flow tracking* methods for Android that monitors how *tainted data* is delivered from a source to a sink within an application or across different applications. Static information-flow tracking [42, 7, 52, 12, 35] analyzes application code to artificially explore every data-leak path, whereas dynamic information-flow tracking [20, 29, 16, 56] runs applications in a monitored environment to track real data flow. Although both static and dynamic approaches have drawbacks (e.g., implicit flow, native code, and emulation awareness), they are the state-of-the-art approaches to detect possible privacy leaks.

In this paper, we depict a novel attack that exploits application updates: a *CrOss-Update Privacy-LEak attack*, called COUPLE in short. The COUPLE attack separates necessary permissions and logic to leak private data across different versions of the same application, and let them collude together to bypass popular Android security mechanisms through the application update mechanism of Google Play. For example, if an old version of an application has the `READ_SMS` permission to access the user's SMSs, and the new version has the `INTERNET` permission, but without `READ_SMS`, then the application can exfiltrate the previously read SMSs outside the user's device. Through this implicit information channel across application boundary, COUPLE can bypass virtually all the proposed Android security mechanisms, as far as we know (Section 4).

We introduce three COUPLE attacks according to their channels and functionalities: *direct*, *indirect*, and *concurrent*. First, in a direct COUPLE attack, the updated version of a malicious application directly communicates with *a Linux process* spawned from its old version via the `fork()` system call, storing collected sensitive data in memory. The forked process can stay alive even after an application update because it is outside the management of the Activity Manager (Section 3.1).

Second, in an indirect COUPLE attack, the updated version of a malicious application accesses its *private resources* (e.g., a file or database (DB) within the application's internal storage) that contain sensitive data collected by an older version of the application. The private resources remain after an application update because the Android system tries to prevent possible damage to such resources when updating applications [54].

Third, unlike direct and indirect COUPLE attacks that are sequential, a concurrent COUPLE attack allows an adversary to simultaneously collect and send out sensitive data with only permissions for data collection. An adversary can use this attack to steal time-limited data, such as a one-time password (OTP) via SMS and the current location. This attack leverages a time of check to time of use (TOCTTOU) problem when using an important component of the Android system: a network socket. We identify that the Android system checks the `INTERNET` permission when creating a socket, but does not check the permission when using the created socket to establish connection or to exchange data.

Existing detection methods have difficulties in determining the malice of an application conducting the COUPLE attack, since they only inspect a snapshot of each Android application or system. No detection method (1) considers the correlation between permission request changes at updates, (2) simultaneously analyzes multiple versions of ap-

plication code to discover separated flow, and (3) continuously runs an application in a monitored environment across application updates. They should accept considerable computation and storage overhead to satisfy the three requirements for detecting the COUPLE attacks with *dummy updates* (Section 7).

To mitigate the COUPLE attacks, we develop a lightweight security extension, called BREAKUP, that inspects information-flow across application updates. BREAKUP's method is similar to information-flow tracking mechanisms [58, 33, 32] in spirit, but it only considers how applications use permissions to minimize system overhead. It maintains permission-use histories associated with processes and private resources of individual third-party applications. To prevent the COUPLE attacks at application updates, BREAKUP uses the information to kill processes and remove private resources that possibly leak sensitive information. In addition, we added some code to the Linux kernel for Android to eliminate the TOCTTOU problem of network sockets.

This work achieves the following contributions:

- **New security problem.** To the best of our knowledge, this is the first study that considers security problems of application updates on Android.
- **Stealthy attack.** The COUPLE attack is stealthy; its malicious data flows and behaviors are separated across version updates. Analyzing individual versions is insufficient to reveal the suspiciousness of applications conducting the COUPLE attack.
- **Efficient countermeasure.** We develop a lightweight solution against the COUPLE attacks with a time overhead below 5%.
- **Large-scale measurement study.** We survey all available versions, from the original version to the latest, of 2,009 different applications (28,682 versions in total) to study the feasibility of the COUPLE attacks in terms of the frequency of application updates and permission changes.

The remainder of this paper is organized as follows. Section 3 presents some background information and our assumption. Section 4 depict the details of the COUPLE attacks. Section 5 describes our countermeasure against the COUPLE attacks. Section 6 analyzes applications collected from Google Play. Section 7 discusses server-side countermeasures against the COUPLE attacks. Section 2 introduces related studies of our work. Lastly, Section 8 concludes this paper.

2. Related Work

In this section, we discuss prior studies of existing threats as well as Android security enhancements related with our work.

2.1. Threats

We first discuss closest related threats to COUPLE attack in terms of their strategies such as how modules of malware collaborate with each other, and how an attack is deployed in timely manner to break or circumvent the security.

Application-level privilege escalation. Researchers have considered two application-level privilege-escalation attacks: confused deputy attack and collusion attack. To conduct a confused deputy attack, a malicious application should discover vulnerable applications that open public interfaces. Detection systems for the confused deputy attack analyze each

application to know whether it has such public interfaces, statically [27, 60, 14, 36, 13] or dynamically [24, 18, 55]. To conduct a collusion attack, more than one application should create a channel for combining their restricted permissions. XManDroid [10] monitors all communication channels in the Android system to detect confused deputy and collusion attacks by using corresponding permissions and intents.

Attack through system-wide update. We explain Pileup [54] that considers updates as our work did. Pileup is a security vulnerability that leverages Android platform updates to escalate privilege. In Pileup, a malicious application requests security-critical permissions that are newly defined in a new version of Android, but are not defined in old versions, also called dormant permissions [50]. When a user who uses an old version of Android installs such an application, the Android system allows installation because the system knows nothing about the permissions. Later, when the user installs the new version of Android, the malicious application can use the granted permissions to perform sensitive operations. Although Pileup is a serious and critical security problem, demanding a platform update for attacks is a strong assumption.

Dynamic code loading. Dynamic code loading is a well-known technique to circumvent malware detection systems [45]. With it, a malicious application can download and execute malicious logic after it has passed inspections. Since the COUPLE attacks also change their behaviors via updates, they resemble each other. However, unlike the COUPLE attack, dynamic code loading cannot change permission requests or use Google Play to host malicious logic. Therefore, it should request all dangerous permissions at installation, which can be prohibited or suspected by security systems.

2.2. Security Enhancements

Next, we discuss security enhancements. Due to the ever growing popularity of malware and its diversity, a huge number of monitor, detection, and prevention systems have been proposed.

Permission-based security enhancement. Security researchers have considered how to fortify and extend the permission-based security model of Android. Kirin [19] modifies the Android application installer to restrict application installation according to user-defined security rules. Instead of preventing installation, Apex [41] allows users to selectively grant permissions when installing applications. Saint [43] allows application developers to define security policies to access the public interfaces of their applications. CrePE [15] extends the Android permission model to be aware of the context of a device, such as location and time. Lastly, Dr. Android and Mr. Hide [31] provides fine-grained permissions using an inline reference monitor.

System-level security enhancement. We discuss DroidBarrier [2] and FireDroid [48]. DroidBarrier provides a security model for ensuring strong authentication for application processes. Although the security model can identify processes spawned without going through Zygote, it cannot prevent COUPLE because the forked process only conveys collected data to the next version of the application without directly accessing the boundary of the protected application. FireDroid implements a policy-based system call interposition mechanism. However, this mechanism lacks information about different versions of applications thus cannot properly prevent COUPLE.

Information-flow tracking. Numerous researchers develop static and dynamic taint analysis tools that track information flow within an application or across applications to detect unauthorized leakage of sensitive data. In theory, static taint analysis [42, 7, 52, 12, 35] can explore all data leakage paths. However, in practice, this method has limitations: investigators cannot fully obtain the source code of a malicious application and it takes much time when analyzing complex source code. In contrast, dynamic taint analysis [20, 29, 56, 16] demands no source code and only inspects the paths that an application has accessed during execution. However, an adversary can develop a malicious application that recognizes whether dynamic analysis is performed and behave as a legitimate application to evade analysis. In addition, some researchers refine information-flow tracking by using user intention [57], permission-use events [59], or user interface workflow [39]. Lastly, most state-of-the-art dynamic tracking systems [21, 9] provides interfaces to inject code to track or monitor applications and processes without any system modification.

Advanced malware detection. Recent works extensively show studies of malware detection with promising results. Feature-based malware detection scheme [47, 44] uses main characteristics of real-world malware to provide fast and reliable detection and classification. More advanced works employs Hidden Markov model to effectively detect and classify malware [38, 11]. Some researches focus on intents as a feature to distinguish malicious applications form benign ones [30, 22]. Lastly, a work provides native code level malware detection [1]. Despite the numerous efforts in this area, none of the schemes can detect COUPLE because of its unique attack pattern.

3. Background and Assumption

In this section, we explain some characteristics of Android exploited to develop the COUPLE attack. We also explain our assumption.

3.1. Process Management and fork()

We briefly explain how Android manages processes and how we can create an unmanaged process by calling the `fork()` system call. When starting an application, the Android system first contacts the Activity Manager to check whether it should create a new process to launch the application. If no process is executing the application, the system requests Zygote to fork a new process, which contains a Dalvik virtual machine (VM) instance and preloaded classes [34]. The Activity Manager then loads a main thread (ActivityThread) on the created process. The thread starts some components, e.g., activity and service, and manages their life cycles by registering itself to the Activity Manager and communicating with the service via a binder remote procedure call (RPC).

A new process should be registered to the Activity Manager to manage its life cycle. However, when an application directly creates a process by calling the `fork()` system call via the native development kit (NDK), the forked process is not registered to the Activity Manager service such that the Android system has no mean to manage it.[3] We observe that the forked process remains alive when a user or the system kills the corresponding application, regardless of the way it was killed (e.g., swiping away on the recent

[3] `https://groups.google.com/forum/#!topic/android-ndk/sjIiMsLkHCM`

Table 1. Example of Google Play's permission groups and their permissions

Group	Permissions
Calendar	READ_CALENDAR, WRITE_CALENDAR
Contacts	READ_CONTACTS, WRITE_CONTACTS
Device/App History	GET_TASKS, READ_HISTORY_BOOKMARKS
Device ID, Call Info.	READ_PHONE_STATE
Identity	GET_ACCOUNTS, MANAGE_ACCOUNTS, READ_PROFILE, WRITE_PROFILE
Location	ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION
Phone	CALL_PHONE, PROCESS_OUTGOING_CALLS, READ_CALL_LOG, WRITE_CALL_LOG
Photos/Media/Files	READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE
SMS	READ_SMS, RECEIVE_MMS, RECEIVE_SMS, RECEIVE_WAP_PUSH, SEND_SMS, WRITE_SMS
Wi-Fi Conn. Info.	ACCESS_WIFI_STATE
Other	ADD_VOICEMAIL, ACCESS_WIFI_STATE, BLUETOOTH, CHANGE_WIFI_STATE, CLEAR_APP_CACHE, GET_PACKAGE_SIZE, INSTALL_SHORTCUT, INTERNET, NFC, RECEIVE_BOOT_COMPLETED, ···

task list or touching a kill button in application setting). Moreover, the forked process is alive even after the Android system updates or replaces the application, whereas other legitimate processes of the application terminate. To abort the forked process, a user must use an adb shell to send a kill command or reboot the device.

3.2. Binder/Intent and fork()

A forked process has an important limitation: it cannot use binder or intent.[4] A process should have binder helper threads to use binder or intent. However, a forked process cannot have them, because the `fork()` system call only duplicates the current thread of the process that initiates it. Furthermore, to send an intent, an application needs to use a binder. Consequently, a forked process demands other methods to interact with other applications, such as using the command-line tool `am`.

3.3. Auto-update and Permission Group

Auto-update. Updating an application to the latest version is important for enjoying new functions, avoiding crashes due to bugs, and preventing an adversary from exploiting security vulnerabilities. To provide up-to-date applications to users, Google Play supports an option "auto-update apps" by default to automatically update installed applications. The option allows the user's device to update installed applications when new versions are available on Google Play.

However, an auto-update can be an attack channel because a user cannot check how an application has changed before updating. If an adversary adds sensitive permission requests when developing a new version and uploads it to Google Play, a user's device

[4] http://stackoverflow.com/questions/26309046/android-native-code-fork-has-issues-with-ipc-binder

would automatically update the application without user confirmation if no security check policy exists.

Google Play has a strict policy to avoid the security problem of auto-updates. It stops an auto-update when the permissions requested by an updated application differ from the permissions granted to the old version of the application. Google Play then asks whether the user wants to update the application by displaying the changed permission requests. Although the explained policy mitigates the security problem, it can annoy Android users by compelling them to frequently check updated applications.

Permission group. Recently, Google Play introduced permission groups that simplify the Android permission model by grouping similar permissions, which further simplify auto-updates [26]. When a user installs an application, instead of displaying the individual permissions the application requests, Google Play displays the permission groups to which the requested permissions belong. For example, when an application requests the `READ_SMS` permission only, Google Play shows that the application requests the permission group `SMS` to use SMS, MMS, or both.

We observe that Google Play's permission groups differ from the Android permission groups defined in `Manifest.permission_group` [4]. For example, `Manifest.permission_group` assigns the two permissions `READ_CALL_LOG` and `READ_CONTACTS` to the same group `SOCIAL_INFO`, but Google Play assigns the `READ_CALL_LOG` permission to the `Phone` group and the `READ_CONTACTS` to the `Contacts` group. To identify how Google Play organizes permission groups, we develop a simple application that demands all normal and dangerous permissions defined in `Manifest.permission` [3], upload it to Google Play, and check its permission groups. Table 1 is a part of the result obtained with Google Play version 5.4.12.

When a new version of an application additionally requests a permission that belongs to a permission group already granted to the old version, Google Play automatically updates the application without user confirmation. If an installed application has obtained the `READ_CONTACTS` permission and its updated version additionally requests the `WRITE_CONTACTS` permission, Google Play automatically updates the application.

However, security researchers have criticized the concept of permission groups because it allows an application to become a malicious application silently [53]. For example, the latest version of Facebook (33.0.0.45.19) has the `READ_SMS` permission. Thus, at the next update, it can obtain the `WRITE_SMS` permission without user confirmation to modify or delete SMSs stored in a user device. We believe that one possible solution to this problem is preventing critical permissions, e.g., `CALL_PHONE` and `SEND_SMS`, from being auto-updated.

3.4. Assumption

We assume a victim user who turns on the "auto-update apps" option or frequently updates a malicious application installed on the user's device. An application update is an essential process of the COUPLE attack, and an auto-update guarantees the malicious application that conducts the COUPLE attack to be updated. Google Play turns the auto-update option on by default [26], and we expect that many Android users keep the option. Note that the auto-update is only necessary to increase the probability of successful attacks. The COUPLE attack is possible even when a victim manually updates applications.

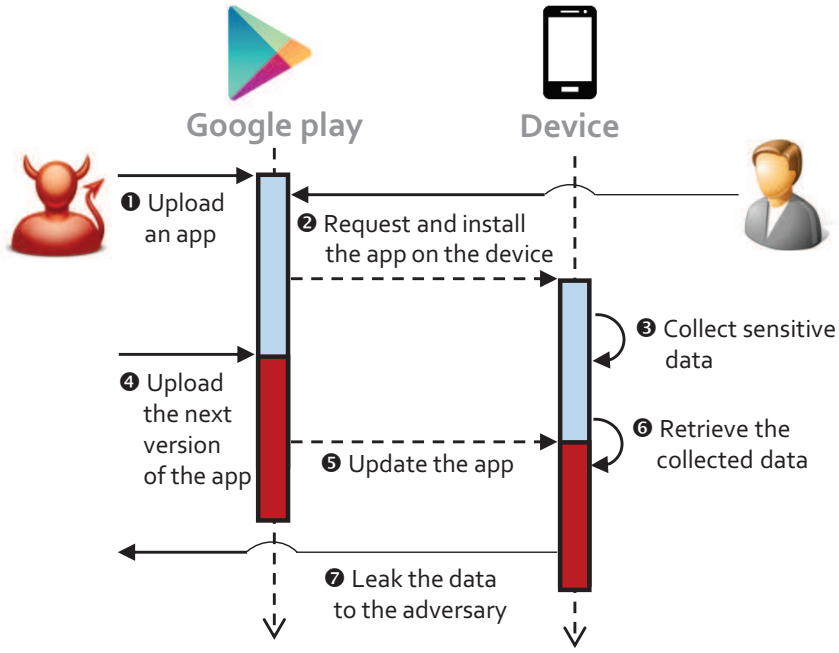


Fig. 1. Overview of the COUPLE attack

4. Couple Attacks

In this section, we describe the COUPLE attack that old and new versions of a malicious application collude with each other to leak private data. We show three types of the COUPLE attacks based on the method that connects the two versions of the application: direct, indirect, and concurrent.

4.1. Overview

Fig. 1 illustrates the work flow of a simple COUPLE attack. An adversary first separates the permissions and logic of a malicious application into two versions (an example shown in Fig. 2) and uploads the first version of the malicious application for collecting sensitive data to Google Play (❶). A user requests and installs the application on the user's device through Google Play (❷). Once the user starts the installed application, the application gathers data corresponding to the granted permissions and waits until the next version of the application is installed on the system (❸). The adversary uploads the second version of the malicious application for sending out data (❹), and the system updates and runs the second version (❺). At last, the second version retrieves the data collected by the first version (❻) and sends the collected data to the adversary (❼).

We classify the COUPLE attacks into *direct* COUPLE, *indirect* COUPLE, and *concurrent* COUPLE attacks. Table 2 summarizes three types of COUPLE in terms of attack channels, secrecy, and reliability, and the freshness of data that the attacks can leak. We will explain the details of each in the following sections.

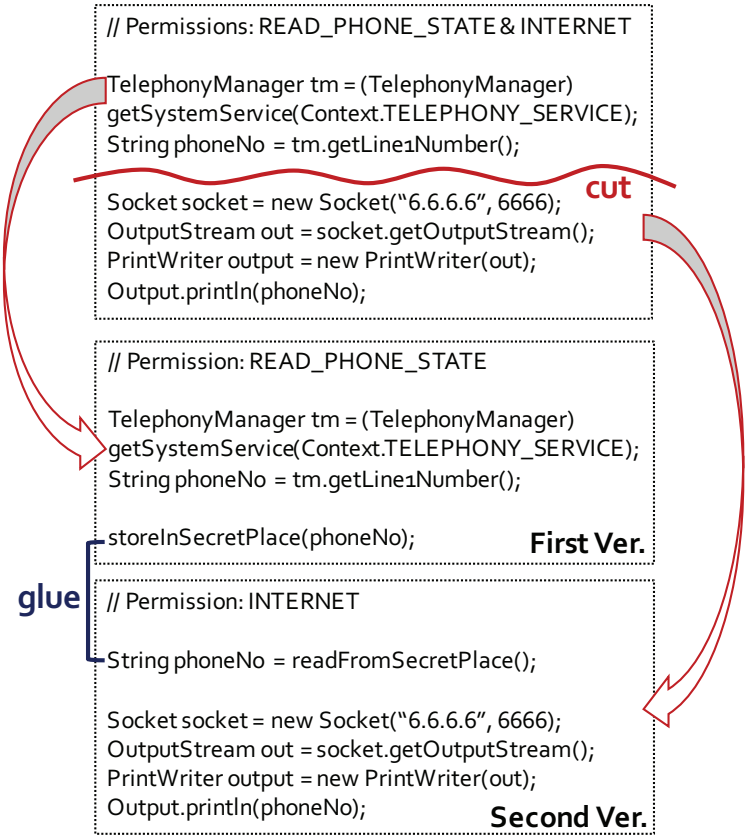


Fig. 2. Separating the permissions and logic of a malicious application into two versions. The first version only has permissions and logic for data collection, whereas the second version only has permissions and logic for data transmission.

4.2. Direct Attack

We introduce a direct COUPLE attack that is based on direct interaction between a process from an old version of a malicious application and a process from a new version.

Overall procedure. We explain the overall procedure of a direct COUPLE attack with an example described in Fig. 3. When the Android system installs and executes the first version of a malicious application developed by an adversary, the application starts a service that collects sensitive data (e.g., a phone number) with the granted permissions (`READ_PHONE_STATE`) (❶) and stores the data in its memory. The application service then calls the `fork()` system call to create a child process that is an exact copy of it, i.e., the child process also has a copy of the collected data in memory (❸). The adversary uploads the second version of the application to Google Play, which only requests the `INTERNET` permission and has logic to use network sockets (❹). The system automatically updates the application while killing every process of the old version except the forked process. Since the update, the forked process has no longer been able to collect

Table 2. Comparison between three types of COUPLE

Type	Channel	Secrecy	Reliability	Data freshness
Direct	Process	High	Low	Low
Indirect	File	Low	High	Low
Concurrent	Process+Socket	High	Low	High

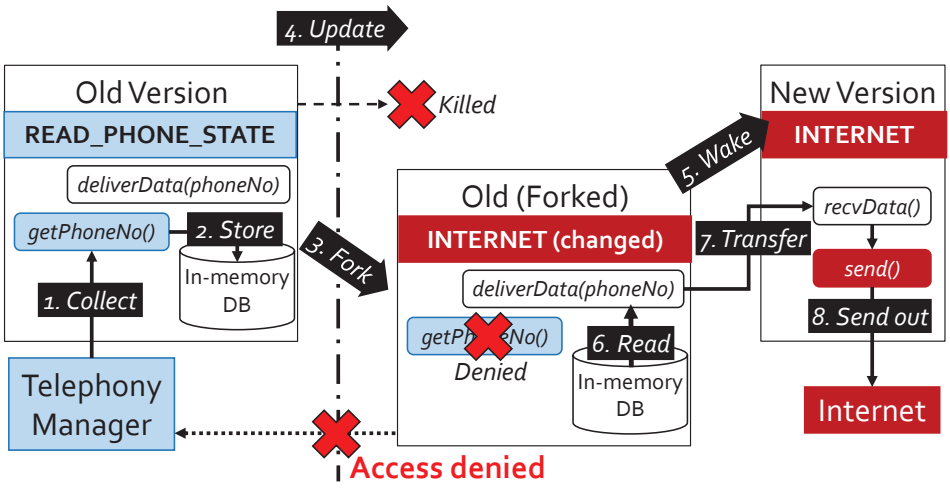


Fig. 3. Overview of a direct COUPLE attack

sensitive data because it now has no permission for data collection. The forked process wakes a service of the second version of the application (❺), reads the stored data from memory (❻), and conveys the data to the waken service via inter-process communication (IPC) (❼). Lastly, the service of the second version sends the received data to the adversary's server (❽).

Identifying update and initiating new version. To deliver data as soon as possible to a new version, a forked process of a malicious application should identify when the malicious application is updated and should activate the new version. First, the forked process can identify whether the application has been updated by inspecting specific files in the application's internal storage. For example, an Android application has a symbolic link `/data/data/<package name>/lib` to a directory `/data/app-lib/<package name>-*` that contains custom library files for the application. We observe that '`*`' of the directory's name is switched between 1 and 2 whenever the application is updated to avoid crashes between old and new library files. So, by monitoring the changes of `lib`, the forked process can know that the system has updated the application such that the next version is available on the device.

Next, the forked process can initiate the new version by sending an intent via the command-line tool `am`. In Android, a newly installed or updated application is in a stopped state such that it cannot receive any messages. The forked process should either wait until a user runs the new version or wake the new version by sending an intent. However, it

cannot directly send any intent due to its imperfect binder information (Section 3.2). To work around this problem, we let the forked process use `am` that supports various system actions, such as start an activity or a service, force-stop a process, and broadcast an intent. Executing the command "`am startservice -n <package name>/<service name>`" via the `system()` function can start the new version's service.

IPC method. An adversary needs some IPC methods to connect old and new versions of a malicious application. We explain three IPC methods used to develop the COUPLE attacks: intent, Unix domain socket, and Android shared memory (ashmem). First, an intent is the default IPC method of the Android system. It allows an application to launch the components of other applications, such as activities and services, while sending data. In a direct COUPLE attack, as explained in the prior section, the forked process uses the command-line tool `am` to send an intent that contains the collected data to a service of the new version. Then, the Activity Manager service of the Android system finds the target service and invokes the service's `onStartCommand()` method. Lastly, the service extracts data embedded in the received intent.

Second, a Unix domain socket is a facility to exchange data for processes on the same system. A direct COUPLE attack uses a Unix domain socket for exchanging data between the forked process from the first version of the application and the second version of the application. Specifically, the forked process creates a Unix domain socket before waking a service in the second version. Right after the service has run, it prepares another socket to communicate with the forked process. As soon as communication between the forked process and the service is established, the forked process transfers security sensitive data stored in its memory to the service.

Third, ashmem allows a process to create a memory region that can be shared with other processes. To share the memory region, processes should know its name and a file descriptor to access it. An adversary's strategy is that the first version of the application creates a shared memory region to store data to which the application aims. When the second version of the application runs, the forked process sends the file descriptor to the second version via Unix domain socket connection. The second version can then access the shared memory with the region's name and the received file descriptor.

4.3. Stealthy Direct Attack

Preventing the direct COUPLE attack explained in the prior section is relatively easy. During an application update, a monitoring system can determine which processes are possibly involved in a direct COUPLE attack according to their *same user ID (UID) or process name*. The system then will kill such processes to prevent possible COUPLE attacks.

However, in this section, we will show how an adversary can *assign different UIDs and names* to each of the processes involved in a direct COUPLE attack. This makes the COUPLE attack stealthy such that we demand a countermeasure that will be explained in Section 5.

Changing UID. First, by using an *isolated process*, an adversary can create a process whose UID differs from its parent process. An isolated process is a new feature of Android that allows an application to create a *zero-permission* process to run an untrusted component [8]. The UID of an isolated process differs from its parent process and is chosen from a pool of isolated UIDs. An application can isolate its service by adding an `android:isolatedProcess="true"` attribute to the service in its manifest file.

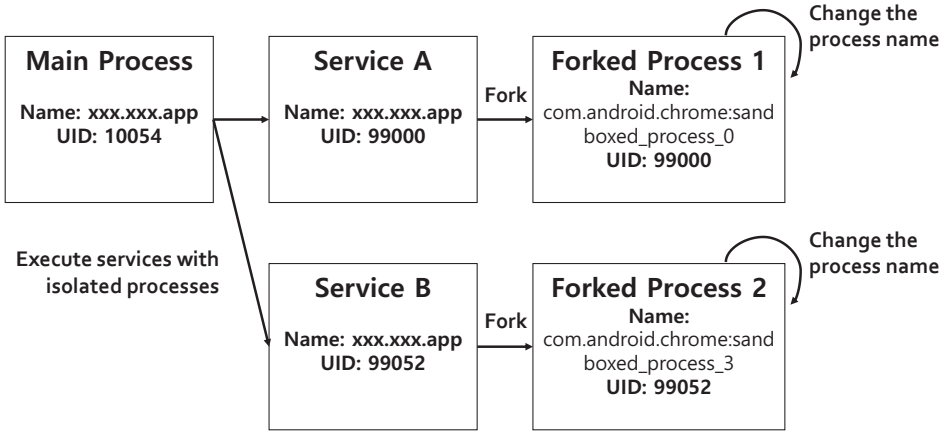


Fig. 4. An example of complicating a direct COUPLE attack to be stealthier

However, as shown in Fig. 4, an adversary can use the isolated process feature to manipulate the UID of an attack service. When the system executes the service for the COUPLE attack, an isolated process for the service runs with a different UID. It leads the subsequent processes forked from the service to also have the UID. Therefore, a simple countermeasure that terminates a group of processes having the same UID no longer works.

Changing process name. Next, an adversary can change the name of any process by manipulating its first command-line argument `argv[0]`. Originally, all processes belonging to an application inherit the package name of its application. Thus, even if the adversary lets the forked process have a different UID, the system still can eliminate the threat of the COUPLE attack by terminating all processes with the package name. However, if the process changes its name, all the above mitigations become ineffective. In specific, the adversary can use a pointer to access and modify the global variable `environ`. By inspecting the content of the global variable, the forked process can identify the address of the first command-line argument and replaces it with the desired name. For example, the process can disguise itself as a legitimate process by having a well-known process name such as `com.android.chromesandboxed.process._<NUM>`, a typical name of an isolated process of Chrome browser.

4.4. Indirect Attack

We introduce an indirect COUPLE attack that uses an indirect channel for communication between old and new versions of a malicious application. Notable examples of an indirect channel are private files and DBs stored in the internal storage of an application. An indirect COUPLE attack is more reliable than a direct COUPLE attack, though it may encounter some delay in sending out data.

Overall procedure. Fig. 5 depicts the overview of an indirect COUPLE attack. The first version of a malicious application developed by an adversary starts a service to retrieve all contacts with the `READ_CONTACT` permission (❶). The service writes the data into

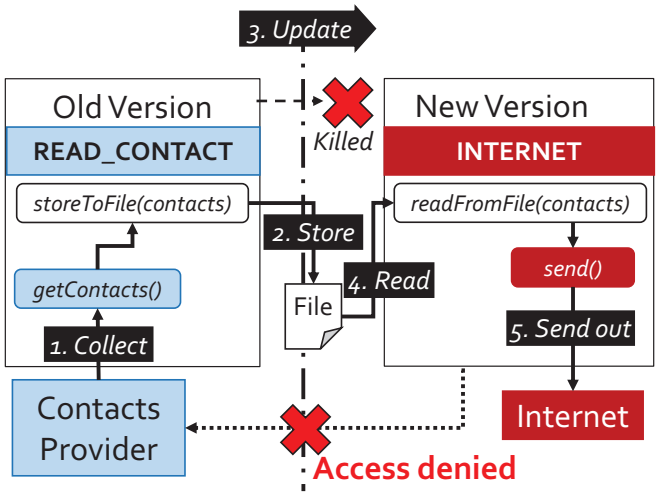


Fig. 5. Overview of an indirect COUPLE attack

a file in its internal storage, i.e., `/data/data/<package name>/` (❷). Later, the adversary uploads the second version that only requests the `INTERNET` permission and has logic to use network sockets to Google Play, and Google Play automatically installs the updated application (❸). When the second version runs, it starts a service to read the contacts data from the file in the local storage (❹), and send the data to the adversary's server (❺).

Reliability of indirect attack. An indirect COUPLE attack uses a reliable channel, a private resource stored in an application's internal storage. The Android system should preserve such resources during application updates because they may contain some important data to run the application. Thus, an indirect COUPLE attack is more reliable than a direct COUPLE attack that can fail due to an unscheduled termination of a child process.

Delay in data leaks. In an indirect COUPLE attack, data leaks from the second version of a malicious application can be postponed, because the attack has no method to wake the second version from a stopped state. The indirect COUPLE attack has no forked process to send an intent to the second version such that the attack cannot directly run the second version. Therefore, data delivery would start after a user runs the second version or reboots the device if the application has registered a broadcast receiver to listen for the broadcast intent `ACTION_BOOT_COMPLETED`.

4.5. Concurrent Attack

The COUPLE attacks explained so far are *sequential*; they collect sensitive data first and send out the collected data later, but cannot conduct both at the same time. Hence, an adversary cannot use them to steal real-time data, such as peeking at an OTP via SMS or monitoring the current location. In this section, we describe a concurrent COUPLE attack that allows a malicious application to simultaneously collect and send out sensitive data even if it has no permission for either operation.

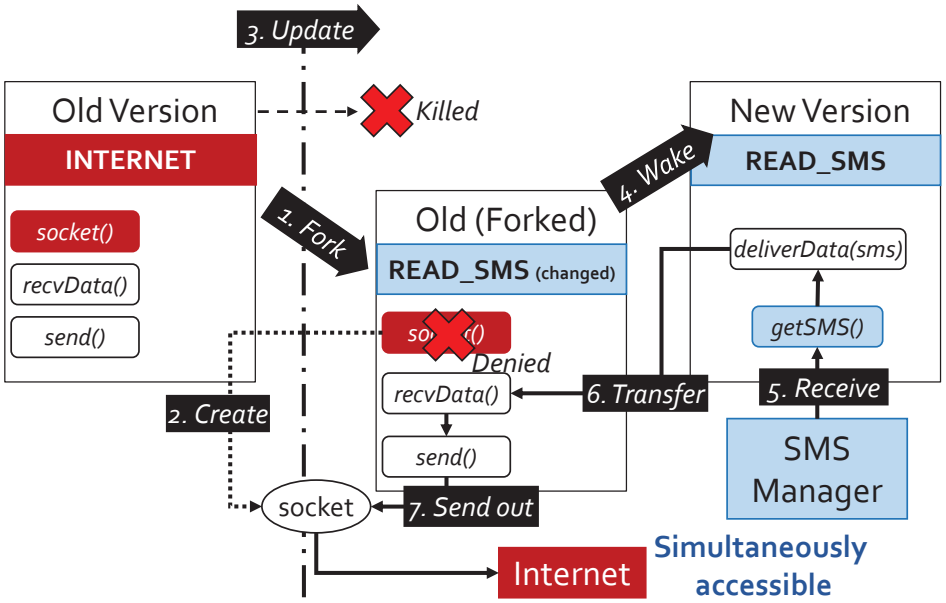


Fig. 6. Overview of a concurrent COUPLE attack

TOCTTOU problem in Internet use. We explain the details of the `INTERNET` permission and its TOCTTOU problem. In the Android system, an application should obtain the `INTERNET` permission to create a network socket to connect to external servers or clients, known as *Android paranoid networking*. If an Android user grants the application the `INTERNET` permission, the Android system adds the application to the group ID `AID_INET`. Later, when the application attempts to access the Internet, it calls the `socket()` system call that eventually calls a kernel function `inet_create()` to create an inet socket. Unlike Linux, Android's `inet_create()` has additional code to deny socket creation when the effective group ID of a calling process is not `AID_INET`, so an application can create an inet socket only when its effective group ID is `AID_INET`. After creating an inet socket, the application may call other system calls, e.g., `bind()`, `listen()`, `accept()`, and `connect()`, to connect to others.

However, a TOCTTOU problem occurs when using the Internet because *the Android system checks the effective group ID of a calling process only when it creates a socket*. An application should have the `INTERNET` permission when creating a socket, but does not need to keep the permission when using the socket to connect to servers, to wait for clients, or to send or receive data packets. Consequently, an adversary can perform a concurrent COUPLE attack that creates a socket using the first version and connects to the socket while collecting sensitive data using the second version. To avoid such a security problem, Linux checks the effective user and group IDs of a calling process not only when the process calls `open()`, but also when it calls `read()`, `write()`, and `ioctl()`. We believe that the Android system needs to take a similar approach.

Details of concurrent attack. Fig. 6 describes a concurrent COUPLE attack that exploits the TOCTTOU problem of the `INTERNET` permission. In contrast to a direct attack, an

Table 3. Example of permissions for data collection and transmission

Data collection	Data transmission
ACCESS_WIFI_STATE	INTERNET
READ_PHONE_STATE	CALL_PHONE
READ_SMS	SEND_SMS
ACCESS_FINE_LOCATION	BLUETOOTH
READ_EXTERNAL_STORAGE	WRITE_EXTERNAL_STORAGE
READ_CALENDAR	NFC
READ_CONTACTS	
READ_HISTORY_BOOKMARKS	
...	

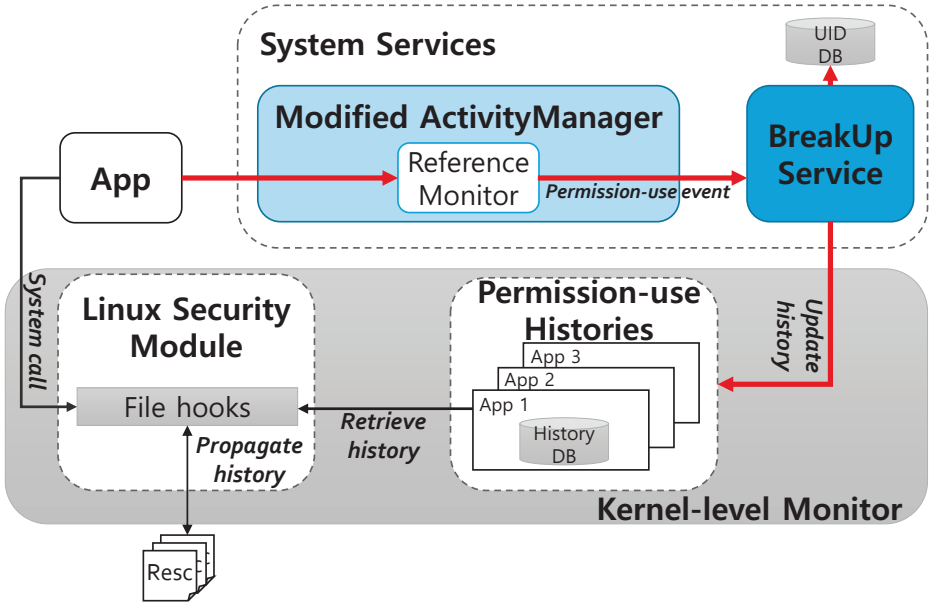
adversary requests the `INTERNET` permission at the first version of a malicious application, and requests a permission to collect sensitive data (e.g., `READ_SMS`) at the second version of the application. The first version starts a service that forks a child process (❶), and the forked process creates an inet socket with the `INTERNET` permission (❷). The adversary uploads the second version of the application implemented to collect sensitive data to Google Play, and the application is updated (❸). The forked process can no longer create a new inet socket, but it can use the already created socket. When the forked process recognizes that Google Play has updated the application, it wakes a service of the second version with the same method used in a direct COUPLE attack (❹). The service of the second version of the application then reads all SMS messages (❺) and transfers them to the forked process via an IPC (❻). Finally, the forked process sends the received data to the adversary's server through the socket that was created before the update (❼).

5. Countermeasure: BreakUp

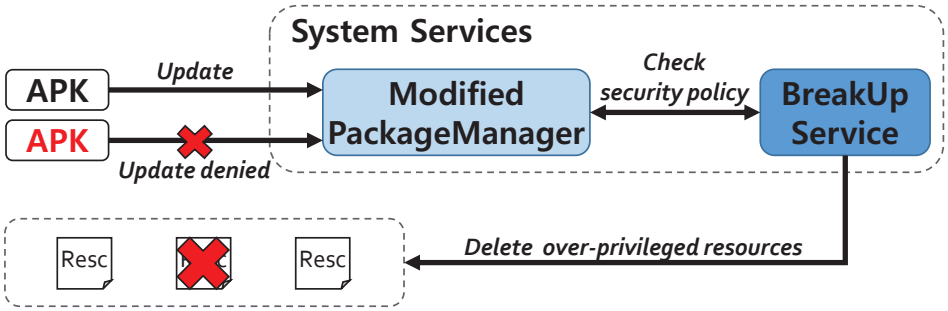
In this section, we explain a runtime security extension, called BREAKUP, that runs on an actual device to prevent the COUPLE attacks. BREAKUP keeps track of processes and resources of installed applications that possibly have security sensitive data. Whenever an application updates, BREAKUP eliminates suspicious processes and over-privileged resources not to leak the data to the next versions of the application.

5.1. Architecture Overview

BREAKUP is a system to detect and prevent the COUPLE attacks (Fig. 7). It consists of four components: BREAKUP *Service*, *modified Activity Manager*, *kernel-level monitor*, and *modified Package Manager*. First of all, the BREAKUP Service is a core component that manages interactions between other components. Second, the modified Activity Manager monitors how individual applications use permissions during their runtime. Third, the kernel-level manager propagates the permission-use histories of individual applications to their private resources by intercepting system calls, e.g., `read()`, and `write()`. Fourth, the modified Package Manager selectively deletes private resources with sensitive data while terminating every forked process at application updates via the BREAKUP Service to prevent the COUPLE attacks.



(a) Runtime management of permission-use histories



(b) Update-time policy enforcement

Fig. 7. Overview of BREAKUP architecture

Source and sink permissions. We summarize some source and sink permissions that can be used for COUPLE. We first choose 10 permissions among the lists of mostly checked permissions and most common unnecessary permissions [23]. We further select 13 other permissions classified as dangerous, e.g., `READ_SMS` and `READ_CALL_LOG`. In total, BREAKUP considers 23 permissions consisting of 17 source permissions and six sink permissions (Table 3).

5.2. Preventing Direct Attacks

We explain how BREAKUP prevents a direct COUPLE attack. The basic strategy of BREAKUP against a direct attack is to kill all forked processes at update, which corresponds to the

Table 4. Source lines of code (SLOC) we have added or changed

	Android framework		
	JAVA	**C++**	**Kernel**
SLOC	730	204	369

application update strategy of the Android system. When updating an application, the Android application installer terminates all running processes of the application to avoid getting into a confused state while the new versions of the application gets installed [6]. However, as described in Section 3.1, the installer cannot terminate arbitrarily spawned processes because it has no knowledge about them, letting a direct COUPLE attack possible. BREAKUP solves this problem by killing all active forked processes at application update.

BREAKUP has two challenges to identify all forked processes of an application: they can have different UIDs or process names as explained in Section 4.3. To overcome these challenges, BREAKUP maintains the UIDs belonging to the active isolated processes of each application (Fig. 7a) and terminates the processes with the maintained UIDs or the application's UID when the application is being updated.

5.3. Preventing Indirect Attacks

We further explain how BREAKUP protects the system from indirect COUPLE attacks. Similar to the idea used for preventing direct attacks, BREAKUP removes resources assumed to have security sensitive data during an application update. To track over-privileged resources, BREAKUP maintains a *permission-use history* representing the source permissions each application actually uses. Whenever an application accesses to sensitive data (e.g., SMS and call logs) during runtime, BREAKUP updates its per-app permission-use history DB. BREAKUP propagates the permission-use history of an application to a resource (e.g., a file) when the application writes data into it, since the written data may contain sensitive information. Also, when an application reads data from a resource, BREAKUP adds the permission-use history stored in the resource to the application's permission-use history. Later, when an application loses permissions at an update, BREAKUP finds the application's resources associated with the lost permissions and deletes them to prevent potential privacy leakage.

5.4. Implementation Details

We explain details of how we implement BREAKUP. We implemented and evaluated BREAKUP on a Nexus 5 phone with Android (kernel version 3.4.0). Table 4 summarizes the lines of code we have either added to or modified in the Android framework and the Linux kernel. Code modifications were kept to a minimum.

Tracking application UIDs. We explain how BREAKUP maintains UIDs for running processes. BREAKUP adds a hook in startProceessLocked() function in the ActivityManagerService. Whenever an application asks the modified Activity Manager to create a process, the hook notifies BREAKUP *Service* a pair of the UID of the process and the

host application name to store it in the UID database. Note that as long as the process is alive, the UID cannot be used for other application. Thus, BREAKUP *Service* maintains the UID-app record until the UID is claimed by another application. If the hook identifies the creation of a process for another application, it implies that there no longer exist running processes for the previous application. Therefore, BREAKUP *Service* replaces the old record with a new pair of UID and the application name.

Creating permission-use history. We describe how BREAKUP creates permission-use history DBs within the kernel-level monitor. The kernel-level monitor dynamically creates a permission-use history DB for an application when the first process for the application starts. BREAKUP keeps permission-use history DBs only for third-party applications by skipping system UIDs to reduce performance degradation. To track the creation of third-party processes, we added a hook into the `copy_process()` kernel function that will be invoked by the system at spawning a process. When the hook is invoked, the kernel-level monitor checks whether an application has its permission-use history DB. If the check fails, BREAKUP creates a new DB for the application.

Updating/propagating permission-use history. We first explain how BREAKUP updates permission-use histories. We added a callback function in the Reference Monitor of the Activity Manager to notify the BREAKUP Service of permission-check events. For example, when an application has successfully accessed a resource that demands a permission P_A through the Reference Monitor, the callback function provides the BREAKUP Service information on the application and P_A. The BREAKUP Service then informs the kernel-level monitor that the application has used P_A. The kernel-level monitor finds the history DB for the application to update the permission-use history. Throughout the process, the BREAKUP Service filters out already reported permission-check events to avoid performance degradation due to frequent communication between the user and kernel.

Next, we depict how BREAKUP propagates permission-use histories between processes and resources. BREAKUP uses the `security_file_permission()` function to intercept every file access event. Upon a write access to a file, the kernel-level monitor adds all permissions in the permission-use history DB for the application into the `xattr` of the file. When an application reads a file, the kernel-level monitor propagates permissions in the `xattr` of the file to the history DB for the application. The kernel-level monitor then reports the change of the permission-use history made by the file read event to the BREAKUP Service.

Deleting permission-use history. We show when BREAKUP discards a permission-use history DB. The kernel-level monitor deletes the permission-use history DB for an application when every process belongs to it has terminated. To intercept termination events for every third-party process, we placed a hook in the `do_group_exit()` kernel function. When the hook is invoked, and if the terminated process is the last process of an application, the kernel-level monitor destroys the permission-use history DB for the application.

Clearing sensitive data at update. We illustrate how BREAKUP eliminates the possibility of sensitive information leak at application updates. BREAKUP checks two conditions : (1) the old version has no sink permission and (2) the new version requests one or more sink permissions. If an update meets the conditions, the BREAKUP Service first finds all forked processes by the package name of the application and terminates them. Next, the BREAKUP Service traverses the private directories of the application to check the *xattr*s of all files. If the BREAKUP Service notices that the permission-use histories in *xattr*s of

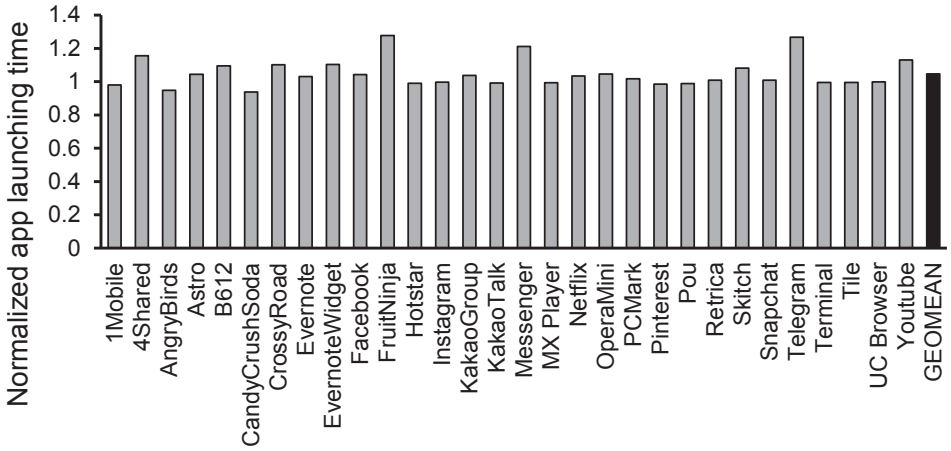


Fig. 8. Elapsed time of launching applications on BREAKUP normalized to that of Android

Table 5. The scores returned by PCMark running on Android and BREAKUP. We performed each experiment 10 times.

Test	Android	BREAKUP	Reduction (%)
Work performance	3874.5	3803.5	1.8
Web browsing	3670.9	3634.8	1.0
Video playback	3989.8	3969.4	0.5
Writing	3406.4	3329.9	2.2
Photo editing	4522.8	4354.3	3.7

some files include permissions that the new version does not have, the BREAKUP Service deletes those files.

5.5. Performance Evaluation

We evaluate the performance of BREAKUP in terms of application launch time, runtime overhead, and update time. First, we measured launch time of 30 popular applications on BREAKUP to estimate the overhead of initializing permission history DBs for the applications. On average, the launch time increased by 4.7% compared with unmodified Android (Fig. 8).

Second, we used a benchmark application, `PCMark for Android`[5], to check overall runtime overhead of BREAKUP. `PCMark` shows relative scores of an Android device by executing five realistic applications. As shown in Table 5, BREAKUP decreases benchmark scores by only 0.5%–3.7%.

Lastly, we measured update time of the 30 applications on BREAKUP to identify the overhead caused by enforcing the security enforcement, killing forked processes, and deleting over-privileged files. The applications that we used for this evaluation created

[5] `http://www.futuremark.com/benchmarks/pcmark-android`

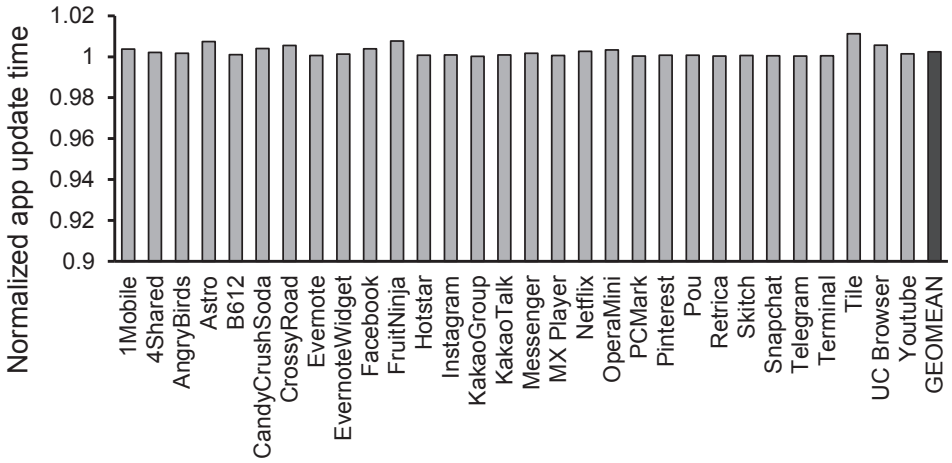


Fig. 9. Elapsed time of updating applications on BREAKUP normalized to that of Android

about 47 files on average during their execution; we deleted all the files to estimate the worst-case performance. Fig. 9 shows that BREAKUP has negligible overhead (0.2%).

5.6. Limitation

Since BREAKUP is a coarse-grained information flow tracking system, it may delete some files without sensitive data at application updates. At worst, BREAKUP will delete all files that an application created during its execution, to essentially be a newly installed application. However, as Section 6.2 shows, a small number of updates are treated as the COUPLE attacks (0.015%). Therefore, we believe that these limitations of BREAKUP are acceptable.

6. Analysis of Google Play Apps

In this section, we study the feasibility of the COUPLE attacks by collecting and analyzing all available versions of applications downloaded from Google Play. We first explain how we collect such applications and describe our analysis results.

6.1. Data Collection

We build our Google Play application crawler based on the PLAYDRONE [51] open source project. However, PLAYDRONE is only capable of retrieving the latest application version. To capture real attack cases in official applications, we need to crawl the total version history of an application starting from its very first release on Google Play. To achieve this goal, we implement two additional features that help in crawling the total version history of an application.

```
https://android.clients.google.com/fdfe/purchase
Method:      POST
Parameter:   ot=1
             doc=<package name>
             vc=<version code>
Header:      Authorization: <auth cookie>
             X-DFE-Device-Id: <android id>
             ......
```

Fig. 10. HTTP request to obtain a URL to download a specific version of a target application from Google Play

First, through our experiments, we found that, after proper device registration and authentication with Google Play, sending the HTTP request in Fig. 10 will trigger the checkout process at Google Play that returns either a valid download cookie and URL (if an application with given `<package name>` and `<version code>` exists) or a message indicating the application does not exist. The version codes for each application can be enumerated by decrementing the version code of the latest version by 1 each time.

Second, Google Play only allows downloading of applications that are suitable to the registered device configuration. For example, certain versions of an application might be hidden from devices with `x86-64` CPU architecture due to lack of support. To avoid missing application versions because of this customization feature, we crawled all publicly available Nexus factory images and created artificial device profiles with the union of their CPU architectures, feature set, loaded libraries, and OpenGL extensions. We used these device configurations for all application crawling activities.

6.2. Data Analysis

In March 2015 we collected 2,009 Google Play applications with all available version histories from the `apps_topselling_free` category (28,682 versions in total). Fig. 11 shows the boxplot summarizing the number of versions of individual applications. The median application in our dataset had nine versions. Among them, we decided to focus on 1,606 applications with ≥ 3 versions, where 3 is the first quartile of the boxplot, to avoid errors due to toy applications with a small number of versions. The total number of versions analyzed was 28,100.

Time interval between updates. We analyzed how frequently the applications were updated. For each application, we measured the average time difference between the creation times of two adjacent versions. On average (median), the applications in our dataset were updated every 330.5 hours (Fig. 12). Among the 1,606 applications analyzed, 382 applications (23.8%) had been updated within one hour at least once. Thus, frequently updating applications (e.g., several hours) is not a distinguishing feature of an attack.

Frequency of permission-changing updates. We estimated the frequency of application updates with permission changes, i.e., permission addition, deletion, or both. For each application, we calculated the number of permission changes over the number of version

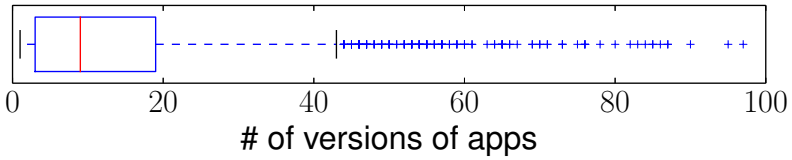


Fig. 11. The number of versions of individual applications

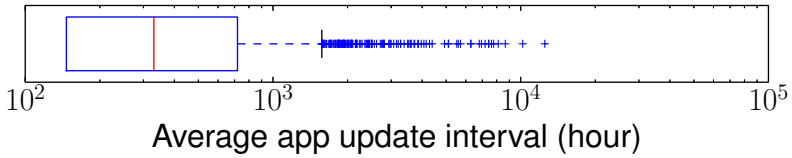


Fig. 12. Average time intervals between application updates. We calculated the difference between the creation times of every two adjacent versions of the same application.

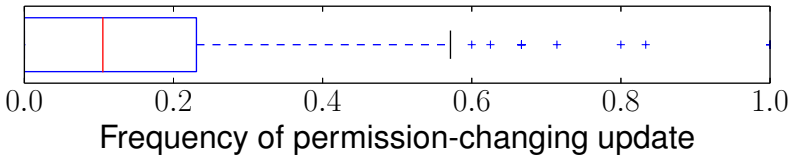


Fig. 13. Frequency of permission-changing updates of individual applications

updates. The median frequency of permission-changing update was 0.11 (Fig. 13). This implies that when an application is updated nine times, its permissions will change once.

The number of auto-updates. We analyzed the number of application updates that can be automatically performed by Google Play (Section 3.3). Among the 26,494 application updates inspected, the number of updates with permission additions was 2,165 (8.2%). We further checked the permission additions of individual updates to recognize the possibility of auto-updates. Among them, 600 updates (27.7%) could be automatically updated by Google Play.

The possibility of the COUPLE attacks. We found four updates that could lead to the COUPLE attacks. When "Magnetic balls bubble shoot" (7), "Kids Piano Games FREE" (8), "Dominoes Elite" (14), and "Pink Kitty bow Clock Widget" (17) requested a sink permission `INTERNET`, they did not request source permissions that they had before (e.g., `GET_ACCOUNTS` and `READ_PHONE_STATE`). Thus, they were able to secretly leak sensitive data, e.g., telephone number and UID, at those versions if they had forked a process or stored the data in private resources.

We have manually analyzed the four applications by running and updating them via an adb shell and confirmed that they were benign applications. They did not fork any child processes. Also, they either did not create any files or downloaded music files from their servers. Deletion of the music files by BREAKUP made no problem: the updated applications downloaded the music files again. Therefore, we believe that the COUPLE attacks are not yet discovered and performed by adversaries.

7. Discussion

In this section, we discuss a few server-side security methods against the COUPLE attacks and how attackers can evade them with dummy updates.

Analysis of permission changes. Analyzing the changes of permission requests in every application update is the easiest countermeasure against the COUPLE attacks. The COUPLE attacks frequently makes updates to rotate source and sink permissions, so these permission-changing patterns can be used to distinguish the COUPLE attacks. However, attackers can manipulate permission request changes with dummy updates. Between every important update, attackers can release a few dummy updates with no or small permission request changes to conceal such patterns.

Static analysis of multiple versions. Static analysis of multiple versions of the same application could recover separated information flow across application updates. For example, if investigators successfully convert multiple versions of the same application into different applications, they can use a scheme for statically checking ICC (such as [52]) to detect COUPLE attacks. But, still, attackers can use dummy updates to make the analysis complex and native code to secure information-flow channels [7].

Monitoring execution across updates. If a detection server runs every application across version updates for dynamic information flow tracking, it can detect COUPLE attacks. However, due to a large number of applications, long-term monitoring of every application is almost impossible. The server can reduce such overhead by executing an application only around every update. But, this technique is ineffective because dummy updates can increase the number of application updates. Consequently, a security system running on an actual client device, such as BREAKUP, is necessary to cope with the COUPLE attacks.

Runtime permissions. Android introduced the runtime permission system in Android 6.0, Marshmallow [5]. Unlike the previous permission system, it allows users to grant or revoke permissions dynamically while applications are running. However, the change cannot be a direct solution to the COUPLE attack because custom users are still prone to grant permissions without doubts. In addition, due to slow update or deployment of new Android versions [40], the complete deployment for all Android smartphone users will take several years.

8. Conclusion

In this paper, we considered a novel attack exploiting application updates: COUPLE. The COUPLE attacks allowed a malicious application to leak sensitive data via cross-update side channels. Compared to conventional attacks, the COUPLE attacks were easy to perform and stealthy. We developed BREAKUP, a lightweight security extension running on an actual device; its time overhead was below 5%. Lastly, our in-depth analysis of 2,009 applications with all available versions showed the feasibility of the COUPLE attacks.

References

1. Alam, S., Qu, Z., Riley, R., Chen, Y., Rastogi, V.: Droidnative: Automating and optimizing detection of android native code malware variants. Computers & Security 65, 230–246 (2017)

2. Almohri, H.M.J., Yao, D., Kafura, D.: Know what is executing on your android. In: Conference on Data and Application Security and Privacy (2014)
3. Android Developers: Manifest.permssion. http://developer.android.com/reference/android/Manifest.permission.html
4. Android Developers: Manifest.permssion_group. http://developer.android.com/reference/android/Manifest.permission_group.html
5. Android Developers: Requesting permissions at run time. http://developer.android.com/intl/ko/training/permissions/requesting.html
6. Android googlesource: Packagemanagerservice. https://android.googlesource.com/platform/frameworks/base/+/483f3b06ea84440a082e21b68ec2c2e54046f5a6/services/java/com/android/server/pm/PackageManagerService.java
7. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y.L., Octeau, D., McDaniel, P.: FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2014)
8. Backes, M., Bugiel, S., Hammer, C., Schranz, O., Von Styp-Rekowsky, P.: Boxify: Full-fledged app sandboxing for stock Android. In: USENIX Security Symposium (2015)
9. Backes, M., Bugiel, S., Schranz, O., von Styp-Rekowsky, P., Weisgerber, S.: ARTist: The Android Runtime Instrumentation and Security Toolkit. In: European Symposium on Security and Privacy (EuroS&P). IEEE (2017)
10. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R., Shastry, B.: Towards taming privilege-escalation attacks on Android. In: Network and Distributed System Security Symposium (NDSS) (2012)
11. Canfora, G., Mercaldo, F., Visaggio, C.A.: An hmm and structural entropy based detector for android malware: An empirical study. Computers & Security 61, 1–18 (2016)
12. Cao, Y., Fratantonio, Y., Bianchi, A., Egele, M., Kruegel, C., Vigna, G., Chen, Y.: EdgeMiner: Automatically detecting implicit control flow transitions through the Android framework. In: Network and Distributed System Security Symposium (NDSS) (2015)
13. Chan, P.P., Hui, L.C., Yiu, S.M.: DroidChecker: Analyzing Android applications for capability leak. In: ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec) (2012)
14. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in Android. In: International Conference on Mobile Systems, Applications and Services (MobiSys) (2011)
15. Conti, M., Nguyen, V.T.N., Crispo, B.: CRePE: Context-related policy enforcement for Android. In: Information Security Conference (ISC) (2010)
16. Cox, L.P., Gilbert, P., Lawler, G., Pistol, V., Razeen, A., Wu, B., Cheemalapati, S.: SpanDex: Secure password tracking for Android. In: USENIX Security Symposium (2014)
17. Davi, L., Dmitrienko, A., Sadeghi, A.R., Winandy, M.: Privilege escalation attacks on Android. In: Information Security Conference (ISC) (2010)
18. Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., Wallach, D.S.: QUIRE: Lightweight provenance for smart phone operating systems. In: USENIX Security Symposium (2011)
19. Enck, W., Ongtang, M., McDaniel, P.: On lightweight mobile phone application certification. In: ACM Conference on Computer and Communications Security (CCS) (2009)
20. 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: USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2010)
21. Fan, W., Sang, Y., Zhang, D., Sun, R., Liu, Y.: Droidinjector: A process injection-based dynamic tracking system for runtime behaviors of android applications. Computers & Security 70, 224–237 (2017)

22. Feizollah, A., Anuar, N.B., Salleh, R., Suarez-Tangil, G., Furnell, S.: Androdialysis: Analysis of android intent effectiveness in malware detection. Computers & Security 65, 121–134 (2017)
23. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: ACM Conference on Computer and Communications Security (CCS) (2011)
24. Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission re-delegation: Attacks and defenses. In: USENIX Security Symposium (2011)
25. Felt, A.P., Ha, E., Egelman, S., Haney, A., Chin, E., Wagner, D.: Android permissions: User attention, comprehension, and behavior. In: Symposium on Usable Privacy and Security (SOUPS) (2012)
26. Google Play Help: About app permissions. https://support.google.com/googleplay/answer/6014972
27. Grace, M., Zhou, Y., Wang, Z., Jiang, X.: Systematic detection of capability leaks in stock Android smartphones. In: Network and Distributed System Security Symposium (NDSS) (2012)
28. GSMArena.com: Android global market share now exceeds 80%. http://www.gsmarena.com/android_worldwide_marketshare_crosses_80_for_the_first_time-news-7171.php
29. Hornyack, P., Han, S., Jung, J., Schechter, S., Wetherall, D.: "these aren't the droids you're looking for": Retrofitting Android to protect data from imperious applications. In: ACM Conference on Computer and Communications Security (CCS) (2011)
30. Idrees, F., Rajarajan, M., Conti, M., Chen, T.M., Rahulamathavan, Y.: Pindroid: A novel android malware detection system using ensemble learning methods. Computers & Security 68, 36–46 (2017)
31. Jeon, J., Micinski, K.K., Vaughan, J.A., Fogel, A., Reddy, N., Foster, J.S., Millstein, T.: Dr. Android and Mr. Hide: Fine-grained permissions in Android applications. In: ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM) (2012)
32. Jia, L., Aljuraidan, J., Fragkaki, E., Bauer, L., Stroucken, M., Fukushima, K., Kiyomoto, S., Miyake, Y.: Run-time enforcement of information-flow properties on Android (extended abstract). In: European Symposium on Research in Computer Security (ESORICS) (2013)
33. Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M.F., Kohler, E., Morris, R.: Information flow control for standard OS abstractions. In: Symposium on Operating Systems Principles (SOSP) (2007)
34. Lee, B., Lu, L., Wang, T., Kim, T., Lee, W.: From Zygote to Morula: Fortifying weakened ASLR on Android. In: IEEE Symposium on Security and Privacy (Oakland) (2014)
35. Lu, K., Li, Z., Kemerlis, V.P., Wu, Z., Lu, L., Zheng, C., Qian, Z., Lee, W., Jiang, G.: Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting. In: Network and Distributed System Security Symposium (NDSS) (2015)
36. Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G.: CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In: ACM Conference on Computer and Communications Security (CCS) (2012)
37. Marforio, C., Ritzdorf, H., Francillon, A., Capkun, S.: Analysis of the communication between colluding applications on modern smartphones. In: Annual Computer Security Applications Conference (ACSAC) (2012)
38. Mariconti, E., Onwuzurike, L., Andriotis, P., De Cristofaro, E., Ross, G., Stringhini, G.: MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. In: The Network and Distributed System Security Symposium 2017 (2017)
39. Nadkarni, A., Enck, W.: Preventing accidental data disclosure in modern operating systems. In: ACM Conference on Computer and Communications Security (CCS) (2013)
40. Nate Swanner: This is what android fragmentation looks like in 2015. http://thenextweb.com/insider/2015/08/05/this-is-what-android-fragmentation-looks-like-in-2015/#gref

41. Nauman, M., Khan, S., Zhang, X.: Apex: Extending Android permission model and enforcement with user-defined runtime constraints. In: ACM Symposium on Information, Computer and Communications Security (ASIACCS) (2010)
42. Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., Traon, Y.L.: Effective inter-component communication mapping in Android: An essential step towards holistic security analysis. In: USENIX Security Symposium (2013)
43. Ongtang, M., McLaughlin, S., Enck, W., McDaniel, P.: Semantically rich application-centric security in Android. In: Annual Computer Security Applications Conference (ACSAC) (2009)
44. Palumbo, P., Sayfullina, L., Komashinskiy, D., Eirola, E., Karhunen, J.: A pragmatic android malware detection procedure. Computers & Security 70, 689–701 (2017)
45. Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., Vigna, G.: Execute this! analyzing unsafe and malicious dynamic code loading in Android applications. In: Network and Distributed System Security Symposium (NDSS) (2014)
46. Reed, B.: 99% of mobile malware targeted Android devices last year. http://bgr.com/2014/01/21/android-mobile-malware-report/
47. Ruiz-Heras, A., García-Teodoro, P., Sánchez-Casado, L.: Adroid: anomaly-based detection of malicious events in android platforms. International Journal of Information Security 16(4), 371–384 (Aug 2017), https://doi.org/10.1007/s10207-016-0333-1
48. Russello, G., Jimenez, A.B., Naderi, H.: FireDroid: Hardening Security in Almost-Stock Android. In: Annual Computer Security Applications Conference (2013)
49. Schlegel, R., Zhang, K., Zhou, X., Intwala, M., Kapadia, A., Wang, X.: Soundcomber: A stealthy and context-aware sound trojan for smartphones. In: Network and Distributed System Security Symposium (NDSS) (2011)
50. Sellwood, J., Crampton, J.: Sleeping Android: Exploit through dormant permission requests. In: Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM) (2013)
51. Viennot, N., Garcia, E., Nieh, J.: A measurement study of Google Play. In: ACM Sigmetrics (2014)
52. Wei, F., Roy, S., Ou, X., Robby: Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In: ACM Conference on Computer and Communications Security (CCS) (2014)
53. XDA Forums: Play Store permissions change opens door to rogue apps. http://www.xda-developers.com/play-store-permissions-change-opens-door-to-rogue-apps/
54. Xing, L., Pan, X., Wang, R., Yuan, K., Wang, X.: Upgrading your Android, elevating my malware: Privilege escalation through mobile OS updating. In: IEEE Symposium on Security and Privacy (Oakland) (2014)
55. Xu, R., Saidi, H., Anderson, R.: Aurasium: Practical policy enforcement for Android applications. In: USENIX Security Symposium (2012)
56. Yan, L.K., Yin, H.: DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In: USENIX Security Symposium (2012)
57. Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P., Wang, X.S.: AppIntent: Analyzing sensitive data transmission in Android for privacy leakage detection. In: ACM Conference on Computer and Communications Security (CCS) (2013)
58. Zeldovich, N., Boyd-Wickizer, S., Kohler, E., Mazières, D.: Making information flow explicit in HiStar. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2006)
59. Zhang, Y., Yang, M., Xu, B., Yang, Z., Gu, G., Ning, P., Wang, X.S., Zang, B.: Vetting undesirable behaviors in Android apps with permission use analysis. In: ACM Conference on Computer and Communications Security (CCS) (2013)
60. Zhou, Y., Jiang, X.: Detecting passive content leaks and pollution in Android applications. In: Network and Distributed System Security Symposium (NDSS) (2013)

Beumjin Cho received the B.S. in computer engineering from Dongguk University, Korea, in 2013. He is currently working toward the Ph.D. at Pohang University of Science and Technology (POSTECH). His research interests include mobile system security, operating systems, and access control.

Sangho Lee received the B.S. in computer engineering from Hongik University, Korea, in 2006, the M.S. degree, and Ph.D. degree in computer science and engineering from Pohang University of Science and Technology (POSTECH), Korea, in 2008 and 2013, respectively. From 2013 to 2015, he was a Postdoctoral Research Associate at POSTECH. He is currently a Postdoctoral Fellow in the School of Computer Science at Georgia Tech. His research interests are all aspects of computer security including system security, web security, and privacy protection.

Meng Xu received the B.S. in computer science from Nanyang Technological University, Singapore, in 2014. He is currently working towards the Ph.D. at Georgia Institute of Technology. His research interests include software diversity, application sandbox and mobile system security.

Sangwoo Ji received the B.S. in computer engineering from Pohang University of Science and Technology (POSTECH), Korea, in 2015. He is currently working toward the Ph.D. at POSTECH. His research interests include mobile system security, and authentication.

Taesoo Kim is an Assistant Professor in the School Computer Science at Georgia Tech. He is interested in building a system that has underline principles for why it should be secure. Those principles include the design of a system, analysis of its implementation, and clear separation of trusted components. He holds a B.S. from KAIST (2009), a S.M. (2011) and a Ph.D. (2014) from MIT.

Jong Kim received the B.S. in electronic engineering from Hanyang University, Korea, in 1981, the M.S. degree in computer science from the Korean Advanced Institute of Science and Technology, Korea, in 1983, and the Ph.D. degree in computer engineering from Pennsylvania State University in 1991. He is currently a professor in the Division of IT Convergence Engineering, Pohang University of Science and Technology (POSTECH), Korea. From 1991 to 1992, he was a research fellow in the Real-Time Computing Laboratory of the Department of Electrical Engineering and Computer Science, University of Michigan. His major areas of interest are fault-tolerant computing, parallel and distributed computing, and computer security.

Received: July 28, 2017; Accepted: December 22, 2017.