Burn After Reading: Expunging Execution Footprints of Android Apps

Junliang Shu(✉), Juanru Li, Yuanyuan Zhang, and Dawu Gu

Lab of Cryptology and Computer Security, Shanghai Jiao Tong University, Shanghai, China
s.junliang@gmail.com

Abstract. Mobile apps nowadays are consuming and producing a mass of sensitive data. In response, a wide variety of privacy protection techniques and tools have been proposed since mobile users have the escalating privacy concerns. However, only a few privacy protection schemes consider how to thoroughly erase the runtime information of an app after its execution. Various traceable vestiges, called execution footprints, are kept by the device which could be used to steal and speculate user's privacy. We argue that a mobile operating system should not only establish sound isolation between different apps but also need to provide a fine-grained execution footprint expunging mechanism to ensure using an app confidentially. To achieve this target, MIST, a modified Android OS, to generate fine-grained data expunging policies, is designed and implemented. MIST is a lightweight ephemeral container, which does not require the support of specialized hardware or operation mode and it will be disposed of securely when in use apps. In this container, MIST persistently tracks every message generated by the app and then it deletes them during and after the execution. Experiments based on 200 apps show that execution footprints still have been neglected by the Android OS even after the app removal. By utilizing the expunging mechanism MIST provided, those footprints are erased to guarantee a private and confidential execution.

1 Introduction

The contradiction between forensics requirements and privacy protection has brought to the public since the FBI-Apple encryption dispute has burst out. Although robust cryptography schemes have been deployed to the mainstream mobile operating systems (Android and iOS), the unceasing attempts keep being made by investigators or attackers who try to break the protection. In particular situation, merely exposing the truth that some specific apps (such as healthcare

We would like to thank the anonymous reviewers for their valuable comments and helpful suggestions. This paper is partially supported by the Key Program of National Natural Science Foundation of China (Grant No. U1636217), the National Key Research and Development Program of China (Grant No. 2016YFB0801200), and a research grant from the Ant Financial Services Group.

M. H. Au et al. (Eds.): NSS 2018, LNCS 11058, pp. 46–63, 2018.
https://doi.org/10.1007/978-3-030-02744-5_4

app, life-style apps, and apps strongly relevant to user's privacy) have been execution can lead to a grave infringement on the privacy of the device owner. To refer individual behavior of the user, the primary sources of information exposure are **execution footprints**, which are modifications of system status due to an app's execution. The execution of an app always changes the system status and leaves an abundance of information on the device temporarily or permanently. These modifications may reveal various kinds of information (e.g., the name of the app) and can be used as digital evidence (e.g., whether a particular app has executed on the device).

Unfortunately, mobile operating systems often fail to erase such footprints to hide execution traces and protect user privacy. As a result, an experienced analyst can indicate not only the identity of the app but also the related operations. Take Android, the most widely used mobile OS, as an example. We reveal that except regular footprints such as files, many kinds of information, although unobtrusive, still indicate the execution of real app. With a study of 100 favorite Android apps mainly focusing on those related to Inter-Process Communication (IPC) or interactions with the OS, we investigated how apps generate footprints, and found When an app relies on intensive system services, some information will unintentionally leak to the system without being noticed. Even if under specific privacy protection enhancements, the surveillance state could still obtain data through conducting brute-force decryption, and also deduce relevant information from some encrypted but featured data. Also, we observe that Android does not provide a cautious data expunging policy and execution footprints remain for most apps even after the uninstallation.

To address this issue of data footprints, the critical steps include determining the sources where the execution footprints generate, and expunging remained footprints thoroughly after the execution/uninstallation. In particular, to execute an app on Android without being perceived by malicious observations as offline forensic analysis or online side-channel eavesdropping, a *private mode*, or say a private execution of the apps, is demanded to conceal the performance of the app and eliminate all those *footprints* left in the system. Private mode is firstly implemented as a *private browsing mode*, a standard security feature provided to prevent information leakage on modern web browsers such as Firefox and Chrome. It ensures that personal information (e.g., browsing histories, cookies, and cache information) are cleared once by the browsing session ends. However, to anonymize the execution of an app on mobile device is much more sophisticated. It involves a myriad of interactions with the OS such as file I/O operation and API invoking with sensitive permission requirements. A systematical solution for this problem is expected to conceal the execution information of specific apps.

To port such private execution function to Android, we design and implement Mist, a modified Android OS that enables an ephemeral execution for universal apps. With Mist, users can launch an app and execute it without being perceived by a later forensic analysis, despite the skillful analyst that could fully control the system and gain every aspect of the system information. During

the ephemeral execution, the input and output of the app execution are managed delicately by fine-grained monitoring policies of Mist. For instance, Mist adopts a temporal partition encryption scheme to re-direct every file operation to a secure and ephemeral container. Mist also determines how to control the inter-process communication (IPC) during the app's execution: when the app generated data flow to the external environment, it either blocks the data flow or labels the system service as a tainted process. After the execution, Mist immediately sanitizes any possible leaked information. In this way, Mist proves that no footprints remain after the execution and thus protects the privacy of the user.

Compared with current privacy schemes on Android, Mist has the following advantages: (1) Mist conducts a fine-grained footprint monitoring based on an investigation of real-world Android devices and apps; (2) Mist implements a lightweight ephemeral execution to expunge footprint comprehensively, and it does not require the support of specialized hardware or operation mode (e.g., Trusted Execution Environment). To validate the reliability and usability of Mist, we conduct experimental evaluation with 200 popular Android apps. The evaluation first reveals many remaining data (according to the monitoring) as footprints and then it utilizes a set of forensic analysis tools to check whether typical footprints can be detected if the app executes in Mist. The results demonstrate that Mist expunges those footprints thoroughly against existing forensic analyses. The evaluation also shows that the performance overhead of Mist is acceptable for most application scenarios.

2 Footprint Expunging

The installation, execution, and uninstallation of an app modify the status of the device temporarily or permanently. Such modifications are defined as execution footprints if they can be used as digital evidences to refer specific behavior of the user (e.g., whether a particular app has been executed on the device). Mobile operating systems are expected to erase such footprints to hide execution traces and protect user privacy. However, modern mobile OS such as Android fail to achieve this target. In this section we first discuss the deficiency of Android system on expunging execution footprint and then present our enhanced strategy of practical footprint expunging.

2.1 Footprints of Android Apps

The Android OS provides a series of access control strategies to prove that each app only executes in an isolated environment. Ideally, the OS should guarantee that the status of the system keeps consistent whether an app has been installed or not. Apparently, this is infeasible since many apps must register themselves to the system to handle specific requests (e.g., write files to the disk and register themselves to some system services). Thus, a practical solution is to execute an uninstallation process and erase every vestige of the app. Nevertheless, current

app uninstallation mechanism of Android does not guarantee such requirement. As shown in Table 1, recent studies related to execution footprints indicated that at least 11 items must be sanitized. Otherwise, each of them represents a class of footprint that may help track specific user behavior.

Table 1. Representative footprints of Android

Item	Footprint pattern
Memory [15,18,21–24,28]	Kept in memory after the execution
Files [18–20,27,28]	Written to flash memory in plaintext
IPC [8,32]	Transferred through IPC
Process status [31,33]	Identified by other processes
Battery usage	App name kept after the execution
Network usage	App name kept after the execution
Screen [14]	Screenshot buffer captured by other apps or the system
Microphone [11]	Sound captured by other apps or the system
Sensor [13,16]	Sensor data captured by other apps or the system
Keyboard input [6]	App identity referred by a third-party input method
System log	App identity recorded in system log files

However, only by enumerating such items we cannot understand the footprint issue comprehensively. From the viewpoint of footprint origins, we classify all execution fingerprints into four types and study them in a systematic way:

Footprint in Memory: Memory pages of an app process contain plenty of sensitive information that may reveal the identity of their creator or even the exact behavior of it [15,21,22,24]. For instance, distinguishable strings, code segments, images, GUI layouts can be used to deduce certain user behavior.

Whereas, few app takes the initiative to actively erase used memory pages, making most sensitive data left in memory even if the running process is terminated. Android system does not consider this as an issue either, leaving a feasible attack window for the advanced cold-boot attack [10].

Footprint on eMMC: Most Android devices store files in the flash memory of embedded MultiMedia Card (eMMC). Files created by apps usually carry a lot of relevant information: both the content and the metadata (e.g., file name, directory structure) can be used as the evidence of app's execution [19,27]. Once an attacker obtains the device, he can directly visit this information through dumping the partition image with forensic techniques (even if the partition is encrypted, it is possible to recover the image through guessing the password [1]). Unfortunately, Android lacks a secure data wiping mechanism and data written to flash memory is difficult to be wiped securely [4,26]. Therefore, sensitive files often remain on the eMMC for a long time (even after a factory reset [25]).

Footprint in IPC: An Android app interacts with other apps or system processes frequently through IPC (mainly through Android's binder). Sensitive information may be leaked to other processes or remains in memory/flash [32] depending on the specific kind of IPC. In either case, footprints left on the device after the execution of their creators.

Side-channel Footprint: Except the memory pages and files directly and actively generated by an app, there are some unobtrusive footprints produced by the Android OS unintentionally. An adversary can also collect these footprints and use them to infer the behaviors of individual apps. In general, these footprints are divided into two major types: **runtime side-channel footprints** that can only be obtained while an app is running, and **legacy side-channel footprints** kept in memory or flash storage after the execution.

- *Runtime side-channel footprints.* A runtime side-channel footprint is a kind of information that can be gather by a (malicious) app with normal privileges [11,31]. Under the protection of Android's sandbox, typical runtime side-channel footprints include process name and UID of running apps. In Android, the system uses the package name as the process name of the app. By gathering process name through shell command `ps`, a malicious app easily obtains footprint of all the running apps. The sensors and microphone are also sources of data leakage. Previous researches have demonstrated the feasibility of inferring identity from such side-channel footprint [5,9,16,17,31].
- *Legacy side-channel footprints.* A legacy side-channel footprint is a kind of information related to the identity of the host app and is often kept by the Android system. Most kinds of these footprints are package names and UIDs logged by specific system monitors, which record system events such as the usage of system resources. some Android system components keep the sensitive runtime footprint of apps unintentionally. Although the threat model in this paper assumes that the OS and the device are both trustful, these footprints are somehow accessible without permission requests, leading an attacker to infer the behaviors of the device owner.

2.2 Footprint Expunging Strategy

The essential requirement for a robust footprint expunging strategy is that after the execution (i.e., the app process is terminated), even the user herself, knowing the execution details, could not able to recover any usable evidence of activities conducted on the device. Hence, the expunging strategy requires that any secret would not persist explicitly.

To achieve such goal, we consider four design principles summarized from real-world threats when sanitizing different footprints. These principles are based on either previous best practices or our observations. If a footprint sanitization scheme violates any of these principles, information may not be cleaned and thus can be identified as a footprint.

1. *Sensitive data in non-persistent memory should have a lifetime as short as possible.* Consider that an attacker could later acquire administration privilege to spy upon the entire memory space, the runtime execution data in memory must be wiped out to counter such attack. Hence, at the end of its lifetime, the data should be appropriately erased.
2. *Sensitive data should be encrypted before being stored in the persistent storage medium.* It believed that the secure deletion of disk data is not able to be fulfilled on the state-of-the-art storage medium of mobile devices. Therefore, any data written to the flash memory must be encrypted beforehand. The encryption key should be ephemeral and must only be kept in non-persistent memory. Thus this kind of data can be cryptographically erased when the execution is over by wiping the temporary key.
3. *The interaction of app and the OS should be censored so that sensitive data leakage can be filtered.* Confidential information sent from the app to the system processes is hard to clean thoroughly (the cleaning may crash the system). Therefore, a filter applied on IPC can block the leakage beforehand.
4. *The execution itself should be side-channel resistant.* In other words, its runtime information should not be gathered by a concurrently executed malicious program.

3 MIST

In this section, we present our solution to erase the app footprint on Android. We design and implement Mist, a system based on Android OS to offer a robust and comprehensive footprint expunging. Mist provides Android user the ability to run apps within an ephemeral container, leaving no recognizable footprint after being executed.

The design of Mist follows the four principles mentioned in Sect. 2.2. Mist achieves footprint expunging through providing the following features: (1) Mist offers in-time memory elimination right after the termination of the protected app to wipe the footprints left in memory. (2) Mist encrypts files to secure the footprints left in the persistent storage medium. (3) Mist monitors the interaction between the protected app and system services with fine-grained access control policy to restrict sensitive information leakage. (4) Mist obfuscates the identity of the protected app and control the access to specific system resources to avoid side-channel information eavesdropping and gathering.

3.1 Footprint Discovering

Mist conducts a comprehensive system monitoring strategy to record every possible behaviors related to footprint generating. As marked in Fig. 1, Mist implements an ephemeral container to keep monitoring sensitive information within this restricted environment. Several system components of Android are modified to achieve the design goals of footprint discovering and expunging, and the original app execution model has been re-implemented with additional or alternative steps.

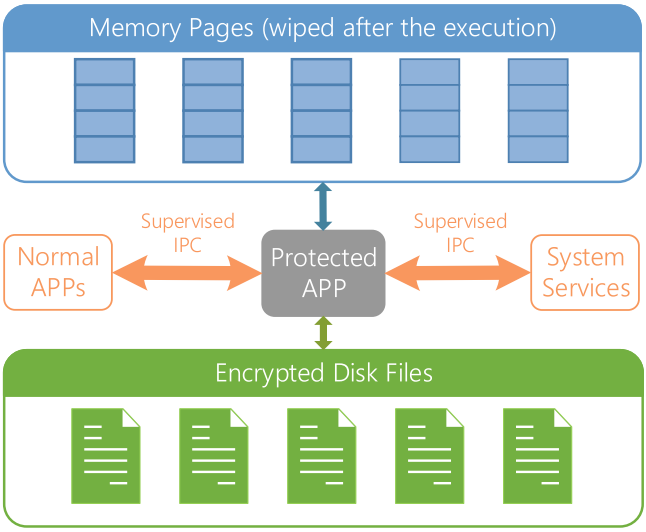

Fig. 1. The ephemeral container of MIST

Memory Monitoring. MIST records all used memory pages as footprints. Since MIST modifies the original Android OS and has the full control of the device, and thus it has the privilege to monitor every memory page used by the protected app directly.

File System Monitoring. MIST monitors an app's file I/O operations through checking all possible directories the app could write. Due to the restriction of Android sandbox, an app can only write files to several directories including its own private directory in the *userdata* partition, the SDcard, and a few temporary directories (e.g., `/data/local/tmp`). Hence the monitoring of an app's file I/O operations is implemented through checking these directories, and all data files modified by the monitored app are considered directly as footprints.

IPC Monitoring. In Android, there are many different ways to achieve data exchange between processes such as using `Intent` and `AIDL`, which based on Android's *Binder*. We monitor all IPC communication channels to investigate which system services a common Android app may use and what data these IPCs may transfer. We first modified the userspace library *libbinder.so*, which is loaded by most apps to implement IPC supervision. We also add monitoring code to the `IPCThreadState::talkWithDriver` function in `IPCThreadState.cpp` so that IPCs through a kernel driver can be supervised.

Side-Channel Footprint Monitoring. As discussed above, the side-channel footprints produced by the Android system may be stored in different system

files. To discover such side-channel footprints, we use *Inotify* to monitor the entire file system during the execution of an app. All files that have been modified during the execution will be analyzed manually, and those contain recognizable data such as packages names and UIDs are considered as footprints.

3.2 Footprint Expunging

Mist executes an app in an ephemeral container and footprints are monitored and cleaned during the entire lifetime of the app. As shown in Fig. 2, to run an app, Mist first assigns to it an encrypted partition and installs the app with obfuscated identities (package name and UID). Then the execution is monitored to block sensitive information leakage through IPC or system logging. Finally, Mist eliminates memory pages of processes of both the executed app and some tainted services at the end of the execution, and removes the app through disposing runtime residue in system logs and erasing encrypted key. In this way, Mist sanitizes the execution footprints of the app thoroughly.

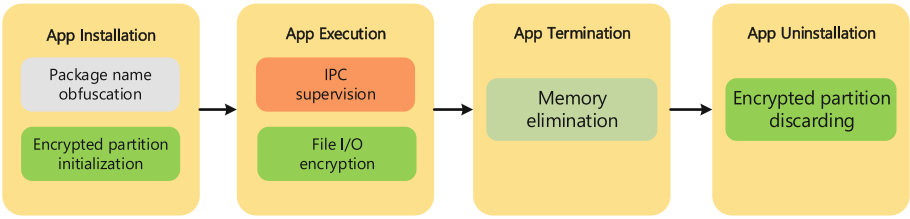

Fig. 2. App execution within the ephemeral container of Mist

Memory Page Elimination. Smart devices use volatile RAM to store in-memory data, hence by rebooting the device could the OS erase the remained data in memory thoroughly. Modern smartphones, however, are usually designed to work without a reboot for a long period. Once the attacker obtains the device, he can use memory dump tools such as *fmem* [12] and *crash* [2] to extract all physical memory pages (with root privilege), or physically access the memory (without root privilege). Most users would not reboot their devices for days, which makes the memory footprint issue even more severe.

Mist rewrites the corresponding memory space with null byte when a process terminates to deallocate the in-memory data generated securely. A kernel module is implemented to fulfill this task: it monitors the memory deallocation function in Android kernel, and whenever a monitored app exits or is killed, the kernel module reads the layout of virtual memory from `/proc/<pid>/maps` and calculates the address range to be eliminated. Then it passes the address of virtual memory should be eliminated and the PID of target process to a memory sanitization stub to erase all contents on those addresses.

File-System Encryption. MIST sets up an exclusive encrypted partition with an ephemeral key for each protected app. After that, all file operations of the protected app are redirected to this partition. MIST modified the `PackageInstaller` to change the default private directory to an exclusive encrypted partition which is set up before the installation. Then it uses `dm-crypt` to create this partition with AES-256-CBC and initialize an ephemeral key, which is then kept only in memory and will be wiped immediately after the uninstallation. During the execution, MIST hooks all file related operations in `libc` to redirect them to the encrypted partition, making sure that every recognizable footprints are only stored in this secure zone. When the app is uninstalled, or the system reboots, the ephemeral key for this encrypted partition is discarded and thus the sensitive data is cryptographically erased, leaving no recognizable footprints in flash memory.

IPC Supervision. MIST uses a configurable access control policy to supervise the IPC between the protected app and system services. The control policy only focuses on those IPCs initiated by the protected app and contain recognizable footprints, leaving others untouched. And MIST treats different IPCs with two strategies: message blocking or tainted process terminating. An IPC is blocked if it broadly contains sensitive information (e.g., individual strings related to the app), or it triggers a permanent changing of the system status (e.g., the modification of system databases or logs). For instance, if the app sends an SMS, the relevant system service will writes a record into system databases. However, the record in the database file is kept as a fingerprint due to the file wiping issue. As a result, MIST chooses to abandon such messages. For those IPCs that contain no sensitive information and only affect the status of the system temporarily (e.g., messages only kept in system memory), MIST monitors and restarts those tainted processes after the execution of the protected app, triggering the memory elimination of those processes accordingly.

Side-Channel Footprint Sanitization. Generally, MIST adopts an identity obfuscation strategy to protect sensitive information leakage against side-channel analysis. In detail, a protected app installed in MIST is assigned an obfuscated package name and a corresponding UID. We modified the `PackageInstaller` system service to assign a randomly generated package name during the installation. By doing so, the installed app runs like a new identity so that the information gathered by both malicious apps and system services would not leak the original identity. In addition, MIST utilizes relevant system functionalities to clean those side-channel footprints left in the memory of system services. For instance, the outputs of both `logcat` and `dmesg` produce sensitive information, and we can use shell commands provided by the system to clean them (`logcat -c and dmesg -c`).

For those side-channel footprints based on specific system resources such as sensor, microphone and input method, we disable these interfaces for other processes when an protected app is running. Besides, memory pages

in the cache containing verifiable information will be removed if they are no longer needed. We can manually clean them through using a shell command `echo 3 >/proc/sys/vm/drop_caches`.

4 Evaluation

This section presents the evaluation results of Mist to show that it provides a comprehensive, usable, and efficient execution mode to expunge footprints of apps. In particular, we measure Mist from three aspects: **(1)** how comprehensively can Mist discover execution footprints of popular apps; **(2)** whether the footprint expunging policy works effectively, and would it affect the normal execution of both the app and the system; **(3)** what is the performance overhead of Mist.

4.1 Discovered Footprints of Popular Apps

Traces of executions are various and some of them are very stealthy on Android device, some execution footprints are often out of regular execution scope of an app. To discovering as many as footprints (especially those recorded unintentionally by the system or reside in system buffers) we conduct comprehensive experiments to help study how common app behaviors generate footprints. We collect the top 100 apps (covering 27 different types and each of them has been downloaded for at least one million times) from Google Play market for our experiments and execute them automatically with the `monkey tool` on two commodity Android devices (Nexus 5X and Nexus 6P) with Mist. Apparently, we found memory pages and files in app's sandbox and on the SDcard as footprints. Interestingly, we also have observed many stealthy footprints in IPC and system files.

Footprints in IPC: After tracing all the IPC interfaces during the execution of these apps, we pick up representative IPC interfaces and analyze them manually to find whether these IPCs leak recognizable runtime data of a individual app or not. Table 2 shows part of the analysis of those most frequently invoked IPC interfaces due to the limitation of this paper. Among these IPC interfaces, we found only a part of them leak sensitive information to the system databases. In response, we can define corresponding access control policy for those interfaces (as the **Policy** column in Table 2 illustrated). According to the analysis, not all IPC interfaces are forbidden for a protected app. Only those IPC interfaces that leak information to flash storage should be blocked (otherwise the corresponding system services for flash I/O should be modified).

Side-channel Footprints: Previous experiments [5,9,14,16,17,31] have shown that the screenshot, keyboard input, sensor data and microphone data can all make running apps distinguishable from other apps. Our experimental results demonstrate that many system logging services also leave legacy information on the device. After analyzing the result of whole file system monitoring, we then

Table 2. The (partial) analysis of most frequently invoked IPC interfaces

IPC interface	Leaked footprint	Access control policy
`IContentProvider.insert`	in flash storage	Forbidden
`IActivityManager.startActivityAsCaller`	in memory	Restart[a]
`IPackageManager.getPackageInfo`	in memory	Restart[a]
`IActivityManager.broadcastIntent`	in memory	Restart[a]
`IGraphicBufferProducer.DEQUEUE_BUFFER`	-	Allowed
`DisplayEventConnection.REQUEST_NEXT_VSYNC`	-	Allowed
`IGraphicBufferProducer.DETACH_BUFFER`	-	Allowed
`IActivityManager.activityResumed`	-	Allowed
`IContentProvider.query`	-	Allowed

[a]Restart the corresponding system service of IPCs.

manually analyse the modified files and find there are three kinds of data related to the identity of apps which have been executed:

- The *Batterystatus* file `/data/system/batterystatus.bin` is used to record the battery usage of each app since last full charge. Device owner can check it through *Setting menu* or shell command `dumpsys batterystats` for more detailed information. All these information discloses the behaviors of specific apps.
- The *netstats* file (`/data/system/netstats/`) is used to record the network-data usage of each app. We can also infer which apps have been executed through these files.
- The *task* file (`/acct/uid/<uid>/tasks`) contains the ever used PID of a specific UID (an individual app), and the change of this file can be used to confirm the execution of apps.

Besides the footprints left on the disk, there are also some system services keep recognizable footprints in their log outputs. For instance, Android inherits the logging system from Linux, providing a various way for viewing system log. We can use shell command `dmesg` to read kernel logs messages. Another logging system is the `logcat` system of Android system debug output, which gives an abundance of information about happened activities on the device Such kind of log messages are generated by the system services and cannot be cleaned by the user. A less noticed place is the Linux page cache, which also holds the data that may expose the execution of apps. The role of the Linux page cache is to speed up access to files on the drive. In other words, the Linux page cache always keeps the data related to files opened by running app.

4.2 Usability Evaluation

Expunging. After we proved that many footprints exist in current Android app execution model, we evaluate the usability of Mist (i.e., whether Mist expunges

typical footprints) with real-world apps collected from Google Play. We utilizes five forensic analysis tools to detect footprint: Oxygen mobile forensic Suite is a mobile forensic software for logical analysis of smartphones and PDAs developed by Oxygen Software. Andriller is software utility with a collection of forensic tools for smartphones. Andriller can extract and analyse data stored in the userdata partition if the device is rooted. UFS Explorer is a data recovery software for data loss cases of different complexity. Here we use it to analysis the disk dump to recover deleted data related to executed apps. LiME is a widely used Linux kernel module to dump RAM contents of the device. In most cases, LiME is used to perform live memory analysis. In addition, we developed Side-channel FP Collector, an Android forensics tool based on our study of side-channel footprints of Android. It extracts the side-channel data and restores the behaviors of executed apps. We use these tools to analyze the following data as suspicious information leakage sources:

- memory dump of the Android system;
- dump of the *userdata* and the *SDcard* partitions;
- system database files;
- battery usage log;
- network usage log;
- files in `/acct/uid`;
- output of `logcat`.

We executed the tested apps in a device with Mist and in a native Android device, respectively. Then we used forensic tools to analyze both devices and compared the analyzing results. The results are shown in Table 3. For the native Android device, forensic tools detected many footprints listed in the Table. On the contrary, Mist expunged those footprints thoroughly and none of the forensics tools could discover useful footprint.

Compatibility. To test the compatibility of Mist, we expand our samples mentioned in Sect. 4.1 to the top 200 apps from Google Play. Each of them is executed on a Nexus 5X with Mist for 10 min with the `monkey tool`. We find some remarkable failures in this experiment and here we provide our analysis to these failures and our improvement to Mist.

In the beginning, we found that 80 out of 200 selected apps crashed. We analyze the crash log and find that most of them are caused by the inconsistent between origin package name and the obfuscated one. Some apps use system API `getPackageName` to get its own package name during runtime, and sometimes such package name is used to interact with components of the app. For example, the dynamically generated string "getPackageName() +'.a.b.c'" represents to a specific class within the app, app can use this string to visit this class through reflection mechanism. But in Mist, the return value of `getPackageName` has been changed to the obfuscated package name, which is inconsistent with the original one. Because we do not modify the class name synchronously, the app will throw the `ClassNotFound` exception when visiting the class through

Table 3. Footprint expunging using Mist

Forensics tools	Data source	Native android	Mist
Oxygen Forensics Suite	Disk files	photos taken by executed apps;	-
	System databases	SMS records and account records related to executed apps;	-
	Call logs	logs that contain package and activity names;	-
Andriller	Disk files	files that contain package names, recognizable strings and icons related to executed apps;	-
UFS Explorer	Deleted files	xml files, icons and pictures related to executed apps;	-
LiME	Memory dump	package names and recognizable strings left in memory;	-
Side-channel FP Collector	Battery usage	records that contain package names, icon and run time of executed apps;	-
	Network usage	records that contain package names and icons of executed apps;	-
	ACCT files	UIDs and PIDs of executed apps;	-

`getPackageName`. This problem also happens when the app visits resources defined in the *xml* files. Another problem is that some apps may conduct an Activity invoking through an `Intent` with hard-coded Activity name, which also involves the package name inconsistency.

To address this issue, we further modified the implementation of Mist: we modified the `findClass` function in the `BaseDexClassLoader` class to change the obfuscated package name to the original one and modified the function `startActivityMayWait` in package `com.android.server.am` to change the hard coded package name to our obfuscated one. Note that these package name patching only happen in the memory of the protected app and will not conflict with the installation process, and thus solves the inconsistency problem. After solving these inconsistent problems, All the 200 selected apps work correctly with Mist.

Since Mist provides a series of extra security features, it also brings some inconveniences to the users of protected apps. Users should reinstall the protected apps before the execution and some system services are disabled due to the IPC supervision. But considering Mist is designed for the users who have additional security demands, such compromises are acceptable while chasing the goal of leaving none footprint in the device.

4.3 Performance Overhead

The specific execution of an app under Mist inevitably introduces performance overhead. To evaluate the overhead of Mist precisely, we run the top 200 apps from Google Play automatically on both normal Nexus 6p and Nexus 5x with

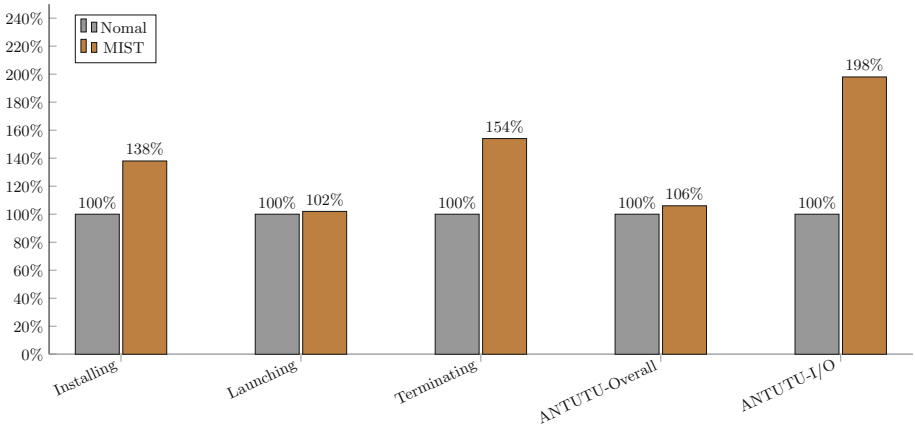

Fig. 3. Performance overhead of MIST

Mist using `Monkey`, and record the time each app spends on the installation, launching, and terminating stage, respectively. We also use the `AnTuTu` Benchmark (a state-of-the-art Benchmark suite in Google Play) to evaluate the running overhead of Mist.

The results of performance overhead are shown in Fig. 3. Among the three typical stages of app lifetime: *installing*, *launching*, and *terminating* we can see that the mostly influenced stage is the installation. The introduced encrypted partition operation in this stage brings a 38% (6.23 s) overhead on average to set up an encrypted partition. However, since the setup of the encrypted partition has been done before the execution, it will not affect the normal execution of the app. Moreover, after the execution, the memory elimination brings a 54% (0.06s) overhead on average. Considering that these operations are only executed once for an execution, we believe that Mist does not produce a significant impact on user experience.

Except those operations happened before and after the execution, From Fig. 3 we can see that the use of Mist only slightly affect the normal execution: performance overhead for both the launching and the overall execution of an app are less than 6%. Note that although the overhead for the file I/O operation is 98% (due to the file encryption) according to the AnTuTu I/O testing, this is the maximum overhead since seldom app continuously reads or writes file during its execution as the Benchmark testing. Therefore, we argue that the overhead of file I/O is acceptable for most application scenarios.

5 Related Work

Private execution is first introduced to web browsers as private browsing mode. This mode guarantees that an attacker who takes control of the machine after the user exits private browsing can learn nothing about the user's actions while in

private browsing. Although private browsing mode has already been supported by four major browsers (Internet Explorer/Edge, Firefox, Chrome, and Safari), recent study by Aggarwal *et al.* [3] points out that private browsing is used differently from how it is marketed. Xu *et al.* further shows that Chrome and Firefox do not correctly clear some of their browsing footprints [29], and they propose Ucognito, a prototype system to enhance existing private browsing mode of browsers. Similar to Mist, Ucognito adopts a filesystem overlaying approach with a sandbox filesystem to assure no persistent modification is stored. However, it only focuses on web browsing and is not universal.

To Generalize private execution on commodity systems, researchers proposed a myriad of design schemes. Lacuna [8] is a comprehensive system that allows users to run programs in *private sessions* of desktop and server Linux systems. It uses a special *ephemeral channel* to isolate the protected program and peripheral devices while making it possible to delete the memories of this communication from the host. However, it relies on a modified QEMU-KVM hypervisor to achieve this functionality and is quite heavyweight, and thus is demanding to be ported to Android devices. PrivExec [18] is an operating system service for private execution. It allows any application to execute in a private execution mode where storage writes will not be recoverable by others during or after execution. PrivExec only requires the modification of the operating system and is promising to be ported to Android platform. However, it does not consider the forensic deniability issue. TpriVexeC [7] improve the performance of private execution via keeping both I/O and runtime data of private applications in memory only. But it does not consider that the secure deallocation and sensitive runtime data may be leaked if memory forensic analysis is employed.

For Android system, CleanOS [28] is a representative system that fulfills memory encryption based protection. It identifies and tracks sensitive data in RAM and on stable storage and encrypts it. CleanOS leverages a trusted, cloud-based service to manage encryption keys, and evicts a key to the cloud when the data is not in active use on the device. Because it mainly focuses on in-memory objects encryption and does not consider the comprehensive behaviors of an app thoroughly, CleanOS may either leak sensitive information via system services interaction (file I/O, IPC, system API invoking) or be incompatible with many standard functionalities provided by the OS. AppShell [30] is a practical Android app private execution solution that supports both in-memory and on-disk data protection by transparently encrypting the data, which requires neither framework modification, nor the root privilege. However, it is possible that sensitive data may reside in places that cannot be touched by AppShell.

Compared with them, Mist provides a more comprehensive footprint expunging mechanism on Android. We demonstrated it in a comparison between Mist and previous solutions related to footprint expunging in Table 4: except Mist, other solutions all fail to achieve footprint expunging on Android for violating at least five items.

Table 4. Comparison to other footprint expunging solutions

	MIST	PrivateDroid	CleanOS	Lacuna	Privexec	Ucognito
Memory	Y	N	Y	Y	Y	Y
File	Y	N	Y	Y	Y	N
IPC	Y	N	N	Y	Y	N
Process status	Y	N	N	Y	N	N
Battery usage	Y	N	N	-	-	-
Network usage	Y	N	N	-	-	-
Screen	Y	N	N	Y	N	N
Microphone	Y	N	N	-	-	-
Sensor	Y	N	N	-	-	-
Keyboard input	Y	N	N	-	-	-
System log	Y	N	N	Y	N	N

[1] "-" indicates that the tested solution works on Linux, which contains no such footprint.

Some researchers use sandboxing techniques to protect Android app against the attack from malwares. Android app sandboxing provides each app a separated execution environment so that third-party apps can't detect and tamper the execution of protected apps. According to our threat model, we assume the attacker has physical access to the device, which means the attacker can analyze the disk and memory data directly. That's why MIST provides features that not considered by sandboxing techniques such as filesystem encryption and memory page elimination.

6 Conclusion

In this paper, we present a privacy enhancement system MIST to achieve the goal of execution footprint expunging of Android apps. MIST adopts a comprehensive footprint detecting and expunging policies, and works with real-world Android devices. We evaluate MIST with popular Android apps and demonstrate that MIST can eliminate most execution footprints compared with regular Android OS.

References

1. What if the FBI tried to crack an Android phone? We attacked one to find out. https://theconversation.com/what-if-the-fbi-tried-to-crack-an-android-phone-we-attacked-one-to-find-out-56556
2. White Paper: Red Hat Crash Utility. http://people.redhat.com/anderson/crash_whitepaper/

3. Aggarwal, G., Bursztein, E., Jackson, C., Boneh, D.: An analysis of private browsing modes in modern browsers. In: USENIX Security Symposium, pp. 79–94 (2010)
4. Albano, P., Castiglione, A., Cattaneo, G., De Santis, A.: A novel anti-forensics technique for the android os. In: 2011 International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA), pp. 380–385. IEEE (2011)
5. Cai, L., Chen, H.: Touchlogger: inferring keystrokes on touch screen from smartphone motion. HotSec **11**, 9 (2011)
6. Chen, J., Chen, H., Bauman, E., Lin, Z., Zang, B., Guan, H.: You shouldnt collect my secrets: thwarting sensitive keystroke leakage in mobile IME apps. In: 24th USENIX Security Symposium (USENIX Security 15), pp. 657–690 (2015)
7. Djoko, J.B., Jennings, B., Lee, A.J.: Tprivexec: private execution in virtual memory. In: Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy, pp. 285–294. ACM (2016)
8. Dunn, A.M., et al.: Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels. In: Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), pp. 61–75 (2012)
9. Fawaz, K., Feng, H., Shin, K.G.: Anatomization and protection of mobile apps location privacy threats. In: 24th USENIX Security Symposium (USENIX Security 15), pp. 753–768 (2015)
10. Halderman, J.A., et al.: Lest we remember: cold-boot attacks on encryption keys. Commun. ACM **52**(5), 91–98 (2009)
11. Jana, S., Narayanan, A., Shmatikov, V.: A scanner darkly: protecting user privacy from perceptual applications. In: 2013 IEEE Symposium on Security and Privacy (SP), pp. 349–363. IEEE (2013)
12. Kollár, I.: Forensic ram dump image analyser. Master's Thesis, Charles University in Prague (2010)
13. Li, L., Zhao, X., Xue, G.: Unobservable re-authentication for smartphones. In: NDSS, pp. 1–16 (2013)
14. Lin, C.C., Li, H., Zhou, X.Y., Wang, X.: Screenmilker: how to milk your android screen for secrets. In: NDSS (2014)
15. Lin, Z., Rhee, J., Wu, C., Zhang, X., Xu, D.: Dimsum: discovering semantic data of interest from un-mappable memory with confidence. In: Proceedings of NDSS (2012)
16. Michalevsky, Y., Boneh, D., Nakibly, G.: Gyrophone: recognizing speech from gyroscope signals. In: 23rd USENIX Security Symposium (USENIX Security 14), pp. 1053–1067 (2014)
17. Nan, Y., Yang, M., Yang, Z., Zhou, S., Gu, G., Wang, X.: Uipicker: user-input privacy identification in mobile applications. In: 24th USENIX Security Symposium (USENIX Security 15), pp. 993–1008 (2015)
18. Onarlioglu, K., Mulliner, C., Robertson, W., Kirda, E.: Privexec: private execution as an operating system service. In: 2013 IEEE Symposium on Security and Privacy (SP), pp. 206–220. IEEE (2013)
19. Peters, T.M., Gondree, M.A., Peterson, Z.N.: Defy: a deniable, encrypted file system for log-structured storage (2015)
20. Reardon, J., Marforio, C., Capkun, S., Basin, D.: User-level secure deletion on log-structured le systems. In: Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security, pp. 63–64. ACM (2012)
21. Saltaformaggio, B., Bhatia, R., Gu, Z., Zhang, X., Xu, D.: Guitar: piecing together android app guis from memory images. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 120–132. ACM (2015)

22. Saltaformaggio, B., Bhatia, R., Gu, Z., Zhang, X., Xu, D.: VCR: app-agnostic recovery of photographic evidence from android device memory images. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 146–157. ACM (2015)
23. Saltaformaggio, B., Bhatia, R., Zhang, X., Xu, D., Richard III, G.G.: Screen after previous screens: spatial-temporal recreation of android app displays from memory images. In: USENIX Security Symposium, pp. 1137–1151 (2016)
24. Saltaformaggio, B., Gu, Z., Zhang, X., Xu, D.: Dscrete: automatic rendering of forensic information from memory images via application logic reuse. In: 23rd USENIX Security Symposium (USENIX Security 14), pp. 255–269 (2014)
25. Shu, J., Zhang, Y., Li, J., Li, B., Gu, D.: Why data deletion fails? a study on deletion flaws and data remanence in android systems. ACM Trans. Embed. Comput. Syst. (TECS) **16**(2), 61 (2017)
26. Simon, L., Anderson, R.: Security analysis of android factory resets. In: 4th Mobile Security Technologies Workshop (MoST) (2015)
27. Skillen, A., Mannan, M.: On implementing deniable storage encryption for mobile devices (2013)
28. Tang, Y., Ames, P., Bhamidipati, S., Bijlani, A., Geambasu, R., Sarda, N.: Cleanos: limiting mobile data exposure with idle eviction. In: Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), pp. 77–91 (2012)
29. Xu, M., Jang, Y., Xing, X., Kim, T., Lee, W.: Ucognito: private browsing without tears. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 438–449. ACM (2015)
30. Yajin, Z., Kapil Singh, X.J.: Appshell: making data protection practical for lost or stolen android devices. In: IEEE/IFIP Network Operations and Management Symposium. IEEE (2016)
31. Zhang, N., Yuan, K., Naveed, M., Zhou, X., Wang, X.: Leave me alone: app-level protection against runtime information gathering on android. In: 2015 IEEE Symposium on Security and Privacy, pp. 915–930. IEEE (2015)
32. Zhang, X., Ying, K., Aafer, Y., Qiu, Z., Du, W.: Life after app uninstallation: are the data still alive? data residue attacks on android. In: Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, California, USA(2016)
33. Zhou, X., et al.: Identity, location, disease and more: inferring your secrets from android public resources. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pp. 1017–1028. ACM (2013)