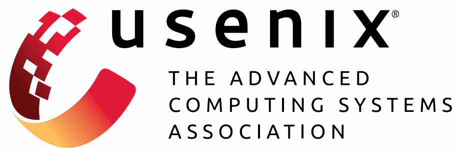


Peep With A Mirror: Breaking The Integrity of Android App Sandboxing via Unprivileged Cache Side Channel

Yan Lin, *Jinan University;* Joshua Wong, *Singapore Management University;*
Xiang Li and Haoyu Ma, *Zhejiang Lab;* Debin Gao, *Singapore Management University*

https://www.usenix.org/conference/usenixsecurity24/presentation/lin-yan

This paper is included in the Proceedings of the
33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the
33rd USENIX Security Symposium
is sponsored by USENIX.

Peep With A Mirror: Breaking The Integrity of Android App Sandboxing via Unprivileged Cache Side Channel

Yan Lin[1], Joshua Wong[2], Xiang Li[3], Haoyu Ma[3]*, Debin Gao[2]

[1]*Jinan University, Guangzhou, China*
[2]*Singapore Management University, Singapore, Singapore*
[3]*Zhejiang Lab, Hangzhou, China*

Abstract

Application sandboxing is a well-established security principle employed in the Android platform to safeguard sensitive information. However, hardware resources, specifically the CPU caches, are beyond the protection of this software-based mechanism, leaving room for potential side-channel attacks. Existing attacks against this particular weakness of app sandboxing mainly target shared components among apps, hence can only observe system-level program dynamics (such as UI tracing). In this work, we advance cache side-channel attacks by demonstrating the viability of non-intrusive and fine-grained probing across different app sandboxes, which have the potential to uncover app-specific and private program behaviors, thereby highlighting the importance of further research in this area.

In contrast to conventional attack schemes, our proposal leverages a user-level attack surface within the Android platform, namely the dynamic inter-app component sharing with package context (also known as DICI), to fully map the code of targeted victim apps into the memory space of the attacker's sandbox. Building upon this concept, we have developed a proof-of-concept attack demo called ANDROSCOPE and demonstrated its effectiveness with empirical evaluations where the attack app was shown to be able to successfully infer private information pertaining to individual apps, such as driving routes and keystroke dynamics with considerable accuracy.

1 Introduction

In the past decade, Android has become the predominant mobile operating system and is boasting the largest market share in global smartphone sales volume [4]. With Android smartphones serving as ubiquitous personal handheld devices, countless personal private data (e.g., user email and contact list) and a variety of persistent identifiers (e.g., IMEI) are accessed and processed daily. The emergence of the Android ecosystem has also attracted numerous malicious attacks [12], a large portion of which either gain device control [13, 23] or steal private user information such as chats [37] and geographic locations [22]. To protect user privacy from unauthorized access, Android sandboxes user-space apps to prevent them from interacting arbitrarily with each other and the key system components. When first introduced in Android 5, the app sandboxing mechanism was implemented on top of the mandatory user-based access control of SELinux [43]. It inherits the UID/GID permission model of Linux and enforces each app to be run in an isolated process space by assigning a unique UID at installation time. By default, two isolated apps (without a pre-established trust relationship) cannot interact; any app is not allowed to access system components or resources without the corresponding permissions. Subsequent Android versions continued to improve their security mechanisms, e.g., Android 10 introduced Scoped Storage [14] which restricts an app's access to external storage.

Nevertheless, it has been shown in real-world practices that an app may circumvent Android's sandboxing and gain access to protected data without user consent, which is mostly achieved by means of covert or side-channel techniques. In particular, covert channels enable communication between two colluding apps so that one app can share its permission-protected data with another app lacking those permissions [27]. The research community has previously explored the potential for covert channels in Android using local sockets and shared storage [32], as well as other unorthodox means such as vibrations and accelerometer data [6]. However, typical covert channels can be easily detected by dynamic analysis, e.g., TaintDroid [15] and XManDroid [10]. Side channels, on the other hand, occur when there are alternate means to access the protected resource that is not audited by security mechanisms. Side-channel attacks take advantage of unintended leakage of information through features such as power consumption, electromagnetic emissions, timing, or even sound. The typical scenarios of side-channel attacks involve inferring sensitive data (e.g., encryption keys and

*Corresponding author

passwords) and private user information (examples including gender [33], identity [44], and even driving routes [35]). Attempts have also been made to exploit cache-based side channels for the purpose of breaching Android's app sandboxing. For instance, ARMageddon launched side-channel attacks on `libinput.so`, a system-shared library, to monitor keyboard presses [30]. Zhang et al. used the same strategy on another system-shared library, namely `libinputflinger.so` [50]. However, existing work only managed to exploit shared code across different processes due to Android's system architecture; in addition, these adversarial approaches are susceptible to various system factors and noises in practice [47], limiting their capabilities to rough estimations of what is happening at a coarse granularity.

In this paper, we propose ANDROSCOPE, a fine-grained side-channel attack that compromises the Android app sandboxing mechanism. Our attack leverages a user-level attack surface of the Android platform, namely the dynamic code loading mechanism using package context (DICI). Specifically, while the previous work explored the possibility of exploiting DICI in code reuse attacks [17], our investigation targets a more fundamental question, namely "*how code of another app is accessed by components within the current sandbox during* DICI", and find that *when an app attempts to invoke external methods via* DICI*, it loads the corresponding code files (belonging to the victim app) into the memory space of the attack app as a side effect*. We confirmed that this undocumented feature applies to all types of executables on the Android platform, including Dalvik Executable (Dex), ahead-of-time (AOT) compiled code, and third-party native libraries, and such code loading turns out to be successful even if the external invocation request is rejected (due to proper security measures). This observation thus enables carefully crafted cache-based side-channel attacks to completely break app sandboxing and permission-based access control, allowing a stealthy app to, for the first time, monitor detailed runtime behavior of other apps non-intrusively without any unusual privileges.

We tested the effectiveness of ANDROSCOPE on monitoring the method execution of a number of open-source apps and confirmed that ANDROSCOPE could effectively extract sensitive and app-specific user behavior. As two representative use cases, we demonstrate that ANDROSCOPE successfully infers driving routes taken by a navigation app and keystroke dynamics of a communication app. In conclusion, we make the following contributions.

- For the first time, we identified another layer of security threat exposed by Android's undocumented DICI mechanisms, which could lead to inter-app privacy leakage that bypasses Android's app sandboxing.
- We have implemented a proof-of-concept attack framework called ANDROSCOPE, which leverages the DICI vulnerability to conduct non-intrusive inter-app runtime behavior monitoring via the cache-based side channel with zero privilege. ANDROSCOPE shows its capability to directly probe a victim's private code, surpassing prior methods that relied on shared libraries and system APIs.
- We conducted a comprehensive evaluation on the effectiveness of ANDROSCOPE using real-world devices and apps, which confirmed the validity of DICI-based sandbox breach and suggested that the issue may be universal to all Android-based systems. The result also demonstrates that ANDROSCOPE addresses concerns about background noise as raised in previous approaches focused on the shared libraries and system APIs.

2 Background

2.1 Android App Sandboxing

App sandboxing is a fundamental security feature of the Android operating system, which helps protect the device and user data by isolating each app from both other apps and the rest of the system. This ensures that apps cannot interfere with each other's operations or access unauthorized resources and data. The construction of the app sandbox is based on three mechanisms, namely Discretionary Access Control (DAC) [11], Mandatory Access Control (MAC) [41], and the Android permissions [16] mechanism. Both DAC and MAC are inherited from Linux. Specifically, DAC restricts access to resources based on user and group identity. By assigning each application a unique UNIX user ID (UID) and a dedicated directory, Android runs each app in its own independent user space with the file system set up in a way that each app can only access its own data and resources. MAC further enforces finer-grained protections by dictating whether an in-process action is allowed based on a set of pre-defined policies concerning the security contexts (i.e., collections of security labels that classify resources) of the involved parties. Last but not the least, Android permissions control data sharing among apps of UIDs and system resources (such as camera, GPS, contacts, and etc.). An app must declare permissions in advance (or, in the case of dangerous permissions, explicitly) so that invocations can be granted.

Android's use of MAC policy has evolved over time to meet the increasing demands of mobile security. In the early versions of Android (before Android 4.3), it relied heavily on DAC and allowed apps to run with the permissions of the user who installed them. This provided only limited security as apps had broad access to the privileged resources. With the release of Android 4.3, SELinux was introduced as an additional layer of security, but the implementation was permissive by default, i.e., it logged policy violations but did not actively enforce policies. Android finally switched its SELinux to enforcing mode since Android 5 to actively enforce the pre-defined security policies.

2.2 Dynamic Inter-App Component Sharing with Package Context

The initial purpose of DICI is for inter-app communication so that one app can leverage resources from other apps during its execution. Listing 1 shows an example of DICI used in plugin implementation in the app with package name *kl.ime.oh*, which intends to invoke a method named `predict` that belongs to the `M` class of a Chinese keyboard plugin app called *klye.hanwriting*. Specifically, the API `createPackageContext` is called at Line 7 to create a context of *klye.hanwriting*. The second parameter of this API is known as flags, allowing developers to specify how should the package context be created, most notably:

- 0x0001 (*CONTEXT_INCLUDE_CODE*), allowing access to the code in the loaded package.
- 0x0002 (*CONTEXT_IGNORE_SECURITY*), ignoring any security restrictions. When it is enabled with *CONTEXT_INCLUDE_CODE*, code is loaded even if it unsafe to do so. features of the accessed resources.

```
1  public boolean loadHW() {
2      try {
3          if (M.i.getPackageManager().
               ↪ getPackageInfo(
4          "klye.hanwriting", DoneRec).versionCode
               ↪  < 19){
5              return false;
6          }
7          Class<?> loadClass = M.i.
               ↪ createPackageContext("klye.
               ↪ hanwriting", 3).getClassLoader()
               ↪ .loadClass("klye.hanwriting.M");
8          Class[] clsArr = new Class[3];
9          clsArr[DoneRec] = String.class;
10         clsArr[1] = Character.TYPE;
11         clsArr[2] = Integer.TYPE;
12         this.m1 = loadClass.getMethod("predict"
               ↪ ,clsArr);
13         ......
14         return true;
15     } catch (Throwable unused) {
16         return false;
17     }
18 }
```

Listing 1: Example of direct inter-app code invocation

In this example, the value 3 ($0x0001|0x0002$) enables both flags *CONTEXT_INCLUDE_CODE* and *CONTEXT_IGNORE_SECURITY* on loading the class *klye.hanwriting.M* on-the-fly. The method object is acquired with the class object containing it (Lines 9-13). Android first locates the APK file based on the package name provided, and then opens it and dynamically loads `classes.dex` into the current address space.

Despite the lack of explicit documentation, DICI is a legitimate mechanism used by various apps published on Google Play Store (see more details in Section 3.6). It is important to emphasize that *even if the method invocation fails (for valid reasons like lack of permissions), the file loading operations would have been completed as a side effect and would not be revoked as long as the initiating app is not terminated*. Our tests confirmed that this undocumented component sharing mechanism is viable on the most recent Android 14.

Existing work [17] pointed out that this mechanism can be used to plagiarize supposedly-protected functionalities, but the potential risk of side-channel attacks was overlooked. As a matter of fact, both `base.vdex` and `base.odex` are used by the Android Runtime for actual execution of apps, such that the overhead of repeatedly verifying and optimizing `classes.dex` can be avoided. With these observations, we investigate how the DICI mechanism could be used to launch side-channel attacks with the purpose of fully circumventing Android app sandboxing.

2.3 CPU Caches and Side-Channel Attacks

Program execution often has temporal and spatial locality, i.e., the most recently accessed memory addresses as well as nearby addresses are often accessed in the near future. To exploit locality, modern CPU architectures use caches to store recently accessed code and data. These caches are often organized into multiple levels with different sizes and access speeds. For example, on ARM CPUs, there are commonly two levels of caches L1 and L2, with L1 being faster and smaller while L2 being larger and slower. On multi-core CPUs, lower-level caches (L2) are often shared among multiple CPU cores. Modern caches are usually organized with an N-way associative table. The basic unit of memory allocation in a cache is called a line or cache line of a typical size of 64 bytes.

An adversary could infer secret information about a running program by observing its use of CPU caches, which is typically called cache side-channel attacks. Initially, cache side-channel attacks were performed on cryptographic algorithms [25, 26, 36, 40, 45]. Previous studies have explored different types of cache side-channel attacks: EVICT+TIME [38], PRIME+PROBE [38], FLUSH+RELOAD [48], and EVICT+RELOAD [19]. EVICT+TIME and PRIME+PROBE allow an adversary to determine which cache sets are used during the victim's computation and have been exploited to attack cryptographic algorithms [38]. In particular, in EVICT+TIME attacks, the attacker first causes the victim to run, preload its cache lines, and establish a baseline execution time. The attacker then evicts a cache line of interest and runs the victim code again, with a variation in execution time indicating that the line of interest was accessed. PRIME+PROBE attacks pre-load every cache line in the target cache set with its own memory blocks so that the adversary can make sure her future memory

accesses will be served by the cache unless some of the cache lines are evicted by the victim program during its execution. Therefore, her own cache misses will reveal the victim's cache usage.

FLUSH+RELOAD [48] allows an attacker to determine which specific instructions are executed and which specific data is accessed by the victim program. Specifically, in FLUSH+RELOAD attacks, the adversary shares some physical memory pages (e.g., through shared libraries and system components) with the victim. By issuing flush instructions on certain virtual address ranges, the adversary can flush the (physical) cache lines that correspond to this address range out of the entire cache hierarchy. Therefore, any future reading (RELOAD) of the cache lines will be slower because they are loaded from the memory unless they have been accessed by the victim. Since some ARM processors do not have a flush instruction, EVICT+RELOAD has been proposed by replacing the flush instruction in FLUSH+RELOAD by eviction. This paper makes use of FLUSH+RELOAD to trace the execution of an app in Android.

3 ANDROSCOPE

In this section, we first provide an overview of the proposed attack starting with the threat model. We then describe the individual components of the attack in detail.

3.1 Threat Model

In the attack scenarios assumed by ANDROSCOPE, the adversary's general goal is to passively eavesdrop behavior of another victim app. The adversary is assumed to be local, i.e., the attack process is co-located with the victim process on the same physical device, and are both executed within the same operating system environment. Nonetheless, the attack process is considered to be isolated from the victim by app sandboxing and is not allowed to break such isolation via any intrusive approaches. We assume that the physical device is a multi-core system, and the attack and victim processes can run simultaneously on different cores. The attack process can create a few threads that run continuously in the background, but cannot attempt to obtain root privileges or request any special permissions. The adversary could perform offline reverse engineering of the victim app and correlate the memory addresses of methods to be targeted by its runtime surveillance.

3.2 Overall Workflow

Rather than leaking system-wide information via shared code components as demonstrated in previous work, ANDROSCOPE is designed to monitor app-specific methods to circumvent the protection of app sandboxing. In particular, utilizing the DICI mechanism enables ANDROSCOPE to load

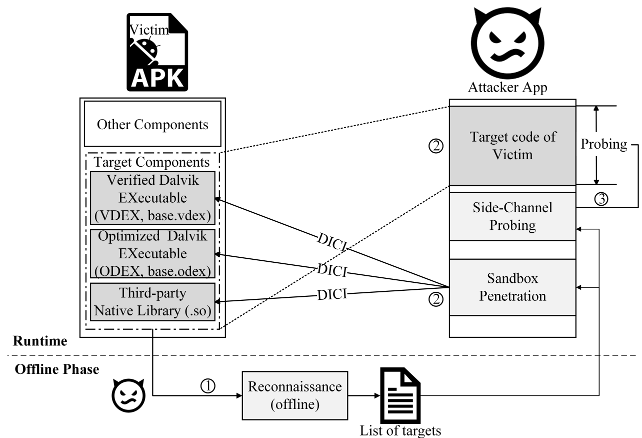


Figure 1: Overview of ANDROSCOPE

all the necessary executables of a victim app into the attacker's own memory space, creating a side-channel attack surface for the attacker process to probe any method of the victim as if observing it from inside using an endoscope (hence the name "ANDROSCOPE").

Figure 1 illustrates the overall workflow of the proposed attack. ANDROSCOPE consists of three phases to create an app-specific side-channel attack.

- **Phase 1: Reconnaissance (Section 3.3)** In order to detect sensitive information through side-channel probing, the adversary first determines which of the victim app's executable files are required and the offsets of the particular methods that are of interest. This is done offline with the adversary obtaining and conducting reconnaissance on the victim app's APK in advance.

- **Phase 2: Sandbox Penetration (Section 3.4)** With the reconnaissance results from Phase 1, the adversary deploys ANDROSCOPE (likely via a cover app with seemingly reasonable functionality) to detect whether the targeted victim app has also been installed. Upon a positive answer, ANDROSCOPE would leverage the DICI mechanism to load necessary code files of the victim app. This ensures that the ANDROSCOPE process shares physical code pages with the victim app of the DICI-loaded code files.

- **Phase 3: Side-Channel Probing (Section 3.5)** After the prior steps, the adversary finally launches side-channel probing of FLUSH+RELOAD to detect fine-grained traces of executions occurring within the victim app's code region, and thus gaining awareness of the sensitive runtime status of the victim.

To achieve the expected attack goal, a number of technical details are critical for the implementation of ANDROSCOPE. Note that our study was carried out on two smartphones as

Table 1: Experimental Devices

	Xiaomi Mi 11 Lite	Pixel 4a
Soc	Snapdragon 732G	Snapdragon 730G
Architecture	big.LITTLE	big.LITTLE
Processors	2x Kryo 470 Gold as big cores 6x Kryo 470 Silver as little cores	

listed in Table 1, which are both mounted with Qualcomm's ARMv8-A CPUs that adopt big.LITTLE architectures. Both devices have the cache line flushing instruction unlocked by default, making it available in user space.

3.3 Reconnaissance

Recall that the purpose of the reconnaissance phase is to determine which parts of a victim code are of interests in the side-channel attack. While the specific methods to be targeted can be selected via a careful manual analysis, the adversary must still be able to parse different types of executable files such that the linear address of these methods at runtime can be correctly resolved.

Today, the Android platform uses a runtime environment called Android runtime (ART) to execute methods in apps. ART uses ahead-of-time (AOT) compilation and, starting from Android 7.0, uses a combination of AOT, just-in-time (JIT), and profile-guided compilations. Specifically, before Android 8.0, an app is installed without any AOT compilation, but an Optimized Dalvik EXecutable (ODEX) named `base.odex` is generated with only Dalvik bytecode. When the device is idle and charging, a compilation daemon AOT-compiles frequently used code to machine code based on a profile generated during the first few runs, and inserts the instructions into the ODEX file. Starting from Android 8.0, however, ART stopped merging the DEX files into `base.odex` but introduced a VDEX file called `base.vdex` to store the copy of original DEX files. In addition, an increasing number of Android developers are incorporating third-party native libraries in their applications for code reuse, CPU-intensive tasks, and other purposes. According to the new runtime environment, the adversary therefore needs to parse all the aforementioned types of files so that the address of any potential targets of our attack can be obtained.

3.3.1 VDEX file Parsing

In order to obtain the target addresses of methods executed by the Dalvik Virtual Machine (DVM), we employ the vdexExtractor [5] utility for the parsing of VDEX files. vdexExtractor represents a command-line tool designed for the decompilation and extraction of Android Dex bytecode from VDEX files. These VDEX files are produced in conjunction with ODEX files during the optimization of bytecode by the dex2oat ART runtime compiler.

More specifically, we analyze the output file generated by vdexExtractor and extract the offset value (referred to as `vdex_offset`) corresponding to the method of interest based on its method name. It is worth noting that each method is subject to an additional fixed offset of 0x40 on our devices. The extra offset 0x40 is to account for a 4-byte magic number and a 4-byte version field at the beginning of the file. Consequently, the ultimate address utilized in the FLUSH+RELOAD attack is computed as follows: `vdex_base_address + 0x40 + vdex_offset`. Here, `vdex_base_address` is determined by examining the mapped location of `base.vdex` of the victim within the `/proc/self/maps` directory of the attacker.

3.3.2 ODEX file Parsing

We employ the oatdump utility to acquire the target offset, denoted as `odex_offset`, pertaining to a native-compiled method subject to AOT compilation. Furthermore, our investigation reveals the presence of a constant offset of 0x1000 (which corresponds to the ELF header page) that is uniformly applied to each native-compiled method on our devices. The ultimate address computation is achieved by summing the values of `odex_base_address`, which is ascertained by inspecting the mapped location of `base.odex` associated with the victim within the `/proc/self/maps` directory of the attacker, with 0x1000 and `odex_offset`.

3.3.3 Third-Party Native Library Parsing

Parsing a native library can be accomplished with ease by employing Executable and Linkable Format (ELF) analysis tools like objdump and readelf. These tools facilitate the retrieval of method offsets. In this paper, we make use of objdump to disassemble the native library and obtain the offset of the methods.

3.4 Sandbox Penetration

The most fundamental intention of Android's app sandboxing is to prevent any user process from accessing another process's private resources (e.g., code, data, config files). However, our observation on the undocumented DICI mechanism, especially regarding its side effect of dynamic code loading, puts a question mark on the effectiveness of app sandboxing.

Here we select `Organic Maps`, an open-source app from F-Droid, as an exemplary target to illustrate how DICI can be exploited to penetrate Android app sandboxing. Listing 2 shows the code snippet used for this mock attack testing, where the invocations at Lines 3 and 4 implement DICI in the exact same way as described in Section 2.2, only that the parameter `victimPackageName` is set to `app.organicmaps`, the package name of the victim app. We find that these two

```
pubic class MainActivity{
    public void LoadVictim(String victimPackageName, String victimClass, Optional<String> victimMethod
        ↪ ){
        Context ike = this.createPackageContext(victimPackageName,  Context.Context_INCLUDE_CODE |
            ↪ Context.CONTEXT_IGNORE_SECURITY);
        ClassLoader loader = ike.getClassLoader();
        /*only loading third-party native libraries needs to execute the following code*/
        Class utils = loader.loadClass(victimClass);
        Method method = util.getDeclaredMethod(victimMethod.get(), String.class);
        method.setAccessible(true);
        method.invoke(null,"/sdcard/Documents");
    }
}
```

Listing 2: Code snippet of attack app

```
72c5fc3000-72c71b9000 r-xp 00000000 /data/app/.../app.organicmaps/lib/arm64/liborganicmaps.so
72c77fc000-72c8842000 r-xp 002b8000 /data/app/.../app.organicmaps/oat/arm64/base.odex
72c885f000-72c91be000 r--p 00000000 /data/app/.../app.organicmaps/oat/arm64/base.vdex
72c950c000-72c9512000 r-xp 0024f000 /data/app/.../com.example.AndroScope/base.apk
72ca449000-72ca4d8000 r-xp 0002d000 /data/app/.../com.example.AndroScope/oat/arm64/base.odex
72ca4db000-72ca9af000 r--p 00000000 /data/app/.../com.example.AndroScope/oat/arm64/base.vdex
```

Figure 2: Partial memory mapping of the attacker app with DICI

simple invocations are sufficient for the adversary to load all the VDEX and ODEX files of `Organic Maps` into the memory space of the attacker process. On top of this, we also find that by invoking Java methods that would eventually call a native method located in a particular third-party native library, the specific native library will be mapped to the attacker process as well. As shown in Lines 6 - 9 of Listing 2, we implemented such an invocation using the reflection mechanism with parameters `victimClass` and `victimMethod` set, respectively, to `app.organicmaps.MwmApplication` and `nativeSetSettingsDir`. The Java method referred by this particular reflection is known to eventually raise a Java Native Interface (JNI) call of the function `ToNativeString` in library `liborganicmaps.so`. Figure 2 presents part of the memory mapping of the ANDROSCOPE attack process after running the mock attack snippet. We can see that, beyond its own executable, all three code files of the victim `Organic Maps`, namely the `base.odex`, the `base.vdex`, and the `liborganicmaps.so`, have been mapped into the address space of the attacker app.

We repeated the same experiments for 1,000 apps downloaded from AndroZoo, a growing collection of Android applications from a few sources, including the official Google Play. The results indicate that ANDROSCOPE can successfully load the VDEX and ODEX files of all the tested apps by leveraging the DICI mechanism. In short, by using the DICI mechanism, ANDROSCOPE can directly load a victim's private code. This enables it to surpass prior methods relying on shared libraries and system APIs on resolution and resilience.

3.5 Side-Channel Probing

Compared to x86, ARM poses several technical challenges for cache side-channel attacks. For starters, not all ARM processors support the CLFLUSH instruction directly. However, ARM v8 offers instructions to flush data and instruction caches, allowing ANDROSCOPE to adopt an instruction sequence as given in Listing 3 to launch FLUSH+RELOAD attack.

```
// Cleans and Invalidates data cache by
    ↪ address to Point of Coherency
asm volatile ("DC CIVAC, %0" :: "r"(address));
// Ensures completion of the cleaning and
    ↪ invalidations
asm volatile ("DSB ISH");
// Synchronize the fetched instruction stream
asm volatile ("ISB");
```

Listing 3: Flush instruction in the FLUSH+RELOAD attack

Another challenge is that many ARM CPUs do not have a data-inclusive shared last-level cache (LLC). Green et al. [18] have also shown that there are other features in the ARM CPU implementation that make an attack more difficult. For instance, ARM devices implement inclusive and non-inclusive caches alike with both properties co-existing even in the same cache hierarchy. Fortunately, most ARM CPUs are cache-coherent. When a process accesses a cache line not currently cached in its own core, the CPU will try to fetch it from other cores in case other processes are accessing it. If successful, the resulting access will be much faster than accessing from

memory. This feature has been exploited in recent work [30] and in ANDROSCOPE as well.

One additional challenge is regarding the big.LITTLE architecture [31] adopted by many ARM CPUs consist of a System-on-a-Chip (SoC) made from two discrete computing clusters, one low-power group of cores and one high-power group. It appears that *only cores among big and LITTLE have the proper cache coherency protocol that can be exploited*, which requires the attacker and victim to be *on different classes of cores*. Further, we find it interesting that flushing takes significantly more time on the big cores (likely due to their cache replacement policy). As a result, ANDROSCOPE pins the attack threads on the “LITTLE” cores, both for performance consideration and also for reasons that high-value victim apps tend to exploit “big” cores for better user experiences.

Additionally, the existing threshold calibration method lacks reliability across devices from various manufacturers and models. Therefore, we have devised an adaptive strategy to accurately calculate the runtime cache hit/miss threshold on any given device. Specifically, we dynamically adjust the hit/miss threshold during runtime by monitoring the occurrence of false positives following a FLUSH+RELOAD operation. If the number of false positives exceeds a predefined threshold, we decrease the hit/miss threshold accordingly.

Last but not the least, given that side-channel probing is, after all, not a deterministic metric, we introduce a simple voting mechanism to improve the accuracy, which is by all means to monitor multiple addresses in the same method and come to the cache hit/miss conclusion according to the majority. Due to the minimum cache line size, we avoid monitoring more than one address in the same 64-byte code snippet. Moreover, our tests show that ARM CPUs tend to trigger the pre-fetching mechanism when a continuous segment of address space is accessed more than four times in a short period, causing any subsequent side-channel accesses to contribute nothing but false positives. Therefore, we also avoid probing more than four addresses of the same targeted method.

Note that for ARM processors that do not have a flush instruction, ANDROSCOPE makes use of the EVICT+RELOAD approach [46] to carry out its cache side-channel attacks.

3.6 DICI in Real-World Apps

To understand the usage of DICI in real-world Android apps (a different experiment from that described in Section 3.4 where DICI was used by our attacker app), we utilize Soot [21] and FlowDroid [7] to construct and parse the call graph in 1,000 apps downloaded from AndroZoo, tracing how DICI is employed in the analyzed apps to gain insights into the prevalence and utilization of DICI. In summary, we find that DICI is used by 227 apps to load code in Google Mobile Services (GMS), including YouTube Kids[1], XPlayer[2], and QR & Barcode Scanner[3]. Additionally, we find that 412 apps call `createPackageContext` to obtain the context of other applications without loading the code, allowing them to access data stored in the *SharedPreferences* of other applications.

4 Evaluation

Now we present our experiences in applying ANDROSCOPE to attack two real-world applications, `Organic Maps` and `Briar`, for the purpose of inferring sensitive user behavior. The two victim applications are chosen due to them being primary targets of recent attacks, which enables us to compare with related work and reveal the unique offering of ANDROSCOPE in attack scenarios involving unauthorized, app-specific behavior inferring. Table 2 summarizes the prior work on the two applications as well as the new perspectives ANDROSCOPE offers. Although our attack scheme shares some common objectives with prior work, it offers two important unique properties. First, it does not require any specific permission, unlike prior work that leverages sensor readings. Second, it captures finer-grained behaviors of the targeted victim app in particular, unlike prior work that leverages system-level method invocations which include noise from other apps. The two apps also allow us to demonstrate our capability in monitoring all three types of victim code, including ODEX, VDEX, and native libraries.

To evaluate the extent and precision to which ANDROSCOPE can monitor app-specific behaviors across different app sandboxes, we implemented our scheme demo as a regular third-party app to carry out the mock attacks. In the following subsections, we go into details of each experiment and explain the specific target methods to be monitored, the intermediate results in detecting method executions, and the resulting overall capability in inferring user behaviors.

Recall that ANDROSCOPE monitors multiple addresses in one method and applies simple output voting to improve our accuracy of CPU cache side-channel monitoring. When attacking `Organic Maps`, our attack app leverages up to four targets within a single method at least four cache lines apart and uses an output voting threshold of three out of four; whereas for `Briar`, ANDROSCOPE uses up to three targets within a method at least two cache lines apart with an output voting threshold of two out of three. This is because that methods of `Briar` are generally shorter than those of `Organic Maps`. We further discuss such practical settings at the end of this section.

[1]MD5: ecad46e3eb7cf31430e0c5f25e9d860f
[2]MD5: fe9cc8e71e0857db0c2ef3bb0b2a983b
[3]MD5: 8628586a3984cf74673019d59d3311dd

Table 2: Comparison with Previous Work on Inferring App Behavior

App		Organic Maps	Briar
Sensitive behavior to be inferred		Route taken in driving navigation	Keystroke dynamics in typing
Information leveraged	Prior work	Gyroscope, accelerometer, and magnetometer [35]	Timing of system-level method execution [30]
	ANDROSCOPE	Timing of app-level method execution	
		Methods from ODEX and native libraries	Methods from VDEX
Unique offering by ANDROSCOPE		• No permissions needed • Detecting start/end of navigation session	• No permissions needed • Detecting start/sending of a message

4.1 Inferencing Driving Routes

In this subsection, we investigate the potential of tracking users' mobility without explicitly requesting permissions to access the phone sensors or location services. More specifically, we demonstrate that ANDROSCOPE can accurately recover the route taken by the victim device user by tracing the execution of methods in the navigation app `Organic Maps`. Our observation is that when the vehicle approaches a turn, a turning notification audio will be played, and the detection of the execution of such audio-playing methods enables our attack app to recover the timestamps of each turning, which can be used to infer the route. Here we assume that the attacker has knowledge of the travel area of the victim (e.g., known to live in Boston) via other means such as analyzing IP addresses or social networks.

Monitored methods. We monitor driving-related methods encompassing route initiation, directional transitions, moments of immobility, and culmination. The primary objective of this study is to ascertain the effectiveness of ANDROSCOPE in elucidating the execution of these procedures. The details about these methods can be found in Table 3. Notably, the first three methods are intrinsic to a third-party native library, while the last method pertains to the ODEX file. When the vehicle approaches a directional change, both `GetDirectionTextId` and `requestAudioFocus` are invoked. In cases where vehicular motion is interrupted, for instance, at traffic lights, method `OnViewportChanged` is further scrutinized to determine whether the vehicle remains in motion along its designated route or is at a standstill. Lastly, method `nativeGenerateNotifications` serves the dual purpose of signaling the commencement and culmination of a driving session.

Implementation. Besides the technical details described in Section 3.4 and 3.5, our attack app leveraged the same algorithm proposed by SENSOR [35] to process the logged cache hit timestamps for the selected methods and subsequently infer driving routes indicated by the gathered data. This design is to make our approach as comparable as possible with the previous work of the same attack purpose.

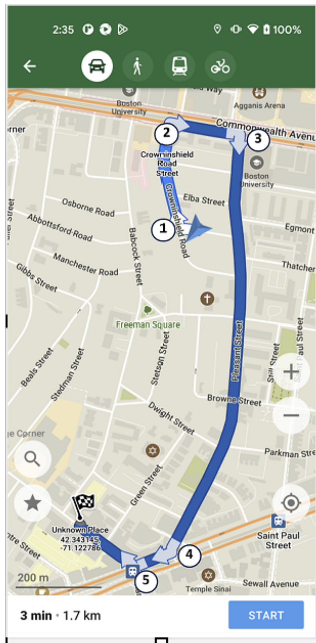
A sample route. Figure 3 illustrates an example of the tested driving route and its corresponding side-channel cache-hit timestamps for the monitored methods. The particular route had five turns (Figure 3a) which resulted in six cache hits for methods `GetDirectionTextId` and `requestAudioFocus`, out of which the first five indicated turns while the last one corresponded to an arrival notification. When the vehicle was stationary, we can see that there was no cache hit for method `OnViewportChanged`, whereas method `nativeGenerateNotifications` was always invoked. This enables us to deduct vehicle stationary time to arrive at a more accurate driving duration between two consecutive turns.

Intermediate results in detecting turnings. We randomly chose 33 routes in two cities, namely Boston and Waltham (provided by Narain et al. in their work of SENSOR), and simulated the driving using Appium [1] by replaying the GPS coordinates along with the timestamps for both two devices Listed in Table 1. Result of ANDROSCOPE in correctly identifying the turns can be found in Table 4, where we report that our attack accurately identified the turns for 30 and 28 routes (out of 33) in Boston and Waltham respectively. Among the mistaken ones, ANDROSCOPE missed one turn for two routes and falsely identified an additional "turn" for one route in Boston. For Waltham, ANDROSCOPE identified one additional "turn" for three routes and failed to identify any turns for two routes. The timing of all correctly identified turnings, on the other hand, were all detected accurately.

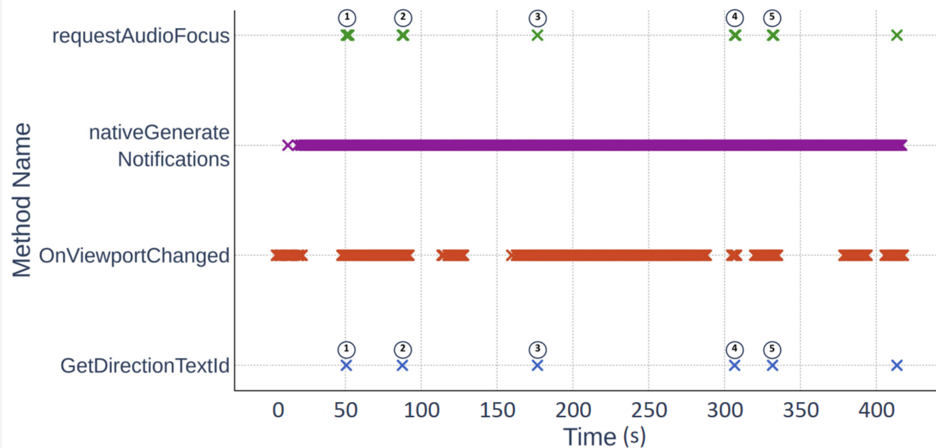
Overall results in route reconstruction. We further demonstrate the effectiveness of ANDROSCOPE by reconstructing routes instead of solely focusing on the number of recoverable turns, utilizing other information obtained by ANDROSCOPE, such as timestamps between turns. To perform a fair comparison with SENSOR, we remove the additional information SENSOR has (e.g., angles of the route curve) and construct a setting with comparable inputs for SENSOR and denote it as SENSOR_OT. The percentage of routes that can be inferred by ANDROSCOPE, SENSOR, and SENSOR_OT are shown in Table 5, which indicate that the timestamps obtained by ANDROSCOPE can be used to effectively infer the route of driving compared to SENSOR_OT. For example, 43% and 47% of routes can be inferred by ANDROSCOPE for mobile phones Xiaomi Mi 11 Lite and Pixel 4a respectively, while SENSOR_OT can only infer 37% of the routes for Boston. For routes in Waltham, ANDROSCOPE can infer 64% of them

Table 3: Method to be monitored

Method Name	Description	Type
`GetDirectionTextId`	Indicates that the vehicle is approaching a turn	`liborganicmaps.so`
`OnViewportChanged`	Indicates whether the vehicle continues moving along the route or is stationary	`liborganicmaps.so`
`nativeGenerateNotifications`	Indicates that the navigation started and ended	`liborganicmaps.so`
`requestAudioFocus`	Indicates that the vehicle is approaching a turn	`base.odex`



(a) Route example



(b) FLUSH+RELOAD side-channel result

Figure 3: Route and FLUSH+RELOAD result.

for both mobile phones while SENSOR_OT can only infer 54% of them.

Comparison with Traditional Approach. We find that when a turn is approaching, the monitored method `requestAudioFocus` of `Organic Maps` eventually sends a request to the Android service `AudioManager` by invoking `android.media.AudioManager.requestAudioFocus` located in */system/framework/arm64/boot-framework.oat*. This file contains compiled code for core framework classes and methods provided by the Android framework and is shared across user processes. This means that in order to achieve the same attack described in this section like ANDROSCOPE, existing work regarding cache-based side-channel attacks against Android devices would have to probe the underlying system (framework) API as mentioned above to implicitly infer the navigation behaviors of `Organic Maps`. However, the audio focus mechanism provided by `AudioManager` serves a wide spread of different types of apps, such as `Spotify` and `WhatsApp`, enabling them to request exclusive or temporary control over the audio output of the device. As such, the execution of the aforementioned framework API alone can mean various types of events (other than navigation), e.g., viewing short videos or hearing voice messages, causing the existing Android side-channel work to be trapped in significant noise interference.

To give an illustrative example, we built a customized app called `NoiseApp` which is designed to continuously invoke `android.media.AudioManager.requestAudioFocus` in the background upon running (simulating music playing apps like `Spotify`). Then, in the presence of `NoiseApp`, the effectiveness of ANDROSCOPE is compared with the traditional Android side-channel approach which can only resort to framework API monitoring (utilizing the same FLUSH+RELOAD approach). The result showed that during the navigation of the same route in Figure 3, the compared traditional side-channel approach picked up a total of 12,089 cache hits for the targeted framework API, out of which 12,083 were actually "noises" contributed by `NoiseApp`; on the other hand, such noises had no impact on ANDROSCOPE because it observes the method `requestAudioFocus` within the code base of `Organic Maps` and is able to confirm the de-

Table 4: Routes distribution under ANDROSCOPE. *Correctly identified* means all identified turns correspond to ground truth. *Extra one* means ANDROSCOPE mistakenly identified one additional turn while *Miss one* means ANDROSCOPE misses one turn compared to ground truth. *Failed* means ANDROSCOPE cannot identify any turns at all.

City	Xiaomi Mi 11 Lite				Pixel 4a			
	Correctly identified	Extra one	Miss one	Failed	Correct identified	Extra one	Miss one	Failed
Boston	30	1	2	0	30	1	2	0
Waltham	28	3	0	2	28	3	0	2

Table 5: Percentage of routes that can be inferred by ANDROSCOPE, SENSOR, and SENSOR_OT

City	ANDROSCOPE		SENSOR	SENSOR_OT
	Xiaomi	Pixel 4a		
Boston	43%	47%	67%	37%
Waltham	64%	64%	93%	54%

tection with the observation on another in-app private method, namely `GetDirectionTextId`.

In conclusion, there are two main aspects in which ANDROSCOPE goes beyond existing work, including (1) ANDROSCOPE demonstrates the capability of directly probing a victim's private code, surpassing previous methods relying on system APIs; and (2) ANDROSCOPE dismisses concerns about background noise as raised in the existing approach.

4.2 Inferring Keystroke Dynamics

In this subsection, we demonstrate that ANDROSCOPE can be used to launch keystroke timing attacks as well. The high-level observation is that by monitoring the callback methods of keyboard pressing using our attack, an adversary could obtain precise timing of each key press, which could be subsequently used to infer the actual words and sentences typed. Our work differs from the prior work of ARMageddon [30] in that we monitor app-level methods instead of system-level methods, which not only eliminates noise introduced by other apps when they execute the same system-level method implementations, but also enables our attack to gain more knowledge on what is being input by the user (e.g., characters or spaces). In addition, the method-level difference also makes ANDROSCOPE immune to counter-measures like KeyDrown [42] which runs through the same code path in the shared library for all, fake and real, keystrokes.

Monitored methods. As depicted in Figure 4, `Briar` transforms the "send" icon into an attachment icon in the absence of text within the chat box, which is facilitated through the invocation of the `updateViewState` method. Simultaneously, the method `onTextIsEmptyChanged` is invoked whenever a transition occurs within the chat box, either from an empty state to a non-empty one or vice versa. Consequently, we leverage cache hit events associated with these two methods to signify the commencement and dispatch of a message. Furthermore, the act of entering a character or depressing the space bar triggers the execution of `onTextChanged` method, whereas method `countLeadingWhiteSpace` is executed exclusively when the current count of leading white spaces, measured from the cursor position to the last character typed, is zero.

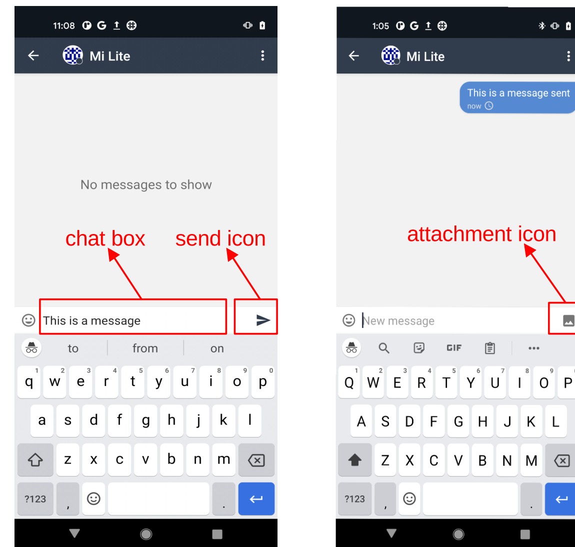


Figure 4: Example of the main page of Briar

As shown in Listing 4, `Briar` implements its own overloaded version of the `onTextChanged` function, which is called to notify that, within the parameter string *s*, the *count* characters beginning at *start* have just replaced old text that had length *before*. Specifically, if *isEmpty* is true (indicating that the text was previously empty), it checks whether the characters that were added (count) contain any leading whitespace characters (Lines 3 - 8). If there are non-whitespace characters among the added ones, it means the text is no longer empty. If *isEmpty* is false and *before* is greater than 0 (indicating that characters have been replaced), it checks from the beginning of the text (index 0) to see if the entire text consists of whitespace characters (Lines 8 - 15). Note that after a spacebar is pressed, the current Android system will set *offset* to zero. When subsequently the next letter is typed, function `countLeadingWhiteSpace` will not be executed.

Table 6: Method to be monitored in Briar

Method Name	Description	Purpose
`onTextIsEmptyChanged`	Gets called when the chat box changes from empty to non-empty or from non-empty to empty	Indicates the start of a message and the sent of a message
`updateViewState`	Updates the composite button functionality. When there is no text in the chat box, the button becomes an 'attachment' icon. When it is filled with text, the button becomes a 'send' icon	Indicates the start of a message and the sent of a message
`onTextChanged`	Gets called whenever a letter is typed, deleted, or when the space bar is pressed	Used together with method `countLeadingWhitespace` to determine the start of a new word.
`countLeadingWhitespace`	Counts the number of leading whitespaces from the current cursor position up until the last character typed	Used together with method `onTextChanged` to determine a space has been entered and indicates the start of a new word

```
1  public void onTextChanged(CharSequence s, int
       ↪ start, int before, int count) {
2     if (emptyTextAllowed || listener == null)
         ↪ return;
3     if (isEmpty) {
4        if (countLeadingWhitespace(s, start,
            ↪ count) < count) {
5           isEmpty = false;
6           listener.onTextIsEmptyChanged(false);
7        }
8     } else if (before > 0) {
9        // Characters have been removed or
            ↪ replaced - check from the start
10       int length = s.length();
11       if (countLeadingWhitespace(s, 0, length)
            ↪  == length) {
12          isEmpty = true;
13          listener.onTextIsEmptyChanged(true);
14        }
15    }
16 }
```

Listing 4: Code snippet of function onTextChanged

A sample message. For instance, consider the message "This is a message sent". Upon pressing the space bar following the word "This", the value of *offset* was set to zero. Subsequently, when the letter *i* was typed, in accordance with the established policy, the code lines in the else condition (Lines 10 - 13 of Figure 4) were not executed, thereby precluding the execution of the `countLeadingWhiteSpace` method. However, upon typing *s*, the value of *offset* became one since there was no white space immediately ahead of it, prompting the invocation of the `countLeadingWhiteSpace` method.

It is important to note that throughout this entire process, the `onTextChanged` method was consistently invoked. Consequently, these two methods were employed to ascertain the length of an individual word being typed. The method that we probed on Google Pixel 4a can be found in Table 6 and all four methods were located in the VDEX file.

The side-channel cache-hit result on these four methods is shown in Figure 5. We can see that ANDROSCOPE accurately identified the starting and sending time of a message from the cache hits obtained for methods `onTextIsEmptyChanged` and `updateViewState`. By analyzing the cache hit result for methods `onTextChanged` and `countLeadingWhitespace`, we can see that a cache hit on method `onTextChanged` and no hit on method `countLeadingWhitespace` can be used to indicate the first letter of a new word.

Results of keystroke timing detected We computed the inter-keystroke timing difference between the ground truths and ANDROSCOPE attacks with five human typing cases and 20 simulated (and replayed) typing cases, and the result is shown in Figure 6. We can see that the timing difference between the ground truth and our attack was trivial, indicating that the obtained cache-hit timestamps may be directly used as resources for keystroke timing attacks. More specifically, the average keystroke timing difference between the ground truth and our side-channel attack was only around 0.00005 seconds while the average time for inter-keystroke was around 0.25 seconds.

Comparison with Traditional Approach. Armageddon scans two addresses in the default AOSP keyboard, namely `Latin-IME.odex`, to showcase the capability of word length detection. This requires inspecting the memory mapping file (i.e., */proc/pid/maps*) of the system application `com.android.inputmethod.latin`. However, accessing files like */proc/pid/maps* which belong to another app's process requires root privilege (e.g., with a rooted Android device). The proposed approach, ANDROSCOPE, eliminates this requirement and does not need any additional permissions.

4.3 Flexibility of ANDROSCOPE

Now we further investigate the flexibility of ANDROSCOPE in probing the execution of a method under different configurations to better support the effectiveness of our approach. Specifically, we set the maximum number of addresses to be probed in a particular method to be five, and the minimum gap between two consecutive addresses be one cache line, two cache lines, and four cache lines, respectively. We tested 24

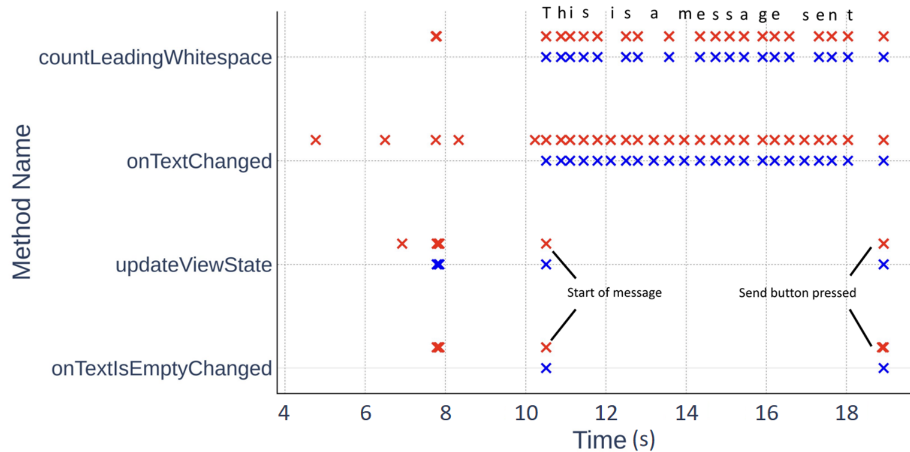


Figure 5: User input reference. The blue x is the ground truth obtained by instrumenting the app, which means the method is really executed. The red x means the cache hit.

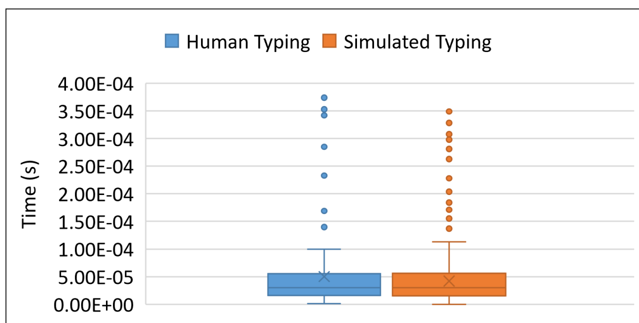


Figure 6: Keystroke time difference between side channel and ground truth

methods in the VDEX file, ODEX file, and third-party native library in apps `Dolphin` [2] and `Briar`, with the F1 score of ANDROSCOPE in identifying the execution of each method under different configurations shown in Figure 7. Here, "forward" means the monitored addresses are ordered from low to high, and ANDROSCOPE probes the lower one first; whereas "backward" means the addresses are ordered reversely and the higher address is probed first. We can see that although the F1 score differs from method to method[4], for most methods we can always find a configuration under which ANDROSCOPE can work with a satisfying accuracy. This indicates that when launching the proposed attack, an adversary is in fact quite flexible w.r.t. the detailed attack strategy, the technical arsenal, as well as the selection of target methods.

For certain methods, we observed suboptimal performance, notably in methods 4, 19, and 20. Several factors may contribute to this suboptimal performance, including but not limited to (1) The potential for cache pollution when a small method shares a cache line with a preceding large method; (2) Cases where a frequently executed method with a short duration may exceed the probing resolution capabilities of ANDROSCOPE; (3) Situations where multiple probed addresses share the same cache set, resulting in conflicts.

[4]The reasons of the relatively unstable results may include the pre-fetching mechanism, layout, and sequence of binary code in the address space, sizes of individual methods, starting/ending offset of a method from the nearest cache line boundary, and etc. We leave the more thorough investigation regarding this observation as a future work.

4.4 Performance Overhead

To evaluate the runtime overhead of ANDROSCOPE, we conducted experiments by running our attack app on a series of devices listed in Table 1, with it kept in the background for the whole period. We then used one of the most popular benchmarks, `Geekbench-6` [3], to assess the devices both with and without *appside* running. Specifically, `Geekbench` runs a series of tests on a processor and measures the time it takes for the processor to complete these tasks. The faster the CPU completes the tests, the higher the `Geekbench` score is.

We configured ANDROSCOPE to probe different numbers of addresses, namely 276, 151, and 104 addresses in Dolphin, and ran `Geekbench-6` 10 times to obtain the average score. As shown by the results presented in Figure 8, ANDROSCOPE reduced the benchmark scores respectively by 18.09%, 14.98%, and 12.78% for the Xiaomi Mi 11 Lite with the three experiment configurations. For the Pixel 4a, ANDROSCOPE decreased the benchmark score by 27.49% when probing 276 addresses and by 23.13% and 20.14% when probing 151 and 104 addresses. The overhead mainly comes from the side-channel probing. We expect such overhead to decrease when ANDROSCOPE is configured to probe its target addresses less frequently or when the attack is launched on more recent (and powerful) devices.

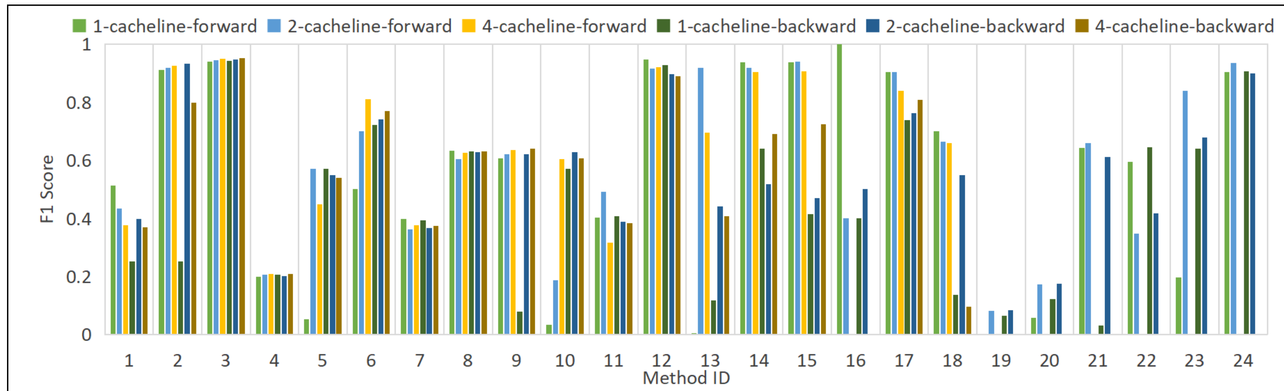


Figure 7: F1 score distribution under different configurations

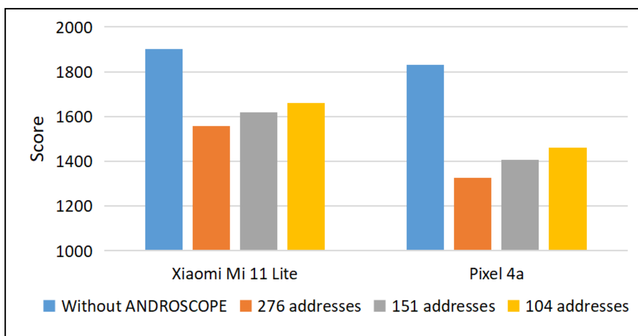


Figure 8: Performance of ANDROSCOPE evaluated with Geekbench Benchmark

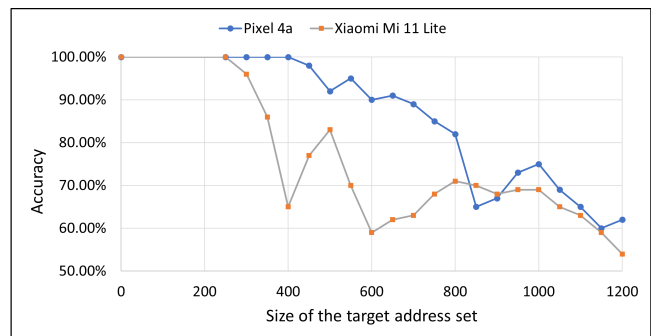


Figure 9: Throughput of ANDROSCOPE, where "0" for the size of ANDROSCOPE's target address set indicates that the subject app's button click was monitored.

4.5 Number of Concurrent Addresses Monitored

With this evaluation, we intend to demonstrate the capability of ANDROSCOPE in monitoring multiple code addresses simultaneously. Currently, our FLUSH+RELOAD component occupies one CPU core and conducts sequential polling on a given set of target addresses. Therefore, here we progressively increased the size of ANDROSCOPE's target address set (+50 a time until the total amount of targets reaches 1,200), with one particular address corresponding to a button click event of a subject app always included; then, the subject app was manually used with the being monitored button clicked for 100 times, to see how many of the clicks were identified. As can be found in Figure 9, the accuracy of our approach was maintained at 100% for Pixel 4a and Xiaomi Mi 11 Lite before the number of scanned target addresses reached 400 and 250, respectively. Beyond these thresholds, the accuracy decreased gradually until down to 62% and 54% respectively when ANDROSCOPE scans 1,200 target addresses simultaneously. It is important to highlight that the previous case studies on `Organic Maps` and `Briar` (Section 4.1and 4.2) only involve scanning 11 and five addresses, respectively. This underscores that scanning 400/250 addresses is sufficient in real attack scenarios to enable ANDROSCOPE to achieve full capability.

4.6 Countermeasures

There are several steps one could take to help mitigate the side-channel attack that we discover.

Firstly, DICI usage can be restricted in several ways. Android OS could introduce an enhanced access control mechanism by adding a declaration to the AndroidManifest file. This declaration would define which legitimate apps are allowed to use DICI or determine which specific parts of the code (such as classes) can be accessed by other applications through DICI. By doing so, Android developers and system administrators can exercise greater control over the scope and permissions of DICI usage, ensuring that sensitive data and functions are protected from unauthorized access.

We rely on reading the */proc/self/maps* file to gather vital details about the VDEX, ODEX, and third-party native libraries of victim apps that are mapped into the attacker's address space. Restricting access to this sensitive information significantly raises the complexity of the proposed attack.

We modify the Android kernel to enhance security by selectively omitting memory mapping information for files located in other apps' private directories. Specifically, we revise the `show_map_vma` routine in `task_mmu.c` to not display memory mapping information when the resolved package name does not match with the package name of the current task.

We also evaluated the runtime overhead of this countermeasure by running `Geekbench-6`. The corresponding scores with and without countermeasures are 1,840 and 1,863, respectively, demonstrating a negligible runtime overhead.

Another way to mitigate the FLUSH+RELOAD attack we proposed is to prevent physical memory sharing among apps. This can be achieved by implementing techniques like the *copy-on-access* method proposed by Zhou et al. [51]. With this approach, a state machine is employed to meticulously monitor the sharing of each physical page among distinct security domains, such as containers. Whenever a shared page is accessed by any security domain, an immediate page copy operation is triggered. This proactive measure serves as a strong defense against FLUSH+RELOAD attacks by disrupting the mechanism that attackers could exploit.

5 Related Work

5.1 Cache Side-Channel Attacks on ARM

Most of the existing work so far has focused on x86 processors. For ARM, we are aware of some papers that specifically explore the security of caches. ARMageddon [30] makes use of the flush instruction provided by ARM v8 to launch FLUSH+RELOAD attacks to probe user input. It also introduces PRIME+PROBE and EVICT+RELOAD attacks for ARM processors that do not have the flush instruction. Zhang et al. [50] give a systematic exploration of vectors for FLUSH+RELOAD attacks on ARM processors and use the FLUSH+RELOAD attacks to trace software execution. AutoLock [18] explores how the *AutoLock* feature found in some ARM processors could be used to thwart some cache timing attacks. It also shows how attackers can overcome the feature and perform EVICT+TIME and PRIME+PROBE attacks. TruSpy [49] analyzes timing cache side-channel attacks on ARM TrustZone and exploits cache contention between the normal world and the secure world to leak secret information from TrustZone protected code using PRIME+PROBE attack. Lee et al. [28] explore FLUSH+RELOAD attacks on the ARM FPGA by exploiting Accelerator Coherency Port (ACP) and iTimed [20] makes use of PRIME+PROBE and FLUSH+RELOAD to attack Apple A10 Fusion System-on-a-Chip (SOC). Wang et al. [46] make use of EVICT+RELOAD attacks to extract a user's password and PIN.

5.2 Side-Channel Attacks on Mobile Sensors

IMU sensor readings on mobile devices have been used to infer sensitive information too. Abdul Rehman et al. [24] proposed and developed an application on the Android platform that runs in the background and gathers information in accelerometer, gyroscope, and magnetometer to infer the keystrokes being typed. Some work [8, 34, 39] used the accelerometer sensor to infer password inputs. Sashank et al. [35] utilized the accelerometer, gyroscope, and magnetometer to infer the route of driving. However, these approaches are known to be very sensitive to random noise and heterogeneous human behavior patterns.

6 Discussion

6.1 Other Scenarios of ANDROSCOPE

In addition to monitoring method execution, ANDROSCOPE exhibits the capability to infer sensitive data. A notable illustration of this capability lies in the DICI mechanism, which may be employed to access data stored in the *SharedPreferences* of another application. *SharedPreferences* serves as a frequently utilized repository for storing key-value pairs, and this repository can encompass sensitive information [9, 29].

6.2 Limitations

Due to the nature of DICI, an app can add noises to ANDROSCOPE by invoking the monitored methods in the victim app. This potential for noise arises from the fact that other apps can inadvertently trigger the same methods that ANDROSCOPE is monitoring. Since we FLUSH+RELOAD multiple functions at the same time, the likelihood of these functions being called together by another app is very low. Hence, most background noise of this kind can be eliminated.

Secondly, a Java method will be compiled using the just-in-time (JIT) compiler and stored in a code cache if it is frequently executed. This optimization mechanism enhances the performance of the application by allowing frequently used methods to be readily available in compiled form. However, this code cache is specific to each individual application and is not shared between the attack and the victim apps. Consequently, ANDROSCOPE cannot be used to launch an attack against this type of method. We observe the activation of JIT compilation after methods have been executed more than 100 times. During periods of device idleness and charging, Android initiates the re-compilation of applications based on the aggregated profile generated during initial runs. At this stage, high-frequency methods are incorporated as native components within the ODEX file. Consequently, frequently executed methods may experience temporary inaccessibility when located inside the code cache due to JIT compilation. However, AndroScope regains access to such code

by patiently waiting and subsequently reloading the ODEX associated with the targeted application.

Thirdly, when an app targets Android 11 (API level 30) or higher and queries for information about other apps installed on a device, the system filters this information by default. In this case, the attacker needs to manually use the `<queries>` element in the manifest file to declare the need for package visibility, so that DICI can work. The `<queries>` element allows developers to specify which packages or components they need to access, granting their apps the necessary permissions to interact with these elements. It's a security measure introduced by Android to protect user privacy and data. While some application stores, like the Google Play Store, may remove the malicious app, attackers can employ various methods to install it, including through advertising and phishing.

7 Responsible Disclosure

We have responsibly informed Google about the risk posed by DICI regarding its potential to bypass Android app sandboxing and infer private information related to individual apps through cache-based side-channel attacks. Google informed us that they have completed their assessment on June 7, 2024, have determined that the issue is of moderate severity, and will pass it to their feature team to fix in an upcoming release. It remains unclear to us how this issue is to be fixed.

8 Conclusion

In this paper, we propose ANDROSCOPE, a side-channel attack scheme that fully circumvents the application sandboxing mechanism of Android by leveraging the DICI mechanism supported by the platform. The experimental results on a number of real-world apps showed that ANDROSCOPE can be used to successfully infer fine-grained user behavior by probing app-specific methods executed in the victim processes, with the attack app not conducting any intrusive operations, obtaining root privilege or requesting any unusual permissions.

Acknowledgements

We thank our shepherd and the anonymous reviewers for their valuable comments and suggestions that have helped improve our paper. This research / project is supported by the National Research Foundation, Singapore, and the Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Proposal ID: NCR25-DeSCEmT-SMU), as well as the National Natural Science Foundation of China (Grant Nos. 62302193 and 61932011) and the Guangdong-Hong Kong Joint Laboratory for Data Security and Privacy Preserving (Grant No. 2023B1212120007). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the National Research Foundation, Singapore, and the Cyber Security Agency of Singapore.

References

[1] Appium. https://github.com/appium/io.appium.settings/tree/master.

[2] Dolphin. https://dolphin-emu.org/.

[3] Geekbench webpage. https://www.geekbench.com/.

[4] statista. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/.

[5] vdexextractor. https://github.com/anestisb/vdexExtractor.

[6] Ahmed Al-Haiqi, Mahamod Ismail, Rosdiadee Nordin, et al. A new sensors-based covert channel on android. *The Scientific World Journal*, 2014, 2014.

[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[8] Adam J Aviv, Benjamin Sapp, Matt Blaze, and Jonathan M Smith. Practicality of accelerometer side channels on smartphones. In *Proceedings of the 28th annual computer security applications conference*, pages 41–50, 2012.

[9] Laura Bello-Jiménez, Alejandro Mazuera-Rozo, Mario Linares-Vásquez, and Gabriele Bavota. Opia: A tool for on-device testing of vulnerabilities in android applications. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 418–421. IEEE, 2019.

[10] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. *Technische Universität Darmstadt, Technical Report TR-2011-04*, 2011.

[11] Haining Chen, Ninghui Li, William Enck, Yousra Aafer, and Xiangyu Zhang. Analysis of seandroid policies: Combining mac and dac in android. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 553–565, 2017.

[12] Sen Chen, Lingling Fan, Guozhu Meng, Ting Su, Minhui Xue, Yinxing Xue, Yang Liu, and Lihua Xu. An empirical assessment of security risks of global android banking apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1310–1322, 2020.

[13] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In *Information Security: 13th International Conference, ISC 2010, Boca Raton, FL, USA, October 25-28, 2010, Revised Selected Papers 13*, pages 346–360. Springer, 2011.

[14] Android Documentation. Scoped storage. https://source.android.com/docs/core/storage/scoped.

[15] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–29, 2014.

[16] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638, 2011.

[17] Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. Borrowing your enemy's arrows: the case of code reuse in android via direct inter-app code invocation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 939–951, 2020.

[18] Marc Green, Leandro Rodrigues Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. Autolock: Why cache attacks on arm are harder than you think. In *USENIX Security Symposium*, pages 1075–1091, 2017.

[19] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive {Last-Level} caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912, 2015.

[20] Gregor Haas, Seetal Potluri, and Aydin Aysu. itimed: Cache attacks on the apple a10 fusion soc. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 80–90. IEEE, 2021.

[21] Laurie Hendren, Patrick Lam, Jennifer Lhotak, Ondrej Lhotak, and Feng Qian. Soot, a tool for analyzing and transforming java bytecode. *World Wide Web, http://www. sable. mcgill. ca/soot/tutorial/-pldi03/tutorial. ps*, 2003.

[22] Guangwu Hu, Bin Zhang, Xi Xiao, Weizhe Zhang, Long Liao, Ying Zhou, and Xia Yan. Samldroid: a static taint analysis and machine learning combined high-accuracy method for identifying android apps with location privacy leakage risks. *Entropy*, 23(11):1489, 2021.

[23] Google Inc. Privilege escalation in google android. https://source.android.com/security/bulletin/pixel/2021-01-01., 2021.

[24] Abdul Rehman Javed, Mirza Omer Beg, Muhammad Asim, Thar Baker, and Ali Hilal Al-Bayatti. Alphalogger: Detecting motion-based side-channel attack using smartphone keystrokes. *Journal of Ambient Intelligence and Humanized Computing*, pages 1–14, 2020.

[25] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. In *European Symposium on Research in Computer Security*, pages 97–110. Springer, 1998.

[26] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[27] Butler W Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[28] Heemin Lee, Sungyeong Jang, Han-Yee Kim, and Taeweon Suh. Hardware-based flush+ reload attack on armv8 system via acp. In *2021 International Conference on Information Networking (ICOIN)*, pages 32–35. IEEE, 2021.

[29] Xiaodong Lin, Ting Chen, Tong Zhu, Kun Yang, and Fengguo Wei. Automated forensic analysis of mobile applications on android devices. *Digital Investigation*, 26:S59–S66, 2018.

[30] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. {ARMageddon}: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, 2016.

[31] Arm Ltd. big.little. https://www.arm.com/why-arm/technologies/big-little.

[32] Claudio Marforio, Hubert Ritzdorf, Aurélien Francillon, and Srdjan Capkun. Analysis of the communication between colluding applications on modern smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 51–60, 2012.

[33] Yan Michalevsky, Dan Boneh, and Gabi Nakibly. Gyrophone: Recognizing speech from gyroscope signals. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 1053–1067, 2014.

[34] Emiliano Miluzzo, Alexander Varshavsky, Suhrid Balakrishnan, and Romit Roy Choudhury. Tapprints: your finger taps have fingerprints. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 323–336, 2012.

[35] Sashank Narain, Triet D Vo-Huu, Kenneth Block, and Guevara Noubir. Inferring user routes and locations using zero-permission mobile sensors. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 397–413. IEEE, 2016.

[36] Michael Neve, Jean-Pierre Seifert, and Zhenghong Wang. A refined look at bernstein's aes side-channel analysis. In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, pages 369–369, 2006.

[37] Lucky Onwuzurike and Emiliano De Cristofaro. Danger is my middle name: experimenting with ssl vulnerabilities in android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 1–6, 2015.

[38] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006.

[39] Emmanuel Owusu, Jun Han, Sauvik Das, Adrian Perrig, and Joy Zhang. Accessory: password inference using accelerometers on smartphones. In *proceedings of the twelfth workshop on mobile computing systems & applications*, pages 1–6, 2012.

[40] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. *Cryptology ePrint Archive*, 2002.

[41] Ravi Sandhu and Pierangela Samarati. Authentication, access control, and audit. *ACM Computing Surveys (CSUR)*, 28(1):241–243, 1996.

[42] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. Keydrown: Eliminating software-based keystroke timing side-channel attacks. In *Network and Distributed System Security Symposium*. Internet Society, 2018.

[43] Asaf Shabtai, Yuval Fledel, and Yuval Elovici. Securing android-powered mobile devices using selinux. *IEEE Security & Privacy*, 8(3):36–44, 2009.

[44] Laurent Simon, Wenduan Xu, and Ross Anderson. Don't interrupt me while i type: Inferring text entered through gesture typing on android keyboards. Privacy Enhancing Technologies Symposium Advisory Board, 2016.

[45] Junko Takahashi, Toshinori Fukunaga, Kazumaro Aoki, and Hitoshi Fuji. Highly accurate key extraction method for access-driven cache attacks using correlation coefficient. In *Australasian Conference on Information Security and Privacy*, pages 286–301. Springer, 2013.

[46] Daimeng Wang, Ajaya Neupane, Zhiyun Qian, Nael B Abu-Ghazaleh, Srikanth V Krishnamurthy, Edward JM Colbert, and Paul Yu. Unveiling your keystrokes: A cache-based side-channel attack on graphics libraries. In *NDSS*, 2019.

[47] Minjun Wu, Stephen McCamant, Pen-Chung Yew, and Antonia Zhai. Predator: A cache side-channel attack detector based on precise event monitoring. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 25–36. IEEE, 2022.

[48] Yuval Yarom and Katrina Falkner. {FLUSH+RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. In *23rd USENIX security symposium (USENIX security 14)*, pages 719–732, 2014.

[49] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou. Truspy: Cache side-channel information leakage from the secure world on arm devices. *Cryptology ePrint Archive*, 2016.

[50] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. Return-oriented flush-reload side channels on arm and their implications for android devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 858–870, 2016.

[51] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 871–882, 2016.