CASE: Comprehensive Application Security Enforcement on COTS Mobile Devices

Suwen Zhu
Stony Brook University
suwzhu@cs.stonybrook.edu

Long Lu
Stony Brook University
long@cs.stonybrook.edu

Kapil Singh
IBM Research
kapil@us.ibm.com

ABSTRACT

Without violating existing app security enforcement, malicious modules inside apps, such as a library or an external class, can steal private data and abuse sensitive capabilities meant for other modules inside the same apps. These so-called "module-level attacks" are quickly emerging, fueled by the pervasive use of third-party code in apps and the lack of module-level security enforcement on mobile platforms.

To systematically thwart the threats, we build CASE, an automatic app patching tool used by app developers to enable module-level security in their apps built for COTS Android devices. During runtime, patched apps enforce developer-supplied security policies that regulate interactions among modules at the granularity of a Java class. Requiring no changes or special support from the Android OS, the enforcement is complete in covering inter-module crossings in apps and is robust against malicious Java and native app modules. We evaluate CASE with 420 popular apps and a set of Android's unit tests. The results show that CASE is fully compatible with the tested apps and incurs an average performance overhead of 4.9%.

1. INTRODUCTION

Mobile operating systems, including Android and iOS, enforce application-level sandbox, which assigns each app a separate security identity and maintains proper isolation among apps. While effectively mitigating attacks among apps, the existing sandbox becomes useless in face of the emerging *in-app threats*, whereby a module of an app, such as a third-party library or an external class, adversely affects the rest of the app or manipulates the underlying OS. For instance, a recent study [21] found that, in both Android and iOS apps, some popular advertisement libraries turned rogue and have been stealing user private information. Another report [1] revealed that more than 18,000 Android apps covertly upload SMS messages from mobile devices to remote servers, unbeknown to users and even app developers, caused by a widely used in-app payment library. In-app threats also lead to the formation of mobile botnets and use native code for hiding the malicious activities [12].

We refer to this type of attacks as "*module-level attacks*", where a module can be as small as a class inside an app. Such attacks are hardly detectable because they do not violate the existing security enforcement mechanisms on mobile platforms, which operate at the application level and cannot distinguish app modules. Module-level attacks are also difficult to prevent. App developers often have to blindly trust external code and libraries included in their apps due to the incapability of imposing separate security restrictions on them. These attacks have been quickly emerging and evolving as the integration of external SDKs and libraries becomes nearly universal in mobile app development.

Previous research studied certain instances of module-level attacks, but failed to cover the entire class of attacks. Some of the proposed solutions isolate a particular type of malicious module, such as advertisement or JNI libraries [18, 16, 20], in ways inapplicable to other types of malicious modules. Other solutions introduce component-awareness to the Android permission checks [23] but cannot control interactions among components. Besides, the proposed mitigations only apply to coarse-grained app modules, such as a compilation unit. They also require changes to Android OS, which impedes wide adoption in the largely fragmented Android market.

To fully prevent module-level attacks, we propose CASE (Comprehensive Application Security Enforcement), an automatic app patching tool for app developers to enable module-level security in their apps built for COTS Android devices (*i.e.,* no OS or middleware changes are needed). CASE's enforcement recognizes app modules at a very fine granularity: *a class for Java code or a JNI library for native code*. In patched apps, each app module can have its own security identity and developer-defined security policies are enforced on a per-module basis, for example, "module A cannot access or reflect on any field or method of module B" or "module A cannot use permission P assigned to the app", where A can be defined in an untrusted library, B can be a sensitive class, and P can be solely intended for a special module's use. To the best of our knowledge, this fine-grained module-level enforcement is the first of its kind. It is essential to stop the emerging module-level attacks.

Realizing this enforcement requires mediation of all the possible channels via which a module may interact with another module or the underlying system. Realizing this mediation in a complete, robust, and efficient fashion is quite challenging given the design requirement that, the mediation must operate purely in user space to avoid modifying Android framework or OS. This requirement not only rules out OS-level support but also attracts potential manipulations and evasions by malicious code in apps, such as a rogue library. To overcome these challenges, we first design a novel module-level mediation scheme for Android apps, namely *dual-layer interception*, which checks all cross-module interactions unexpected to developers. We then propose two self-protection mech-

MobiSys'16, June 25 - 30, 2016, Singapore, Singapore

ISBN 978-1-4503-4269-8/16/06...$15.00

DOI: http://dx.doi.org/10.1145/2906388.2906413

anisms for the user-space mediation, namely *native-safe pages* and *concealed handler*, which together prevent the most powerful attack permitted in our threat model—arbitrary native code trying to bypass the mediation.

We build CASE as an automatic app-patching tool that developers can simply drop in their toolchain. CASE takes a compiled app as input, along with a developer-supplied policy describing (dis)allowed inter-module crossings. It then inserts the core mediation library into the app and rewrites the app's boot routine so that during runtime the mediation and its self-protection mechanisms are properly initialized prior to app execution. It finally signs the patched app with a developer key and produces a releasable package. The whole process happens without any developer assistance, such as code annotation or design refactoring, which usually limits adoption in practice and is error-prone. Since CASE operates directly on compiled apps without requiring changes to app design, not only app developers but also IT administrators and security-savvy app users can employ CASE to protect apps from module-level attacks (discussed in § 5).

We evaluate CASE in terms of its deployment cost and compatibility, security and robustness, and runtime overhead, using 420 popular apps and a set of unit tests. CASE has successfully prevented all the disallowed inter-module crossings in these tests. On average, CASE delays app launch by 0.174 seconds, increases memory usage by 1.19MB, and slows down app execution by 4.93%. The results indicate that CASE is complete, robust, and efficient for defending module-level attacks in real-world.

In summary, this work makes the following contributions:

- A study of the emerging module-level attacks, identifying the demands and challenges for defenses;
- Three novel techniques to achieve complete, robust, and efficient mediation of inter-module crossings in user-space for Android apps, which defends against the module-level attacks in their most powerful forms;
- A drop-in tool that app developers can easily adopt in their toolchain to enable module-level security enforcement in new or existing apps.

The rest of the paper is organized as follows: we discuss the module-level attacks and explain the defense challenges in §2; we explain the design of CASE in §3, centering around the three core techniques; in §4 we share our implementation experience and insights; we report the analysis and evaluation of CASE in §6; we contrast the related works in §7 and conclude the paper in §8.

2. BACKGROUND

2.1 Example Attacks

We use CamX, a hypothetical camera app, as an example to illustrate module-level attacks and the lack of defense. CamX, in addition to capturing photos, provides three value-adding features: image filters, cloud backup, and social network integration. Instead of implementing these features by herself, CamX's developer imports a 3rd-party Java class that encapsulates the image filters, along with a JNI library, and uses the SDKs provided by the cloud and social-network vendors (*e.g.,* Dropbox and Facebook).

Module-level Attacks: As increasingly observed in reality [21, 1, 12], external modules included in apps can be tainted or malicious. Assuming the image filter class in CamX is harmful, it can freely access other modules as well as the system resources meant for these modules. For example, manipulating the integrated SDKs, the harmful class can steal users' files in the cloud storage or manipulate their social network accounts. It can also abuse the camera or Internet access granted to the app. Leveraging its JNI library, a malicious module can theoretically access any user-space data or code loaded in the app's memory space.

In general, module-level attacks can happen in two forms: module-to-module and module-to-system. The first form involves a module stealing data or abusing code of another module. For instance, the malicious image filter class in CamX can use Java reflection to stealthily invoke the file download APIs in the cloud SDK or read the objects that record user profiles inside the social network SDK. The malicious class may also invoke the Android framework APIs for accessing the Internet or the camera. The second form involves a module directly accessing system-managed resources, including virtual memory, file systems, and the app runtime. For instance, a malicious JNI library can scan memory and files for sensitive data. It also can interfere with the execution of the app's Java code by manipulating the virtual machine.

Although gradually gaining attention among app developers and users, module-level attacks remain largely unstoppable because the existing security mechanisms on mobile platforms treat apps as the minimum security entities and cannot regulate or recognize app modules.

2.2 Defense: Requirements and Challenges

Defending against the emerging module-level attacks demands a new layer of security enforcement, on top of the conventional per-app enforcement, that acts on individual modules inside apps. To support modules of various sizes and types, the new enforcement has to allow modules to be defined at a fine granularity, such as a Java class. It must intercept and mediate all instances of the module-to-module and module-to-system interactions in an app that may lead to module-level attacks. The enforcement should not require changes to mobile OS or middleware, which are known to severely slow down and limit real-world adoption, needless to say enlarging the already bloated system software layer on mobile platforms. While operating in the user space, the enforcement needs to protect itself against malicious modules (either Java or native code) running inside same apps.

To meet the above requirements, a potential module-level security enforcement must overcome the following challenges:

($C1$) Intercept cross-module interactions completely and practically: There are two typical approaches to security interception: dynamically monitoring applications at a privileged software layer (*e.g.,* inside the language runtime or OS), or statically instrumenting applications' code. However, neither meets the aforementioned requirements. The first approach enjoys a full view and control of application execution, but it requires changes to system software. The second approach, though only modifying applications, cannot provide complete interception, particularly in cases where malicious code use obfuscation techniques to confuse and evade static instrumentation. Therefore, a new interception mechanism is needed for capturing all cross-module interactions, such as a reflective field read or an API invocation. However, the language runtime or OS does not explicitly track these interaction, and worse, malicious modules may obscure them.

($C2$) Prevent implicit module crossing: Though not ubiquitous used, native code (in the form of JNI libraries) is commonly found in popular Android apps [20]. Such code can directly access the virtual memory of the containing app, including the regions where the language runtime (*e.g.,* Dalvik VM) saves its internal data, such as

the class definitions and the raw representations of objects. Therefore, malicious native code can stealthily manipulate other app modules via direct memory access. For instance in CamX, to access the Java object storing the session key in the social network SDK, the malicious JNI code can locate the raw object in memory and then retrieve its sensitive data fields. Such cross-module access is implicit in that it happens at the virtual memory level without involving the Dalvik VM. Although implicit module crossing in itself represents a security violation (*i.e.,* abusing VM's internal data), preventing it is challenging as it requires protection over VM's memory regions against malicious code running in the same process at the same privilege level as the VM.

($C3$) **Protect interception mechanism in user space:** A malicious module may attempt to subvert the user-space interception mechanisms in the following ways. First, it may tamper with the "hooks" that the interception mechanism places in memory, such as those in the Global Offset Tables (GOTs) and the function preambles. The hooks redirect code executions to security checks before allowing them to enter any sensitive routines (*e.g.,* accessing a protected module). If malicious code reverts or removes the hooks, it bypasses the security checks entirely. Second, a malicious module may locate the sensitive routines in memory and directly invoke them, evading any hooks and security checks. Third, even if the hooks are protected and the sensitive routines are hidden, a malicious module may direct code executions to the locations immediately following the successful checks, as if the executions underwent the hooks and passed the security checks. Therefore, a robust interception mechanism needs to protect its own data and code (*e.g.,* hooks and security checks) from app code sharing the same memory space.

3. ENABLING MODULE-LEVEL SECURITY

3.1 CASE Overview

To stop the emerging module-level attacks, we build CASE, which enables security regulations on individual modules inside Android apps. CASE is intended primarily for app developers' use. It enforces developer-supplied security policies that describe (dis)allowed cross-module interactions. CASE identifies app modules at the granularity of a class (Java code) or a JNI library (native code). CASE mediates both module-to-module (*e.g.,* reading a member field of a different class) and module-to-system (*e.g.,* making a direct system call) access during an app execution. Rather than modifying the Android Framework or OS, CASE retrofits the module-level security enforcement into apps via automatic app patching. This design has two main advantages: first, it makes CASE compatible with all COTS Android devices and thus is *readily deployable*; and second, it minimizes developers' assistance and thus is *easily adoptable* (*i.e.,* developers only need to provide desired policies without having to modify app code or understand the enforcement mechanism).

Threat Model: We adopt a threat model that is both realistic and permissive in the context of defending module-level attacks. We trust the OS as it is locked via hardware features on COTS mobile devices. Although malicious apps may in theory comprise the OS by exploiting rare vulnerabilities, such apps are out of scope for this work because module-level attacks become pointless in these apps. On the other hand, we assume the existence of the most powerful app-level adversaries. The adversaries can gain full control over one or more modules in an app and aim at: (*i*) penetrating into a sensitive module in the app; (*ii*) abusing the privileges and

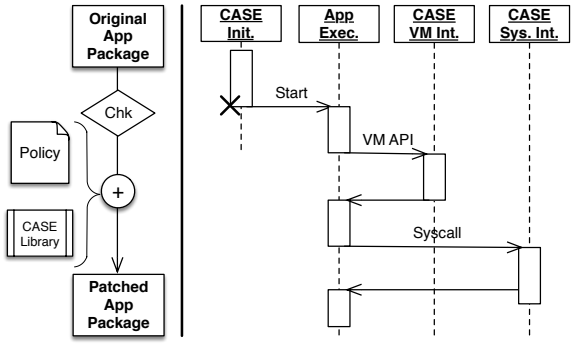

Figure 1: CASE workflow: static patching phase (left) and dynamic enforcement phase (right)

resources granted to the entire app. Tainted or malicious modules can contain arbitrary Java and native code, which is loaded inside the same app VM and process as the targeted modules as well as CASE.

Workflow: CASE consists of two main components, an automatic app patcher and a runtime enforcement library, which function in the app building phase and app execution phase, respectively (Figure 1). In the first phase, the CASE patcher, streamlined in the app building toolchain, takes as input the compiled app executable and the developer-supplied policies. The patcher checks the native library, if any, in the executable for disallowed instructions (explained in § 3.4). It then injects the policies and the CASE enforcement library into the app package. Subsequently, the patcher rewrites the app boot routine so that the self-initialization of the enforcement library is always performed prior to the original startup sequences of the app. The patcher finally signs the app using the developer's key and produces a releasable app package.

The second phase starts when the patched app is launched and CASE's self-initialization is triggered. The initialization sets up the interception mechanism for cross-module interactions (§ 3.3). Additionally, it enables the protection measures for the interception mechanism to prevent evasions and tampering by malicious modules in the app (§ 3.4 and 3.5). During the app execution, the enforcement library monitors every cross-module interaction that warrants a check as described by the policies. It denies any forbidden module crossings and alarms the app user of the event.

3.2 Overall Design and Rationale

CASE addresses the open problem of "*completely and robustly mediating cross-module interactions*". We achieve *completeness* using a mechanism called the *dual-layer interception*. It was driven by our observation that, module crossings can happen at two orthogonal layers: (*i*) the VM layer where the module-to-module interactions happen; (*ii*) the system call layer where the module-to-system interactions happen. Module-to-module interactions trigger the VM internal functions in charge of class resolution or JNI transition; similarly, module-to-system interactions trigger system calls. By interposing a set of selected VM internal functions and system call interfaces, the dual-layer interception monitors all module crossings encountered during an app execution. Without requiring any support from the VM or the OS, the CASE enforcement library realizes the function interposition via in-memory code patching and GOT hooking. The details about the set of interposed functions and the interposition techniques are explained in § 3.3.

To ensure the *robustness* of the interception against malicious modules, we introduce two protection measures, namely the *native-safe pages* and the *concealed handler*. The former prevents evasion

of the interception and the latter protects the interception mechanism against tampering.

To evade dual-layer interception, a sophisticated malicious module may perform implicit module crossing by sidestepping the VM or the regular system call interface, and therefore, avoids the interposed functions. Specifically, malicious native code can directly locate and parse the raw presentation of a Java object in memory, and in turn, manipulate the member fields or methods in a VM-agnostic fashion. The malicious module can also make implicit system calls by locating and jumping to the system call entry points without explicitly invoking the interposed system call wrappers. We design the native-safe pages to prevent implicit module crossing. The key rationale behind the design is that implicit module crossing cannot succeed without direct memory access to VM's internal data or system call entry points. The native-safe pages are regular memory pages with special page-level protection attributes dynamically set by the CASE enforcement library. Allocating VM's internal data in the native-safe pages prevents direct access by app code. Keeping system call wrappers and entry points in the native-safe pages hides them from direct execution.

In addition to evasion, tampering is the other way to bypass the interception. Since operating in user space, a malicious module may directly abuse CASE's internal data and code. For instance, it can try to overwrite memory data critical to the interception, including the policy rules and internal states. It may also try to manipulate CASE's interception routines, such as bypassing the security checks. To prevent potential subverting of CASE, we introduce two security properties, namely *interception integrity* and *interception invisibility*, which apply to CASE's internal data and code, respectively. The intuition is that no tampering is possible if malicious code cannot modify the internal data or observe (find in memory) the internal code.

To achieve interception integrity, the CASE enforcement library write-protects the memory pages for its static data and function hooks (both GOTs and patched code) during the self-initialization phase. Malicious code cannot override this page protection thanks to the system call interception. To achieve interception invisibility, CASE uses a novel invocation mechanism for stealthily invoking its interception routines. The mechanism, called the *concealed handler*, repurposes the standard POSIX signal handling to hide sensitive code in user space memory while allowing other code to invoke the hidden code without knowing its address. Using the concealed handler, CASE hides its interception routines and other internal code that app code must not directly call or partially execute.

Next, we delve into the detailed design and usage of CASE's three core techniques: the dual-layer interception, the native-safe pages, and the concealed handler. We explain how they tackle the major challenges associated with enforcing module-level security in user space (§ 2.2).

3.3 Dual-layer interception

A key insight that inspired the dual-layer interception is that, despite their large variety, inter-module crossings in apps take place at either of the two software boundaries: the boundary between two modules (*e.g.,* when accessing data or code in a object of a different class) and the boundary between a module and the underlying OS (*e.g.,* when accessing IPC, network, or hardware sensors), as illustrated in Figure 2. To capture all inter-module crossing that warrant security checks, the dual-layer interception covers a set of selected functions located at either the language runtime layer or the system call layer. We refer to the set of functions as $MinSet$ (Table 1). Derived from our manual examination of all Dalvik internal APIs

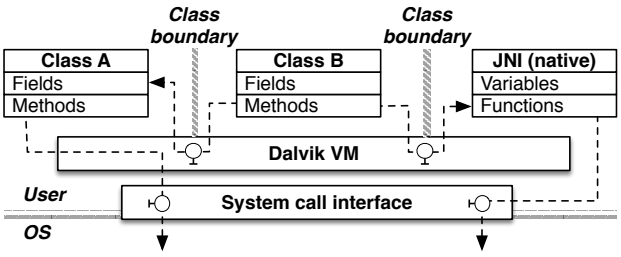

Figure 2: The dual-layer interception

for class and JNI management, $MinSet$ represents the minimum set of functions that either module-to-module or module-to-system crossings have to trigger.

Table 1: Functions in $MinSet$

Function	Library	Purpose
ioctl	libc	IPC interception
open	libc	file system isolation
link	libc	file system isolation
unlink	libc	file system isolation
connect	libc	socket access
dlopen	linker	self-protection
dlclose	linker	self-protection
dlsym	linker	self-protection
mmap	libc	self-protection
mremap	libc	self-protection
munmap	libc	self-protection
mprotect	libc	self-protection
fork	libc	process management
execve	libc	process management
dvmPlatformInvoke	libdvm	JNI interception
*Method_invokeNative	libdvm	reflection interception
*Constructor_constructNative	libdvm	reflection interception
*Field_getField	libdvm	reflection interception
*Field_setField	libdvm	reflection interception
*Field_getPrimitiveField	libdvm	reflection interception
*Field_setPrimitiveField	libdvm	reflection interception
dvmLookupClass	libdvm	class managing
loadClassFromDex0	libdvm	class managing
dvmFindPrimitiveClass	libdvm	class managing
findMethodInListByDescriptor	libdvm	class managing
findMethodInListByProto	libdvm	class managing
dvmFindVirtualMethodByName	libdvm	class managing
dvmFindStaticField	libdvm	class managing
dvmFindInstanceField	libdvm	class managing
sigaction	libc	self-protection

* denoting the common prefix "Dalvik_java_lang_reflect_".

At the system call layer, $MinSet$ contains the system call wrappers that apps use for accessing system-managed resources, including `open` for file systems and sensors, `ioctl` for driver-exposed interfaces and Binder IPC, `socket` and `connect` for sockets and network. Intercepting these system calls allows for full mediation of a module's access to system-managed resources. In addition, $MinSet$ also contains a few system calls that are not directly related to enforcing security policies but critical for preventing attacks against CASE. These include `mmap` and `mprotect` for memory management and protection (*e.g.,* write-protecting trampoline code, GOTs, and policy rules), and `sigaction` for exclusively handling the sensitive signals as the concealed handler requires (§3.5). Unlike the previous works that intercept system calls enforcing security policies [25], our interception is immune from malicious native code running in the same process. Besides, it also

mediates module-to-module interactions, which no previous work has achieved.

At the language runtime layer, $MinSet$ contains selected APIs from two Android runtime libraries, namely the Dalvik VM (`libdvm`) and the dynamic linker and loader (`linker`). The selected internal APIs in `libdvm` are used by the VM whenever an object's member fields and methods are being accessed or when a JNI library is being invoked. Intercepting these APIs allows for security checks on all kinds of module-to-module interactions inside apps. On the linker's side, $MinSet$ contains APIs for dynamically loading executables and looking up symbols in loaded executables, such as `dlopen` and `dlsym`. Intercepting the linker APIs is necessary for covering newly loaded modules and securing loaded modules, including the CASE enforcement library.

CASE additionally employs standard function interception methods, namely GOT hooking and dynamic binary patching. The former is used whenever applicable whereas the latter is needed when an interception target is a private API or does not have an exported symbol (*i.e.,* the target does not have any GOT entry). Although these interception methods have been widely used, they are intrinsically vulnerable in adversarial environments as the one CASE faces (*e.g.,* sharing a memory space with untrusted or malicious native code). Our key contribution to the interception methods lies in the self-protection techniques that make them robust against powerful adversaries (*e.g.,* malicious code running inside the same process).

The dual-layer interception overcomes the challenge $C1$, acting as the underpinnings to enable CASE's complete module-level security mediation.

3.4 Native-safe pages

A malicious module may bypass the dual-layer interception via implicit module crossing, whereby it interacts with the OS or another module without using or triggering the functions in $MinSet$. For instance, a malicious module can directly initiate a system call by jumping to the system-entering instructions (*e.g.,*`SYSENTER` and `SWI`) inside the system call wrappers in `libc`. It may also directly read and parse the VM's internal data in memory without using any VM APIs, and in turn, inspect another module. In general, implicit module crossing is possible because native code in apps can execute arbitrary instructions and has unchecked access to an app's entire user-space memory. Benign apps never need to carry out implicit module crossing.

To prevent implicit module crossing, we introduce the native-safe pages. By converting the memory pages where the system call wrappers and the VM internal data reside into the native-safe pages, the CASE enforcement library ensures that apps' native code cannot directly execute or access the system call wrappers and the VM internal data. The native-safe pages are built on the basic page-level memory protection without affecting app's normal functioning. They switch to enforcement mode only for the duration of JNI execution and remain in permissive mode (*i.e.,* no performance overhead) otherwise. CASE supports two types of native-safe pages (Figure 3):

Native-safe pages for system call wrappers: These pages store and protect the default system call wrappers. Since they are frequently accessed, keeping them locked and only unlock them for each permitted call can cause significant app delays, even if the enforcement is not always-on (*i.e.,* only in effect during app native code execution). Instead of repeatedly locking and unlocking these pages, our design only performs a single page protection adjustment at each switch between the permissive mode and the enforcement mode. In permissive mode, these pages are unlocked and freely accessible. When entering the enforcement mode (*i.e.,* switching to JNI execution), CASE locks these pages (*i.e.,* set the `PROT_NONE` bit) and duplicates these pages at a hidden and random location in memory[1], as shown in Figure 3. Every subsequent system call by the app triggers a page fault, which is transparently handled by the concealed handler (§ 3.5), where the call undergoes the security checks and, if passed, is forwarded to the hidden system wrappers. This design takes advantage of the self-contained and location-independent nature of the system call wrappers (*i.e.,* they do not reference external code or data via relative offsets, and therefore, can function normally when relocated to different memory locations.)

Obviously, the security of this optimization hinges on the confidentiality of the duplicated wrappers' locations in memory, which is assured as follows. First, as part of the interception invisibility enabled by the concealed handler (§ 3.5), the CASE enforcement library's internal code and data, including the whereabouts of the duplicated system call wrappers, are kept secure and secret from app code. Second, the hidden pages are surrounded by guard pages that prevent brute-force memory searches by malicious native code. Without requiring memory locks, the hidden pages remain unreachable for all but the checked invocations of system call wrappers. This design achieves efficient mediation and protection of the system call interface.

Using duplicated, hidden system call wrappers also solves two additional issues, namely cyclic page lock and thread unsafety, that we would run into if we simply lock the wrappers. First, without duplicated wrappers, no user-space code, including the CASE enforcement library, can make system calls once the native-safe pages for the system call wrappers have been locked. As a result, CASE cannot call `mprotect` to unlock any native-safe pages when needed, leaving the app execution in a cyclic lock. Second, without duplicated wrappers, CASE needs to unlock the native-safe pages for each checked system call, which exposes the pages and the system call interfaces to all concurrent threads of the app that may execute malicious native code.

Native-safe pages for VM internal data: These pages store and protect the internal data of the VM, including the class hierarchies and the raw object pools (Figure 3). They ensure that the resident data are only accessible to the intended code, namely the VM's internal APIs. Similar to their counterparts for system call wrappers, the native-safe pages for VM data are only effective when JNI code is running and use the basic page-level memory protection but in a different fashion. These pages are locked prior to any JNI code execution, which should never directly access data on these pages. They are unlocked when the app execution switches from JNI back to Java code.

However, this intuitive form of protection can break normal app executions when the VM happens to access its data during a page lockdown. We observe that, the VM may access a locked data page only when a Java thread is running concurrently with the thread that triggered the page lock and is running JNI code. CASE avoids this adverse impact on the VM by interleaving the native and Java threads. The interleaved executions start when a VM thread encounters a page fault due to the locked native-safe pages. CASE immediately suspends the VM thread and let the active native thread(s) continue. After the native threads return or 100 microseconds (*i.e.,* tuned to Linux's `DEF_TIMESLICE`) expire, whichever comes first, CASE suspends all native threads, unlocks the native-safe pages, and resumes the previously suspended VM thread. It repeats the preemption and switches back and forth between Java

[1]It duplicates the virtual memory mappings rather than the content.

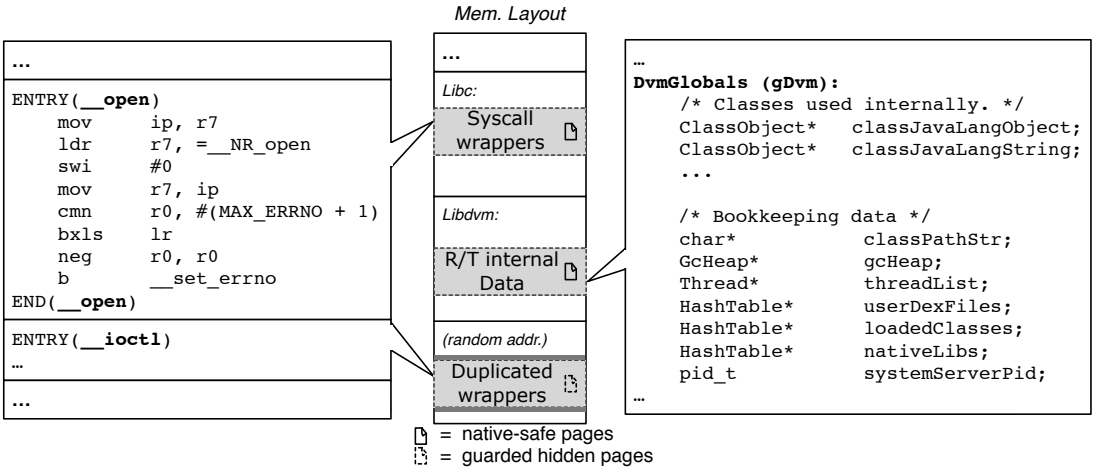

Figure 3: The native-safe pages for system call wrappers and the internal data of the runtime

and native threads until the JNI execution finishes or the app enters the single-thread execution. CASE carries out the thread interleaving and scheduling purely in the user-space by using the `SIGSTOP` and `SIGCONT` signals. It identifies the correct instances to raise these signals by monitoring an app's thread creation and JNI invocation, a capability enabled by the dual-layer interception (*i.e.,* it covers the related system calls and the JNI bridge APIs). Note that the interleaved execution rarely happens in practice as normal apps' JNI code are usually single-threaded and invoked synchronously by Java code. We tested 20 popular apps from Google Play that use JNI and found none of them triggering the interleaved execution.

With the aforementioned properties, the native-safe pages stop both forms of implicit module crossing in apps and overcome challenge $C2$.

3.5 Concealed handler

The two security properties, interception integrity and invisibility, guarantee CASE's robustness against tampering. Interception integrity is straightforward and achieved by write-protecting the memory pages for CASE's static data and code. In comparison, interception invisibility is challenging because it requires the interception routines to be hidden and protected from app code, despite the shared memory address space. CASE achieves interception invisibility via the concealed handler.

Using the standard POSIX signals, the concealed handler enables blind invocation and orderly execution of its (sensitive) code—untrusted code running in the same process can invoke the concealed handler without knowing its address in memory or influencing its execution. Since the concealed handler's code or data are undiscoverable in memory, malicious code cannot invoke it partially (*e.g.,* jumping to the middle of the function) or out-of-order (*e.g.,* ROP-style execution), nor access its associated data. In general, the concealed handler can be used by many user-space security enforcement or monitoring systems to create invisible yet invokable sensitive routines.

To create a concealed handler that encapsulates the sensitive routines, the CASE enforcement library first allocates two memory pages at random locations (one for code and one for data) during its initialization phase. It then loads a master dispatch function, the sensitive routines, and their associated data to these pages. Next, the dispatch function is registered as the handler for `SIGSEGV` (*i.e.,* for page faults). In theory, any standard signal suffices, but having the master function handle the page fault signal allows for tight integration with the native-safe pages. No pointers to these pages are saved in use-space memory as they may leak the hidden locations. Only the OS knows the address to the master dispatcher as the registered signal handler. The system call interception prevents app code from reading or changing the handler.

CASE's concealed handler contains the sensitive routines essential to its enforcement mechanism, including the resolver for the hidden system call wrappers and the native thread orchestrator, as discussed below.

System call wrapper resolver: Resolving the addresses of hidden system call wrappers takes place in a transparent fashion: the caller is neither aware of the indirection (*i.e.,* the system call is serviced by a duplicated wrapper) nor able to extract the address of the hidden system call wrapper. As shown in the example trampoline in Figure 4, when the native-safe pages are in permissive mode, the load instruction loads the address of the original wrapper into the `r7` register (Line m). The register is then used as the control flow transfer target (Line j), which happens after a successful security check. The system call interposition finishes as if the native-safe pages do not exist. In contrast, when the native-safe pages are in the enforcement mode, the load instruction on Line n causes a page fault because the native-safe pages for the original system call wrappers are now locked. The page fault instantly triggers the concealed handler. Next, the wrapper resolver in the concealed handler is activated, which looks up the address of the duplicated wrapper from its secret table stored in the hidden data page. It then loads the secret address of the hidden wrapper into `r7` and finally returns the control back to the trampoline. The trampoline continues from the fault instruction (Line n), and if the security check succeeds, jumps to the address saved in `r7` (Line j), which now points to the hidden wrapper. It is worth noting that, the caller as well as the trampoline are not aware of the system call redirection and wrapper resolution process. The design of the trampoline allows for transparent system call redirection and resolution, which happen only when the native-safe pages are locked. The secret value cannot leak because it is saved only in `r7` between Line m and j, where no return or indirect jump exists that may be exploited.

Native thread orchestrator: This routine is triggered when a page fault occurs on the native-safe pages where the VM internal data are stored. The orchestrator first checks if the page fault is caused by apps' native code, and if so, terminates the thread and raises an alert (*i.e.,* app's code should never directly access the VM internal data). If the fault occurs on a VM thread, the orchestrator performs

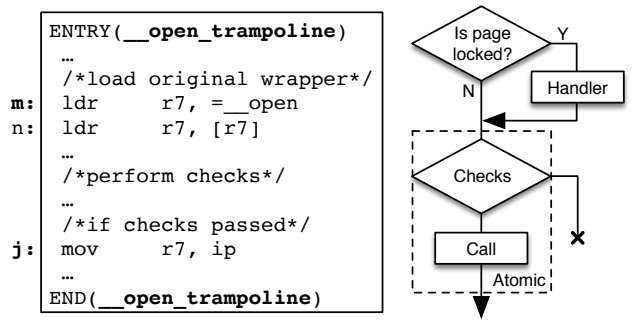

Figure 4: An example system call trampoline, showing the transparent and blind invocation of the wrapper resolution routine (not shown) in the concealed handler.

the thread interleaving as described in § 3.4. The reason for placing the orchestrator in the concealed handler is two-fold. First, the orchestrator needs to perform the sensitive operations (*e.g.,* unlocking the native-safe pages), which are disallowed by the dual-layer interception if performed outside of the concealed handler. Second, the sensitive code in the orchestrator must be protected from abuse by malicious code.

The concealed handler enables the interception invisibility for CASE and, together with the interception integrity measures, solves challenge $C3$.

4. IMPLEMENTATION HIGHLIGHTS

Due to the space constraints, we only highlight the major implementation details that are necessary for others to reproduce and expand our system.

CASE **injection & initialization:** CASE is mostly written in C with a few helper classes in Java (*e.g.,* the call stack dumper for Java code). It is compiled into a JNI library along with a few Dalvik class files. Our automatic app patcher employs the widely used Apktool to inject CASE into an app. The patcher first adds (or rewrites) a subclass of the `android.app.Application` in the app. The subclass is automatically invoked by the OS to initialize the app itself as well as additional VM instances, if any, created during the app execution (*e.g.,* an `isolatedProcess` or a remote service). The `Application` subclass initializes CASE by setting up the dual-layer interception, the native-safe pages, and the concealed handler prior to app code execution. This way, we can ensure that CASE is always invoked at app entry point and protect CASE from attacks at the initialization phase. If the app contains native code, the patcher then verifies that the code does not use system call instructions. The verification runs conservatively on ambiguous instruction sets (*e.g.,*`x86`) in favor of soundness: any legitimate instruction sequence that starts with a system call instruction and proceeds to an indirect control transfer is deemed as dangerous and triggers an alert. The patching process is fairly straightforward and does not modify the original app code, and therefore, does not cause compatibility issues or broken apps. The patched app can run on all the Android versions that the app originally targets.

Since app developers are our main target as the users of CASE, we allow app developers to provide their own key to sign the app after the rewriting process. As app developers are expected to have a good understanding of their own app, as well as the expected functionality of the untrusted or potentially vulnerable modules, they are capable of making reasonable policy rules. We also assume that the app developers are able to appropriately handle resource access denial, for example, by either notifying the device users or by silently terminating the app. Although the app needs to be repackaged if its security policies are updated, we believe such updates occur far less often than app update itself, and would not bring noticeable overhead to app's updating process.

Function call interception: CASE uses two standard dynamic function call interception techniques, namely GOT hooking and binary patching. The basic functioning of both techniques are well known. However, we gained new insights into using the techniques on Android for enforcing security policies.

Our prototype uses GOT hooking for intercepting invocations of system call wrappers. The technique is particularly convenient to implement on Android, whose dynamic linker resolves all GOT entries ahead of time (*i.e.,* no support for lazy binding) and locks the memory pages afterwards. Therefore, CASE overwrites all the GOT entries that correspond to the functions in $MinSet$ at once during the CASE initialization phase without having to force the linker to resolve the symbols. We also patch any newly loaded libraries by monitoring all `dlopen` system calls. Since all system calls are exported and resolved from GOT at the time of use, we can easily achieve complete mediation.

On the other hand, since there are no centralized function pointer tables for VM-level APIs, we choose binary patching to intercept VM-level APIs in $MinSet$. Although not as easy to implement, the binary patching technique offers two unique benefits over the GOT hooking technique. First, it is quick to apply and takes effect globally inside a process. For example, patching VM APIs only takes a linear memory copy (*i.e.,* overwriting the old code page with a branch instruction to the trampoline) and does not involve any table lookups or per-function treatment as GOT hooking does. It is effective on all upcoming API calls regardless of the calling module. Second, it can intercept calls to private functions (in this case, private VM APIs), which are not exported to GOT at all. These functions exist because they are supposed to be only used by the Dalvik VM internally (*i.e.,*`libdvm.so`). External API calls are secured by the native-safe pages.

Native-safe pages: The native-safe pages store the security-critical resources that do not have any dominant accessors, including system call wrappers and the VM's internal data. The implementation of the native-safe pages can directly influence the security and robustness of the entire CASE system. The location of the native-safe pages for the system call wrappers have to remain undecidable by adversaries. It is known that, even if ASLR is present, the memory pages allocated using `mmap` are typically continuous in virtual memory and thus deterministic. CASE forces `mmap` to allocate a randomly located page sequence by providing a random base address in user space and then verifying the allocation result. In addition, CASE makes the first and the last page in the sequence non-accessible and uses them as guard pages to prevent linear memory search by malicious code.

The system call wrappers that are not security-sensitive, but are located on the same memory pages as functions in $MinSet$, have to be handled carefully. We do not monitor these functions, but since these pages will be locked at the time the execution enters native threads, we copy them into the native-safe pages as well. We silently forward the non-sensitive system calls to their counterparts in the native-safe pages without performing any policy checks. In addition, the native-safe pages for the internal runtime data have to cover not only the data regions of `libdvm` (*e.g.,*`.data` and `.bss`) that are page aligned, but also the heap area for the VM's book-keeping data that is pre-allocated by the Zygote process and not mixed with apps' data.

Concealed handler: The concealed handler ensures the hidden functions and data are not referenced by any user-space pointers. The hidden code and data pages required by the concealed handler are allocated during the initialization phase in a similar fashion as the native-safe pages for system call wrappers. All global data items used by concealed handler are organized into an array, which is always allocated at the beginning of the hidden data page. The data are referenced via relative offset to the array's starting address, which is also the base address of the hidden data page. This implementation detail allows for efficient address resolution and easy data access. Before being mapped to the hidden code page, the concealed handler's code is localized with the hardcoded array address adjusted to the address of the hidden data page. Therefore, during runtime, concealed handler's code always finds its data via simple array indexing.

CASE registers the concealed handler as the handler for the relevant signals, (*e.g.,*`SIGSEGV`, `SIGSTOP`, and `SIGCONT`). The original handlers are saved, and can invoked by the concealed handler when a signal is not meant to be handled by CASE. This is critical for preserving Dalvik's signal handling chains. When a signal is raised as a result of CASE's enforcement, concealed handler parses the `ucontext` object on the stack, which describes the execution context when the fault occurred, including the instruction pointer, the register values, *etc.* The concealed handler uses the instruction pointer to determine which routine should handle the signal. It may also receive implicit parameters from the registers and return values back by overwriting the registers (*e.g.,* passing the address of a hidden system call wrapper via the register). All the routines in the concealed handler are kept short and do not perform asynchronous operations or raise signals, a requirement imposed by the OS on signal handlers.

5. DISCUSSION

Policy Specification: This paper focuses on proposing the new module-level enforcement mechanism while leaving the formal specification and automatic generation of policies for future work. However, for the sake of demonstration, we include below a basic language specification that we currently use to compose policies for CASE:

```
Rule::= {Sub Op Obj Cond}
Sub ::= class_ID|JNI_lib_name
Op  ::= read|write|execute
Obj ::= {sys_resource|vm_resource} x Mod
Mod ::= scope_modifier
Cond::= runtime_conditions
```

We designed CASE's policy specification to be comprehensible and easy to use for intended users, namely app developers. They can define policies to restrict untrusted modules in two simple steps: (1) identifying the modules with the class names or executable names; (2) choosing the accessible resources and the access conditions for the modules. For example, an app developer can define a policy that allows a third-party library in her app (*i.e.,*`Sub = com.example.lib.*`) to send/receive SMS to/from a particular number (*i.e.,*`Obj = SMS, Mod=Number:1234567, Op = read & write`). The policy can also require user's presence at the moment of the SMS access (*i.e.,*`Cond=User_Active & App_Foreground`). This policy prevents an untrusted library, which only needs limited SMS capability (e.g., for user registration), from stealthily sending SMS to unexpected numbers (*e.g.,* a premium number) or reading SMS meant for other apps (*e.g.,* a 2nd-factor authentication code). We note that, to define policies, app developers do not need any knowledge about how the untrusted modules work but just a high-level understanding of the modules' expected behaviors. Developers should have this high-level understanding of the modules because they have chosen to use the modules in their apps. Furthermore, with the help of a simple policy generator[2], app developers would not need to understand the policy syntax or write raw policies. Instead, the tool would assist developers via GUI to choose modules, resource types, access scope, and conditions, and eventually create policy stubs based on developers' selections. Therefore, we do not expect policy definition would prevent app developers (or users), including those with little security expertise, from making use of CASE to protect their apps.

Porting CASE to ART: We used Dalvik as the reference VM for implementing CASE because it was the default Android runtime when our project started. Although ART has replaced Dalvik in recent Android releases, we confirmed that the Dalvik-based design of CASE applies to ART as well and porting CASE to ART does not involve design changes. This is because, despite the major difference introduced by the ahead-of-time compilation, the two runtimes largely share the same in-VM interfaces for class and object management and use the same JNI bridge, which allows the dual-layer interception, the native-safe pages, and the concealed handler to remain effective for ART.

Alternative Usages: While it is mainly designed for app developers, CASE can be alternatively used by the IT administrators and end users to enforce module-level security policies on apps that, for instance, using apps containing untrusted or unwanted modules. Although it is challenging for average users to identify app modules and define proper policies, we believe that tech-savvy users and IT administrators should be able to describe modules using packages names and class names (i.e., easily retrievable and human readable) without understanding app designs or needing developer assistance. However, when not used by app developers, CASE cannot produce app packages signed by the original app developer keys, which may cause inconvenience to automatic app updates.

Memory Layout Disclosure: The effectiveness of the native-safe pages and the concealed handler partly relies on the confidentiality of app memory layout. There have been known channels through which the memory layout of a process can be observed or inferred. Our design considers and blocks these channels. For instance, certain system calls take user-space addresses as parameters, and when receiving an inaccessible address, they return a error code rather than triggering a page fault. By employing the dual-layer interception, CASE prevents attackers from exploiting these system calls to bypass guard pages that surrounding hidden pages. Similarly, CASE blocks direct access to the `/proc/self/maps` files by JNI code, preventing memory mapping disclosures.

6. ANALYSIS AND EVALUATION

We conducted a series of experiments to evaluate CASE in terms of its completeness and robustness, compatibility with COTS apps, and runtime overhead. Our experiments involved the following sets of apps and test inputs:

($S1$) a set of 420 apps, selected from the 50 most downloaded apps across 26 function categories from Google Play and HiAPK (a top alternative Android market), excluding games and apps that do

[2]Building this tool should only involve straightforward development effort and is out of the scope of this paper.

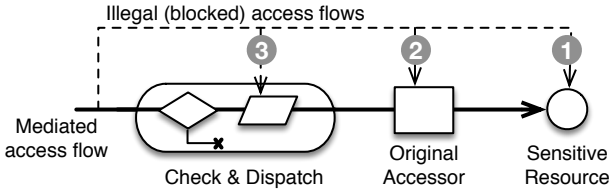

Figure 5: Potential evasion or tampering of CASE

not use any sensitive permission (*i.e.,* an indicator of low security relevance);

($S2$) a set of 20 apps randomly drawn from $S1$ for semi-manual, in-depth study and tests;

($S3$) a set of in-house and Android's unit tests for stress and coverage testing.

We first analyzed the *completeness and robustness* of CASE's enforcement (§ 6.1). We then examined CASE's *deployment cost and compatibility* by exercising our automatic app patcher against $S1$ and subsequently testing the patched apps (§ 6.2). The results show that patching CASE into legacy and COTS apps is fast, effortless, and free of compatibility issues. We finally measured the *runtime overhead* of CASE in three settings: enforcing generic security policies on a large number of apps in $S1$; enforcing app-specific policies (*i.e.,* manually tailored for each app) on the 20 apps in $S2$; and stress testing the critical paths of CASE using $S3$ (§ 6.3).

The results indicate that CASE is efficient for practical use. All experiments were conducted on an LG Nexus 5 phone running Android Kitkat.

6.1 Security Analysis

For benign apps, it is safe to assume that their code follows a regular pattern. This indicates that the app code solely relies on system calls for accessing system-managed resources, and the Dalvik VM for Java-level cross-class interactions. We propose *dual-layer interception* to monitor app's sensitive operations by intercepting APIs at two layers, namely the system call wrappers and the interfaces exposed by the Dalvik VM. We identified a minimum set of functions that cover all cross-boundary interactions. This way, we could effectively and completely mediate all security-critical activities on both layers.

In addition to the comprehensive security enforcement, CASE defends against the most powerful adversaries at app-level, who may attempt either bypassing security checks or tampering with the CASE enforcement mechanism itself. The adversaries are allowed to deploy malicious code, either in Java or in native form, in enforced apps alongside CASE. Figure 5 generalizes possible evasions into three types. We show that CASE is designed to provide countermeasures to all of them and achieve robustness.

First, malicious native code may attempt to directly access the sensitive resources (①) without invoking the typical resource accessors. For example, it may perform system calls using embedded assemblies rather than the `libc` wrappers; it may also locate the VM's internal data items in memory (*e.g.,* class table) and manipulate the raw data without calling VM APIs. **Countermeasure**: CASE forbids embedded system call instructions in apps. This is checked by the automatic patcher at app rewriting phase. For sensitive VM data, we allocate *data native-safe pages* to hold these data. The native-safe pages are hidden at random memory location, and protected with guard pages to prevent memory scan. We allow the native-safe pages to be accessible when Java threads are running, since code compiled from high-level languages (*i.e.,* Java) can not directly access virtual memory. When native threads starting, we immediately lock the pages and prevent access to VM data.

Second, malicious code may try to bypass the function interception by directly invoking the resource accessors (②). For example, native code can invoke the system call wrappers or VM-level APIs via function pointers or indirect jumps. **Countermeasure**: CASE disables the default system call wrappers while native code is running. Even if malicious native code can acquire the function pointers to the system call wrappers, directly invoking them will cause a page fault. Similarly, we use *code native-safe pages* to keep a hidden copy of system call wrappers. Only legitimate system calls that go through our trampolines are redirected to these code native-safe pages. On the other hand, although the VM-level APIs are not protected as the system call wrappers, we lock the memory pages that hold VM internal data essential to these APIs. Any unchecked invocation of such APIs eventually will hit the locked data native-safe pages and fail.

Third, sophisticated malware may launch control-flow attacks against the trampolines (③), where intercepted calls to functions are checked for policy compliance and dispatched to the original accessors if the check passes. For example, malware may perform ROP-style tricks that coerce the control flow to jump to the middle of a trampoline right after the checkpoint, continue on the affirmative branch, and later return to the malicious code when the resource access finishes. **Countermeasure**: CASE ensures the atomicity of its security checks and trampolines. For system calls, we design the *concealed handler* mechanism to only expose the address of the hidden system call wrappers if the execution arrives at the trampolines via the first instruction and goes through the entire check. For VM APIs, the data native-safe pages remain locked during the entire execution of native code, which strictly blocks any access to the VM's internal data by native code.

6.2 Deployment Cost and Compatibility

The automatic app patcher was designed to minimize the efforts and cost associated with deploying CASE in practice. It helps developers to enforce fine-grained policies and least-privilege principle among different modules in their apps without having to implement the complex enforcement mechanisms by themselves. To measure the performance of the app patcher, and more generally, the overhead and compatibility of CASE's rewriting approach, we ran the patcher through the 420 apps in $S1$. We used a set of generic policies that reflect some common security needs of most apps, rather than specifically defining a policy for each app:

($P1$) `com.package execute (dlopen)x(Path: /system/lib/) always`

($P2$) `com.package.module read|write (FileSys)x(Path:$APPHOME/module/) always`

($P3$) `com.package.main read (Location)x(NULL) always`

($P4$) `com.package read|write (Internet)x(IP:xxx.xxx.xx.xxx) always`

($P5$) `com.package.module1 read (fieldID)x(com.package.module2) module1==module2`

On average, each app takes 6.34 seconds to be patched and the size of the app executable increases by 65.58 KB or 4.22%. Such costs are generally affordable given that the patching process happens only once per app install in an offline fashion and today's mobile devices are usually equipped with ample storage.

6.3 Runtime Overhead

The runtime overhead of CASE-enabled apps include the approach-intrinsic overhead and policy-specific overhead. The former is universal to all CASE-enabled apps whereas the latter depends on the nature and complexity of enforced security policies.

Table 2: Unit testing results

Operation	Number of runs	Crossed interface	Intercepted API	Per-operation time (ms)			Per-operation overhead (%)
				original app	CASE-enabled app	overhead	
Open a file	1,000	module-to-system	open	0.310512	0.314719	0.004207	1.35%
Modify the value of a global variable in another package	100,000	module-to-module	setField	0.021202	0.022097	0.000895	4.22%
Query 200 contacts	500	module-to-module	ioctl	38.181994	38.920963	0.738970	1.94%
Network access	10,000	module-to-system	connect	0.080651	0.087197	0.006546	8.12%

Table 3: End-to-end performance tests on $S2$

	Package name	**Activity**	**Original app (ms)**	**CASE-enabled app (ms)**	**Overhead (ms)**	**Overhead**
1	com.jb.mms	.ui.MainPreference, .ui.NotifyPreference	400.00	408.00	8.00	2.00%
		.ui.ComposeMessageActivity	356.00	370.00	14.00	3.93%
2	com.facebook.orca	.auth.LoginScreenActivity, .auth.SilentLoginActivity	978.00	1,018.00	40.00	4.09%
		.threadlist.ThreadListActivity, .creation.CreateThreadActivity	962.00	1,037.00	75.00	7.80%
3	com.fsck.k9	.activity.MessageList	312.00	318.00	6.00	1.92%
4	com.snda.inote	.activity.NoteViewActivity, .activity.NoteEditActivity	1,266.00	1,287.00	21.00	1.66%
5	com.yelp.android	.ui.activities.ActivityNearby, .ui.activities.ActivityBusinessListResults	333.33	357.00	23.67	7.10%
6	wind.android	.setting.activity.MoreAppSettingctivity, .setting.activity.SystemSettingActivity	352.00	370.00	18.00	5.11%
7	com.symantec.android.spot	.ui.EulaActivity	473.00	492.00	19.00	4.02%
8	com.corewillsoft.usetool	.chrome.browser.ChromeTabbedActivity	360.00	376.00	16.00	4.44%
9	com.zhangdan.safebox	.activities.MainActivity	584.00	632.00	48.00	8.22%
		.activities.card.AddCreditCardActivity, io.card.payment.CardIOActivity	902.00	908.00	6.00	0.67%
10	com.tecace.cameraace	com.android.camera.CaptureActivity	244.44	248.89	4.45	1.82%
		com.tecace.photogram.PEffectActivity	774.00	782.00	8.00	1.03%
11	com.piriform.ccleaner	.ui.activity.CleanActivity, .ui.activity.CleanCacheActivity	234.00	266.00	32.00	13.68%
		.ui.activity.AppManagerActivity	166.00	178.00	12.00	7.23%
12	com.gemini.calendar	.Month	236.25	242.50	6.25	2.65%
13	cn.ecook	.MainTab, .ui.SearchUser	730.00	796.00	66.00	9.04%
14	com.hotelvp.tonight	.activities.CityListActivity	260.00	280.00	20.00	7.69%
15	cn.dict.android.pro	.activity.UserGuidanceActivity, .activity.TranslationActivity	556.00	605.00	49.00	8.81%
16	com.fstop.photo	.ListOfFoldersActivity	840.00	882.00	42.00	5.00%
17	com.jiubang.go.backup.ex	.recent.summaryentry.SingleInfomation-ViewActivity, .BackupProcessActivity, .ReportActivity	806.00	850.00	44.00	5.46%
		.RestoreProcessActivity, .ReportActivity	300.00	326.00	26.00	8.67%
18	com.tvkdevelopment.nobloat	.activities.BackedUpApps	412.00	414.00	2.00	0.49%
		.activities.Blacklist	408.00	432.00	24.00	5.88%
19	org.mightyfrog.android. simplenotepad	.FolderView, .NoteEditor, .Checklist	366.00	382.00	16.00	4.37%
20	com.qo.android.am3	com.quickoffice.mx.FileSystemListActivity, com.quickoffice.mx.FileListActivity, com.quickoffice.mx.SaveFileAsActivity	390.00	434.00	44.00	11.28%
Average			14,001.02	14,691.39	690.37	4.93%

First, we measured the approach-intrinsic overhead in the same experiment setup as the compatibility test, where the 420 patched apps in $S1$ were automatically exercised with the generic policies enforced. The average delay in cold app startup is 0.174 seconds (a 17.6% slowdown). The timed period starts from the moment when the app launch intent is sent out to the moment when the default activity-ready Logcat message is captured. The slowdown is a result of loading the CASE library and initializing the enforcement mechanisms. The absolute delay is hardly noticeable in practice and varies marginally across different apps. The increases in peak CPU `utime` and peak CPU `stime` are 4.47% and 9.01%, respectively. They represent the worst case CPU utilization increases in user and kernel modes. The increased memory utilization, 1.19MB on average (`Uss`), matches with the sizes of CASE library and the native-safe pages. To measure CASE's impact on battery consumption, we conducted two drain tests on a fully charged battery: the first test repeatedly ran the unpatched apps in $S1$ till the battery died and the second test did the same but on the patched apps in $S1$. The first test lasted 5.52h while the second test lasted 5.19h (20 minutes 2 seconds less), indicating a 6.43% battery overhead of CASE.

Our second experiment consists of a series of stress testing. We created a set of unit tests (Table 2), each repeatedly carrying out a critical operation that CASE intercepts. The unit tests contain common security checks of different types to simulate the enforcement of comprehensive security policies. Compensating the previous end-to-end overhead evaluation, these unit tests offer insights into CASE's worst-case overhead associated with enforcing certain class of policies listed in $P1$ through $P5$. The results are shown in Table 2. The average per-operation overhead is less than 3.91%, which is unnoticeable in practice because the absolute delays are at the microsecond scale and not accumulative. Moreover, we notice that such overhead is negligible in the much longer end-to-end operations (*e.g.,* accessing files after obtaining the descriptor).

Third, to determine the end-to-end operation overhead of CASE, we performed semi-manual experiments on a sufficient yet manageable size of apps. For each app in $S2$, we exercised the selected features of the app in order to measure the relative delays caused by CASE's security enforcement. The tested features were selected based on the consideration that they should start from a user input event and proceed nonstop (*i.e.,* no further user input or asynchronous wait is needed) till the end marked by certain detectable UI changes. For instance, a testing feature of Yelp is the search of nearby restaurants. It begins with a touch event on the search button, then proceeds to read the GPS data and query the remote server, and eventually displays the search result in a list view.

Fully executing such a feature serves as an end-to-end test of CASE's enforcement overhead. It also allows for semi-automatic and precisely timed tests: we first identify a feature's triggering UI (*e.g.,* a button) and the terminating UI (*e.g.,* a result view); with this input, our UI automator then carries out the feature and times its execution in two separate runs—one on the original version and the other one on the CASE-enabled version of the app. The relative increase in the time spent for executing a feature quantifies CASE's end-to-end performance overhead. Table 3 shows the end-to-end testing results on the apps in $S2$. Across all the 20 apps, CASE incurred an average enforcement overhead of 4.93% (Table 3).

7. RELATED WORK

To mitigate threats imposed by malicious or vulnerable apps, several solutions have been proposed to enhance Android app security. These solutions either extend the underlying Android framework or instrument the app to include inline mediation hooks.

OS-level protection: MockDroid [3] and DeepDroid [22] mock sensitive data items when apps are not allowed to access them. AirBag [24] provides a virtualized app runtime environment that mediates apps' access to system resources. [4] prevents privilege escalation attacks by validating all inter-app communication (ICC) against app permissions. FlaskDroid [5] and SEAndroid [19] mediate component interactions for security enforcement. TaintDroid [9] and AppFence [11] dynamically track sensitive information within an application to detect privacy violations. ASM [10] and ASF [2] extend Android security with programmable interfaces and multi-layer access control. All these works introduce new app protection and security enforcement capabilities to the Android OS. In comparison, CASE addresses a different line of threats, namely the module-level attacks. It enables fine-grained class-level security mediation for the first time, without modifying the OS.

App-level protection: Different Inline reference monitors (IRMs) were proposed to control app's access to sensitive APIs through bytecode rewriting [7, 8, 13] or native code interposition [25]. Similar to CASE, they do not require modifications to the Android framework or OS, but unlike CASE, their mediation may be circumvented by malicious code using reflections or native code. DeepDroid [22] and FireDroid [17] use `ptrace` to monitor an app's behavior. However, these solutions require mobile devices to be unlocked and rooted, which severely limits their practical deployment and arguably weakens device security. AppCage [26] creates app-level sandboxes to constrain app code. While it does not require OS changes or root privileges, its enforcement applies to individual apps and cannot recognize app modules.

Component-level protection: There have been some proposals to split monolithic apps into mutually distrusting components [18, 20, 23] to administer independent security policies for these components. AdSplit [18] isolates advertisement libraries into separate processes and maintain UI consistency. Similarly, NativeGuard [20] runs JNI libraries in separate processes in isolation to the containing app. These process-based isolations work well on independent libraries, but they are too rigid and coarse to be applied to libraries or external code that are closely integrated with apps. Moreover, they suffer from high IPC overheads while requiring stubs to be created for each component. Another relevant body of research, including Apex [14], CRePE [6] and Saint [15], enables configurable and contextual security policies for regulating an app's interaction with other apps and its access to system-managed resources. In comparison, CASE recognizes app modules and allows security policies to be defined and enforced at module level—a currently missing security capability essential for defending against the module-level attacks. Aurasium [25] intercepts resource-accessing system calls via GOT hooking. CASE's dual-layer interception also intercepts system calls, though for a different purpose. In addition, CASE's interception covers the VM-level APIs, captures inter-module crossings, and more importantly, is protected by the native-safe pages and the concealed handler and is robust against malicious Java and native code.

8. CONCLUSION

We have presented the design, implementation and evaluation of CASE, a system that for the first time enables fine-grained module-level security in Android apps. The enforcement does not require any changes in the OS and is robust against malicious Java and native code, thanks to the three novel core techniques: dual-layer interception, native-safe pages, and concealed handler. We implemented a prototype of CASE for Android. Our evaluation against

420 real apps shows that CASE effectively and efficiently enforces module-level security policies, and is suitable for quick and wide adoption to defend against the emerging module-level attacks.

9. ACKNOWLEDGMENT

We thank the anonymous reviewers for their insightful comments. We thank our shepherd, Ardalan Amiri Sani, for his guidance on the final paper revisions. This project was supported by the National Science Foundation (Grant#: CNS-1421824) and the Office of Naval Research (Grant#: N00014-15-1-2378). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation or the Office of Naval Research.

10. REFERENCES

[1] Chinese taomike monetization library steals sms messages. http://researchcenter.paloaltonetworks.com/2015/10/chinese-taomike-monetization-library-steals-sms-messages/.

[2] Michael Backes, Sven Bugiel, Sebastian Gerling, and Philipp von Styp-Rekowsky. Android security framework: Extensible multi-layered access control on Android. In *ACSAC*, 2014.

[3] Alastair R. Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. MockDroid: Trading privacy for application functionality on smartphones. In *HotMobile*, 2011.

[4] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on Android. In *NDSS*, 2012.

[5] Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *USENIX Security*, 2013.

[6] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. CRePE: Context-related policy enforcement for Android. In *ISC*, 2011.

[7] Benjamin Davis and Hao Chen. RetroSkeleton: Retrofitting Android apps. In *MobiSys*, 2013.

[8] Benjamin Davis, Ben Sanders, Armen Khodaverdian, and Hao Chen. I-ARM-Droid: A rewriting framework for in-app reference monitors for Android applications. 2012.

[9] 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 Trans. Comput. Syst.*, 2014.

[10] Stephan Heuser, Adwait Nadkarni, William Enck, and Ahmad-Reza Sadeghi. ASM: A programmable interface for extending Android security. In *USENIX Security*, 2014.

[11] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: Retrofitting Android to protect data from imperious applications. In *ACM CCS*, 2011.

[12] Trend Micro Inc. Library file in certain android apps connects to c&c servers. http://blog.trendmicro.com/trendlabs-security-intelligence/library-file-in-certain-android-apps-connects-to-cc-servers/, 2012.

[13] Jinseong Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster, and Todd Millstein. Dr. Android and Mr. Hide: Fine-grained permissions in Android applications. In *ACM SPSM*, 2012.

[14] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: Extending Android permission model and enforcement with user-defined runtime constraints. In *ACM ASIACCS*, 2010.

[15] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in android. In *ACSAC*, 2009.

[16] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. AdDroid: Privilege separation for applications and advertisers in Android. In *ACM ASIACCS*, 2012.

[17] Giovanni Russello, Arturo Blas Jimenez, Habib Naderi, and Wannes van der Mark. FireDroid: Hardening security in almost-stock Android. In *ACSAC*, 2013.

[18] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. AdSplit: Separating smartphone advertising from applications. In *USENIX Security*, 2012.

[19] Stephen Smalley and Robert Craig. Security Enhanced (SE) Android: Bringing flexible MAC to Android. In *NDSS*, 2013.

[20] Mengtao Sun and Gang Tan. NativeGuard: Protecting Android applications from third-party native libraries. In *ACM WiSec*, 2014.

[21] Symantec. Ad library behind pulled ios apps also used in android development. http://www.symantec.com/connect/blogs/ad-library-behind-pulled-ios-apps-also-used-android-development, 2015.

[22] Xueqiang Wang, Kun Sun, Yuewu Wang, and Jiwu Jing. DeepDroid: Dynamically enforcing enterprise policy on Android devices. In *NDSS*, 2015.

[23] Yifei Wang, Srinivas Hariharan, Chenxi Zhao, Jiaming Liu, and Wenliang Du. Compac: Enforce component-level access control in Android. In *ACM CODASPY*, 2014.

[24] Chiachih Wu, Yajin Zhou, Kunal Patel, Zhenkai Liang, and Xuxian Jiang. AirBag: Boosting smartphone resistance to malware infection. In *NDSS*, 2014.

[25] Rubin Xu, Hassen Saidi, and Ross Anderson. Aurasium: Practical policy enforcement for android applications. In *USENIX Security*, 2012.

[26] Yajin Zhou, Kunal Patel, Lei Wu, Zhi Wang, and Xuxian Jiang. Hybrid user-level sandboxing of third-party Android apps. In *ACM ASIACCS*, 2015.