Hybrid User-level Sandboxing of Third-party Android Apps

Yajin Zhou†; Kunal Patel†, Lei Wu†, Zhi Wang‡, and Xuxian Jiang†∗
†North Carolina State University  ‡Florida State University  ∗Qihoo 360
{yajin_zhou,kmpatel4, lwu4}@ncsu.edu, zwang@cs.fsu.edu, xjiang4@ncsu.edu

ABSTRACT
Users of Android phones increasingly entrust personal information to third-party apps. However, recent studies reveal that many apps, even benign ones, could leak sensitive information without user awareness or consent. Previous solutions either require to modify the Android framework thus significantly impairing their practical deployment, or could be easily defeated by malicious apps using a native library.

In this paper, we propose AppCage, a system that thoroughly confines the run-time behavior of third-party Android apps without requiring framework modifications or root privilege. AppCage leverages two complimentary user-level sandboxes to interpose and regulate an app’s access to sensitive APIs. Specifically, dex sandbox hooks into the app’s Dalvik virtual machine instance and redirects each sensitive framework API to a proxy which strictly enforces the user-defined policies, and native sandbox leverages software fault isolation to prevent the app’s native libraries from directly accessing the protected APIs or subverting the dex sandbox. We have implemented a prototype of AppCage. Our evaluation shows that AppCage can successfully detect and block attempts to leak private information by third-party apps, and the performance overhead caused by AppCage is negligible for apps without native libraries and minor for apps with them.

Categories and Subject Descriptors
D.4.6 [Security and Protection]: Access control, Information flow controls

Keywords
Android; Software Fault Isolation; Dalvik Hooking

1. INTRODUCTION
Android has become the leading mobile platform with nearly 85% market share in the second quarter of 2014 [10]. This trend is accompanied by the vigorous increase of third-party Android apps that are available for users to download and install. However, recent studies reveal that third-party apps, including popular ones, could leak private information without user consent or awareness [12,19]. In light of these serious threats, there is a pressing need to strictly confine these apps, especially their access to sensitive data and dangerous operations. However, Android’s existing permission system lacks flexibility for this purpose. An user has to grant all the permissions requested by an app in order to install it, and cannot make any adjustment to those permissions after installation.

To address this limitation, researchers have proposed a number of systems to enable fine-grained control of Android apps. They roughly fall into two categories: the first category consists of solutions that extend the Android framework (and even the kernel) to allow fine-tuning of the apps’ permissions [5,14–16,27,33,37,45,55]. For example, AppFence [27] and TISSA [55] can be configured to return a mock location, instead of the real one, to the apps. Framework enhancement appears to be a natural solution. However, the requirement to update key Android components could strongly impair their practical deployment due to the deep fragmentation of the Android platform [46]. The second category includes the “in-app” mechanisms that control the app’s access to sensitive APIs through (Dalvik) bytecode rewriting [13, 17, 18, 28] or native code intercepting [47]. These systems do not require changes to the Android framework and thus can be readily deployed. Specifically, bytecode rewriting-based approaches insert inline reference monitors to regulate apps’ behaviors. However, dynamically loaded classes (e.g., classes loaded by DexClassLoader) could pose serious challenges to these systems since the dynamically loaded bytecode is not available when statically rewriting the app. On the other hand, Aura- sium [47] intercepts the inter-process communication [2] between the app and remote Android services. Therefore, it can accommodate dynamic classes, but has to bridge the “semantic gap” by reconstructing high-level semantics from the exchanged raw messages, overall a tedious and error-prone task. More importantly, all those systems can be bypassed or subverted by a malicious app using a native library. For example, the app can leverage native code to tamper with the instrumented bytecode [1] or bypass security checks by “restoring the global offset table entries” [47].

In this paper, we present AppCage, a secure in-app mechanism to regulate the app’s access to sensitive APIs and dangerous operations (e.g., making phone calls). Like other in-app mechanisms, AppCage does not need to change the Android framework; nor requires the root privilege, and thus can be readily deployed. Specifically, AppCage synthesizes a wrapper app for each target app (i.e., a third-party app to be confined). The wrapper sets up the dex sandbox for the app by hooking into its instance of Dalvik virtual machine and redirecting sensitive framework APIs to their respective
stubs. The stubs interpose the access to those APIs and enforce the user-defined policies. With API hooking, AppCage naturally supports dynamically loaded classes because the app eventually needs to call those APIs to retrieve sensitive data. However, API hooking alone could be subverted by the app using native code. To address this challenge, AppCage relies on a second sandbox, native sandbox, to confine the app’s native code. The native sandbox leverages software fault isolation (SFI) [42] to ensure that the app’s native libraries (including the system libraries they depend on) cannot escape from the sandbox or directly modify the code or data outside the sandbox. The app thus cannot tamper with the dex sandbox using native code. In addition, native sandbox prevents the app’s native code from directly requesting sensitive data or performing dangerous operations via binder, the Android’s ubiquitous inter-process communication mechanism. Combining those two sandboxes, AppCage can comprehensively interpose the app’s access to key Android APIs and enforce user-defined policies to fine-tune the app’s permissions.

We have implemented a prototype of AppCage and evaluated its effectiveness, compatibility, and performance. Our experiments show that AppCage can successfully detect and block attempts to leak the user’s private data by both malicious and benign-but-invasive Android apps, and the security analysis demonstrates that native sandbox can protect our system from being bypassed or subverted. Moreover, the prototype is shown to be compatible with the popular Android apps we downloaded from the Google Play store, and it incurs negligible overhead for apps without native code and a mild 10.7% overhead for ones with native code. Given the protection offered by AppCage, we consider the performance of AppCage is acceptable for most daily uses.

In summary, this paper makes the following contributions:

- We propose hybrid user-level sandboxes to confine two components of an Android app – the bytecode and the native code. Particularly, dex sandbox relies on API hooking to reliably interpose the bytecode’s access to key framework APIs, while native sandbox applies the proven technology of software fault isolation to confine the native code.
- Our native sandbox leverages both dynamic binary rewriting and static compiler-based binary instrumentation to reduce performance overhead. They enforce the same set of rules and can be seamlessly integrated.
- We have implemented and evaluated a prototype of AppCage. The experiment results show that AppCage is effective and compatible with popular apps, and it incurs acceptable overhead for daily use.

2. BACKGROUND AND THREAT MODEL

In this section, we briefly introduce some key concepts in Android to provide necessary background information of the proposed system, and then present the threat model.

2.1 Dalvik Virtual Machine

Most Android apps are written in the Java programming language and compiled into the bytecode for Dalvik VM. Dalvik VM is a shared library loaded into each running app and is responsible for executing the app’s bytecode. To support later-binding of Java, Dalvik VM maintains a data structure for each Java class in the app (ClassObject) and one for each of its methods (Method). The Method structure in turn contains a pointer to the method’s bytecode. Figure 1 shows a simple class and its representation in Dalvik VM. When Sample.M is called, the VM searches for ClassObject of the class and further method of the target method [11]. It then retrieves the bytecode of the method to decode and execute it.

Dalvik VM allows an app to dynamically load extra classes using DexClassLoader. The app can leverage this convenient capability to change its behavior at run-time. Dynamic class loading poses serious challenges to systems based on the bytecode rewriting because those classes are not available when the app is statically rewritten [17, 18, 28]. AppCage can naturally support this feature because it is based on the API hooking.

2.2 Java Native Interface

Java Native Interface (JNI) defines a framework for bytecode and native code to call each other. Android developers can use NDK to implement part of their apps in native languages such as C/C++. NDK compiles the source files into shared native libraries that can be dynamically loaded into the app by Dalvik VM under the request of the app’s bytecode. Specifically, the VM uses the dl_open function to load the library into the app, and resolves the address of each imported library function using dlsym. These addresses are cached by the VM to avoid duplicated address resolution later. When a native function is invoked by the bytecode, it passes a special data structure of type JNIEnv, which allows the native code to interact with the bytecode [6, 39]. For example, the native function can use JNIEnv->FindClass(“Sample”) to locate the Java class Sample and subsequently call its functions. AppCage needs to interpose those key JNI related functions to (1) intercept the loading of native libraries and prepare the native sandbox for them, (2) and switch the run-time environment when entering and leaving the native sandbox.

2.3 Dynamic Loading and Linking

Android implements its own dynamic loader and linker for native libraries (/system/bin/linker). Unlike its counterpart in the desktop, Android’s loader resolves all the external functions eagerly. For example, if the app’s native code depends on the syslog function in libc.so, the loader will promptly load the library and recursively resolve the function address. Even though Android does not support lazy address resolution, the PLT/GOT structure is still used for dynamic linking. More specifically, the compiler generates a stub in the PLT section for each external function. All calls to that function in the library are redirect to the stub, which simply contains an indirect branch to the address in the associated GOT entry. When a native library is loaded, the loader resolves the address of the external function and fills it in the GOT entry.

2.4 Threat Model

Similar to existing solutions [27, 55], we assume an adversary model where third-party apps are not trustworthy (e.g., they could leak personal information) but some of their features are desired by the user. Nevertheless, this privacy-aware user wishes to regulate...
the apps’ access to the private data and dangerous operations. Our system leverages a utility app that runs on the user’s phone to instrument the target app, and this utility app is trusted. Moreover, we assume that the underlying Linux kernel and the Android middlewares are trusted and attackers do not have the root privilege.

### 3. SYSTEM DESIGN

In this section, we describe the design of AppCage, particularly its hybrid user-level sandboxes: dex sandbox and native sandbox.

#### 3.1 Overview

The goal of AppCage is to interpose key Android APIs to control the third-party app’s access to private data and dangerous operations. Android apps consist of Dalvik bytecode and the optional native libraries. It is necessary to control both components of an app. AppCage provides dex sandbox and native sandbox for this purpose, respectively (Figure 2). To confine bytecode, AppCage hooks into Dalvik VM (libhook.so), and then manipulates its internal data structures to redirect each important Android API to its stub provided by AppCage. At run-time, the stub queries the permission manager whether the operation should be allowed, denied, or prompted to the user for confirmation. If the operation is allowed, the stub will then call the actual API on behalf of the original caller. Figure 2 shows how the framework method \( f \) is interposed by \( \text{stub}_f \). Native sandbox applies software fault isolation to confine the app’s native code. It prevents the app from using native code to subvert dex sandbox or directly request system services through binder, Android’s lightweight remote procedure call mechanism [2]. Note that user libraries may depend on some system libraries such as \( \	ext{l1bc.so} \) or \( \	ext{l1bm.so} \). AppCage provides a set of confined system libraries to the native sandbox. (Dalvik VM is still linked to the original system libraries.) With both sandboxes, AppCage has complete control over the app’s access to sensitive APIs.

**Use case:** Figure 2 illustrates the run-time state of a confined app. In the following, we describe the use case of AppCage and how the sandboxes are initialized. AppCage provides a utility app that runs on the user’s phone. For each third-party app to be installed, it generates a wrapper app that carries the whole original app, the bytecode of the stubs, and \( \	ext{libhook.so} \) (Figure 2). If the app has the native code, it also contains an instrumented copy of the native code. The wrapper requests the same permissions as the original app, and its entry point is assigned to a function in AppCage that is tasked to set up the sandboxes before executing the app [47]. The utility app also monitors the installation of new apps from the official and alternative app stores. For the former, it monitors the directory where the apps normally reside (\( /\text{data/app} \)) and prompts the user to uninstall the original app and replace it with the generated app. The process is mostly automated and the user only needs to click a few buttons when prompted. We cannot intercept the apps from the official app store because that requires the system privilege. For the latter, it can intercept the installation of the app by listening to the INSTALL_PACKAGE intent, and generate and install the wrapper app on-the-fly. The original app is not installed.

#### 3.2 Dex Sandbox

AppCage interposes the app’s access to key framework APIs by essentially hooking those APIs. In contrast to previous systems that rewrite the app’s bytecode for the same purpose [17, 18, 28], AppCage does not require extra efforts to support dynamically loaded classes and can tolerate obfuscation of the app’s bytecode because the app eventually needs to call those interposed APIs to be effective.

AppCage’s API hooking is implemented through direct manipulation of Dalvik VM’s internal data structures. (We will discuss the compatibility issue raised in Section 5.3.) As mentioned in Section 2.1, Dalvik VM maintains a ClassObject data structure for each Java class in the app including those of the framework, through which we can find all the methods of the class. For each framework class that has sensitive methods, we manually create a stub class that contains the same set of stub methods 1. The stub methods query the permission manager whether the operation should be allowed, and call the original methods if so.

Figure 3 shows an example of the SmsManager class, which allows an app to send text messages in the background via the predefined \( \	ext{sendTextMessage} \) method, possibly to premium numbers. To interpose this method, AppCage loads \( \text{SmsHooks} \), the stub class for \( \text{SmsManager} \), into the app, and manipulates the pointers in their corresponding ClassObject so that \( \text{SmsManager}.\text{sendTextMessage} \) points to \( \text{SmsHooks}.\text{sendTextMessage} \) and the pointer to the original method \( \text{SmsManager}.\text{sendTextMessage} \) is stored in \( \text{SmsHooks} \)’s array of direct methods (arrow 1 and 2 in Figure 3, respectively). As such, the app will be redirected to \( \text{SmsHooks}.\text{sendTextMessage} \) when it tries to send a text message and subject to policy check. If the operation is allowed, \( \text{SmsHooks}.\text{sendTextMessage} \) calls the original method, which can be found in \( \text{SmsHooks} \)’s array of direct methods. Unlike virtual methods, direct methods are called directly without dynamic method resolution [11]. Hence, it is guaranteed that the original method will be called. By only manipulating method pointers, our system is compatible with the just-in-time compiling by Dalvik VM, a key technology to improve the system performance.

#### 3.3 Native Sandbox

Android allows its apps to use native code via the JNI interface (Section 2.2). Native code is often used to speed up performance-critical tasks such as rendering of 3D games. However, native code

1Most of this process can be automated. Doing it manually is acceptable since it is only a one-time effort.
could be exploited to subvert security schemes based on bytecode. For example, it can revert the changes to Dalvik VM’s data structures made by AppCage. To address that, we adopt the software fault isolation (SFI) technology to confine the app’s native code. AppCage’s native sandbox provides the following security guarantees: native code cannot write to memory out of the sandbox so that it cannot tamper with dex sandbox (memory read does not pose a threat to AppCage because sensitive data are maintained by Android’s system service app); native code cannot escape from the sandbox; the access to dangerous instructions such as system call is regulated. Since source code of the app’s native binary is not available, we use binary rewriting to implement software fault isolation.

Binary rewriting of the native code can take place at both installation time and run-time: AppCage instruments any native libraries discovered when generating the wrapper app (Section 3.1). It also hooks into the related JNI API to translate native libraries that are unknown during the installation, such as the encrypted libraries or the downloaded ones. Our binaryrewriter enforces the same set of rules as Native Client (NaCl) [38, 48]. Particularly, instructions are grouped into equal-sized bundles (16 bytes), and indirect branch instructions such as indirect call and return must target the boundary of a bundle. Moreover, the instructions inserted by AppCage (bic and xor) to confine an instruction must be put in the same bundle as that instruction so that they cannot be bypassed by indirect branches jumping over them. The rules of NaCl provide our native sandbox a solid theoretical and practical foundation in security. However, our binary rewritertakes care of the app’s native code in the binary format, while NaCl is a compiler plugin and thus requires the access to source code. Code generated by NaCl normally has less overhead than that by our binary rewritertaking care of the app’s native code.

As such, AppCage also utilizes a modified NaCl compiler to confine system libraries that the app’s native code rely on since their source code is readily available.

Even though ARM is a RISC architecture, rewriting ARM binaries is not straightforward: 1.) an ARM binary can mix the ARM instructions (32 bits) and the THUMB-2 instructions (16 or 32 bits). 2.) Constants are often embedded among instructions. We need to identify them and prevent them from being executed. 3.) ARM instructions can directly refer to the program counter (pc), often to read the embedded constants in code. Binary rewriting shifts the instructions around and may cause the wrong pc to be used. To address challenge 2 and 3, AppCage retains the original code section of the app’s native code but makes it read-only and non-executable. The translated code section only contains instructions but not constants, which must be loaded from the original code section instead. AppCage also keeps a mapping between the original pc and the translated pc and converts them when necessary. To rewrite a binary, AppCage first disassembles it, breaks it into basic blocks, and instruments it as required by the native sandbox.

Disassembling Native Code: AppCage recursively disassembles the app’s native code [50] using the exported functions as the initial starting points (the binary may be stripped and may not contain the complete symbol table.) Specifically, it keeps disassembling instructions from a starting point until a return instruction or other terminating instructions. Any targets of direct jumps or call instructions are added to a work list as the new starting points. After exhausting the work list, we start disassembling the leftover gaps with a trial-and-error strategy.

Our disassembler faces two challenges. First, constants are often embedded in-between instructions because they do not fit in the instruction (32 bits at most). To address that, we observe that constants are often collected at the end of functions and referred to with the pc-relative addressing mode, for example,

```assembly
ldr r1, [pc, #184]
```
of branch instructions need to be instrumented to ensure that the app’s native code cannot escape from the sandbox. Direct branches can be handled completely during binary rewriting because their targets are known statically. We only need to verify that they target legitimate instructions in the sandbox and patch the immediate offsets accordingly (case 5 in Figure 4). Indirect branches require validation of the branch targets at run-time because they are unknown during translation. The address of the target may be on the stack (case 6 in Figure 4) or in a register (case 7 in Figure 4). AppCage uses a trampoline to handle the indirect branch instruction. The original instruction is replaced with a direct branch to its associated trampoline. The trampoline retrieves the original branch address and verifies that it lies within the sandbox. If so, the trampoline further converts the address to the translated target, with the help of the mapping between the original pc and the translated pc, and branches to it.

• c. Instructions using pc as a general register: in ARM, pc (program counter) can be directly accessed as a general register. Instructions using pc as the destination operand are in fact indirect branch instructions. This case has been discussed in b). If pc is used as the source operand, its value is decided by the address of the currently executing instruction, and thus is different from the original and intended value. For example, the following instruction at address 0x14e80 is relocated to 0x1748e by the binary rewriter. When the app runs, the pc register has a value of 0x17494, instead of the expected value of 0x14e88 (for historic reasons, pc in ARM is the address of the current instruction plus 8.)

| 14e80: add r1, pc, r1 --> 1748e: add r1, pc, r1 |
| PC: 0x14e88 | PC: 0x17494 |

To address this, we allocate a scratch register, load the original pc into it, and patch the instruction to use the scratch register instead of the current pc (case 8 in Figure 4).
d.) System call instruction: uncontrolled system calls can be exploited to subvert our system, for example, by tampering with the memory protection or calling system services through binder. As such, AppCage disallows direct system calls in the app’s native libraries. The app instead has to access the kernel services through the APIs of the system libraries (e.g., \texttt{libc}). This is barely restrictive because very few apps, if any, rely on direct system calls. To prevent those libraries from being misused, AppCage provides a confined copy of the necessary system libraries to the native sandbox. During rewriting, direct system call instructions in the app’s native code are replaced by branches to a function that terminates the current app.

System Libraries Instrumentation: the app’s native code often relies on system libraries for service (e.g., \texttt{libc.so}). For performance reasons, AppCage uses the NaCl compiler for ARM \cite{38} to sandbox the system libraries. Those libraries are loaded into the native sandbox by a custom loader and linked with the app’s native code (Dalvik VM uses an unconfined version of the system libraries). Those libraries are subjected to the same constraints as the app’s native code: memory writes and branch instructions must target locations within the native sandbox. However, NaCl compiler assumes that the lower 1GB memory is reserved for the sandbox. This cannot be guaranteed by AppCage because the location of native sandbox is unknown until it is initialized at run-time. Hence, we need to customize the NaCl compiler for our purpose.

NaCl uses the \texttt{bic} instruction to clear the most significant two bits of a branch target to ensure it is within the first 1GB of the address space. The last four bits of the target are also cleared to prevent the instrumented code from branching to the middle of an instruction bundle (16 bytes):

\begin{verbatim}
[bic r6, r6, 0x0000000f] /* clear top 2 and the last 4 bits*/
\end{verbatim}

For memory store instructions, NaCl uses the \texttt{tst} and \texttt{streq} instructions to conditionally execute the memory write if the address is inside the sandbox.

\begin{verbatim}
tst r6, #0xc0000000 /* within the first 1GB?*/
streq r6, [r0, #12] /* store to memory if so*/
\end{verbatim}

Because native sandbox is not guaranteed to start at address 0, we modify the NaCl compiler to emit \texttt{bic} and \texttt{err} instructions to confine sensitive instructions, similar to the binary writer. However, we have to keep the immediate values of those instructions undefined during translation because the location of native sandbox is unknown until it’s initialized at run-time. AppCage’s custom loader patches those instructions with the actual location of the sandbox.

JNI Interface: the app’s bytecode and native code can call each other through the JNI interface. However, Dalvik VM and native sandbox have different contexts under AppCage such as stack, heap, and the code section. AppCage needs to intercept the JNI calls and switches the context accordingly. This has to be performed in both directions, from bytecode to native code and vice versa.

Bytecode can load and resolve native functions via Android’s dynamic linker (/system/bin/linker) and call it through the JNI interface. To intercept those calls, we hook the \texttt{dlopen} and \texttt{dlsym} functions in Dalvik VM. The \texttt{dlopen} hook allows us to rewrite native libraries unseen during the installation (e.g., a newly downloaded library). Moreover, when \texttt{dlsym} is called to resolve a native function, we return its associated gate function in place of the target function. The gate prepares the execution context for the sandbox by copying over the parameters and switching the stack and registers. It then enters the sandbox to execute the target function. When the function returns, the gate switches the context back to Dalvik VM. Meanwhile, native code can also call exported bytecode functions through the JNI\texttt{Env} structure. To intercept those calls, we replace the functions in JNI\texttt{Env} with their stubs in the sandbox. The stub switches the execution environment to that of Dalvik VM and then calls the actual JNI\texttt{Env} functions \cite{39}. The direct reference to Java heap is processed through the copy-in and copy-out mechanism \cite{39}.

4. IMPLEMENTATION

We have implemented a prototype of AppCage. In particular, dex sandbox is implemented in the Java and C++ programming language, while native sandbox is implemented in C and the ARM assembly. In this section, we discuss the concrete implementation of this prototype.

4.1 Dex Sandbox

AppCage generates a wrapper app for a third-party app. The wrapper app consists of the original app and the additional components for the sandbox. AppCage features both a dex sandbox for the bytecode and a native sandbox for native code. Dex sandbox hooks into the sensitive framework APIs to enforce the user-define policies. These policies are managed by a separate app (permission manager in Figure 2). Only permission manager can update the user policy database; other apps can read the database through an exported content provider interface. Table 1 lists the operations that our prototype can interpose. It is relatively easy to extend this list by adding more stubs. Permission manager can respond to a request with three verdicts: allow, deny, and prompt-to-user. It can be enhanced to return mock results (e.g., fake location) to improve its compatibility with third-party apps \cite{27, 55}.

Original App Loading: AppCage loads the embedded app into dex sandbox for execution. It leverages the DexClassLoader class for this purpose. However, this class loader is different from the class loader used by Dalvik VM to load the wrapper app and the stubs (PathClassLoader), and only classes loaded by the same class loader can refer to each other (this is also true for apps without AppCage). Therefore, classes in the original app cannot communicate with the stubs. To address this, we change the value of \texttt{pathList} of PathClassLoader to that of DexClassLoader. After that, the classes of the original app will behave like that they are loaded by the same loader as the stubs.

4.2 Native Sandbox

Native sandbox confines the app’s native code into a continuous block of memory space. Figure 5 shows the layout of the app’s address space at run-time. The memory for native sandbox is 256MB aligned so that we can use the simple \texttt{bic} and \texttt{err} instructions to control memory access. AppCage has a custom loader for the native code. It intercepts the app’s requests to load the native code to sandbox it. Specifically, the loader will load both the original code section and the translated code section. The former is marked
as read-only and non-executable. We keep this section so that pc-relative instructions can access the correct memory. The loader also needs to fix various addresses in the translated code, such as those of the indirect jump trampolines.

**Link to System Libraries:** the app’s native code is linked to NaCl-compiled system libraries. NaCl requires that branch targets aligned at the boundary of a bundle (16 bytes). This alignment ensures that the sandboxed code cannot jump into the middle of a bundle and bypass security checks. As such, we need to align return addresses of the native code at the boundary of a bundle. For example, the instruction at offset 0x440c calls __android_log_print@plt, which eventually jumps to the __android_log_print function in the system library. When this function in the system library returns, the last 4 bits of the lr register are masked out by NaCl [38]. When translating this instruction, we need to add padding instructions (nop) to ensure that the return address is 16 bytes aligned (0x22b0 in the example).

Changes of System Libraries: AppCage provides the NaCl-compiled system libraries to the native sandbox. It makes the following additional changes for them to safely run in the sandbox:

*First*, the heap management functions in libc such as malloc are changed to allocate memory from the heap of the native sandbox, instead of the default heap. *Second*, we add security checks in some library functions. For example, the app should not be able to use mprotect to make the code section writable. Another example is the ioctl function in libc, which may be misused to send commands to the key Android services through the binder interface (/dev/binder), bypassing AppCage’s policy check. *Third*, we relocate the thread local storage (TLS) to the native sandbox. Particularly, we allocate a special region inside the sandbox for TLS and change the __get_tls function accordingly. *Last*, callback functions may pose a problem for AppCage. For example, the app can register a callback function to the qsort function in libc. When qsort calls this function, a segmentation fault will be raised because the function is a part of the original code section, which is non-executable (Section 3.3). To address this, we register a signal handler to capture segmentation faults caused by those functions. In the signal handler, we lookup the translated callback function and dispatch to it.

**Indirect Branch Trampolines:** the indirect branch trampoline is fairly involved. It first saves the scratch registers and the status register, and retrieves the branch target from the stack or the register. The branch target (in the original code section) is then converted to the translated pc. At last, the trampoline restores the registers and branches to the target pc. Figure 6 shows a concrete example of the indirect branch trampoline. The trampoline has a non-trivial design because we cannot simply spill the scratch registers to the stack. For example, if the indirect branch instruction calls a function, some parameters may be saved on the stack. Changing the stack passes wrong values to those parameters. If the indirect branch instruction returns from a function, it could return to a wrong location, even out of the sandbox. The trampoline in Figure 6 guarantees the stack is not changed before branching.

Since the indirect branch trampoline is in the native sandbox, we need to prevent it from being misused by the native code. Binary rewriting guarantees that the app’s native code cannot jump to the middle of a trampoline. This is because branches in the translated code can only target addresses in the mapping table of the original and translated pcs, and only the beginning of the trampoline is in this table. However, the native code may achieve the same goal by leveraging the indirect branches in the system libraries since they can target any bundles. To this end, we put the bkpt instruction at the code boundaries of the trampoline, and use b instruction to skip bkpt inside the trampoline [48]. Any attempts to jump into the trampoline from the NaCl-compiled system libraries will be captured because they can only target those boundaries (the bkpt instructions in the trampoline), a security rule enforced by the NaCl compiler.

### 4.3 Native Sandbox Optimizations

To reduce performance overhead, we apply several optimization techniques to the native sandbox.

**Redundant Check Removal:** when translating memory write instructions, we can analyze the register usage in an instruction bundle and remove redundant safety checks. For example, we only need to apply the confinement instructions to the first memory write instruction in the following case:

```
3124: str r3, [r4, #8]  
3128: str r3, [r4, #28]  
/* native sandbox: [9x20000000, 0x45FFFEFF] */  
4469: bic r4, r4, #0xa8000000  
4464: orr r5, r4, #0xa8000000  
4468: str r5, [r4, #8]  
446c: str r5, [r4, #28]  
```

**Calls of System Library Functions:** compilers use the PLT/GOT structure to support dynamic linking. Specifically, calls to an exter-
5.1 Effectiveness of AppCage

We used samples from 15 malware families obtained from the Android Malware Genome Project [52] and online malware [8] repository to evaluate the effectiveness of AppCage. Specifically, we try to trigger their malicious behaviors and check whether our prototype can capture all of them. However, some samples can only be triggered by commands from the defunct remote command and control (C&C) servers. For example, GoldDream waits for the commands to send SMS or make phone calls to premium numbers in the background. To address this, we redirect the traffic to the C&C servers to our local machines at the network level and send commands to these malware samples. Similarly, some malware samples need to be triggered by particular SMS messages. Table 2 lists these samples and the attempted malicious operations by them. AppCage successfully captures all the attempts by them to leak private information and perform dangerous operations. In the following, we present the details of the experiment with the Gone60 malware.

Gone60 was first discovered from the official Android Market in September 2011. When executed, it immediately reads the call logs, contacts and SMS messages in the phone and sends them to the remote server. We sideload this app on our test phone, AppCage intercepts the installation and installs a confined version of it. When the app is started, AppCage detects its attempts to read the sensitive information. Since we have not specified any policy for this app yet, AppCage prompts us to choose whether to allow or deny the access. Figure 7(a) shows the prompt generated by AppCage regarding Gone60’s access to the contacts. Figure 7(b) shows a similar prompt when FakePlayer tries to send SMS to a premium-rated number.

In addition to malware, we also experimented with some benign but invasive apps. These apps are not malicious, but may aggressively access the private information, say, for targeted ads. In our evaluation, AppCage can detect all the accesses and provide users an option to block them. For example, the BestBuy app requests the permissions to access location, send SMS, and make phone calls. The user can leverage AppCage to disallow this app to send SMS but allow it to access location, e.g., to find the local BestBuy stores.

5.2 Security Analysis

In this paper, we assume a threat model in which third-party apps are untrusted or even malicious. In this section, we explore possible attacks to bypass or subvert AppCage and the countermeasures built into our system.

Java Obfuscation: the bytecode of Android apps are often obfuscated. Java-based obfuscation does not affect the effectiveness of our system because dex sandbox is implemented in Dalvik VM unreachable to the bytecode (tamper-resistance), and the app still needs to call the interposed APIs to get private information. For example, Java reflection is often used to hide the actual framework APIs called by the app. AppCage can interpose this behavior because the function call will eventually be dispatched through the method structure in Dalvik VM, where AppCage places its hook.

Dynamic Bytecode: Android apps can dynamically load external bytecode for execution. For example, they can download the bytecode from a remote server and use DexClassLoader or other loaders to execute it. This poses a threat to bytecode-rewriting systems because the new bytecode is not available during rewrite. This attack is not effective against our system for the same reason as Java obfuscation. AppCage is positioned to interpose the dynamically loaded bytecode.

Direct Calls to System Services: In Android, sensitive data are maintained by separate system service daemons and exported to third-party apps through the binder interface. An app can directly access those services through the IBinder interface (in Java) [26] or the raw IPC (in native code). For example, the app can obtain the IBinder interface of the location service to get the current location without using the high-level LocationManager class. AppCage can defeat such attacks by preventing the app from obtaining the IBinder instance of known system services (most legitimate apps
Table 2: AppCage successfully blocks malicious behaviors of samples in 15 malware families

<table>
<thead>
<tr>
<th>Malware</th>
<th>Send SMS</th>
<th>Read/Delete SMS</th>
<th>Phone Calls</th>
<th>Location</th>
<th>Call Logs</th>
<th>Contacts</th>
<th>Internet</th>
<th>IMEI</th>
</tr>
</thead>
<tbody>
<tr>
<td>FakePlayer</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>YZFilter</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GoldDream</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TapSnake</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NickiBot</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DroidKongFu</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BeanBot</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gone60</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmiApps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HippoSMS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zilla</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zane</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spimio</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DroidLive</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gemimi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Some apps used in compatibility test (†: in seconds)

<table>
<thead>
<tr>
<th>App</th>
<th>Size</th>
<th>Package Name</th>
<th>Time†</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job Search</td>
<td>4.7K</td>
<td>com.indeed.android.jobsearch</td>
<td>5.7</td>
</tr>
<tr>
<td>Comics</td>
<td>7.6M</td>
<td>com.iconology.comics</td>
<td>13.5</td>
</tr>
<tr>
<td>chase</td>
<td>1.8M</td>
<td>com.chase.sig.android</td>
<td>10.6</td>
</tr>
<tr>
<td>Domino’s Pizza</td>
<td>7.2M</td>
<td>com.domnosopizza</td>
<td>18.6</td>
</tr>
<tr>
<td>Early Detection Plan</td>
<td>4.6M</td>
<td>com.ndc.edp</td>
<td>13.7</td>
</tr>
<tr>
<td>Super-Bright LED Flashlight</td>
<td>1.5M</td>
<td>com.surpx.ledflashlight.panel</td>
<td>4.8</td>
</tr>
<tr>
<td>Ebay</td>
<td>9.6M</td>
<td>com.ebay.mobile</td>
<td>37.9</td>
</tr>
<tr>
<td>The Weather Channel</td>
<td>7.2M</td>
<td>com.weather.Weather</td>
<td>27.9</td>
</tr>
<tr>
<td>Bug Rush Free</td>
<td>7.1M</td>
<td>com.fourpixels/td</td>
<td>22.5</td>
</tr>
<tr>
<td>Solitaire</td>
<td>8.7M</td>
<td>com.mobilityware.solitaire</td>
<td>18.6</td>
</tr>
<tr>
<td>Average</td>
<td>5.7M</td>
<td>-</td>
<td>17.3</td>
</tr>
</tbody>
</table>

Table 4: Code size increase (†: sizes in kilobytes before and after instrumentation ‡: percentage of code size increased. *: percentage of padding. $: time in seconds.)

<table>
<thead>
<tr>
<th>App</th>
<th>Native Library</th>
<th>Size†</th>
<th>Size‡</th>
<th>% Inc.‡</th>
<th>% Pad*</th>
<th>Time$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ebay</td>
<td>libredlaser.so</td>
<td>529.2</td>
<td>709.7</td>
<td>34.1</td>
<td>15.9</td>
<td>12.2</td>
</tr>
<tr>
<td>AngryBirds</td>
<td>libangrybirds.so</td>
<td>1,283.4</td>
<td>1,639.8</td>
<td>27.8</td>
<td>16.6</td>
<td>27.7</td>
</tr>
<tr>
<td>MiBench</td>
<td>libti.ffmpeg</td>
<td>213.7</td>
<td>283.5</td>
<td>32.3</td>
<td>12.5</td>
<td>5.4</td>
</tr>
<tr>
<td></td>
<td>liblame.so</td>
<td>260.1</td>
<td>347.7</td>
<td>33.7</td>
<td>15.8</td>
<td>5.7</td>
</tr>
<tr>
<td></td>
<td>liblame.so</td>
<td>115.7</td>
<td>147.5</td>
<td>27.5</td>
<td>7.9</td>
<td>2.6</td>
</tr>
<tr>
<td></td>
<td>liblame.so</td>
<td>195.4</td>
<td>257.0</td>
<td>31.3</td>
<td>10.1</td>
<td>5.2</td>
</tr>
<tr>
<td>Nbench</td>
<td>libbench.so</td>
<td>82.2</td>
<td>103.3</td>
<td>29.2</td>
<td>13.1</td>
<td>2.2</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>5.7M</td>
<td>7.1M</td>
<td>20.9</td>
<td>13.1</td>
<td>8.7</td>
</tr>
</tbody>
</table>

5.3 Compatibility of AppCage

AppCage may cause compatibility issues for two reasons: first, some apps may not be accommodating to constraints imposed by AppCage’s sandboxes, particularly the native sandbox. For example, AppCage disallows the app’s native code to directly issue a system call. Meanwhile, dex sandbox should pose few problems because it is, in essential, a set of hooks. Overall, we expect this category of compatibility issues not to be serious because currently not many apps contain native code that has legitimate reasons to directly issue system calls. These system calls usually go through the linked system library (libc for example). In fact, we do not find a case during our evaluation that legitimate user native libraries issue system calls directly. Second, our prototype permission manager only supports three coarse responses: allow, deny, and prompt-to-user. It is known that many apps will not fail gracefully when permissions are removed [25, 27]. Issues in this category can generally be remedied by returning mock results, such as a mock location, instead of an error [27, 55]. However, resources such as network are hard to reconcile this way.

To test the compatibility of our current prototype, we downloaded 50 popular apps from the Google Play store. Some of the apps we tested are shown in Table 3. We first evaluate the compatibility issues caused by the sandboxes by configuring permission manager to always return allow. All the apps can run under AppCage and we have not met any glitches even with exhaustion.

Nevertheless, we still need to sandbox native code. Otherwise, malicious apps can easily subvert our system using a native library.
tive interactions with the app (the result may not be definitive because we may have missed some code paths). We then evaluate the worst case of compatibility by configuring permission manager to always return deny. Under this stringent setting, some apps can only function partially but can still run. Only two apps crashed during our evaluation: Bug Bush Free and Super-Bright LED Flashlight. They immediately crashed when the access to the network and IMEI was denied, respectively. As aforementioned, the compatibility in this case can be improved by returning mock results if and when applicable.

We also evaluate the compatibility of the dex sandbox on different Android versions. We use five different Android versions, i.e., 2.3.6, 4.0.4, 4.1.2, 4.2.2, 4.3 in our test. Our experiments show that the dex sandbox can tolerate the changes of the Dalvik VM in different versions. We only need to slightly change the offset of the Dalvik VM data structure that needs to be hooked. This is implemented by maintaining a mapping table between the version numbers and the offsets of the Dalvik VM data structure that the dex sandbox are interested. Fortunately, this is a one time effort and the number of interested data structures is small (less than 10).

5.4 Performance Evaluation

We also evaluated the overhead introduced by AppCage including its impact to the code size, the installation time and the performance overhead.

**Code Size:** for each target app, AppCage generates a wrapper app that consists of the original app and other components to set up the sandboxes. This adds about 50KB of size to the original app (33KB for the stubs and 18KB for the library). Considering the average size of apps we tested is 5.7MB, the code size increase is less than 1%. There is additional code size increase due to the binary instrumentation for apps with native libraries. Table 4 shows the effect of binary instrumentation on the native code of two Android apps (Ebay and Angry Birds) and two benchmarks (81Bench and 81bench). The average increase is around 30%, and 13.1% of it can be contributed to the padding instructions to align code (Section 4).

**Installation Time:** AppCage will also increase installation time. For example, it needs to generate a wrapper app and sign it. Table 3 shows that the increase to the installation time is about 17 seconds for apps without native code. Moreover, binary rewriting could be time-consuming for large libraries. For example, AppCage spends about 27 seconds to rewrite the 1.2MB native code of Angry Birds (Table 4). Considering that native libraries are normally small and the binary rewriting is performed only once during installation, this increased installation time is acceptable, but maybe frustrating, in the practical deployment.

**Performance Overhead:** We also evaluated the run-time performance overhead introduced by AppCage. For bytecode, AppCage interposes its calls to some framework APIs. We used CaffeineMark, the standard Java benchmark to measure this overhead. The results in Figure 8 show that AppCage’s dex sandbox only introduces negligible performance overhead to the bytecode. We obtained the similar results with an evaluation app that intensively invokes the confined framework APIs. Moreover, AppCage needs to communicate with permission manager (see Figure 2) to retrieve the policies. To measure the overhead introduced by this operation, we develop an app which continuously retrieves the policy for 10,000 times. It costs about 3.9 seconds in total for this app to complete. That is, the average time for each policy retrieval is about 0.39 millisecond. We believe this time introduced by AppCage is negligible that users cannot actually perceive it.

For native code, AppCage adds instructions to confine memory write and branches. To measure the overhead of native sandbox, we used Nbench [7], a computation-intensive benchmark of the CPU, FPU, and memory system performance. The average overhead of native sandbox is about 10.7% (Figure 9). The string sort benchmark has higher performance overhead because it heavily uses the memcpy function in libc to move large blocks of data. libc is built by a modified NaCl compiler and is the main source of the overhead for this benchmark. This is confirmed by the fact that the overhead will reduce dramatically if the benchmark is linked to a plain libc. This overhead can be reduced (to that of the original NaCl for ARM) if we position the sandbox in the same way as NaCl. This is not feasible unless we can change the default Android loader, a design we avoid for easy deployment. Similar to NaCl, there are the tests in Nbench perform better than the native execution. This is probably due to the caching effects of the code bundles [48].

6. DISCUSSION

First, compared to the systems that target the same problem [5, 14–16, 27, 33, 45, 55], AppCage is a more comprehensive and secure solution with its hybrid sandboxes. The ideal solution is for Google to officially support adjustment of permissions after installation [5]. However, there are two major issues that encumber this solution. First, Android is heavily fragmented with many versions and incompatible customization by major vendors [46]. It is not clear whether and when such update will be deployed to most of the users. In contrast, solutions such as AppCage can be immediately deployed. More importantly, as an official solution, the user would expect it to be compatible with all the apps in Google Play. It is a daunting task to update those millions of apps. It is questionable whether Google will ever deploy such a system (the experimental official permission manager has since been removed in a recent update to Android KitKat 4.4.2 [41]). Users likely have higher tolerance to the (inevitable) incompatibility of third-party solutions.

Second, AppCage prompts the user to choose to allow or deny the app’s access to sensitive resources if no policy has been set (Figure 7). Compared to the prompt during installation, this innocent prompt is more effective and users are less likely to ignore it. Particularly, the research by Felt et al. shows that only 12% of
the participants pay attention to the permission request at installation time [20], while 85% of them denied the location request for at least one app [21]. The system can also be extended to support community-based policy definition [35]. Moreover, the signature of the installed wrapper app is different from the original one, which may break the automatic update feature provided by the Google Play. Our system can monitor the version history of the apps on the Google Play and generate the wrapper app if a newer version is available.

Third, our current prototype only supports ARM, the dominate architecture for smartphones and tablets. Intel’s x86 is an emerging architecture for mobile platforms. Our native sandbox can be extended to support x86 using similar techniques. In addition, our prototype does not support self-modify code. We leave it as a future work. Moreover, the indirect branch from the instruction to the target pc on the stack and branches to it using a pop instruction (Figure 6). This introduces a race condition where the target pc could be manipulated by another thread. The race condition can be avoided with an extra scratch register, and instructions to spill and restore the register. It would further complicate the design of the trampoline. However, the window of vulnerability is only four instructions, and the chance of successfully attacking it in practice is low. In fact, this race condition also affects other similar systems [32, 49].

At last, since Android version 5.0, it uses a new runtime (ART) which leverages the ahead of time optimization to convert dex bytecode to a native binary. AppCage could be extended to support the ART runtime since this runtime still has the corresponding data structures to represent the Java classes and methods as the Dalvik VM (Figure 1). Accordingly, our system could take similar methods to hook these data structures in the ART runtime (libart.so). We take the ART support as one of the future work.

7. RELATED WORK

Android App Security: the first category of related work includes systems that expose privacy risks of third-party apps and systems to confine those apps. For example, researchers discover that private information could be leaked by benign apps [19], ad libraries [23], vulnerable apps [43], and malicious apps [52]. To mitigate this threat, researchers have proposed solutions to detect malicious or privacy-leaking apps. DroidRanger [54] and RiskRanker [24] are two systems to detect malicious apps on the official and alternative Android markets. Those systems can detect malicious app behaviors before they are installed on the user devices. However, they usually suffer from false negatives and false positives. On contrast, the interposition of AppCage cannot be bypassed by third-party apps.

There are also systems that extend the Android framework to provide fine-grained control of third-party apps at run-time [14–16, 27, 33, 55]. For example, AppFence [27] and TISSA [55] can return mock results of the sensitive resources such as the location. User-driven access control is a promising solution to provide in-context and non-destructive permission granting [36]. While these systems may solve the problem in theory, the requirement to modify the Android framework significantly limits their practical deployment. In contrast, our system does not have such requirement and can be readily deployed. From another perspective, researchers have proposed to monitor and confine third-party apps in the user space with bytecode rewriting [17, 18, 28] or native library interposing [47] (Similar system also exists on other platforms [31]). For instance, AppGuard [13] is a closely-related system that instruments the target app and detours security-relevant methods to their guards functions through virtual machine internal data structure manipulation, a similar design as our dex sandbox (Section 3.2). Unfortunately, like other systems, it could be subverted by leveraging the native code. The native sandbox in AppCage is specifically designed to prevent this attack.

Software Fault Isolation AppCage leverages the software fault isolation (SFI) technology to sandbox native code. SFI has been widely researched and deployed [22, 29, 32, 34, 38, 42, 48, 50, 51]. Most of these systems target the x86 architecture [22, 29, 32, 34, 48, 50]. AppCage is designed for the ARM architecture, which has a different set of challenges (Section 3.3). ARMor [51] is a system providing SFI for the ARM architecture. However, it does not support dynamic linking, and has a high performance overhead unsuitable to our system. Native Client for ARM [38] provides a customized compiler to generate confined ARM binaries. It thus requires source code access which is not available for the app’s native code. ARMLock [53] implements an efficient fault isolation solution. However it requires the support from kernel space and thus cannot be used without the change to the phone’s firmware.

AppCage’s native sandbox uses static binary translation to enforce the same proven rules of NaCl.

Robusta [39] and Arabica [40] are two closely related systems. They leverage Native Client and the JVMTI (JVM Tool Interface) to sandbox the native libraries of JVM, respectively. They have different assumptions than AppCage: Robusta requires to recompile the native code of the libraries, and Arabica needs the support of JVMTI that is unavailable in Dalvik VM. Klinkoff et al. propose a SFI mechanism to protect managed code and the.NET runtime from the unmanaged code (or native code) [30]. They isolate unmanaged code in a separate process. Similarly, the NativeGuard [41] leverages the process boundary to isolate untrusted native libraries. AppCage takes a different design by isolating native code in the same process. While process-based isolation could be used in our system, one disadvantage is that every JNI call is transformed to a RPC call cross the process boundary, which is expensive and thus infeasible for our use cases.

8. CONCLUSION

We have presented the design, implementation, and evaluation of AppCage, a system to interpose and regulate third-party Android apps with hybrid user-level sandboxes, dex sandbox and native sandbox. Together, they enable AppCage to securely interpose the app’s access to key APIs and services. We have implemented a prototype of AppCage. Our evaluation shows that AppCage can successfully detect and block the attempts to leak private data or perform dangerous operations by malware and invasive apps, and it also has an acceptable overhead, especially for apps without native code.

Acknowledgements The authors would like to thank the anonymous reviewers for their insightful comments that helped improve the presentation of this paper. This work was supported in part by the US National Science Foundation (NSF) under Grants 0855036 and 0952640. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

9. REFERENCES