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 interposing [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, Aurasium [47] interposes 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

[*]Part of the work was done when the first author was an intern at Qihoo 360.

ASIA CCS'15, April 14–17, 2015, Singapore..

ACM 978-1-4503-3245-3/15/04 ...$15.00.
http://dx.doi.org/10.1145/2714576.2714598.

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.

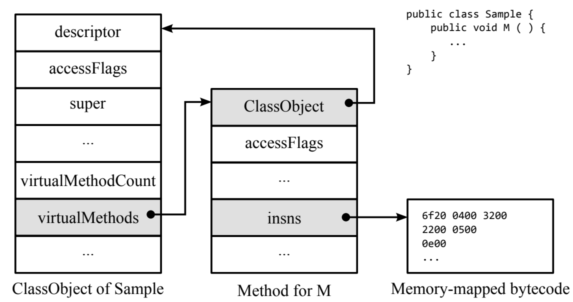


Figure 1: `ClassObject` and `Method` in Dalvik

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 `dlopen` 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 `__android_log_print` function in `liblog.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 [9] 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

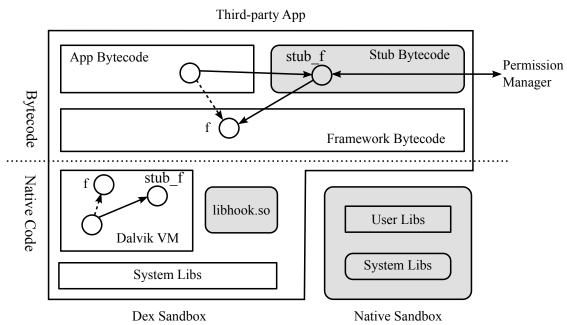


Figure 2: System architecture of AppCage

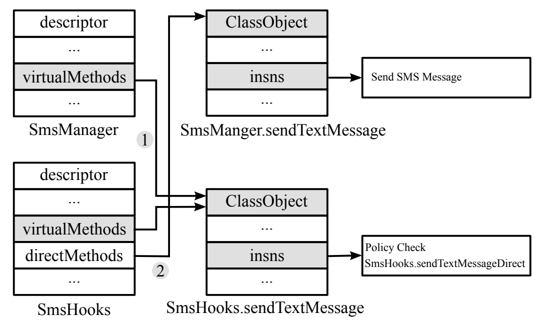


Figure 3: Dalvik hooking of `SmsManager.sendTextMessage`

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 middleware 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 `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 `libc.so` or `libm.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 `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 (`/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 `sendTextMessage` method, possibly to premium numbers. To interpose this method, AppCage loads `SmsHooks`, the stub class for `SmsManager`, into the app, and manipulates the pointers in their corresponding `ClassObjects` so that `SmsManager.sendTextMessage` points to `SmsHooks.sendTextMessage` and the pointer to the original method `SmsManager.sendTextMessage` is stored in `SmsHooks`'s array of direct methods (arrow 1 and 2 in Figure 3, respectively). As such, the app will be redirected to `SmsHooks.sendTextMessage` when it tries to send a text message and subject to policy check. If the operation is allowed, `SmsHooks.sendTextMessage` calls the original method, which can be found in `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

[1] Most 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) [42] 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 [3] or the downloaded ones. Our binary rewriter 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 `orr`) 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 rewriter needs to handle 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 rewriter. 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 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,

```
ldr r1, [pc, #284]
```

We accordingly treat the targets of `pc`-relative load instructions as constants. However, the heuristic may treat the real code as constants. This is handled at run-time by rewriting those missed instructions on demand. The second challenge is that a binary can consist of functions in the ARM and THUMB/THUMB-2 states. These states have different instruction length and encoding. AppCage needs to identify the instruction state to correctly disassemble them. This is solved as follows: first, branch instructions such as `B`, `BL` with an immediate offset, `BX`, and `BLX` specify the state of the target instruction in the encoding. AppCage decodes the targets accordingly using the target instruction encoding. Second, we use the context of a gap to infer its state. For example, if the instructions before and after a gap are in the THUMB state, the gap likely is also in that state. Third, if the previous heuristics fail to decide the instruction state, we assume the instructions are in the ARM state and start to disassemble them. We will restart the process in the THUMB state if errors are encountered, for example, if there are invalid instructions or the data flow in a peephole is inconsistent [44].

Note that the instruction mode which is determined by the heuristic, e.g., the branch target of an indirect branch instruction, may be wrong during statically disassembling. AppCage records the instruction mode for such instructions and compare it with the actual instruction mode at run-time. If there is a mismatch, then these instructions will be re-disassembled and instrumented at run-time.

Native Code Instrumentation: To rewrite the app's native code, AppCage breaks the disassembled instructions into basic blocks and then instruments each of them according to the instruction types. Figure 4 shows a concrete example for each such case.

a.) Memory write instructions: AppCage requires that memory write instructions can only target addresses within the sandbox. To this end, we position the native sandbox at an address that is 2^n-byte aligned with a length of 2^n. Consequently, we can confine memory writes by fixing the top $32 - n$ bits of the target address to that of the sandbox. For example, if the sandbox is located at the range of `[0x40000000, 0x5FFFFFFF]`, we can use the following two instructions to confine memory writes (assuming `r0` is the base register):

```
bic r0, r0, #0xa0000000  /* clear bits 31,29 */
orr r0, r0, #0x40000000  /* set bits 30 */
```

Moreover, we put two guard pages around the sandbox, one at each end, to accommodate addressing modes with an immediate offset, which is always less than 4,096 in the ARM architecture (see case 1 in Figure 4 for an example). If the target address is calculated from several registers, we first save the address to a scratch register (spill one if necessary) and apply the same instructions as above to confine it. The memory write instruction is then patched to use the scratch register as the target (case 2 in Figure 4).

AppCage normally instruments every memory write instruction. One exception is those instructions with `sp` as the base register and an immediate offset. They are frequently used to update local variables. To reduce overhead, we do not instrument this type of instruction (case 3 in Figure 4), but instead instrument all instructions that update the `sp` register to guarantee that `sp` is within the sandbox (case 4 in Figure 4.)

b.) Branch instructions: the second category of instructions to rewrite consists of branch instructions, used for jumps and calls. Branch instructions can address the target with an immediate offset (direct branch) or with registers (indirect branch). Both types

(1) memory write: immediate offset

```
before:
 e2b8: e5840008   str  r0, [r4, #8]

after:
 f8b0: e3c4420e   bic  r4, r4,  #0xe0000000
 f8b4: e5840008   str  r0, [r4, #8]
```

Before writing data into memory,we mask off some bits of the target address. Since the immediate offset is within the 4K range, we can confine it by adding a guard page before and after the native sandbox.

(2) memory write: register offset

```
before:
 f054: e784e10c   str  lr, [r4, ip, lsl#2]

after:
 1004: e084310c   add  r3, r4, ip, lsl#2
 1008: e3c3320e   bic  r3, r3, #0xe0000000
 100c: e583e000   str  lr, [r3]
```

For instructions using the register offset,we allocate a scratch register (r3 in this case) first to store the target address of memory write, and then we mask off the related address bits. The original instruction is replaced by one using the scratch register.

(3) memory write: SP as the base register with immediate offset

```
before:
 114a0: e58d11d0   str  r1, [sp, #464]

after:
 12f5c: e58d11d0   str  r1, [sp, #464]
```

For instructions using the SP register as the base with immediate offset, we do not instrument them but instead ensure that SP is always within the sandbox by rewriting the instructions that update SP.

(4) computation : update SP

```
before:
 1c284: e084d005   add  sp, r4, r5

after:
 10200: e084d005   add  sp, r4, r5
 10204: e3cdd20e   bic  sp, sp, #0xe0000000
```

We instrument instructions that update SP to make sure SP is always within the native sandbox.

(5) direct branch

```
before:
  dca4: 1affffbf   bne  0xdba8

after:
  f068: 1affffb3   bne  0xef3c
```

We need to patch the target of the direct branch instructions to its corresponding translated target.

(6) indirect branch: branch address on the stack

```
before:
 164f8: e8bd8002   pop  {r1, pc}

after:
 18e10: e8bd0002   pop  {r1}
 18e14: e59ff718   ldr  pc, [pc, #1816] ;0x19534
 ...
 19534: 0001aae4
```

The registers other than PC will be popped first, and then it jumps to the indirect branch trampoline at offset 0x1aae4 (this address is patched by AppCage according to whether the PC is on the stack or in the reigster.) The trampoline fecthes the branch address from the stack and check whether it is valid. If so, it will jump to the address for execution.

(7) indirect branch: branch address in a register

```
before:
 164bc: e12fff1e   bx  lr

after:
 18dcc: e59ff75c   ldr  pc, [pc, #1884] ;0x19530
 ...
 19530: 0001aa88
```

Similar to case 6, it jumps to the indirect branch trampoline. This trampoline validates the branch target in the register and jumps to it for execution, if valid.

(8) using PC as the source register in non-branch instructions

```
before:
 14e80: e08f1001   add   r1, pc, r1

after:
 1748c: e52d3004   push  {r3}
 17490: e59f331c   ldr   r3, [pc, #796] ;0x177b4
 17494: e0831001   add   r1, r3, r1
 17498: e49d3004   pop   {r3}
 ...
 177b4: 00014e88
```

We allocate a scratch register (r3 in this case), spill it when necessary, and load the original PC stored at address 0x177b4 into it. The original instruction is transformed to use this scratch register.

Figure 4: Instrumentation of native code by AppCage
(for simplicity, we assume the native sandbox is at the [0x0, 0x1FFFFFFF] address space range.)

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 `0x1748c` 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    -->    1748c: 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., `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., `libc.so`). For performance reasons, AppCage uses the NaCl compiler for ARM [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 `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):

```
bic r0, r0, 0xc000000f   /*clear top 2 and the last 4 bits*/
```

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

```
tst r0, #0xc0000000      /* within the first 1GB? */
streq r1, [r0, #12]      /* store to memory if so*/
```

Because native sandbox is not guaranteed to start at address 0, we modify the NaCl compiler to emit `bic` and `orr` instructions to confine sensitive instructions, similar to the binary rewriter. 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 `dlopen` and `dlsym` functions in Dalvik VM. The `dlopen` hook allows us to rewrite native libraries unseen during the installation (e.g., a newly downloaded library). Moreover, when `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 `JNIEnv` structure. To intercept those calls, we replace the functions in `JNIEnv` with their stubs in the sandbox. The stub switches the execution environment to that of Dalvik VM and then calls the actual `JNIEnv` functions [39]. The direct reference to Java heap is processed through the copy-in and copy-out mechanism [39].

Table 1: Confined operations by the current prototype

Operation	Permissions
Send SMS	SEND_SMS
Read SMS	READ_SMS
Delete SMS	WRITE_SMS
Phone call	CALL_PHONE
Read contacts	READ_CONTACTS
Write contacts	WRITE_CONTACTS
Read call logs	READ_CALL_LOG
Get location	ACCESS_FINE_LOCATION ACCESS_COARSE_LOCATION
Network	INTERNET
Read IMEI	READ_PHONE_STATE

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 sandboxes. 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 [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 `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 `bic` and `orr` 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

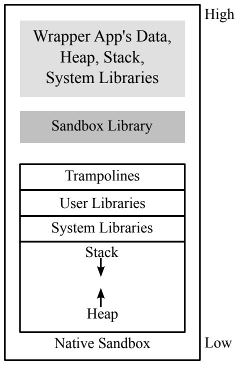


Figure 5: Memory layout of the native sandbox

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 $0x4ae8$ 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).

```
4ae8:   ebfff8de    bl  2e68 <__android_log_print@plt>
-->
22a4:   e320f000    nop {0}
22a8:   e320f000    nop {0}
22ac:   ebfff772    bl  0x7c <__android_log_print@plt>
        /*lr = 0x22b0 */
```

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

```
1a544: e24dd004 sub   sp, sp, #4 ; 0x4
1a548: e92d400f push  {r0, r1, r2, r3, lr}
1a54c: ea000000 b     0x1a554
1a550: e1200070 bkpt  0x0000
1a554: e10f2000 mrs   r2, CPSR //move status to r2
1a558: e92d0004 push  {r2} //save status register on stack
1a55c: ea000000 b     0x1a564
1a560: e1200070 bkpt  0x0000
1a564: e1a03000 mov   r3, r0 //move jump target (r0) to r3
1a568: e59f25a8 ldr   r2, [pc, #1448] ; 0x1ab18
1a56c: ea000000 b     0x1a574
1a570: e1200070 bkpt  0x0000
1a574: e12fff32 blx   r2 // retrieve the pc
1a578: e8bd0004 pop   {r2} //pop status register to r2
1a57c: ea000000 b     0x1a584
1a580: e1200070 bkpt  0x0000
1a584: e129f002 msr   CPSR_fc, r2 //restore status register
1a588: e58d0014 str   r0, [sp, #20] //put new PC on stack
1a58c: ea000000 b     0x1a594
1a590: e1200070 bkpt  0x0000
1a594: e8bd400f pop   {r0, r1, r2, r3, lr}
1a598: e49df004 pop   {pc} //jump to new PC
..
/* patch it to the function to get pc from stack or register*/
1ab18: 00000000
```

Figure 6: An example of indirect branch trampoline

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 `pc`s, 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.

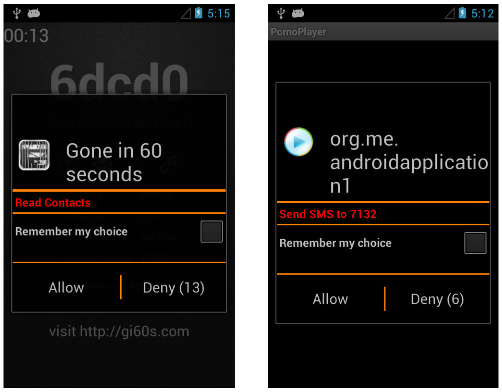
4.3 Native Sandbox Optimizations

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:

```
3324: str r3, [r4, #8]
3328: str r3, [r4, #20]
-->
/* native sandbox: [0x40000000, 0x5FFFFFFF] */
4400: bic r4, r4, #0xa0000000
4404: orr r4, r4, #0x40000000
4408: str r3, [r4, #8]
440c: str r3, [r4, #20]
```

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



(a) Gone60 malware: reading contacts

(b) FakePlayer: sending SMS to a premium number

Figure 7: AppCage detects malicious behaviors

nal function are patched to the function's stub in the PLT section, which indirectly branches to the function address in the associated GOT entry. The GOT entry is normally filled in by the dynamic linker after it resolves the address of the external function. In AppCage, the GOT entries are patched by our own custom loader and are guaranteed to be in the sandbox. Therefore, we do not need to rewrite the indirect branch instructions inside the PLT section (the GOT section is set to read-only after initialization.)

Function Return Optimization: ARM binaries often use the `pop` instruction to return from a function. Our binary rewriter treats the `pop` instruction using `pc` as the target operand as an indirect branch, which has a relatively heavy instrumentation (Section 4.2). Considering function return is highly frequent, we need to avoid the time-consuming indirect branch trampoline. To this end, we fetch the branch target into the `lr` register and add instructions to confine the value of `lr` before jumping to it. In particular, we mask off the last four bits of the return address to align it with the NaCl instruction bundle. The following instructions illustrate this optimization:

```
pop {r3, r4, pc}
-->
pop {r3, r4]
pop {lr}
/* native sandbox: [0x40000000, 0x5FFFFFFF] */
bic lr, lr, #0xa000000f
orr lr, lr, #0x40000000
bx  lr
```

Address Cache in Indirect Branch Trampolines: indirect branch trampolines need to look up the mapping between old `pc` and translated `pc`. Before looking up the table, we first search a small cache of the previously converted `pcs` (8 entries). This simple optimization has a high hit rate (90% for the `nbench` [7] benchmark.)

5. EVALUATION

In this section, we evaluate the effectiveness, compatibility, and performance of our prototype. The experiments are based on a Google Nexus S phone running Android 4.1.2 (build number JZ054K).

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

Malware	Send SMS	Read/Delete SMS	Phone Calls	Location	Call Logs	Contacts	Internet	IMEI
FakePlayer	√							
YZHC	√	√						
GoldDream	√		√					
TapSnake				√				
NickiBot		√		√	√	√	√	√
DroidKungFu							√	√
BeanBot	√		√				√	√
Gone60		√			√	√	√	
SndApps							√	
HippoSMS	√	√					√	√
Zitmo		√					√	√
Zsone	√	√						
Spitmo		√					√	
DroidLive	√		√				√	√
Geinimi	√		√	√		√	√	√

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

App	Size	Package Name	Time†
Job Search	474K	com.indeed.android.jobsearch	5.7
Comics	3.7M	com.iconology.comics	13.5
chase	1.8M	com.chase.sig.android	10.6
Domino's Pizza	7.2M	com.dominospizza	18.6
Early Detection Plan	4.6M	com.nbcf.edp	13.7
Super-Bright LED Flashlight	1.5M	com.surpax.ledflashlight.panel	4.8
Ebay	9.6M	com.ebay.mobile	37.9
The Weather Channel	7.2M	com.weather.Weather	27.9
Bug Rush Free	13M	com.fourpixels.td	22.2
Solitaire	8.7M	com.mobilityware.solitaire	18.6
Average	5.7 M	-	17.3

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

App	Native Library	Size†	Size†	%Inc.‡	%Pad∗	Time♯
Ebay	libredlaser.so	529.2	709.7	34.1	15.9	12.2
AngryBirds	libangrybirds.so	1,283.4	1,639.8	27.8	16.6	27.7
MiBench	libjpeg6a.so	213.7	283.5	32.3	12.5	5.4
	liblame.so	260.1	347.7	33.7	15.8	5.7
	libmad.so	115.7	147.5	27.5	7.9	2.6
	libtiff.so	195.4	257.0	31.5	10.1	5.2
Nbench	libnbench.so	82.2	103.3	29.2	13.1	2.2
Average	-	-	-	30.9	13.1	8.7

do not do this anyway). Moreover, the app can use native code to open the `binder` device and send commands to the system services using the `ioctl` system call. AppCage adds security checks in the `ioctl` function of `libc` to prevent it, and it disallows the native code to issue system calls directly (Section 3.3). Note that we only block the `ioctl` system call directly issued from user native libraries or the (confined) system libraries linked with user native libraries. The original communication channels to remote system services through the binder interface remain unchanged since they are going through original system libraries.

Tampering with Dalvik VM: the app may tamper with Dalvik VM using native code since they share the same address space [1]. For example, it can remove all the hooks of AppCage in Dalvik VM to disable the policy check. This attack can be foiled by our system because the app's native code is sandboxed and cannot write memory out of the sandbox. In addition, it cannot escape from the sandbox to execute untranslated code.

Attacking Native Sandbox: the app may also try to attack the native sandbox. For example, it may try to load unsandboxed native code or break out of the sandbox. Our system design is secure against those attacks. First, AppCage hooks into the Android's dynamic loader. It intercepts any attempt to load a native binary and ensures that all the loaded user binaries are instrumented and sandboxed. Second, even though AppCage uses both a binary rewriter and the customized NaCl compiler to instrument native code, they enforce the same set of rules as the Native Client, which has been proved to be reliable and secure [38,48], despite a few fixed implementation issues.

Synthetic Attack: to further evaluate the security of the native sandbox, we create several synthetic attacks that violate the native sandbox rules. The first one is to change the loaded app bytecode at run-time, i.e., by writing to the memory mapped bytecode. Our system can capture such an attempt since this attack invokes the mmap system calls (Section 3.3) to remap the memory space containing the bytecode as readable and writeable. The second one is to manipulate the Dalvik internal data structure to change the control flow of the app code using the native library at run-time. For instance, it manipulate the method pointer inside the Dalvik VM to invoke the sensitive APIs. Our system blocks this attack since it writes to the memory space outside the native sandbox.

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 [2]. 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 exhaus-

[2]Nevertheless, we still need to sandbox native code. Otherwise, malicious apps can easily subvert our system using a native library.

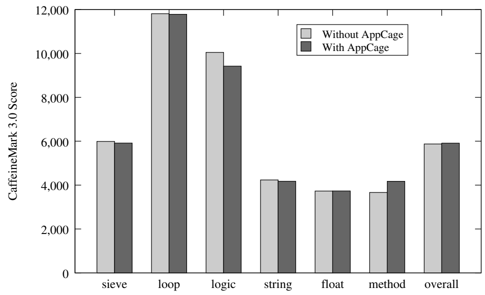


Figure 8: CaffeineMark with and without AppCage

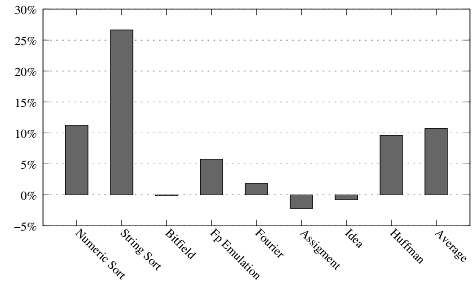


Figure 9: Normalized overhead of the native sandbox

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 Rush 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 sanbox can tolerant 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 (`MiBench` and `Nbench`). 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 `memmove` 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 three 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 securer 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 [4].) 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 *in-context* 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 app, 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 trampoline temporarily stores 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-disruptive 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 source code of the native 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 run-time 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

[1] Android Security Analysis Challenge: Tampering Dalvik Bytecode During Runtime. `https://bluebox.com/technical/android-security-analysis-challenge-tampering-dalvik-bytecode-during-runtime/`.

[2] Binder. `http://developer.android.com/reference/android/os/Binder.html`.

[3] DroidKungFu Malware. `http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu.html`.

[4] Googler: App Ops Was Never Meant For End Users, Used For Internal Testing And Debugging Only. http://www.androidpolice.com/2013/12/11/googler-app-ops-was-never-meant-for-end-users-used-for-internal-testing-and-debugging-only/.

[5] Hidden Permissions Manager Found in Android 4.3, Lets You Set the Rules. http://www.engadget.com/2013/07/26/hidden-permissions-manager-android-4-3/.

[6] JNI Tips. http://developer.android.com/training/articles/perf-jni.html.

[7] Linux/Unix nbench. http://www.tux.org/~mayer/linux/bmark.html.

[8] Mobile Malware Minidump. http://contagiominidump.blogspot.com/.

[9] PLT and GOT - the Key to Code Sharing and Dynamic Libraries. https://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html.

[10] Smartphone OS Market Share, Q2 2014. http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[11] The Java Virtual Machine Specification. http://docs.oracle.com/javase/specs/jvms/se7/html/.

[12] Your Apps Are Watching You. http://online.wsj.com/news/articles/SB10001424052748704368004576027751867039730.

[13] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. AppGuard - Enforcing User Requirements on Android Apps. In *Proceedings of 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2012.

[14] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *Proceedings of the 12th International Workshop on Mobile Computing System and Applications*, 2011.

[15] S. Bugiel, S. Heuser, and A.-R. Sadeghi. mytunes: Semantically linked and user-centric fine-grained privacy control on android. Technical Report TUD-CS-2012-0226, Center for Advanced Security Research Darmstadt, November 2012.

[16] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and Fine-Grained Mandatory Access Control on Android for Diverse Security and Privacy Policies. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[17] B. Davis and H. Chen. RetroSkeleton: Retrofitting Android Apps. In *Proceedings of the 11th International Conference on Mobile Systems, Applications and Services*, 2013.

[18] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen. I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications. In *Proceedings of the IEEE Mobile Security Technology*, 2012.

[19] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[20] A. P. Felt, E. Hay, S. Egelman, A. Haneyy, E. Chin, and D. Wagner. Android Permissions: User Attention, Comprehension, and Behavior. In *Symposium on Usable Privacy and Security*, 2012.

[21] D. Fisher, L. Dorner, and D. Wagner. Short Paper: Location Privacy: User Behavior in the Field. In *Proceedings of the 2nd Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2012.

[22] B. Ford and R. Cox. Vx32: Lightweight User-level Sandboxing on the x86. In *Proceedings of the USENIX 2008 Annual Technical Conference*, 2008.

[23] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe Exposure Analysis of Mobile In-App Advertisements. In *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2012.

[24] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the 10th ACM International Conference on Mobile Systems, Applications and Services*, 2012.

[25] E. Gustafson, K. Kennedy, and H. Chen. Quantifying the Effects of Removing Permissions from Android Applications. In *IEEE Mobile Security Technologies*, 2013.

[26] H. Hao, V. Singh, and W. Du. On the Effectiveness of API-Level Access Control Using Bytecode Rewriting in Android. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security*, 2013.

[27] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These Aren't the Droids You're Looking For": Retroffiting Android to Protect Data from Imperious Applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.

[28] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications . In *Proceedings of 2nd ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2012.

[29] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, 2001.

[30] P. Klinkoff, E. Kirda, C. Kruegel, and G. Vigna. Extending .NET Security to Unmanaged Code. In *International Journal of Information Security*, 2007.

[31] B. Livshits and J. Jung. Automatic Mediation of Privacy-Sensitive Resource Access in Smartphone Applications. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[32] S. McCamant and G. Morrisett. Evaluating SFI for a CISC Architecture. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[33] M. Nauman, S. Khan, M. Alam, and X. Zhang. Extending Android permission model and enforcement with user-deïňĄned runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, 2010.

[34] M. Payer and T. R. Gross. Fine-Grained User-Space Security Through Virtualization. In *Proceedings of the 7th International Conference of Virtual Execution Environments*, 2011.

[35] B. Rashidi, C. Fung, and T. Vu. RecDroid: A Resource Access Permission Control Portal and Recommendation Service for Smartphone Users. In *Proceedings of the ACM MobiCom workshop on Security and privacy in mobile environments*, 2014.

[36] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, and H. J. Wang. User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.

[37] G. Russello, A. B. Jimenez, H. Naderi, and W. van der Mark. FireDroid: Hardening Security in Almost-Stock Android. In *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013.

[38] D. Sehr, R. Muth, C. L. Biffle, V. Khimenko, E. Pasko, B. S. Yee, K. Schimpf, and B. Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[39] J. Siefers, G. Tan, and G. Morrisett. Robusta: Taming the Native Beast of the JVM. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[40] M. Sun and G. Tan. JVM-Portable Sandboxing of Java's Native Libraries. In *Proceedings of the 17th European Symposium on Research in Computer Security*, 2012.

[41] M. Sun and G. Tan. NativeGuard: Protecting Android Applications from Third-Party Native Libraries. In *Proceedings of the 7th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2014.

[42] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-based Fault Isolation. In *Proceedings of the 14th ACM symposium on Operating systems principles*, 1993.

[43] R. Wang, L. Xing, X. Wang, and S. Chen. Unauthorized Origin Crossing on Mobile Platforms: Threats and Mitigation. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, 2013.

[44] X. Wang, C.-C. Pan, P. Liu, and S. Zhu. SigFree: A Signature-free Buffer Overflow Attack Blocker. In *Proceedings of the 15th IEEE Symposium on Security and Privacy*, 2006.

[45] C. Wu, Y. Zhou, K. Patel, Z. Liang, and X. Jiang. AirBag: Boosting Smartphone Resistance to Malware Infection. In *Proceedings of the 21st Network and Distributed System Security Symposium*, 2014.

[46] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The Impact of Vendor Customizations on Android Security. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, 2013.

[47] R. Xu, H. Saidi, and R. Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the 21th USENIX Security Symposium*, 2012.

[48] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009.

[49] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity & randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, 2013.

[50] M. Zhang and R. Sekar. Control-Flow Integrity For COTS Binaries. In *Proceedings of the 22rd USENIX Security Symposium*, 2013.

[51] L. Zhao, G. Li, B. D. Sutter, and J. Regehr. ARMor: Fully Verified Software Fault Isolation. In *Proceedings of the 11th International Conference on Embedded Software*, 2011.

[52] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.

[53] Y. Zhou, X. Wang, Y. Chen, and Z. Wang. ARMlock: Hardware-based Fault Isolation for ARM . In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, 2014.

[54] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the 19th Network and Distributed System Security Symposium*, 2012.

[55] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, 2011.