NaClDroid: Native Code Isolation for Android Applications

Elias Athanasopoulos[1], Vasileios P. Kemerlis[2], Georgios Portokalidis[3], and Angelos D. Keromytis[4]

[1] Vrije Universiteit Amsterdam, The Netherlands
i.a.athanasopoulos@vu.nl
[2] Brown University, Providence, RI, USA
vpk@cs.brown.edu
[3] Stevens Institute of Technology, Hoboken, NJ, USA
gportoka@stevens.edu
[4] Columbia University, New York, NY, USA
angelos@cs.columbia.edu

Abstract. Android apps frequently incorporate third-party libraries that contain native code; this not only facilitates rapid application development and distribution, but also provides new ways to generate revenue. As a matter of fact, one in two apps in Google Play are linked with a library providing ad network services. However, linking applications with third-party code can have severe security implications: malicious libraries written in native code can exfiltrate sensitive information from a running app, or completely modify the execution runtime, since all native code is mapped inside the same address space with the execution environment, namely the Dalvik/ART VM. We propose NaClDroid, a framework that addresses these problems, while still allowing apps to include third-party code. NaClDroid prevents malicious native-code libraries from hijacking Android applications using Software Fault Isolation. More specifically, we place all native code in a Native Client sandbox that prevents unconstrained reads, or writes, inside the process address space. NaClDroid has little overhead; for native code running inside the NaCl sandbox the slowdown is less than 10% on average.

Keywords: SFI, NaCl, Android

1 Introduction

Android is undoubtedly the most rapidly growing platform for mobile devices, with estimates predicting one billion Android-based smartphones shipping in 2017 [12]. App developers have great incentives to release their software without delays, ahead of their competitors; for that reason, they frequently decide to incorporate already available third-party code, typically included in the form of native-code libraries, loaded at run time. For example, ∼50% of apps are linked with a library providing ad services [25], while native-code libraries are heavily used among the most popular applications [32].

Including untrusted, third-party code in an application has severe security consequences, affecting both the developers and the users of apps. *Lookout*, a smartphone-centric security firm, recently identified 32, otherwise legitimate, apps in Google Play that were linked with a malicious ad library [7]. They estimated the number of affected users to be in the range of 2–9 million, and their advice was the following: *"Developers need to pay very close attention to any third-party libraries they include in their applications. Unsafe libraries can put their users and reputation at risk."* [7]

The current design of the Android runtime allows a malicious *native* library, included in an otherwise legitimate app, to: (a) exfiltrate private information from the app; or (b) change the functionality of the app. To address the above, we propose NaClDroid: a framework that provides strong confidentiality and integrity guarantees to the Android execution environment. NaClDroid introduces a sandbox for running native third-party code that has been packaged with an app. This enables us to retain support for native code, while concurrently preventing it from arbitrarily reading process memory, tampering with the Dalvik runtime[5], or directly accessing operating system (OS) interfaces, like system calls. We employ Software Fault Isolation [33] (SFI) and sandbox all native library code using Google's Native Client (NaCl) [27,37]. Therefore, we separate the runtime, Dalvik VM, from the native part of the app, while at the same time permitting the use of native code through JNI. Note that this architectural model is already used by Google Chrome for running untrusted browser extensions [5]; in this work, we extend it to the mobile setting.

We briefly explain how NaClDroid protects Android apps from a malicious library. Consider, a legitimate instant messaging (IM) application; it is written in Java and provided for free. To compensate, it includes a third-party, native-code library to display advertisements that generate revenue for its authors. The third-party library is mapped to the same address space with the rest of the VM executing the app, and can access all app information just by reading memory (e.g., sensitive information like user passwords and discussions). Note that whether encryption is used to transmit data is inconsequential, since the information is available in plaintext in process memory. NaClDroid's sandbox confines the library, so that is can only access its own memory region(s) and data exchanged with the app through JNI, thereby making it impossible to exfiltrate sensitive information.

As a second example, we can look at Facebook's Android application, which updated itself without going through the app store [9], avoiding verification and analysis of the updated app. Google eventually banned Facebook's official app from its app store [3], and the ban was accompanied with a change in the store terms of service that forbids developers from independently updating their apps. However, this policy update is *not* enforced by the platform. NaClDroid prevents the introduction of new functionality outside the app store, since dangerous

[5] Dalvik has evolved to ART, which supports ahead-of-time compilation. The system we present here works with both Dalvik and ART. Since many Android OSes still support Dalvik, we use the term Dalvik to refer to both systems.

system calls that load new code and modify the VM are no longer possible. Additionally, attacks that attempt to hijack the control flow for activating hidden functionality [34] are also prevented, as NaCl only allows code that has been compiled in a certain way (i.e., unvetted control flows are not possible).

Isolating code, using SFI, in Java-based runtime environments has been demonstrated in the past [28,29]. For Android, in particular, there is NativeGuard [30], which performs isolation without heavily modifying the runtime. In this paper, we present NaClDroid for completeness, as an additional purely SFI-based solution for isolating native code in Android. First, we begin with the underlying principles of NaClDroid. Next, we implement, and evaluate, a Dalvik VM, which is fully equipped with loading and running NaCl-compiled libraries.

Other approaches focus on Android's permission model, which is fundamental to the security of the platform [16,17,24,35]. An app is only allowed to perform actions (e.g., calling or texting) based on the permissions it holds, and these permissions are enforced by the OS. However, applications frequently request a broad set of permissions, users are not sufficiently attentive, and applications can collude with each other. As a result, we are increasingly relying on the analysis of apps to determine whether they are malicious. As these methodologies keep improving, ensuring the integrity of the verified code and the execution environment, under the presence of untrusted native third-party code, becomes even more important, because the modification of the execution *after* review will eventually become the sole avenue for malware authors [7]. NaClDroid is orthogonal to such proposals.

Previous approaches, unlike NaClDroid, do not attempt to provide developers with the means to isolate their apps from third-party code (or control how they interact with it). Furthermore, even though we do not aim to detect or protect from malware, through NaClDroid we confine native code, preventing the abuse of system call interfaces. For example, the DroidKungFu and DroidDream malware use a native library to execute exploits against the OS kernel [36]. These would fail under NaClDroid, since all native code is running under the constraints of a NaCl sandbox.

1.1 Contributions

To summarize, this paper makes the following contributions:

- We experimentally demonstrate the various ways third-party code can tamper with an app and its execution environment.
- We design and implement NaClDroid: an SFI-based framework that ensures the integrity and confidentiality of Android's execution environment. We embed Google's NaCl in Android's runtime to safely allow the use native code; with NaClDroid in place malicious libraries cannot leak information from, or subvert the control flow of, running apps.
- We thoroughly evaluate the proposed framework. Running native code through NaClDroid imposes a moderate slowdown of less than 10% on average.

– In contrast to similar approaches based on a strict review process, completely disallowing third-party code, or relying on code signing, this paper presents a series of challenges and systematic solutions for confining native third-party code linked with apps running in a highly open platform.

1.2 Organization

The rest of the paper is organized as follows. In Section 2, we present background information regarding the Android platform, and through detailed examples we discuss how malicious libraries can leak information from, or subvert control flow of, legitimate apps. Additionally, we discuss the threat model we consider in this paper. We present the architecture of NaClDroid and provide implementation details in Section 3. We analyze the security guarantees offered by NaClDroid, possible attacks against it, and additional hardening in Section 4. We evaluate the performance of our solution in Section 5. Related work is in Section 6, and conclusions in Section 7.

2 Malicious Third-Party Code

Android apps are written in Java, and, once compiled, the produced Java bytecode is transformed to a bytecode variant called Dalvik, which can be executed by the Dalvik VM [11]. Dalvik supports standard Java technologies, such as the Java Native Interface (JNI) [20], and hence all Android apps can attach native code to their bytecode. Native code was originally allowed to enable apps to quickly and efficiently perform computationally intensive tasks, such as cryptographic operations, image recognition, etc.

Android applications may include multiple third-party components, implemented either in native code or in Java, to complement their functionality; this is typical of many apps because it facilitates rapid development and code reuse. For example, it has been estimated that one in two Android applications is linked to an ad library [25]. Linking code in an application is not an unusual practice, and it is the *modus operandi* in many platforms—from desktop to mobile. However, linking code has major security implications in the case of Android. The openness of the platform allows adversaries to craft malicious libraries and implicitly attack a bigger user base than making malware popular. This has been recently demonstrated by researchers who identified malicious libraries that infected more than 32 legitimate applications in Google Play [7].

In the rest of this section, we investigate how malicious, native third-party code can break the integrity and confidentiality of legitimate apps that link to it. We look into how malicious libraries of native code can arbitrarily corrupt the virtual runtime, which executes the app, by re-writing the process virtual address space or exfiltrate sensitive information by reading the process memory. This technique can be used by any malicious third-party code for transforming a legitimate application to a malicious application [7].

```
1  #define MAP "/path/program@classes.dex"
2  char bytecode[4] = {0x90, 0x02, 0x00, 0x01};
3
4  void patch_bytecode(...) {
5      ...
6      mprotect((const void*)p, len, PROT_WRITE);
7      while (...) {
8          if (bytecode_found(p)) {
9              *p = 0x91;
10             break;
11         }
12     }
13 }
14
15 void patch_file(FILE *fp) {
16     ...
17     if (file_found(MAP)) {
18       patch_bytecode(start_address, length);
19     }
20 }
21
22 void nativeInjection(...) {
23     char process_path[16]; FILE *fp;
24     pid_t pid = getpid();
25
26     sprintf(process_path, "/proc/%d/maps", pid);
27
28     fp = fopen(process_path, "r");
29
30     patch_file(fp);
31 }
```

Fig. 1. Native injection. First, the `pid` of the running process is resolved (line 24). Next, the file `/proc/[pid]/maps` is opened (line 28) and scanned for finding the memory area where the Dalvik bytecode has been mapped (lines 15–20). Once found, the area is scanned for a particular opcode sequence, and, once located, the opcode `0x90` (integer addition) is changed to `0x91` (integer subtraction) (line 9).

2.1 Altering the Execution Environment

Native code resides inside the same virtual address space with the Dalvik VM. Therefore, it has direct access to all of the process memory, and can directly read or write anywhere in the process, easily tampering-with the Android runtime.

First, native code can *read* process memory, as it is mapped in the same virtual address space; this can break the confidentiality of the execution environment. Sensitive information that is stored in process memory can be easily exfiltrated by scanning the (virtual) memory footprint of the process. Consider, the example we presented in the introduction: an instant messaging application that uses a native-code library to serve advertisements to its users. The native library can easily scan the process image for sensitive data, like the user's password or message content exchanged with other users. Note that even if the developer of the instant messenger is careful enough to encrypt all conversations exchanged, the library can still gain access to the unencrypted content, since it can read data before reaching the encryption code. Therefore, *all* data manipulated by the app are (potentially) exposed to the library's code.

Second, native code can *write* the process image, as it is mapped in the same virtual address space; this can break the integrity of the execution environment. Native code can directly modify the bytecode or the virtual runtime itself. To better illustrate this, in Figure 1, we show a code sample of a JNI function (`nativeInjection()`, lines 22–31) that scans the app to locate a specific bytecode pattern and replace it with a different one; we have omitted some parts and simplified the code for brevity. The JNI function initially performs a `getpid()` system call (line 24) to retrieve the process ID. It then opens and scans the `/proc/[pid]/maps` file (lines 26–28) to obtain the memory layout of the current process. Function `patch_file()` is responsible for finding where the `.dex` file (i.e., the path indicated by `MAP` in line 1) is mapped in memory, and, finally, calling function `patch_bytecode()`, which replaces the targeted opcode. To do so, it first uses `mprotect` to make the particular region writable and then scans the region to find and replace the respective opcode. In this example, we change the first byte from `0x90`, which is the Dalvik opcode for addition, to `0x91`, which is the opcode for subtraction.

A real-world incident of such behavior is the Facebook app that altered the Dalvik VM to increase the size of one of its internal buffers [9], before it was banned from the Google Play store [3]. Facebook did this to overcome some internal constraints imposed by the Dalvik architecture, but it is evident that the same technique(s) can be used by malicious libraries against legitimate apps.

2.2 Threat model

NaClDroid protects applications from malicious third-party code, contained in shared libraries of native code. Although, in general, malicious third-party code can be implemented in Java as well, as we showed in this section, native code that plugs in with the rest of the code has superior capabilities. First, native code can read or write to the rest of the process image, and, second, it can implement this malicious functionality, silently. More importantly, analysis of the semantics of native code can be substantially harder (when compared to Java bytecode). Therefore, NaClDroid protects only from malicious native code that directly interferes with the process image. Additionally, NaClDroid assumes that the main app is legitimate and trusted. The app may expose interfaces to the third-party code, but it is assumed that the correct use of such interfaces (either through Java or native code) does not compromise the functionality of the application. However, since native code can interact with the rest of the app in unforeseen ways, not through APIs but by directly accessing the process image, NaClDroid confines all native code in an isolated sandbox.

3 The NaClDroid Framework

In this section, we present the design and implementation of NaClDroid. We begin with describing how SFI works and how the sandbox of native code is implemented. Next, we discuss how the new trust domains operate in an Android application. Lastly, we comment on architecture-specific issues.

```
1  void *_r__dlopen(const char *filename, int flag) {
2
3      uint32_t saved_esp = knatp->user.stack_ptr;
4
5      void * addr = (void *) setjmp(kbuf);
6      if (addr == 0) {
7          uint32_t esp = NaClUserToSys(knatp->nap,
8                                       knatp->user.stack_ptr);
9          /* Prepare stack (fname, *fname, flag). */
10         __prepare_stack(...);
11
12         /* Call dlopen().  */
13         __call_fcn(knatp, ...);
14     }
15     /* longjmp continues from here.  */
16     /* Restore esp.  */
17     knatp->user.stack_ptr = saved_esp;
18     /* Return result (i.e., pointer from dlopen()). */
19     return handle_return(addr);
20 }
21 int32_t
22 __syscall_handler(struct NaClAppThread *natp) {
23     struct __trampoline *rt;
24     ...
25     rt = __parse_args(natp);
26     ...
27     if (rt->rtf == rt_FunctionReturn) {
28             longjmp(kbuf, rt->ret);
29     }
30     ...
31     return 1;
32 }
```

Fig. 2. Example of a NaClDroid call. First, we store the current stack pointer of the NaCl thread (line 3). We then use `setjmp()` for saving the current state (line 5) and prepare the stack with the parameters required for calling `dlopen()` (line 10). Once the stack is prepared, we jump to the address of NaCl `dlopen()` (line 13). Next, a custom handler is called for serving the system call (lines 22–32). The handler parses the arguments of the system call (line 25) and if the system call refers to a function return, the return value is taken and passed to a `longjmp()` call, which will resume back to line 17. Finally, we restore the saved stack and return the value taken from `longjmp()` (i.e., the handle of the shared library loaded using `dlopen()`) back to Dalvik.

3.1 Software Fault Isolation

Android apps can host native code, which is mapped in the same virtual address space occupied by the Dalvik VM. Hence, native code can read process memory, leak sensitive data, modify existing bytecode, or change the semantics of the execution environment by directly patching its code [9]. In order to prevent such behavior(s), NaClDroid isolates all native code, using SFI, in a NaCl [37] sandbox (i.e., all native code is compiled with the NaCl toolchain). Therefore, upon execution, although native code is mapped inside the same address space with the running process, it can no longer read or write outside of the sandbox boundaries. Many dangerous system calls (like `mprotect`) are also prohibited.

We further refer to all native code built with the NaCl toolchain as NaCl modules, in order to be consistent with current NaCl terminology.

Dalvik uses the dynamic loader API, namely functions `dlopen()`, `dlsym()`, and friends, for loading external native code. Since all native code is now compiled as NaCl modules, we need to use the NaCl versions of `dlopen()` and friends that can handle the modifications performed by the NaCl toolchain. NaClDroid includes a NaCl shim, namely the `NaClDroid bridge`, which can dynamically load and invoke code hosted in a NaCl module. Thus, when Dalvik invokes `dlopen()` to load a native library, NaClDroid takes over and passes control to the `NaClDroid bridge`, which, in turn, redirects the call to the correct `dlopen()`, for loading a NaCl module. The `NaClDroid bridge` runs in a separate thread, which we will further refer to as the NaCl thread.

When Dalvik invokes the NaCl-compliant `dlopen()` to load NaCl modules, it expects a pointer to the loaded object in return. However, code compiled with NaCl, and thus using `NaClDroid bridge`, is isolated from the rest of the process; returning the pointer practically involves escaping the sandbox, so we need a way to safely communicate the pointer back to Dalvik. This is achieved by using a NaCl system call: i.e., special trampoline code, hosted in a different section, which can transfer data from NaCl modules to the rest of the process.

Based on the above, we now describe in detail the steps taken during a `dlopen()` call, using the example code illustrated in Figure 2. (The code has been simplified for readability.) Once `__r__dlopen()` is called, which is the `dlopen()` implemented by NaClDroid, the following take place. First, we store the current stack pointer of the NaCl thread (line 3)—i.e., the thread running `NaClDroid bridge`. Next, we use `setjmp()` for saving the current state (line 5); once the actual `dlopen()` is called, we are not going to return to the main thread normally, but using a special NaCl system call. We then prepare the stack with the parameters required for calling `dlopen()` (line 10). (Note that we need to jump directly to the address where the NaCl `dlopen()` is located, and, thus, we need to prepare an appropriate stack manually.) Once the stack is prepared, we directly jump to NaCl's `dlopen()` (line 13).

At this point, the execution has been transferred to the NaCl thread. As we have already stated, execution can be returned to the main thread only using a NaCl system call. Once that happens, a custom handler is invoked for serving it (lines 22–32). The handler parses the arguments of the system call (line 25), and if the system call refers to a function return, the return value is taken and passed to a `longjmp()` call, which will resume back to line 17. We then restore the saved stack and return the value taken from `longjmp()` (i.e., the handle of the shared library) back to Dalvik that initially called `dlopen()`. This technique is carried out for loading, resolving (i.e., using `dlsym()`), and invoking functions in native code. Figure 3, shows the control-flow of calling `dlsym()`.

3.2 Native-to-Java Communication

Native code can interact with the Java program. All native functions, loaded through JNI, take as parameter a pointer to a structure, named `JNIEnv`. This

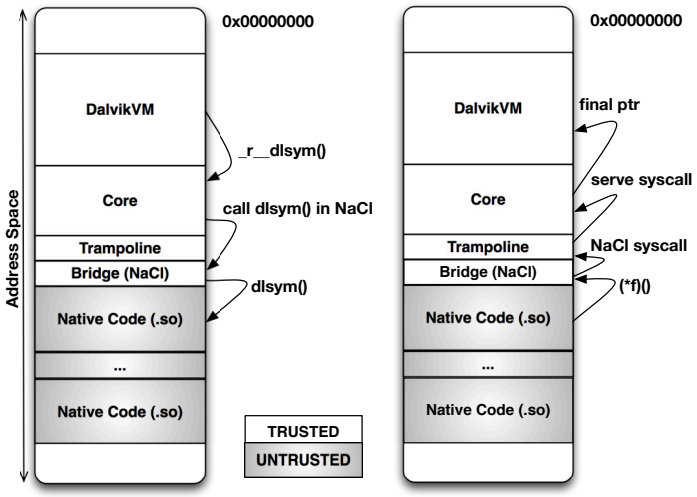


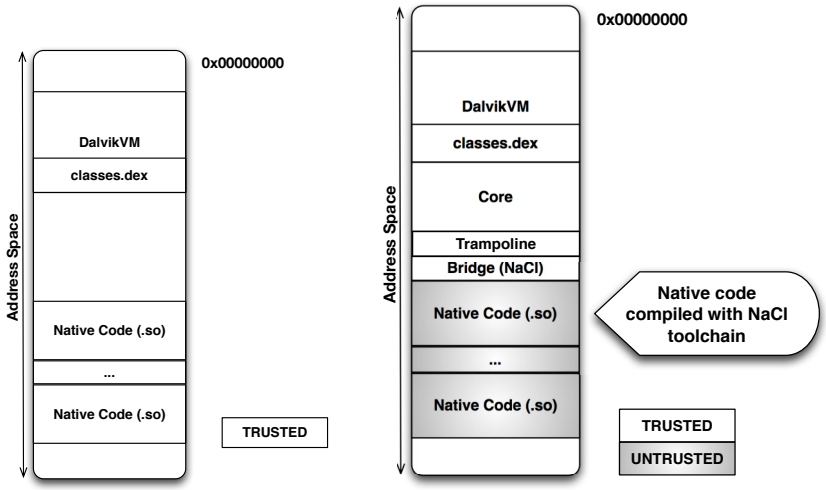
Fig. 3. Implementation of `dlsym()` in NaClDroid. Dalvik VM has been modified and `dlsym()` has been replaced with a wrapper that calls the NaCl-compatible `dlsym()` implemented in `NaClDroid bridge`. The wrapper resolves the symbol inside the untrusted sandboxed library, and by using a NaCl system call the address of the symbol is returned back to the VM.

structure encapsulates all available functionality provided from the Java environment to native code. For instance, a Java string object can be created in native code as follows:

```
jstring s = (*env)->NewStringUTF(env, "jni");
```

The implementation of `NewStringUTF` lies in the trusted domain, which means that it cannot be directly reached by code inside the sandbox. We could follow a similar technique with the one we outlined for `dlopen` earlier in this section. However, note, that in this case the code for the string creation is called by the developer, who is unaware of the existence of the sandbox. This is in contrast with `dlopen`, which is called by the VM under our control.

To allow native code access the API provided by Java we proceed as follows. First, we clone the environment structure, `JNIenv`, and make all function pointers, like the one for `NewStringUTF`, point at placeholder implementations located in NaClDroid bridge. Each native call receives a copy of the structure, and when a call takes place, the placeholder implementation (located in the bridge) is invoked, instead of the actual function located in the Dalvik VM. The placeholder serves the call by parsing all arguments and preparing a NaCl system call, which is the only way to communicate information to the trusted domain. Finally, NaClDroid serves the incoming system call by calling the actual function, getting the result, and communicating it back to the untrusted part.



(a) Dalvik (b) NaClDroid

Fig. 4. The NaClDroid architecture compared to vanilla Dalvik. All dark grey boxes are considered untrusted. We consider the environment, which executes the bytecode as trusted. The bytecode, by itself, cannot modify the trusted VM; native code can modify it and thus all supplied native code is considered untrusted. For ensuring that untrusted code cannot modify the trusted execution, we isolate it in a NaCl sandbox.

Recall, that the trusted domain can return to the sandbox normally, without following a particular procedure.

3.3 Trust Domains

NaClDroid separates Android applications in two parts, one considered trusted and one untrusted. We depict this separation in Figure 4, where NaClDroid is compared with the original Dalvik architecture. All dark gray boxes are considered untrusted. More precisely, we treat all Dalvik bytecode as untrusted, since it can potentially *change* at run-time. However, we consider the VM, which executes the bytecode as trusted. The bytecode, by itself, cannot modify the trusted VM; however, native code can modify it. User-supplied, native code is considered untrusted. Dalvik, NaClDroid, and the bridge, are all trusted. For ensuring that untrusted code cannot modify the trusted execution, we isolate it in a NaCl sandbox. This guarantees that native code cannot read/write outside its mapped memory, and cannot invoke OS system calls. NaClDroid introduces trust relationships in Android software, for the first time.

3.4 Platform Issues

Dalvik targets the x86-32 and ARM architectures. NaCl is also available for both x86-32 (using the segmentation unit) and ARM (pure SFI-based). The Dalvik interface for loading native code is based on dynamic linking. Dynamic linking can be provided by the GNU C Library (`glibc`) [22] or Android's `bionic` [1] library. The latter is an extended version of Newlib [19], which is designed for embedded systems (i.e., it is much more lightweight compared to `glibc`). The NaCl port for ARM is based on Newlib, which does not support dynamic linking. Hopefully, there are thoughts about porting `glibc` [8] or `bionic` [2] to NaCl for ARM. The design of NaClDroid, as outlined in this paper, is based on Dalvik for x86-32, and NaCl supporting dynamic linking through the NaCl port of `glibc` for x86-32. If `glibc` or `bionic` is ported to NaCl for ARM, then NaClDroid can be used *as is*, since it is based on an abstract interface for dynamic linking, like `dlopen()`, `dlsym()`, etc.

4 Security Evaluation

In this section, we discuss potential avenues of attacking NaClDroid and comment on how to tackle each one. In all cases, we assume a legitimate Android app that includes malicious third-party code, contained in a native library.

4.1 Running External Binaries

Android apps are allowed to launch external binaries to access system utilities and third-party programs. This is possible both through native code using one of the `execve` family of system calls, and through the Java `Runtime.exec()` API call. Allowing an app to invoke arbitrary binaries, essentially enables it to download any binary from the Internet and execute it, if permission for network access has been granted. Apparently allowing the arbitrary execution of binaries is overly permissive; a malicious app could use such a binary to launch exploits against the kernel and essentially break the guarantees offered by NaClDroid.

Blocking the execution of binaries is not straightforward. Applications can write in their own directory under `/data`, which permits execution. Preventing execution from `/data` is currently not possible, because it also holds the native libraries included with apps. As a result, disallowing execution from the partition (e.g., by setting the `noexec` flag during mounting) will also prevent loading any shared library using `dlopen`. More importantly, an app can also execute a binary indirectly by simply invoking the shell (`/system/bin/sh -c [my_command]`) or use a shell script that will invoke a binary.

To address these issues we propose permitting the execution of stock binaries only, such as the binaries located under `/system`, with the exception of the shell command that can be used to launch other commands. Executing binaries through native code is not allowed under NaClDroid, since native code now executes in a NaCl sandbox that blocks system calls. We tackle binary execution

from Java code by modifying the `Runtime.exec()` call to disallow applications outside whitelisted directories. In order to block the shell command, Android fortunately provides us with the means to easily block access to it: the stock shell of Android is owned by `root` and belongs to group `shell`. We can prevent most apps from executing it by removing their user ids (`uid`) from the `shell` group in Android, and by removing the executable bit from non-group members.

4.2 Malicious Apps

NaClDroid is not meant for detecting or identifying Android malware, but for protecting legitimate apps from malicious libraries. However, traditional malware can be also substantially confined if running under NaClDroid. By examining a popular malware dataset [38], we identified that over 53% of the malicious applications contain native code, such as native libraries or known exploits (e.g., `rageagainstthecage`) in their `assets` folder. This percentage is one order of magnitude higher than the one expected for Android markets, which is reported to be only about 4.52% (8272 out of the 204,040 studied) [39]. Therefore, we see a trend in malware towards hosting native code, potentially for exploiting and rooting the victims' devices. NaClDroid prevents *all* these exploits, since it prohibits external programs from running, unless they are part of the system tools, and all native code is constrained in a NaCl sandbox.

4.3 JVM Type Safety

Native code can modify data structures hosted in the Java domain through JNI. This can be used for modifying the state of the JVM, by assigning a pointer to a non-compatible type, something often identified with the term *type-confusion attack* [23]. Type safety is orthogonal to bytecode integrity and confidentiality, and, therefore, NaClDroid is orthogonal to existing frameworks that ensure type safety of heterogeneous programs that contain Java and C components [31,21].

4.4 Memory Corruption

Bugs in native code can corrupt memory. These bugs are usually due to memory writes outside the bounds of buffers or due to careless dereferences of pointers. An attacker able to trigger such a bug can change the original control flow of the buggy program and run her own code. There are many bugs that can lead to memory corruption, such as buffer overflows, `NULL` pointer dereferences, integer overflows, etc. In the context of our threat model, there is a critical difference. Memory corruption bugs may be *intentionally* implanted in the malicious application. [34] The attacker can trigger the bug at a later time, hijacking her own program, and modifying the app's behavior with new, malicious code. NaClDroid cannot prevent memory corruption, but can significantly confine it in the memory region where the vulnerable native module has been mapped. As a result, an attacker can only redirect control within the module. This, combined with

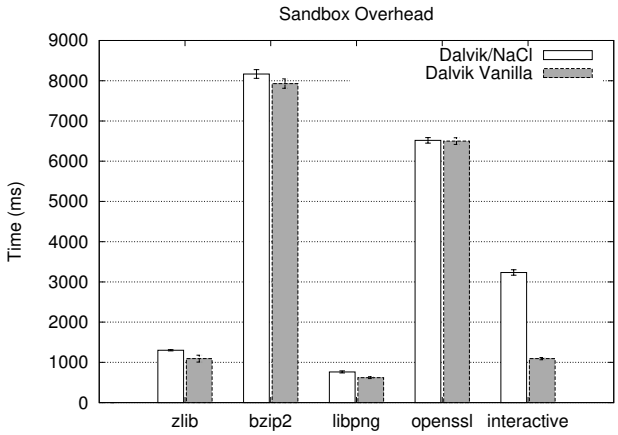


Fig. 5. Evaluation of NaClDroid's NaCl sandbox using popular packages ported on NaCl [4], such as `zlib`, `bzip2`, `libpng`, and OpenSSL. We also use a worse-case scenario, handcrafted test, which involves communicating large amounts of data between Java and native code, labeled as `interactive`.

the fact that there are no memory pages both writable and executable in the module, prevents code-injection attacks. The attacker could still employ code-reuse techniques to alter the functionality of the app, but recall that system calls cannot be executed. Consequently, even if the native library part of the app is compromised, it cannot escape the sandbox.

5 Performance Evaluation

In this section, we experimentally evaluate the performance overhead for sandboxing all native code. We show that, as expected, the overhead of NaClDroid is acceptable and similar to that of similar work [37].

To evaluate the overhead imposed by sandboxing, we use custom Android applications linked with popular packages that have been ported to NaCl [4], such as `zlib`, `bzip2`, `libpng`, and OpenSSL. We also created a custom test, which involves communicating a large number of data objects between Java and native code, which we call `interactive`. Notice that this test is artificially-made and does not follow correctly the paradigm of using native code with Java. Native code is meant to be used for performing heavy computation, and return results back to Java, and not for heavily exchanging data structures between the native and the Java part.

We run each Android application linked with a native library over NaClDroid and over the standard runtime. The workloads for all applications are part of the test programs of each particular package. We depict our results in Figure 5. Note that in all cases, except `interactive`, the run-time overhead is moderate; the NaCl sandbox imposes a slowdown of less than 10%, on average, similarly to

related reports found in literature [37]. However, the overhead becomes significant in the `interactive` test, since this test represents a worst-case scenario, where no computation is actually performed by the library and large amounts of data are copied over JNI. The increased overhead is primarily due to switching between the trusted thread of execution and the untrusted thread, which runs inside the sandbox. Recall that trust domain switching is a complicated process for guaranteeing that execution will leave the sandbox only in a very precise and confined way (see Section 3). Most Android applications do not use native code in that way (i.e., intensively exchanging data, back and forth, between Java and native code), but rather outsource computationally intensive tasks to the native part, and only receive back the final result; `interactive` serves as a micro-benchmark for measuring the overhead of merely switching trust domains.

6 Related Work

Mobile and Android security have received a lot of attention by the research community. In this section, we review related work in various fields of mobile security research.

6.1 Malware

An initial study of mobile threats for Android, iOS, and Symbian, was carried out by Felt et al. [18]; shortly after, larger studies of Android malware were carried out [39,38]. NaClDroid operates orthogonally to detection methods applied on app stores, and focuses solely on prevention at the end-host level. It guarantees that legitimate applications cannot be hijacked by malicious libraries. However, NaClDroid can also implicitly protect the system from traditional malware that aims at rooting the device. Lately, we see a trend in malware towards hosting native code, potentially for exploiting and rooting the victims' devices. NaClDroid prevents *all* such exploits, since it prohibits external programs from running, unless they are part of the system tools, and all native code is constrained in a NaCl sandbox.

6.2 Analysis

The research community has developed various techniques, employing static and dynamic analyses, as well as symbolic execution, to assist the reviewing of mobile apps. Ded [15] decompiles Dalvik programs to Java, which can then be analyzed using numerous, already-available static analysis tools. With the assistance of Ded, researchers managed to study more than 21 million lines of source code. ComDroid [13] also uses static analysis to infer malicious inter-application communication, performed through message passing. Paranoid Android [26] is a system for dynamically analyzing Android applications. In the same fashion, AASandbox [10] combines both static and dynamic analysis for identifying Android malware, while SymDroid [6] is a symbolic execution framework for Dalvik. Finally, TaintDroid [14] uses tainting for the analyzing apps.

The above are some representative works in the field of static and dynamic analysis, and symbolic execution, for mobile apps. NaClDroid complements these systems in the following way. Assuming that applications are thoroughly reviewed before distribution, using such analyses systems, we argue that malware writers will utilize more advanced techniques for modifying legitimate applications at run-time [9] using malicious third-party code [7]. NaClDroid guarantees that an app cleared during reviewing will not deviate by altering the semantics of the underlying execution environment.

6.3 Permission Model

Researchers have also focused in enhancing, and optimizing, the permission model of Android, which is crucial to the security of the platform. Kirin [16] attempts to resolve dangerous combinations of permissions at install time and warn the user. The authors also provide an implementation of a service performing application certification based on Kirin. Stowaway [17] can statically analyze Android apps to identify unnecessary permissions, drastically reducing the capabilities of over-privileged applications. Saint [24] enhances Android's permission model with policies, which are more powerful than static permissions enforced at installation time. Saint policies can assist the trusted communication between applications and components. Aurasium [35] enforces policies through user-level sandboxing. The authors automatically repackage Android apps with custom code, which is able to resolve offensive actions (e.g., calling or texting premium numbers). The great advantage of Aurasium is that it needs no system modifications, since apps are automatically extended to support the framework.

All these frameworks can be easily integrated with NaClDroid, since we make no assumptions about Android's permission and software model. Applications running over NaClDroid are only confined in terms of code integrity and code confidentiality; we do not focus on abuses of the permission model. NaClDroid operates transparently and does not affect, or interfere in any way, with the currently deployed permission model.

7 Conclusion

We presented how malicious, third-party code can hijack legitimate applications and break their integrity and confidentiality. We showed that native code, included in Android apps in the form of a library, can steal sensitive information from the app or even alter its control flow; a capability that has already been taken advantage by apps distributed through Google Play.

To address this issue, we designed, implemented, and evaluated NaClDroid, a framework that leverages SFI and embeds NaCl in Android's runtime to allow apps safely use native code, while at the same time ensuring that the code cannot exfiltrate sensitive data from the app's memory, extend itself, or tamper-with the Dalvik VM. Confining untrusted native code, using SFI, has also been adopted by successful projects like Chrome. In this work, we argue that stricter isolation

is also crucial for the Android platform to safely support native code in apps. We showed that NaClDroid has moderate overhead; our SFI-based implementation imposes a slowdown of less than 10% on average.

Acknowledgements This work was supported by the European Commission through project H2020 ICT-32-2014 "SHARCS", under Grant Agreement No. 644571, and the U.S. Office of Naval Research under award number N00014-16-1-2261. Any opinions, findings, conclusions, and recommendations expressed herein are those of the authors and do not necessarily reflect the views of the European Commission, the US Government, or the ONR.

References

1. Bionic C Library. https://android.googlesource.com/platform/bionic/+/android-4.2.2_r1/libc/README
2. Dynamic Linking in Native Client. http://code.google.com/p/nativeclient/wiki/DynamicLinkingPlan
3. Google bans self-updating Android apps, possibly including Facebook's. http://goo.gl/oKYM8f
4. NaCl ports, http://code.google.com/p/naclports/
5. Native Client in Google Chrome. https://support.google.com/chrome/answer/1647344?hl=en
6. SymDroid: Symbolic Execution for Dalvik Bytecode, http://www.cs.umd.edu/~jfoster/papers/symdroid.pdf
7. The Bearer of BadNews. https://blog.lookout.com/blog/2013/04/19/the-bearer-of-badnews-malware-google-play/
8. Thoughts about porting glibc to NaCl for ARM. native-client-discussion list. Private communication, December 2012
9. Under the Hood: Dalvik patch for Facebook for Android. https://goo.gl/1Hor5F. March 2013
10. Bläsing, T., Schmidt, A.D., Batyuk, L., Camtepe, S.A., Albayrak, S.: An Android Application Sandbox System for Suspicious Software Detection. In: MALWARE (2010)
11. Bornstein, D.: Dalvik VM Internals. In: Google I/O Developer Conference. vol. 23, pp. 17–30 (2008)
12. canalys: Over 1 billion android-based smart phones to ship in 2017, http://www.canalys.com/newsroom/over-1-billion-android-based-smart-phones-ship-2017
13. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing Inter-Application Communication in Android. In: MobiSys (2011)
14. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In: OSDI (2010)
15. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A Study of Android Application Security. In: USENIX Security (2011)
16. Enck, W., Ongtang, M., McDaniel, P.: On Lightweight Mobile Phone Application Certification. In: CCS (2009)

17. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android Permissions Demystified. In: CCS (2011)
18. Felt, A.P., Finifter, M., Chin, E., Hanna, S., Wagner, D.: A Survey of Mobile Malware in the Wild. In: SPSM (2011)
19. Gatliff, B.: Embedding with GNU: Newlib. Embedded Systems Programming 15(1), 12–17 (2002)
20. Gordon, R.: Essential JNI: Java Native Interface. Prentice-Hall, Inc. (1998)
21. Lee, B., Wiedermann, B., Hirzel, M., Grimm, R., McKinley, K.S.: Jinn: Synthesizing Dynamic Bug Detectors for Foreign Language Interfaces. In: PLDI (2010)
22. Loosemore, S., Stallman, R.M., McGrath, R., Oram, A., Drepper, U.: The GNU C Library Feference Manual. Free Software Foundation (2001)
23. McGraw, G., Felten, E.W.: Securing Java: Getting Down to Business with Mobile Code. John Wiley & Sons, Inc., New York, NY, USA (1999)
24. Ongtang, M., McLaughlin, S.E., Enck, W., McDaniel, P.: Semantically Rich Application-Centric Security in Android. Security and Communication Networks 5(6), 658–673 (2012)
25. Pearce, P., Felt, A.P., Nunez, G., Wagner, D.: AdDroid: Privilege Separation for Applications and Advertisers in Android. In: ASIACCS (2012)
26. Portokalidis, G., Homburg, P., Anagnostakis, K., Bos, H.: Paranoid Android: Versatile Protection for Smartphones. In: ACSAC (2010)
27. Sehr, D., Muth, R., Biffle, C., Khimenko, V., Pasko, E., Schimpf, K., Yee, B., Chen, B.: Adapting Software Fault Isolation to Contemporary CPU Architectures. In: USENIX Security (2010)
28. Siefers, J., Tan, G., Morrisett, G.: Robusta: Taming the Native Beast of the JVM. In: CCS (2010)
29. Sun, M., Tan, G.: JVM-Portable Sandboxing of Java's Native Libraries. In: ESORICS (2012)
30. Sun, M., Tan, G.: NativeGuard: Protecting Android Applications from Third-party Native Libraries. In: WiSec (2014)
31. Tan, G., Appel, A.W., Chakradhar, S., Raghunathan, A., Ravi, S., Wang, D.: Safe Java Native Interface. In: ISSSE (2006)
32. Viennot, N., Garcia, E., Nieh, J.: A Measurement Study of Google Play. In: SIGMETRICS (2014)
33. Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L.: Efficient Software-Based Fault Isolation. In: SOSP (1993)
34. Wang, T., Lu, K., Lu, L., Chung, S., Lee, W.: Jekyll on iOS: When Benign Apps Become Evil. In: USENIX Security (2013)
35. Xu, R., Saidi, H., Anderson, R.: Aurasium: Practical Policy Enforcement for Android Applications. In: USENIX Security (2012)
36. Yan, L.K., Yin, H.: DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In: USENIX Security (2012)
37. Yee, B., Sehr, D., Dardyk, G., Chen, J.B., Muth, R., Orm, T., Okasaka, S., Narula, N., Fullagar, N.: Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In: IEEE S&P (2009)
38. Zhou, Y., Jiang, X.: Dissecting Android Malware: Characterization and Evolution. In: IEEE S&P (2012)
39. Zhou, Y., Wang, Z., Zhou, W., Jiang, X.: Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In: NDSS (2012)