Blender: Self-randomizing Address Space Layout for Android Apps

Mingshen Sun[1], John C.S. Lui[1], and Yajin Zhou[2]

[1] The Chinese University of Hong Kong
[2] Qihoo 360 Technology Co. Ltd.

Abstract. In this paper, we first demonstrate that the newly introduced Android RunTime (ART) in latest Android versions (Android 5.0 or above) exposes a new attack surface, namely, the "return-to-art" (ret2art) attack. Unlike traditional return-to-library attacks, the ret2art attack abuses Android framework APIs (e.g., the API to send SMS) as payloads to conveniently perform malicious operations. This new attack surface, along with the weakened ASLR implementation in the Android system, makes the successful exploiting of vulnerable apps much easier. To mitigate this threat and provide *self-protection* for Android apps, we propose a user-level solution called BLENDER, which is able to *self-randomize* address space layout for apps. Specifically, for an app using our system, BLENDER randomly rearranges loaded libraries and Android runtime executable code in the app's process, achieving much higher memory entropy compared with the vanilla app. BLENDER requires no changes to the Android framework nor the underlying Linux kernel, thus is a non-invasive and easy-to-deploy solution. Our evaluation shows that BLENDER only incurs around 6 MB memory footprint increase for the app with our system, and does not affect other apps without our system. It increases 0.3 seconds of app starting delay, and imposes negligible CPU and battery overheads.

Keywords: Android, ROP, ASLR, Blender

1 Introduction

Due to the increasing functionalities of applications (apps for short) on mobile devices, the security and privacy of apps become one major concern. For apps running on the Android system, they are mainly written in Java. However, to enhance compatibility and performance, developers often choose to use the native development kit (NDK) to develop native libraries written in C/C++ and integrate them in their apps. A recent study [14] showed that around 37% of Android apps contain at least one native library. These native libraries are *not memory safe* and may suffer from memory corruption issues [4,5,8]. What is even worse that the potential vulnerabilities [9] in Android system libraries are loaded into every app's process and further expose more attack surfaces, even if the app itself does not contain any native library. Attackers could exploit vulnerabilities in native libraries and execute arbitrary shellcode on stack or launch ROP attack [40] if the stack is not executable.

To mitigate such threat, Address Space Layout Randomization (ASLR) [48] is a widely adopted solution in modern operating systems. If properly implemented, a system with the ASLR protection will randomize the loaded code and data into different

locations. Therefore, attackers cannot infer the memory layout from previous executions or other side channels, raising the bar for successfully exploiting the system.

Android introduced the ASLR protection since version 4.0 and improved the implementation in later versions. However, as indicated by previous research [32], the ASLR support in Android is not complete. First of all, the ASLR protection in earlier Android versions is only effective for system-related processes started at device booting stage, e.g., service management and communication-related processes. Second, the `zygote` process creation model of Android *indirectly weakens the effectiveness of memory layout randomization*. System libraries in different apps inherit shared (and same) memory regions from the parent `zygote` process. Thus, attackers can infer the memory layout from other running apps. This memory layout information helps attackers to initiate attacks and execute any arbitrary code on an Android system. For instance, Lee et al. [32] demonstrated the possibilities of remotely exploiting vulnerable apps and easily bypassing the ASLR protection in the Android system to launch an ROP attack (e.g., return-to-library [23, 36] and return-to-linker attacks). They further proposed a countermeasure called Morula that changes the Android system to randomize the memory layout for apps. Framework enhancement appears to be a natural solution. However, the need to change the Android framework could strongly impair the practical deployment due to the deep fragmentation of the Android platform.

In this paper, we first demonstrate that the newly introduced Android app RunTime (ART) exposes a new attack surface, namely return-to-art (ret2art for short). This attack surface increases the predictability of the memory layout of executable code regions which are the pool of useful ROP gadgets. Then, it further facilitates the construction of malicious payloads since attackers could return to the pre-compiled framework libraries and leverage the well-defined Android framework APIs to perform malicious operations. For instance, attackers could easily construct the payload to send SMS, get GPS locations on behalf of the vulnerable app if the app has corresponding permissions, without the need to understand the tedious details of the binder IPC mechanism and bridge the semantic gaps between the high level framework APIs and low level system calls. This new attack surface is not just in theory, but it is actually a practical threat. A recent study [37] leveraged a similar attack surface to exploit the Android system.

To mitigate this threat, we then propose a user-level solution called BLENDER. Our system provides the capability of memory layout self-randomization to (sensitive) Android apps with high security requirement, without waiting for the changes of the Android framework nor the underlying Linux kernel. Specifically, BLENDER first randomizes memory layout of loaded system libraries which are inherited from the `zygote` process. Then, to prevent the ret2art attack, BLENDER also randomizes the ART executable runtime dynamically at startup time. It ensures that the base addresses of libraries and the ART runtime are unpredictable.

We implement a prototype of BLENDER and evaluate its effectiveness and overhead. Our evaluation shows that apps using our system have a much higher memory entropy than vanilla apps. This means attackers have to try many times to successfully bypass the Android ASLR protection, instead of a single attempt. BLENDER incurs an increase of 6 MB memory footprint for an app. Note that, this only affects apps using our system, and does not affect other ones running on the device, an extra advantage compared with

the system-wide solution [32]. Our system increases 0.3 seconds to the app starting time, and incurs no obvious CPU and battery overhead.

To summarize, this paper makes following contributions:

- We first discover a new attack surface called ret2art attack in recent Android versions. This attack surface provides a large pool for useful ROP gadgets, and facilitates the construction of malicious payloads using high-level framework APIs.
- To mitigate the threat of ret2art attack and weakened ASLR implementation in the Android system, we propose a user-level solution which could *self-randomize* address space layout for both native libraries and the ART runtime of a running app, without the need of framework modification.
- We implement a prototype of the BLENDER system and evaluate the effectiveness and performance overheads. Our experiments show that BLENDER can gain high randomization entropy with only 300 milliseconds delay of the app's startup time, without obvious overhead to the CPU and battery resources.

The paper is organized as follows. In §2, we discuss the background of Android and related attack/defense methods. §3 explains the weakened ASLR mechanism in the current Android system, and we also illustrate conventional ROP attacks and propose a novel ret2art attacks on the latest version of Android. §4 presents the design and implementation of BLENDER. We present the experimental results which show the effectiveness, performance and battery overheads of BLENDER in §5. Finally, we discuss possible limitations in §6, study related work in §7, and §8 concludes the paper.

2 Background

In this section, we briefly introduce the new Android runtime (ART runtime) and the ASLR protection on Android.

2.1 Dalvik VM and ART Runtime

An Android app is a zip file packaged with Dalvik executable code (i.e., `dex` file) and other resources. In previous Android versions (before Android 5.0), Android utilizes the Dalivk virtual machine (DVM) to interpret the Dalvik bytecode at runtime. When an app is started, each Dalvik instance is created and system libraries and app bytecode will be loaded into an individual process. However, creating a new process and fully loading dependent libraries is a time-consuming process, especially on resource-limited mobile platforms. Android optimizes this process by creating the `zygote` process and pre-loading all the system libraries into this `zygote` process when the system is booting. Then all other apps are forked from this `zygote` process and inherit the pre-loaded system libraries (and the Dalvik instance) in the `zygote` process. This optimization improves an app's launch-time, however, defeats the ASLR protection in Android since the system libraries in different apps are shared the same memory layout. Figure 1 shows that the system libraries like `libc.so` and `libart.so` are shared between different apps and their addresses could be predicted by attackers. We will illustrate the way of launching corresponding attacks using the knowledge of predicted address space layout in Section 3.1.

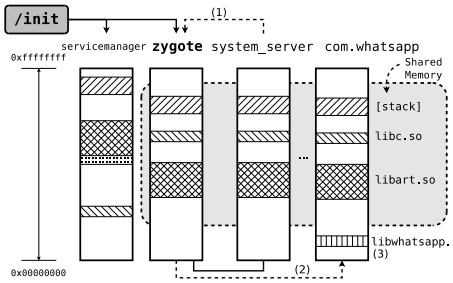

Fig. 1: Android booting and app creation process.

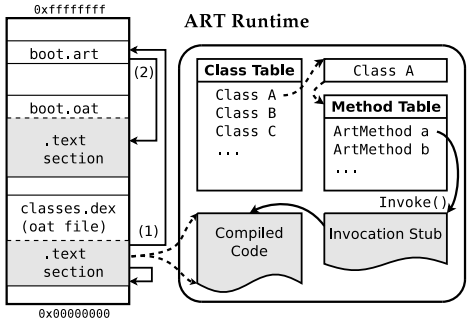

Fig. 2: Android ART runtime and memory structure.

Since Android 5.0, Google optimizes the Android system by introducing a new Android runtime, i.e., the ART runtime. ART introduces an ahead-of-time (AOT) compilation strategy to compile the Dalvik bytecode into native machine code. Due to this optimization, the framework-level APIs in the format of Dalvik bytecode are now converted into native code, and are shared between different apps. The new executable machine code is internally stored in the `oat` file format, which is nearly same with the traditional `ELF` format.

Figure 2 illustrates the flow of code execution of an app by the ART runtime. This runtime introduces three different memory regions into the app's process space. The first one is the `classes.dex` file, which contains an app's logic. The file name has a legacy extension which was inherited from the Dalvik runtime, but it is actual in the `oat` format. The second region is the `system@framework@boot.oat` file (i.e., "ART boot code" short for `boot.oat`). This region contains the compiled executable code of all Android framework bytecode. The third region is a data area and it does not contain any executable code. It is mapped with the `system@framework@boot.art` file (i.e., "ART boot image" internally and is called `boot.art` for short) which contains all necessary objects for bootstrapping the ART runtime. Basically, it provides a mapping table between a framework function and its real address of the executable code. To invoke a framework function in the app, the code will first (1) query the `boot.art` mapping table, then (2) call the actual code in the text section in `boot.oat`. For the ART runtime, there are class tables and method tables which maintain information of all loaded classes and methods. The runtime can call `Invoke()` of the `ArtMethod` in the method table to execute the compiled code through an invocation assembly code stub.

We found that the introduction of the ART runtime exposes a new attack surface due to two reasons. First, the large chunk of pre-compiled framework native code are shared between different apps, and its memory layout is more predictable than other system libraries. Thus, this increases the pool of libraries that could be used as ROP gadgets. Second, the ART runtime exposes all pre-compiled code of the framework functions at predictable locations. *Attackers can utilize this code as payloads and invoke high-level framework APIs more easily than the previous Dalvik runtime*. We will elaborate this form of attack surface in Section 3.2.

2.2 DEP/ASLR Protection on Android

Control flow hijacking is a way to exploit vulnerable program and control the program's execution flow. In old days, attackers usually hijacked the control flow to the data area and executed the prepared shellcode on stack. DEP is a security feature which intends to defeat this type of attack, by disallowing the memory page as writable and executable at the same time. This feature is supported in modern hardware and enabled by default in many operating systems, including the Android system.

Then researchers proposed the *return-oriented programming* (ROP) attack to defeat the effectiveness of the DEP protection. It does not need to inject shellcode into the data area and then mark the data area as executable. Instead, it reuses the already loaded code in the process to launch attack. Specifically, the ROP attack hijacks the program's control flow and jumps to existing executable instruction sequences which end with return instructions. These instruction sequences are called "*gadgets*". By chaining gadgets together, attackers can perform arbitrary operations regardless of the DEP protection. There are many kinds of ROP techniques, e.g., return into binary executable, return into shared libraries and return into non-randomized memory. The most widely used technique is the return-into-library technique, due to the fact that libraries such as `libc` contain functions (or gadgets) for invoking system calls and other functionalities which are useful to attackers.

To defend against ROP attacks, in conjunction with DEP, Address Space Layout Randomization (ASLR) was proposed in a probability manner. The basic idea of ASLR is that addresses of loaded executable, stack, heap and loaded libraries for each new process are randomized. Therefore, attackers cannot easily predict the memory address and jump to a fixed executable address of a gadget for an ROP attack. Although there are several techniques [34] to bypass DEP/ASLR, ASLR indeed makes attacks more difficult and limited.

Android gradually adopted memory layout randomization on stack, library, heap, and dynamic linker in Android 2.3.4, Android 4.0, Android 4.0.3, and Android 5.0 respectively. However, ASLR protection on Android is not as effective as expected due to several reasons. First, only the latest version Android 5.x supports the full ASLR protection, but it only accounts for 12.4 % among all Android devices [6]. Second, even in the case of the full ASLR protection, the `zygote` app creation model still tampers this protection (Section 2.1). Third, the pre-compiled system framework `oat` files increase the pool for ROP gadgets and facilitate the construction of malicious payloads, and introduce a new attack surface.

3 A New Attack: Ret2art

In this section, we discuss how to circumvent the ASLR protection on Android and present a new attack surface introduced by the ART runtime.

3.1 ASLR Circumvention

What went wrong? As discussed in the previous section, all apps are forked from the `zygote` process. This implies that the memory structures of child apps are identical and

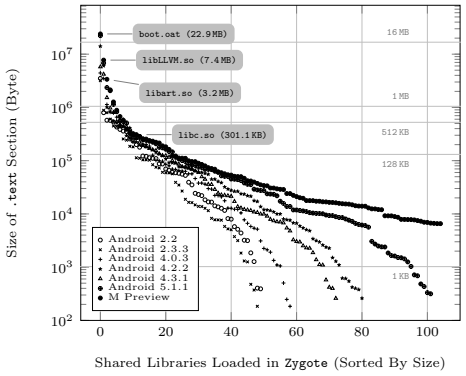

Fig. 3: Increasing .text section sizes of loaded shared libraries in `zygote` for different Android major versions.

Shared Library *	# of ROP Gadgets
libpdfium.so	56 154
libft2.so	7318
...	...
libandroid_runtime.so	1951
libEGL.so	1804
libz.so	1626
libvorbisidec.so	1219
libc.so	1049
...	...
Total	102 311

* Sorted by the number of ROP gadgets.

Table 1: Number of unique ROP gadgets of loaded libraries in the zygote process.

duplicated by the parent `zygote` process. In other words, the base addresses of stacks, common libraries such as `libc.so`, and the dynamic linker are same in every app. Attackers can now easily predict memory layout information of all apps from one single exploited app. Moreover, even if some system libraries are not used by the app, they are still mapped into the app's process because the `zygote` process has loaded them. This further increases the possibility of the success of the ROP attack. In summary, the way that Android app is created *defeats the purpose of ASLR mechanism*.

We discover that the loaded libraries of the `zygote` process provide rich sources of ROP gadgets which every other app will inherit. To quantify the attack surface, we measure the size of text section (or executable section) of system libraries loaded in the `zygote` process for different Android major versions. Figure 3 shows that the number of loaded libraries increases from 50 to about 100, and the largest size of executable section is about 22 MB. This exposes a large number of vulnerable executable instructions for attackers. We then utilize an automatic ROP gadget search tool [7] to find out possible gadgets (i.e., instruction sequences ended with `bx reg`, `blx reg` and `pop {,pc}`) in shared libraries of the `zygote` process. Table 1 shows the number of unique ROP gadgets found by the tool in Android 5.1.1. Two common system libraries `libandroid_runtime.so` and `libc.so` (highlighted in the table) contain around a thousand usable gadgets. Because these two libraries provide basic functionalities for other part of the system, they are stable across different Android versions. Attackers could leverage the found ROP gadgets in them to launch the ROP attack.

How to exploit? To further understand the way to launch the ROP attack on Android, we use an example to illustrate the whole process. Figure 4 shows the flow of this attack. The attack scenario involves two apps. The objective of the first app (App A) is to obtain the current memory layout. This app can be a simple trojan app installed beforehand. Note that one app can access its own memory layout without any privileged permission. By reading the `/proc/self/maps` file, attackers can easily obtain the memory mapping

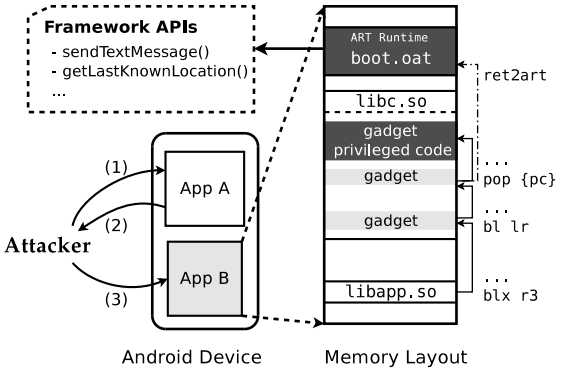

Fig. 4: ROP attack on Android (return-to-library attack and return-to-art attack).

information including library names, base addresses, and protect permissions, etc. The second app (App B) is the target app for an ROP attack, which has a buffer overflow vulnerability (e.g., popular apps like VLC [13] and Adobe Flash [1] have such vulnerability). Attackers first induce users to install the first app (App A) (step 1) to obtain the current memory layout (step 2). Secondly, attackers can craft a chain of gadgets using common libraries such as `libc.so` and `libart.so`, then determine the absolute addresses according to the current memory layout obtained previously. At last, attackers exploit the buffer overflow vulnerability of the legitimate app (App B) to initiate an ROP attack (step 3). By jumping and chaining executable gadgets, attackers can execute arbitrary privileged code for further attacks.

Even though the proposed attack in Figure 4 leverages the first app (App A) to obtain the memory layout information, this information could be obtained through exploiting vulnerabilities in legitimate apps. For example, several known vulnerabilities of the Chrome Browser [4, 5] and Samsung KNOX browser [8] can leak part of the memory information. That means the proposed attack could be launched remotely without the need to install the first app (App A). This conclusion has been demonstrated in the previous research [32] and we will not discuss its details in this paper.

3.2 The New Return-to-ART Attack (ret2art)

When launching the ROP attack, the most complicated part is to design a valuable gadget chain and execute malicious payloads. Traditionally, attackers could leverage particular system calls (e.g., `execve()`) or existing functions in common libraries (e.g., `system()` and `strcpy()` in `libc` library) for this purpose. However, in the context of the Android system, it is hard for attackers to construct meaningful payloads. For instance, if attackers want to send a text message to subscribe to a premium service to make money, or to steal private information from a local database, they have to bridge the semantic gap between the malicious operations and low level APIs. Though the Android framework provides many useful APIs, it is hard for attackers to invoke them since these APIs are in the format of the Dalvik bytecode and cannot be executed directly. Therefore, it is a non-trivial task to construct malicious payloads on Android.

The ART runtime was introduced since the latest Android version 5.0. We found that the design and implementation of the ART runtime *exposes a new attack surface*, which is called *return-to-ART (ret2art) attack*. It eases the construction of malicious payloads and attackers could initiate more powerful and damaging attacks.

What went wrong? Due to the introduction of the ART runtime, the addresses of the pre-compiled native code of the system framework APIs are predictable. First, `boot.oat` and `boot.art` files contain the compiled native code and related metadata of Android framework APIs. These two files are generated by phone vendors before shipping the devices to users, and will not change unless there is a new OTA update image. Therefore, these files are same across all devices using the same firmware image. Second, the base address of `boot.art` is fixed (`0x70000000` for the 32-bit ARM architecture) in the AOSP source code (in the `/build/core/dex_preopt_libart.mk` file [11]). The exact mapping address of the `boot.oat` file is patched when the device is first booted, and will not change unless a system update is performed. The patch offset of the `boot.oat` file is fixed between `-0x01000000` and `0x01000000` as indicated in `/art/build/Android.common_build.mk`. For instance, if the patch offset of `boot.oat` is `0x8000`, then `boot.oat` will be mapped to the fixed address `0x70008000` every time for every app running on the device, until the device updates its firmware image.

The predictable nature of the addresses of loaded oat and art files exposes a new attack surface (the ret2art attack). First, `boot.oat` is loaded by the `zygote` process and inherited by all other apps. Therefore, the base address of the `boot.oat` file is fixed for *every app in each execution*. Second, the `boot.oat` file is mapped as an executable region in memory. It contains abundant number of compiled native code of all methods in the Android framework, and provides a fertile ground for ROP gadgets. According to Figure 3, the size of the executable code in this file is around 22.9 MB. Third, the code offsets for each method are fixed and can be easily located from the structured metadata from either the `boot.art` file or the `boot.oat` file. Therefore, attackers can craft gadgets and jump to the native code offset of a method in the `boot.oat` file. Figure 4 illustrates the basic flow of the re2art attack. Similar with the conventional ROP attack, attackers can hijack the control flow to the ART executable code. This way, attackers can invoke framework APIs in the ART runtime, which facilitates the construction of malicious payloads. For instance, attackers can use the `getLastKnowLocation()` API to obtain any recent geographical location information.

How to exploit? Suppose attackers want to send a text message to achieve an *unauthorized premium services subscription*. *First*, attackers need to get the offset of the `sendTextMessage` method in the `boot.oat` file. This can be achieved by reading the `boot.oat` of the firmware using the `oatdump` tool. Note that, since this offset is only related to particular firmware, attackers could get this knowledge in advance by downloading firmwares from Internet and obtain a mapping table of offsets of interested APIs to the firmware fingerprint. The base address of the `boot.oat` file is fixed after the system is first powered, and could be obtained through another trojan app or information leak vulnerabilities in other apps, and even guessed since the base address is around a fixed location `0x70000000`. Code snippet 1.1 shows an example of the dumped `boot.oat` file. We can find that the code offset of the `sendTextMessage` method is fixed in the `boot.oat` file at `0x02ca944d` (line 11). *Second*, similar to the previous ROP attack,

```
$ adb shell oatdump -oat-file=/system/framework/arm/boot.oat
...
IMAGE PATCH DELTA: -724992 (0xfff4f000)
...
40: Landroid/telephony/SmsManager; (offset=0x015d849c) (type_idx=198) (StatusVerified) (
      ↪ OatClassSomeCompiled)
  ...
  37: void android.telephony.SmsManager.sendTextMessage(java.lang.String, java.lang.String,
        ↪ java.lang.String, android.app.PendingIntent, android.app.PendingIntent)} (
        ↪ dex_method_idx=844)
    OatMethodOffsets (offset=0x015d853c)
      code_offset: 0x02ca944d
    ...
    CODE: (code_offset=0x02ca944d size_offset=0x02ca9448 size=324)...
      0x02ca944c: f5bd5c00  subs    r12, sp, #8192
      ...
```

Code Snippet 1.1: Example of `oatdump` for `boot.oat` file.

attackers can obtain the base address of `boot.oat` file locally or remotely. Combing the obtained offset and the base address, attackers now have the absolute address of the method. *Third*, attackers exploit existing or zero-day buffer overflow vulnerabilities of the target app to hijack the control flow for initiating a ret2art attack. Note that attackers cannot directly jump to this address and execute the code, because the framework code should be executed with the support of the ART runtime. Specifically, the ART runtime executes native methods through an invocation stub code, i.e., the `art_quick_invoke_stub` function defined in the `quick_entrypoints_arm.S` assembly file [12] for the ARM platform. Before invoking this code, attackers have to prepare several registers for related parameters as shown in Table 2. After passing these registers to the `art_quick_invoke_stub` function, the function will finally load the compiled code to a register as a branch address. As shown in Code Snippet 1.2, the address is calculated by summing up `r0` with an offset `METHOD_QUICK_CODE_OFFSET_32` in line 8, that is, the address of `entry_point_from_quick_compiled_code_` field in the `ArtMethod` class. Moreover, `r1-r3` are copied from the stack controlled by attackers, which makes the ret2art attack even easier. Therefore, to initiate a ret2art attack, the attacker can branch (e.g., `blx reg`) to this stub function and invoke the `sendTextMessage` framework API. If the target app has declared the "`SEND_SMS`" permission, attackers can use this technique to subscribe to some premium services, or to spread the trojan app via messages.

4 Blender

In this section, we present the design and implementation of BLENDER, a user-level solution to mitigate threats caused by the weakened ASLR implementation on Android and the new ret2art attack.

Parameter	Description
r0 register	method pointer to the invoke ArtMethod class object
r1 register	argument array or NULL for no argument methods
r2 register	size of argument array in bytes
r3 register	thread pointer
[sp]	address for return value
[sp + 4]	address for shorty character representation of return value

Table 2: Parameter description of "Invocation Stub".

```
1  ENTRY art_quick_invoke_stub
2  ...
3      ldr    r0, [r11]      @ restore method*
4      ldr    r1, [sp, #4]   @ copy arg value for r1
5      ldr    r2, [sp, #8]   @ copy arg value for r2
6      ldr    r3, [sp, #12]  @ copy arg value for r3
7      mov    ip, #0         @ set ip to 0
8      ldr    ip, [r0, #METHOD_QUICK_CODE_OFFSET_32]
9                             @ get pointer to the
                                   code
10     blx    ip             @ call the method
11 ...
12 END art_quick_invoke_stub
```

Code Snippet 1.2: Invocation stub function.

4.1 High Level System Design

Design Requirements Our goal is to provide a user-level solution. Accordingly, we follow several design requirements to balance protection strength, performance, and practical system deployment.

Complete Protection: Our system needs to mitigate the threats introduced by both the `zygote` application creation process and the new ART runtime. This means that our system has to eliminate the predictability of the memory layout for loaded system libraries, and the pre-compiled native code of the framework APIs (the `boot.oat` file).

Lightweight Protection: It naturally requires that our system should be memory- and energy-efficient. The performance overhead should not affect user experience. Moreover, the overhead introduced should not affect the apps without our protection.

Easy Deployment: Our system should maintain the compatibility of existing apps, and not require any change to the Android framework nor the Linux kernel. Also, the changes made to apps for deploying our system should be minimum.

Threat and Trust Model As our purpose is to provide a user-level solution to mitigate the threat of weakened ASLR protection on Android, we assume app developers are trusted. However, their apps or libraries that apps are depending on may have security vulnerabilities and could be exploited by attackers both locally and remotely to arbitrarily read, write, and execute code in app's memory. Their apps have higher security requirements, and they want to provide the self-protection capability to their apps. By deploying our solution, the empowered self-protection capability makes the exploitation of the vulnerabilities in their apps much harder.

Design Overview BLENDER provides protection in two different aspects. First, BLENDER randomizes the addresses of loaded system libraries for apps using our system. This eliminates the possibility that the memory layout of these libraries are predictable. From this perspective, our system provides similar security guarantees as previous work [32], by solely in user space, without making changes to the Android framework. Second, BLENDER deals with the new ret2art attack by randomizing the executable code of the pre-compiled framework APIs (i.e., `boot.art` and `boot.oat` files) in the ART runtime. This is a new security guarantee which is not covered by previous research.

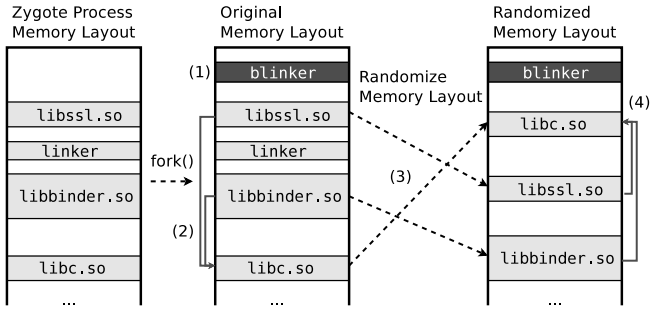

Fig. 5: Overview of BLENDER Library Randomization Module.

Accordingly, BLENDER contains three components: (1) the bootstrap module, (2) BLENDER library randomization module (short for BLENDERLRM), and (3) BLENDER ART randomization module (short for BLENDERART). The bootstrap module takes over the startup stage of an app, and prepares the running environment of our system. Like other user-level solutions [54, 60], this bootstrap module is integrated into the app by simply including a proxy class which extends the `Application` class. Then the bootstrap module will invoke BLENDERLRM to self-randomize the current loaded libraries. After that, it will invoke BLENDERART to rearrange the ART runtime in the memory. Finally, the original app will be loaded and started. Since the bootstrap module has been extensively discussed in previous research [54, 60], in this paper, we will explain BLENDERLRM and BLENDERART, respectively.

4.2 BlenderLRM

Figure 5 illustrates the overview of BLENDERLRM. The main purpose of BLENDERLRM is to randomize the addresses of already loaded system libraries inherited from the `zygote` process, and all other app-provided third-party libraries. For this purpose, BLENDERLRM leverages a customized dynamic linker (named as `blinker`), which first rearranges the already loaded system libraries and then takes over the process of loading app-provided third-party libraries and randomizes their addresses. Note that all the described operations in this section later are only applied to its own process of the app with our system, and does not affect other processes running on the same device.

Rearrange System Libraries Rearranging the system libraries looks straightforward, since all system libraries on the Android with ASLR support should be compiled as position independent code (PIC). This means that these libraries could be loaded into any addresses [3]. We can simply copy the loaded libraries from one location to another one to randomize the loaded addresses of them. However, most, if not all, system libraries are dynamically linked. These dynamically linked libraries depend on other libraries, and their dependencies have been resolved when creating the `zygote` process. Simply mov-

[3] In early versions of Android without ASLR support, system libraries are pre-loaded into fixed locations.

Algorithm 1 Memory Randomization Algorithm

```
1: function RANDOMIZELIBRARIES(libraryDependencyGraph)
2:     sortedNodes ← TOPOSORT(libraryDependencyGraph)
3:     for each n ∈ sortedNodes do
4:         DUPMAP(n)                          ▷ Duplicate memory mapping to a random free space.
5:         for each node m with an edge from n to m do
6:             FIXGOT(m,n)                    ▷ Fix symbol resolution in GOT of m.
7:         end for
8:         SAVELIBRARYINFORMATION(n)
9:         UNMAP(n)                           ▷ Unmap library n from memory mappings.
10:    end for
11: end function
```

ing the system libraries from one location to another location will break the resolved dependencies, and crash the app.

Before presenting our method to solve this challenge, we will describe the background of dynamic linking first to help readers better understand our proposed method. For each dynamically linked library, there is a Procedure Linkage Table (PLT) section (.plt), which contains several stubs to call external functions. For example, suppose library *A* uses the `strcpy` function in `libc`, then there is a stub for the `strcpy` function in the PLT section of library *A*. The functionality of this PLT stub is to load the real address of the `strcpy` (of `libc` in the memory) from the entry of the Global Offset Table (GOT) section, and then jump to it. Each external function used by the library has an entry in GOT, and its real address is resolved by the dynamic linker (i.e., `/system/bin/linker` in Android) when the library is first loaded into the memory and written in the corresponding GOT entry. Note that the dynamic linker in Android does not adopted the "lazy binding" mechanism [20], which is common in the desktop systems, to speed up the app startup stage.

To solve the challenge of dependencies between system libraries, `blinker` generates a dependency graph on the loaded libraries and fixes the wrong addresses in GOT due to library rearrangement. We say that library *A* depends on library *B* if there exists a function call from library *B* to library *A*. For instance, `liblog.so` uses the `strcpy()` function in `libc.so`, and we say `libc.so` depends on `liblog.so`. In the dependency graph, there will be an edge from *A* (e.g., `libc.so`) to *B* (e.g., `liblog.so`). Correspondingly, the GOT section of `liblog.so` should contain an entry of the `strcpy` function pointing to `libc.so`. Figure 6 illustrates the dependency graph of ten common libraries loaded by `zygote`. From the figure, we can see that there are eight libraries which depend on `libc.so`. Therefore, if BLENDERLRM rearranges `libc.so` library to other address, addresses pointing to `libc.so` in GOTs of its dependent libraries needs to be updated. Note that `blinker` itself is statically linked, otherwise it will depend on other system libraries which will be rearranged and a dead lock will be created between `blinker` and its dependent libraries.

After generating the dependency graph, `blinker` rearranges system libraries according to the method described in Algorithm 1. The algorithm takes a library dependency graph as an input. `blinker` first topologically sorts the dependency graph. For each node in the sorted node list, `blinker` first duplicates it into a random free space

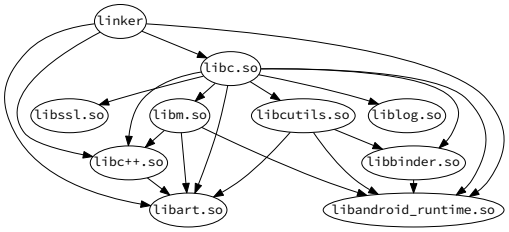

Fig. 6: Dependency graph.

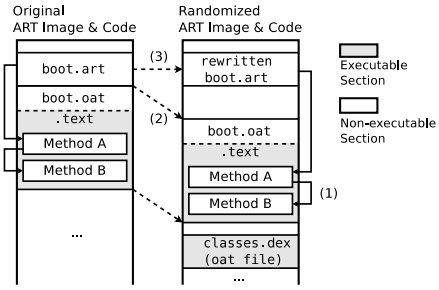

Fig. 7: Overview of BLENDERART.

aligned with the memory page size. Then, `blinker` fixes GOTs of its dependent nodes. Furthermore, `blinker` will store the library information including new base locations, names, dependency information, etc. This information will help `blinker` to link libraries which will be added at later stages. Finally, `blinker` will unmap the original libraries from memory.

Rearrange App-provided Third-party Libraries Besides system libraries, an app may have its own third-party libraries. For instance, the app using the Cocos2d game engine will include the corresponding native libraries in the app. Our system needs to randomize these libraries as well to ensure they have different addresses in different runs. For this purpose, `blinker` takes over the role of the original linker. Specifically, native libraries are loaded into memory by the `dlopen()` function in `libdl.so`. We modify the dynamic linker related function pointers in the GOT section of `libdl.so` to our customized `blinker`. Then, if a new native library is loaded into memory by using the `dlopen()` function, `blinker` will map it into a random address and resolve external function calls.

4.3 BlenderART

As discussed in Section 3.2, the newly introduced ART runtime exposes a new attack surface, due to the fact that the pre-compiled `boot.oat` file is in a fixed memory location after the system is first booted and will not change unless an OTA update is performed [4]. Our system needs to rearrange this `boot.oat` to other locations. However, the differences between the `boot.oat` and other system libraries we discussed in Section 4.2 pose new challenges, and we cannot directly apply the method proposed in Section 4.2 to the `boot.oat` file.

Figure 7 illustrates the workflow of BLENDERART. There are three steps to carry out the ART runtime randomization: (1) patch the `boot.oat` file with an offset, (2) load this patched `boot.oat` file into the memory, (3) fix code addresses of the class linker instance in the ART runtime.

Patch & Load Boot.oat To randomize the loaded address of the `boot.oat` file, two main components in the `boot.oat` file should be patched. First, some branch instruc-

[4] Actually, the app's bytecode in the file classes.dex is also compiled into the native code. However, this compiled native code is loaded into different places each time the app is started.

Algorithm 2 ART Runtime (boot.oat) Patching Algorithm

```
1: function PATCHOAT(oatFile, offset)
2:     for each patch ∈ oatFile.oatPatches do
3:         patchLocation ← GETLOCATION(patch)
4:         *patchLocation ← patchLocation + offset
5:     end for
6:     FIXUPOATHEADER(oatFile, offset)
7:     FIXUPELF(oatFile, offset)
8: end function
```

tions in the `boot.oat` use *absolute* addresses to jump to the target instruction. For instance, suppose method A invokes method B in the framework as shown in Figure 7, the branch instruction jumping from method A to method B uses an absolute address in memory. These absolute addresses should be patched if we want to move the `boot.oat` to another location. Second, the metadata information in the oat file header contains absolute addresses, and need to be patched too.

One natural choice to patch the address is to leverage the binary rewriting tool to disassemble the compiled native code, locate and modify absolute addresses in branch instructions. However, writing a binary rewriting tool from scratch is a tedious and error-prone process. In this work, we take advantage of a convenient interface provided by Google for binary rewriting, which is called the `oat_patches`. When converting the dex bytecode to native code, the ART compiler first translates the dex bytecode into an intermediate representations (MIR), and then compiles it into the low-level intermediate representation (LIR). During the converting stage from MIR to LIR, the compiler records all literals (including code, method, class, and string literals) which contain absolute addresses and can be modified later (implemented in `InstallLiteralPools()` methods in the `codegen_util.cc` file [10] from AOSP). And the literal information will be written into one special ELF section of the final `oat` file, which is called the `oat_patches` section. We leverage the `oat_patches` tool to help us relocate `boot.oat` and patch the original fixed absolute addresses. In fact, this `oat_patches` information is also used by Android to patch the `boot.oat` when the system is first powered on.

BLENDERART first randomly picks a free memory region and calculates the offset (Δ) between the new base address and the original one ($\mathcal{B}$). Algorithm 2 illustrates the procedure to patch the `boot.oat` file. The patching algorithm takes the `oat` file and offset number as input, and will go through all patches and add an offset. The `FixupOatHeader` function is to relocate the metadata of the embedded oat header. The `FixupELF` function is to rewrite the section header information, dynamic symbol section (`dynsym`) and the symbol table section (`symtab`) information. At last, the patched `boot.oat` will be loaded into the memory. Because we already fixup all relocation based on an offset, the load address should be $\mathcal{B} + \Delta$.

Fix Class Linker Data Instance Besides the absolute address in the code area in the `boot.oat` file, some information in the data area of the ART runtime should be patched too. Class linker (i.e., the `ClassLinker` class) is a single global instance maintained by the ART runtime. Since the executable code in the `boot.oat` file has been relocated by our system, several important information maintained by it should be fixed

too. For instance, it maintains a class table (the `class_table_` field), which contains loaded classes information (i.e., the `mirror::Class` class). For each class structure, it contains corresponding methods in the method tables. There are two types of methods: direct methods and virtual methods, which are stored in the `direct_methods_` table and `virtual_methods_` table respectively. The methods in the method table are in the `mirror::ArtMethod` class. There is a pointer sized field contains four entry point addresses. For example, the `entry_point_from_quick_compiled_code_` field of a framework method points to the actual compiled code address of `boot.oat` in the memory. Since `boot.oat` has been relocated, this pointer should be fixed to point to the new address. Finally, BLENDERART changes the old memory region of `boot.oat` to non-executable to ensure data in this file cannot be executed, but can still be accessed by the ART runtime. In theory, we could fully unmap this memory region. However, we then need to fix all the references to this memory region in the ART runtime, which is a time-consuming work. As long as the code area is no-longer executable, it is safe to leave it there since attackers cannot leverage it to construct ROP gadgets.

Optimization Apps with BLENDERART should perform all the previous steps to achieve the ART runtime randomization. However, patching the `boot.oat` file introduces an overhead of around 1.6 seconds which will be shown in Section 5. To reduce this overhead, we cache the randomized `boot.oat` so as to reduce the app's startup time. We design a patched `boot.oat` pool which contains a set of offline patched `boot.oat` files with different random patched offsets. For each execution, our system picks up a patched `boot.oat` file from the pool and loads it into the memory, without patching it online.

4.4 Implementation Details

We prototype our BLENDER system based on Android 5.1 Lollipop (the AOSP tag `android-5.1.0_r1`) for 32-bit ARM architecture. Since the code base of the ART runtime is stable after Android 5.0, our implementation is generic for Android 5.0 and 6.0 versions. The system contains about two thousand lines of code including C/C++ and Java. For the implementation, we reuse the peer-reviewed code from AOSP as much as possible. This will ensure the stability and security of BLENDER. We use the `/dev/random` file as the seed for randomization.

There is no official ART support for Android versions less than 5.0 [5]. Therefore, the Dalvik virtual machine runtime cannot be exploited by using the ret2art attack technique. Although a researcher discovered interpreter exploitation [16] on the conventional JIT based virtual machine, it is still difficult to initiate attacks on the Dalvik runtime. However, the security issue caused by the `zygote` app creation model still exists. To harden the ASLR for old Android versions (before Android 5.0), we port BLENDERLRM to them so as to self-randomize addresses of system libraries inherited from the `zygote` process. Because BLENDER is a user-level solution and provides self-randomization capability to the apps using our system, rather than modifying the source code of the Android framework, app developers could safely deploy our system and their apps immediately get protected.

[5] There is an experimental implementation of the ART runtime in Android 4.4 but is disabled by default.

5 Evaluation

In this section we evaluate the effectiveness of BLENDER by measuring the app memory entropy, and the performance overhead at apps' startup time, execution, memory, and battery usage. The device used in the evaluation is a Nexus 5 device with Quad-core 2.3 GHz CPU, 2 GB memory and 16 GB internal storage. The test device runs the Google official Android firmware which is Lollipop 5.1 with the build number `LMY47D` and the kernel version 3.4.0.

5.1 Effectiveness

The goal of the BLENDER system is to prevent attackers from predicting address space layout of apps. To evaluate the effectiveness of BLENDER, we first discuss from an app's perspective.

To measure the address space layout randomness of shared system libraries, we use the notion of *entropy*. Entropy is a metric to represent the uncertainty of random variables. We apply entropy to measure memory layout randomness, and the library loading addresses are treated as a random variable. We utilize the space layout entropy metric from [32] to evaluate the application randomness. Specifically, for a shared library or runtime image code m, let X_m be a discrete random variable with base addresses $\{x_1, x_2, \ldots, x_n\}$ and $p(x_i)$ is a probability of $x_m = x_i$. The normalized address space layout entropy can be defined as $H(X_m) = -\sum_{i=1}^{n} p(x_i) \frac{\ln p(x_i)}{\ln n}$, and $0 \leq H(X_m) \leq 1$ because of normalization.

App Randomness Because BLENDER only randomizes memory of certain apps with the BLENDER protection, we evaluate the entropy on one app for multiple executions. We define $\{x_1, x_2, \ldots, x_n\}$ as base addresses of the library m, and n is the number of executions for one app. For instance, suppose $n = 10$, we execute the app with BLENDER ten times, and the base addresses of library `libart.so` are totally different. Because each base address is uniformly distributed, the output will have a probability of 0.1. At last, the entropy for the `libart.so` library is $H(X_{libart.so}) = 1$. This means, for the ten times execution of this app, `libart.so` is mapped into different addresses. We calculate the average entropy for all loaded libraries in application A. It is defined as $R(A) = \frac{\sum_{m \in M} H(X_m)}{|M|}$. We measure $R(\mathcal{A})$ on a simple app ($\mathcal{A}$) (generated by the blank app template of Android Gradle 1.2.3 [2]). App $\mathcal{A}$ contains 109 native libraries at runtime, and 108 of them are shared libraries which are inherited from `zygote`. We execute the app without and with BLENDER protection ten times respectively, and record the memory layout after the startup stage. Table 3 shows the results of the average entropy. The average entropy of original app, app with BLENDERLRM only, and app with full BLENDER support are 0.005, 0.981, and 0.991 respectively. The average entropy of the original system is quite low, which shows that there is nearly no randomness in the original app. After using BLENDER with library randomization module, the entropy increases significantly. When adding with the ART runtime randomization module, the entropy increases about 0.1. Although the increased entropy of BLENDERART is small, but the security gain is considerably high because of the large range of executable regions.

Table 3: Entropy Analysis Results.

Mode	App Entropy R(A)
Original App	0.005
BLENDERLRM Only	0.981
BLENDERLRM and BLENDERART	0.991

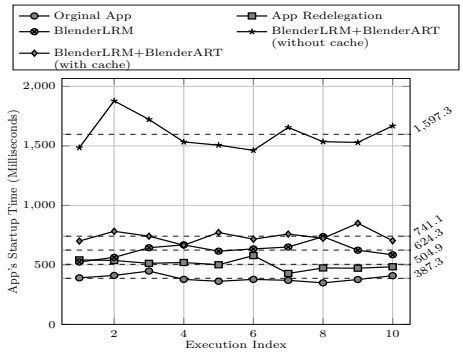

Fig. 8: App's startup time.

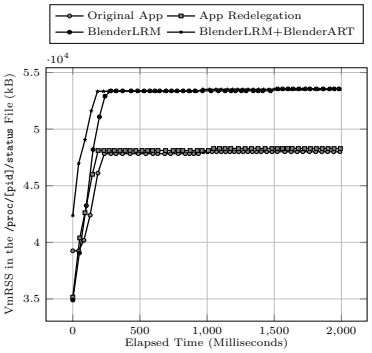

Fig. 9: Memory usages at the startup of apps for different setups.

5.2 Performance

Startup Time Because BLENDER conducts the library and ART randomization when app is first started, we want to evaluate its overhead in terms of the startup time delay, which is crucial for user experience. To quantify the startup time, we conduct experiments on a simple app. We create the app targeting Android 5.1 with one activity (generated by the app template of Android Gradle 1.2.3 [2]). In the app, we override `attachBaseContext` methods in the activity and log the current time (t_1). To accurately calculate the startup elapsed time, we use a UI/application exerciser (monkey tool) to launch this application and record the time (t_0) by reading the `$EPOCHREALTIME` value. $t_1 - t_0$ represents the elapsed time from launch time into application context. We measure the startup time of the original app, app only with the bootstrap module (app re-delegation), app with BLENDERLRM, app with the whole BLENDER without BLENDERART cache, and finally, app with the optimized BLENDER with cache. We execute the app for ten times and record the results. Figure 8 illustrates the startup time (in millisecond) for each launch and the average numbers of different setups. First, because app re-delegation needs to load the app at runtime, it introduces about 120 ms overhead. Second, without using the cache, BLENDERART needs to execute code patching for each time. The startup time is about 1.5 seconds, which is noticeable by normal users. For the system with cache, the startup time is about 740 ms and incurs about 360 ms overhead, which is comparable with Morula [32]. It is worth noting that, this overhead only affects at the app's first startup time (cold start), and will not affect the following launching of the app (warm start) if the app is not killed due to low mem-

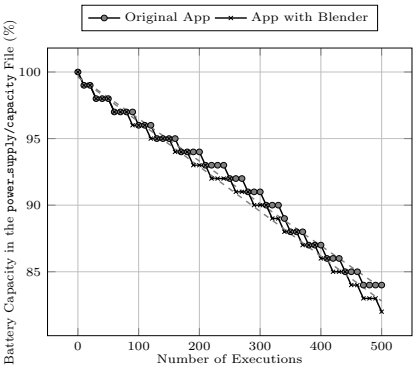

Fig. 10: Battery capacity after multiple executions.

	Baseline	BLENDERLRM	Full BLENDER
CPU	35915	36480	35969
Memory	13900	13846	14653
I/O	5874	5893	5900
2D	330	330	298
3D	1967	2019	1981
Total	57986	58568	58801

Table 4: Benchmark scores.

ory. Moreover, unlike Morula [32], this delay only applies to apps with our protection, and does not affect other apps.

Runtime Overheads BLENDER provides self-randomization capability to apps and the randomization process happens at the app's startup time, it will not affect the runtime performance. We use the Quadrant Standard Edition v2.1.1 to measure the general purpose benchmark for CPU, memory, I/O, 2D, and 3D graphics. Because we cannot get the source code of the benchmark tool, we use apktools [53] to repackage the app and add the BLENDER protection for our evaluation. Table 4 illustrates the benchmark results. Because of startup time randomization, the benchmark results are nearly same.

Memory Overheads We also evaluate the memory usage at runtime for the original app, app with re-delegation, app with BLENDERLRM only, and app with BLENDERLRM and BLENDERART. We create a script to monitor the `/proc/[pid]/status` file which contains all memory information at runtime. Figure 9 shows the VmRSS sizes during the start time to 2000 milliseconds. VmRSS (virtual memory resident set size) represents the portion of memory occupied by a process in memory. At the first 250 ms, the VmRSS value increases from a low level and then becomes stable. The VmRSS values of BLENDERLRM only and BLENDERLRM/ART together are nearly same at runtime, and introduces about 5513 kB (11.5 %) overhead. BLENDER incurs less memory overhead compared to previous mitigation solution Morula [32] by patching the Android which introduces 13 MB for each app.

Battery Overheads Battery consumption is important for mobile devices. Because BLENDER conducts randomization at the startup time of an app, BLENDER will consume more battery than original settings. We conduct the following experiments to measure the battery overhead of the BLENDER system. Firstly, we use a fully charged device (Nexus 5) and set screen as "always on". Then, we launch and close the experiment app (the same app in the performance evaluation experiments) for 500 times with 10 seconds interval. For each execution, we record the current time and the current battery capacity. For the BLENDER evaluation, we use a fully charged device to execute the experiment app with BLENDER installed and record the battery capacity. For both ex-

periments, we obtain the battery capacity by reading the `/sys/class/power_supply/battery/capacity` file. Figure 10 illustrates the remaining battery capacities after multiple number of executions for two apps, and we plot their linear fit as two dashed lines. There is only 1 % more power consumption after 500 executions for about 6400 seconds which is comparable with the Morula system. Therefore, the battery overhead is negligible for normal users.

6 Discussion

Limitations of Caching Patched ART Code To balance security gain and performance overheads, our design caches patched ART code (i.e., `boot.oat`) in a pool. Although attackers can try multiple times to guess the offsets of the `boot.oat` file in the pool, they still cannot obtain the current offset by previous executions or by other side channels. However, this technique decreases the entropy of the randomization. To achieve high entropy randomization, developers can disable utilizing cached code and conduct randomization at runtime. Although this may introduce more startup overhead (less than two seconds), this is still acceptable for apps with high security requirements. Also we may randomize the `boot.oat` file at runtime, such as when the app is idle in the background, to reduce the startup time delay. However, this may need deep understanding of the app's logic and more involvement from the app developer's side.

BLENDER on Other Architectures Because most mobile devices are based on the ARM architecture (99 % according to report [3]), our ret2art attack and BLENDER system are implemented on an ARM-based device. In fact, the latest Android version support other architectures including x86 and MIPS. The only differences are architecture specific source code. Therefore, the weakness of ASLR introduced by `zygote` process creation model still exists. And one can easily write code to initiate ret2art attack on those platforms. For the BLENDER system, one can port to other architectures by translating architecture specific ARM assembly code to the corresponding architecture.

Randomization within Shared Library Another limitation of current system is that BLENDER does not randomize the functions inside a library. This means that if there is a memory leak vulnerability, attackers could know the base address and compute offsets of ROP gadgets to launch an ROP attack. To overcome this potential security problem, we can use method proposed as binary stirring [52] to randomly rewrite the binary code blocks of loaded libraries. However, this method requires disassembling, rewriting and assembling all loaded libraries at launch time of an app. This will introduces considerable overheads. Therefore, we leave it as our future work.

7 Related Work

Security problem in memory is one of the oldest issues in computer security. Previous studies [28,47,50] summarize the attack and defense solutions on memory security. Our work focuses on attacking and protecting weakened ASLR mechanism on Android.

Attacks and Defenses of ASLR Mechanism Because modern operating systems have implemented/deployed ASLR and DEP defense mechanisms by default [24, 30, 48], attackers try many bypassing techniques from different perspectives. Several works [34,

43] focus on bypassing by brute-forcing method. Moreover, leaked pointers, type confusion and use-after-free bugs can be also exploited [41, 42]. Furthermore, by repeatedly abuse a memory disclosure, attackers can map an application's memory layout on-the-fly with dynamically discovered gadgets [44]. There are many return oriented programming techniques described in several papers [34, 36]. Moreover, some researchers [22, 52] proposed to protect memory by introducing high randomization entropy.

Attacks and Defenses on Android Compared to traditional desktop operating system, mobile OS have their domain-specific architecture design which introduces new attack surfaces. For Android, many researches discuss about security issues on permission mechanism [19, 29, 31] of Android. In addition, some work exploit underlying system components on Android [15, 21, 27, 35, 38, 39, 49, 51, 57]. Because there are a number of malware on Android, Zhou et al. [59] provide the characterization and evolution of Android malware. In addition, some systems propose to prevent [46] or detect malware [45]. Moreover, researchers also propose both static analysis systems [26,33,56,58] and dynamic analysis systems [25,55] to assist malware researchers to understand the malicious logic.

Mitigating ASLR on Android Because of the limitations of mobile system, the design and implementation of ASLR mechanism is rather weak. Retouching [17], Morula [32] and LR^2 [18] are three systems which discuss attacking techniques and provide mitigation solutions. Retouching can randomize pre-linked code when deploying Android applications. However, Retouching does not resolve the issue of uniform memory layout introduced by the `zygote` process creation model. Morula proposes a patch for Android source code to randomize all layout of apps after forking from `zygote` and also introduces low overheads. LR^2 proposes a leakage-resilient layout randomization method by introducing transformations as passes on compiler. However, they all have a major deployment issue. Current systems needs to modify Android source code to achieve randomization functionality. Users should replace original firmware with the customized system. Moreover, the system should keep up with the latest Android version with new features and bug fixes. Hence, because of the deployment issues, both users and developers cannot easily adopt this mitigation solution. Our work provides a non-invasive methodology for both developers and users.

8 Conclusion

In this paper, we show that the ASLR protection on Android is weakened due to the `zygote` app creation model. Moreover, we demonstrate a newly discovered attack surface introduced by the ART runtime, and present a novel way to exploit the weakness of the ASLR protection and this new attack surface. Then we propose a non-invasive user-level solution called BLENDER which does not need framework modification. BLENDER *self-randomizes* address space layout for apps, hence raising the bar for successfully bypassing the weakened ASLR protection on Android. We discuss the design, implementation, and present the effectiveness and performance overhead of our system.

References

1. Adobe Flash Use-after-free Vulnerability. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3108.
2. Android plugin for gradle. https://developer.android.com/intl/ru/tools/building/plugin-for-gradle.html.
3. Arm designs one of the world's most-used products. http://www.bloomberg.com/bw/articles/2014-02-04/arm-chips-are-the-most-used-consumer-product-dot-where-s-the-money.
4. CVE-2013-0912. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0912.
5. CVE-2015-1233. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1233.
6. Distribution of android platform versions. https://developer.android.com/about/dashboards/index.html.
7. Ropgadget - gadgets finder and auto-roper. http://shell-storm.org/project/ROPgadget/.
8. Samsung galaxy knox android browser rce. https://www.exploit-db.com/exploits/35282/.
9. Stagefright (bug). https://en.wikipedia.org/wiki/Stagefright_(bug).
10. `codegen_util.cc` file in AOSP. https://android.googlesource.com/platform/art/+/android-6.0.0_r26/compiler/dex/quick/codegen_util.cc.
11. `dex_preopt_libart.mk` file in AOSP. https://android.googlesource.com/platform/build/+/android-6.0.0_r26/core/dex_preopt_libart.mk#36.
12. `quick_entrypoints_arm.S` file in AOSP. https://android.googlesource.com/platform/art/+/android-6.0.0_r26/runtime/arch/arm/quick_entrypoints_arm.S.
13. VLC media player 2.0.4 suffers from buffer overflow. https://trac.videolan.org/vlc/ticket/7860.
14. V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupé, M. Polino, P. de Geus, C. Kruegel, and G. Vigna. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *NDSS*, 2016.
15. A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. What the app is that? deception and countermeasures in the android user interface. In *SP*, 2015.
16. D. Blazakis. Interpreter exploitation. In *WOOT*, 2010.
17. H. Bojinov, D. Boneh, R. Cannings, and I. Malchev. Address space randomization for mobile devices. In *WiSec*, 2011.
18. K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi. Leakage-resilient layout randomization for mobile devices. In *NDSS*, 2016.
19. S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *NDSS*, 2012.
20. S. Chamberlain and I. L. Taylor. The gnu linker, 1991.
21. Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In *USENIX Security*, 2014.
22. Y. Chen, Z. Wang, D. Whalley, and L. Lu. Remix: On-demand live randomization. In *CODASPY*, 2016.
23. S. Designer. return-to-libc attack. *Bugtraq, Aug*, 1997.
24. T. Durden. Bypassing pax aslr protection. *Phrack Magazine*, 59, 2002.
25. W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM TOCS*, 2014.
26. W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *USENIX Security*, 2011.
27. W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *SP*, 2009.
28. Ú. Erlingsson. Low-level software security: Attacks and defenses. In *FOSAD*. 2007.
29. A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security*, 2011.

30. C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security*, 2012.
31. M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS*, 2012.
32. B. Lee, L. Lu, T. Wang, T. Kim, and W. Lee. From zygote to morula: Fortifying weakened aslr on android. In *SP*, 2014.
33. L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *CCS*, 2012.
34. T. Müller. Aslr smack & laugh reference. In *Advanced Exploitation Techniques*, 2008.
35. C. Mulliner, J. Oberheide, W. Robertson, and E. Kirda. Patchdroid: scalable third-party security patches for android devices. In *ACSAC*, 2013.
36. Nergal. The advanced return-into-lib (c) exploits: Pax case study. *Phrack Magazine, Volume 0x0b, Issue 0x3a*.
37. O. Peles and R. Hay. One class to rule them all: 0-day deserialization vulnerabilities in android. In *WOOT*, 2015.
38. A. Razeen, B. Wu, and S. Cheemalapati. Spandex: Secure password tracking for android. In *USENIX Security*, 2014.
39. C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu. Towards discovering and understanding task hijacking in android. In *USENIX Security*, 2015.
40. R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM TISSEC*, 2012.
41. G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib (c). In *ACSAC*, 2009.
42. F. J. Serna. The info leak era on software exploitation. *Black Hat USA*, 2012.
43. H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *CCS*, 2014.
44. K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *SP*, 2013.
45. M. Sun, M. Li, and J. C. S. Lui. Droideagle: seamless detection of visually similar android apps. In *WiSec*, 2015.
46. M. Sun, M. Zheng, J. C. S. Lui, and X. Jiang. Design and implementation of an android host-based intrusion prevention system. In *ACSAC*, 2014.
47. L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *SP*, 2013.
48. P. Team. Pax address space layout randomization (aslr), 2003.
49. D. R. Thomas, A. R. Beresford, and A. Rice. Security metrics for the android ecosystem. In *SPSM*, 2015.
50. V. Van der Veen, L. Cavallaro, H. Bos, et al. Memory errors: the past, the present, and the future. In *RAID*, 2012.
51. T. Vidas, D. Votipka, and N. Christin. All your droid are belong to us: A survey of current android attacks. In *WOOT*, 2011.
52. R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *ASIACCS*, 2012.
53. R. Winsniewski. Android–apktool: A tool for reverse engineering android apk files, 2012.
54. R. Xu, H. Saidi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *USENIX Security*, 2012.
55. L.-K. Yan and H. Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *USENIX Security*, 2012.
56. M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *CCS*, 2014.
57. M. Zheng, M. Sun, and J. Lui. Droidray: a security evaluation system for customized android firmwares. In *ASIACCS*, 2014.
58. M. Zheng, M. Sun, and J. C. Lui. Droidanalytics: a signature based analytic system to collect, extract, analyze and associate android malware. In *TrustCom*, 2013.
59. Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *SP*, 2012.
60. Y. Zhou, K. Patel, L. Wu, Z. Wang, and X. Jiang. Hybrid user-level sandboxing of third-party android apps. In *ASIACCS*, 2015.