From Zygote to Morula: Fortifying Weakened ASLR on Android

Byoungyoung Lee[†], Long Lu[‡], Tielei Wang[†], Taesoo Kim[*], and Wenke Lee[†]

[†]School of Computer Science, Georgia Institute of Technology
[‡]Department of Computer Science, Stony Brook University
[*]MIT CSAIL

In embryology, the **morula** *is produced by the rapid division of the* **zygote** *cell; in Android, each application process is a fork of the* **Zygote** *process.*

***Abstract*—There have been many research efforts to secure Android applications and the high-level system mechanisms. The low-level operating system designs have been overlooked partially due to the belief that security issues at this level are similar to those on Linux, which are well-studied. However, we identify that certain Android modifications are at odds with security and result in serious vulnerabilities that need to be addressed immediately.**

In this paper, we analyze the Zygote process creation model, an Android operating system design for speeding up application launches. Zygote weakens Address Space Layout Randomization (ASLR) because all application processes are created with largely identical memory layouts. We design both remote and local attacks capable of bypassing the weakened ASLR and executing return-oriented programming on Android. We demonstrate the attacks using real applications, such as the Chrome Browser and VLC Media Player. Further, we design and implement Morula, a secure replacement for Zygote. Morula introduces a small amount of code to the Android operating system and can be easily adopted by device vendors. Our evaluation shows that, compared to Zygote, Morula incurs a 13 MB memory increase for each running application but allows each Android process to have an individually randomized memory layout and even a slightly shorter average launch time.

I. Introduction

With over 1.5 million devices activated daily in 2013 [33], Android now owns the largest mobile user population around the globe. However, as past experiences have shown, once a piece of software or platform gains significant popularity, it becomes a hot target for financially or politically motivated attackers. Despite the tremendous efforts by the security research community in reinforcing the security of Android, so far only a few *categories* of security issues pertaining to Android have been thoroughly studied and addressed. Most of these issues are due to the *vulnerable applications* and specific to the *high-level* design concepts adopted in Android, such as the widely debated permission model.

In this paper, we describe a new Android security threat and propose a countermeasure. Unlike previous studies, this threat is enabled by a *low-level* design inside the *Android operating system*. The design was intended to improve the responsiveness of applications at launch-time, but it adversely affects the effectiveness of Address Space Layout Randomization (ASLR). The root cause of the new threat lies in the core routine that each application process goes through when created in Android. Distributed in bytecode form, Android apps rely on the Dalvik Virtual Machine (DVM) for interpretation and runtime support. However, launching a new DVM instance for each new app process can be both time- and resource-consuming. Given the severe constraints of CPU power and memory space on the early generation of mobile devices, Android designers chose to spawn every app process by forking a master process, the Zygote process, which is created at device boot-time and contains a full DVM instance with frequently used classes preloaded. Zygote effectively shortens apps' launch-time. However, a side effect of this long-existing design—all app processes running on a device share a largely identical memory layout inherited from Zygote—poses a great security threat to Android's recently adopted ASLR [22].

ASLR, when properly implemented, loads the code and data of a program into random memory locations such that the process memory layout cannot be deterministically inferred from other executions of the same program or from other co-located processes using the same shared libraries. However, the Zygote process creation model indirectly causes two types of memory layout sharing on Android, which undermine the effectiveness of ASLR: 1) the code of an app is always loaded at the exact same memory location across different runs even when ASLR is present; and more alarmingly, 2) all running apps inherit the commonly used libraries from the Zygote process (including the `libc` library) and thus share the same virtual memory mappings of these libraries, which represent a total of 27 `MB` of executable code in memory.

Both types of memory layout sharing (or leakage) are found on all versions of Android to date. These memory leakages enable attackers to easily bypass ASLR, exposing Android apps to critical attacks such as return-oriented programming (ROP). We identify not only local but also remote attack vectors, which obviate the common need for a pre-installed app to carry out attacks on Android devices. By launching attacks at popular apps, including the Chrome browser and VLC media player, we demonstrate in Section III the critical and realistic nature of this threat. It is also worth noting that we found similar issues in the Chromium OS and Chrome Browser for desktops. For simplicity, this paper only discusses the issues in the context of Android.

Beyond the identification of this critical threat, this paper also proposes a simple and effective countermeasure, namely

Morula, which is implemented via a small modification to the Android OS. Morula eliminates the dangerously high predictability of the memory layout in Android by allowing processes to have individually randomized memory layouts. An intuitive way to carry out this straightforward idea is to abandon the Zygote process and create each application process from scratch. In this case, the dependent libraries, application code, and data are loaded into freshly allocated memory regions each time a process is being created, and therefore the existing ASLR in Android is able to independently randomize the memory layout for each process. However, this intuitive approach (requiring a cold start for each application) can incur a prohibitive performance overhead that prolongs the average app launch time by more than 3.5 seconds. In comparison, Morula allows Android ASLR to achieve the same level of security and shows a slightly shorter average launch time. The core idea of Morula is to adaptively maintain a pool of Zygote processes with distinct memory layouts (i.e., Morula processes), so that when an app is about to start, a unique Morula instance is available as a one-time and pre-initialized process template in which the app can be loaded. Creating and initializing the Morula processes in advance moves the time-consuming operations (e.g., library and class loading) out of the time-critical app launching phase and consequently reduces the app launch time. We also devise two optimization strategies: on-demand loading and selective randomization, to further limit Morula's impact on app responsiveness and system memory usages.

We have built a prototype of Morula on Android 4.2 and evaluated its effectiveness and performance overheads. When Morula is in use, an attacker can no longer bypass Android ASLR with a single attempt; rather, they have to try, on average, a minimum of 18,000 times. Moreover, Morula meets, and sometimes exceeds, its performance goal of keeping app launches as fast as when Zygote is used. However, Morula incurs an increase of 13.7 MB memory usage per running app, as a trade-off for the improved security and optimized app responsiveness. Morula imposes no obvious overhead to other resources, including battery power.

In addition to identifying a new threat and proposing a countermeasure, this paper also conveys an important message: the modifications and extensions made to Linux by Android and other emerging OSes can open new and unique attack surfaces. Some feature- or performance- oriented customizations do interfere or even break existing security mechanisms. Therefore, the new designs incorporated by these OSes need careful security scrutiny. This message contradicts the widely held belief that the security of the low-level Android OS is the same as the security of Linux (because Linux is the OS core of Android).

To summarize, our contributions are as follows:

- Leveraging Android's weakened ASLR, we devised two realistic attacks on real apps, which break ASLR and achieve ROP on current Android systems.
- We designed Morula as a practical countermeasure and implemented it as a small and backward-compatible extension to the Android OS for easy adoption.
- We conducted thorough evaluations of Morula: measuring its enhancement on ASLR and analyzing its overheads in terms of app launch delays, memory use, battery life, etc.

The rest of the paper continues as follows. Section II provides the necessary background information about the new threat and our solution. We discuss the details about the threat and illustrate the possible attack scenarios in Section III. Morula's design and implementation is explained in Section IV, followed by an evaluation in Section V. We then discuss implications and limitations in Section VI, and compare the related work in Section VII. Section VIII concludes the paper.

II. Background

A. App Process Creation on Android

Android inherits its operating system core from Linux. To overcome the unique constraints facing mobile platforms and enable new mobile-specific features, Android's design introduced a new middleware layer on top of traditional Linux and customized the designs of several system management components within Linux. Among these customizations is the Zygote process creation model. At the time of its introduction, Zygote represented a reasonable design choice for improving responsiveness and performance of apps, but we find the design conflicts with ASLR, the critical security mechanism recently adopted by Android. Next we provide the background knowledge on the Zygote process and other factors that contribute to the root cause of the new security threat.

Android apps are packaged in Dalvik bytecode form and rely on the Dalvik Virtual Machine (DVM) for interpretation in runtime. Compiling apps into bytecode for distribution naturally brings cross-platform compatibility to Android. Moreover, compared with native code, bytecode generally produces smaller executable files and guarantees better runtime security, which is favorable to mobile devices. However, similar to other interpretation-based execution, Android's bytecode-based apps can suffer from the performance overhead incurred by the virtual machine during runtime as well as launch-time. To speed up the execution of bytecode, DVM prioritized performance in its design. One example is the adoption of just-in-time compilation at a very early stage in its development.

To improve app launch-time performance, Android employs the Zygote process creation model to avoid the long delay that every app would otherwise have to undergo (i.e., waiting for a fresh DVM instance to be created). Instead of repeating the creation and initialization of DVM for each app, Android does so only once during OS boot-time and keeps the resulting DVM instance as a template process, namely the Zygote process, from which all app processes will be "forked." The Zygote process not only hosts an initialized DVM instance but also loads a large pool of commonly used classes and libraries. As child processes of Zygote, every app inherits the established process context without performing the initialization themselves, and thus takes a significantly reduced amount of time to start. Zygote also enables a system-wide sharing of the memory pages that contain the preloaded code and data, reducing global memory usage. It is also worth noting that Zygote-like process creation and class preloading is not only seen on Android, but also used by the Chrome browser and Chromium OS.

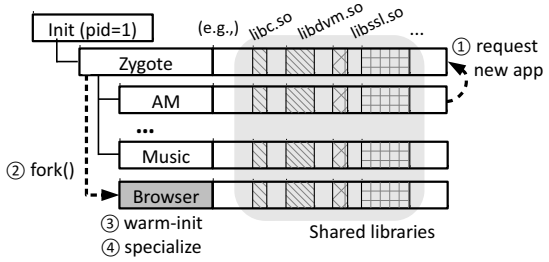


Fig. 1. Zygote process creation model with memory layout representations when launching a browser app. AM represents Activity Manager. The gray region displays a snapshot of the Android's address space layouts for running apps, directly inherited from the Zygote process; each pattern represents a shared library, such as `libc.so` or `libssl.so`, which is located identically among all running apps.

Figure 1 illustrates how Zygote assists with creating a new application process. Upon receiving an application launch request, the Activity Manager in the Android Runtime signals the Zygote process through an IPC socket (①). The latter then invokes the `fork()` system call, creating a new process to host the app (②). Since the DVM instance is already created and loaded with the common libraries by the parent process, the forked child directly proceeds to the `warm-init` stage, where only a few initializations that cannot be done in advance are completed (③). As the final step of the app launch phase, namely `specialization`, the forked process loads the actual app per the information passed from the Activity Manager (④). The UID and GID are set at this time, marking the beginning of app-specific security enforcement. Control is then handed over to the application code via a call to the `main()` method of the Activity Thread.

While reducing the app launch-time delay significantly, the Zygote process creation model poses an obvious side effect to all app processes: as the forked children of Zygote, they all end up sharing not only the common memory content but also the same memory layout. Figure 1 also shows that the shared system libraries are loaded at the same memory locations across different processes. This side effect is by no means harmful by itself but improves the system-wide memory space efficiency. However, the nearly identical memory layout in all processes can be leveraged by an attacker, who, knowing the code addresses in one app can bypass ASLR and launch return-oriented programming attacks against any vulnerable app on the same device[1]. The severity of this threat is elevated by the fact that native code is commonly used in many popular and important apps, each of which heavily relies on native code components for rendering HTML, playing media, accelerating graphics, etc., and such code is historically known for being prone to memory corruptions and address information leakages.

B. Address Space Layout Randomization on Android

ASLR is an attack mitigation technique used by all major commercial operating systems today. It allocates memory regions for both code and data at nearly random locations, making it statistically difficult to predict the memory address of any executable code and writable data. Therefore, it can significantly lower the chance of locating code gadgets in memory, without which return-oriented programming attacks cannot succeed. Although a number of evasion techniques have emerged, most of them require a particular app to leak its address information and have an additional memory corruption vulnerability at the same time, which is a difficult requirement to satisfy. In reality, ASLR, along with data execution prevention, forms today's most effective and practical defense against a broad range of control-flow hijacking attacks.

Android started supporting ASLR only recently, after the OS had experienced a remarkable growth in user population and seen an increasing demand for better security. Despite apps being mainly written in a strong-type language, memory leaks and control-flow hijacking attacks are still more than likely to happen on Android, as shown by the continuous stream of new rooting exploits and vulnerability disclosures [1, 3]. This is partially because of the large amount of native code executing inside every app process. First, Android allows apps that are built with NDK (Native Development Kit) to link native libraries and invoke functions in such libraries through JNI-like interfaces. Apps with complex features, high performance requirements, and legacy codebase tend to have heavy dependence on native code, including many popular apps, such as browsers, media players, games, etc. Second, the runtime libraries and the DVM are implemented in C/C++ and run natively inside app processes without any memory safety assurance. Native code from these two sources, loaded into the same process as the rest of the app, represents a large attack surface, which would have been left exposed if ASLR was not supported or was bypassed.

The adoption of ASLR in Android took several version iterations to complete [32]. Android 4.0 was the major upgrade that first introduced ASLR to the mobile OS. The scope at that time only covered the shared libraries shipped with the OS, such as `libbionic` (Android's implementation of `libc`). Therefore, only these libraries were loaded into randomly allocated memory locations in each process, whereas other code that may also be used as sources for mining ROP gadgets, such as system executables and apps' native libraries, always remained at the same memory location when loaded, creating an easy way to bypass ASLR. As Android added support for PIE (Position Independent Executable) in Android 4.1 a year later, its ASLR finally expanded to cover the remaining libraries, the dynamic linker, and all other executables compiled with the PIE flag. The heap space was also randomized, marking the end of a complete port of ASLR from Linux to Android [22] and winning praise from the security community [17].

Although a large amount of effort was spent to fully incorporate ASLR to Android, unfortunately, the effort failed to include a careful examination of the existing system designs that could cause conflicts with ASLR. In the next section, we will explain these in detail: how the Zygote process creation model—a design choice made to boost app performance—can significantly reduce the effectiveness of ASLR and even allow attackers to bypass the critical security mechanism.

[1]Although the memory layout sharing issue was previously known, it was not considered to be harmful to ASLR as it was seemingly difficult for attackers to take advantage of this issue [28, 31]. However, we found that this can be easily abused, even in popular Android apps (see Section III).

III. Exploiting Android's ASLR

The effectiveness of ASLR in mitigating control-flow hijacking attacks hinges on the fact that attackers possess no knowledge about the memory layout of a target process and have no better way to gain such knowledge than brute force. Naturally, the evasion efforts in various types of OSes so far have all focused on defeating the protection by either stealing the secretive memory layout information through information leaks and side channels [25], or brute forcing on those platforms that cannot randomize memory allocations with enough entropy [39]. However, either direction is by no means easy to pursue; exploitable vulnerabilities that leak memory address information are fairly hard to come by, and a single failure on brute force attempts often results in an application crash that is easily detectable by users. Therefore, successful ASLR evasions have only been seen occasionally in real world attack incidents. As for Android, there has not been a single incident known to the public where its ASLR was bypassed and a ROP exploit was mounted consequently. However, we believe that it will not take long before attackers start exploiting the uniform memory layout among apps for ASLR evasions as it significantly eases the prerequisites and difficulties of launching such attacks on Android.

To explain how much easier evasions have become on the Zygote-forked app processes, we first show the major obstacles that an attacker has to face in order to bypass ASLR on other platforms but are weakened on Android by Zygote. We then systematically discuss the negative impact of Zygote on ASLR. By demonstrating our attacks on real apps, we present two general scenarios where the negatively impacted ASLR can be defeated either remotely or locally, both archiving ROP capabilities. In the end of the section, we provide a quantitative analysis to show that the advantage gained by attackers through exploiting the uniform address space layout is indeed significant enough to carry out realistic attacks.

A. ASLR Bypasses Made (Relatively) Easy

Attempts to bypass ASLR usually serve as the first step of a bigger attack plan with the end goal of launching ROP or other types of control-flow hijacks. This is because exploits that solely obtain the memory mapping information of a process do not yield much gain for attackers, unless the mapping information, such as the load address of a library, can advance another concurrent exploit within the same process context, such as a buffer overflow. Together, two or even more coordinating exploits eventually diverge the original control-flow of the vulnerable program in a way to carry out malicious or unexpected activities.

In reality, such attacks involving chained exploits on different types of vulnerabilities are extremely difficult to design and execute, mostly for two reasons:

- It is quite rare to find an address information leak vulnerability and a memory corruption vulnerability within the same program;
- Even if such vulnerabilities do exist, a hard-to-craft exploit is needed to sequentially trigger the vulnerabilities and simultaneously channel the leaked address information to the second exploitation phase, without crashing the target process.

These two obstacles have maintained a very high bar to prevent ASLR bypasses. As a result, the rare events of successful ASLR bypasses have always drawn the security community's attention and have been seriously handled by software vendors [44].

However, we found that both obstacles on Android are no longer as difficult to circumvent as they are on other platforms, due to the system-wide uniform address layout resulting from the Zygote process creation model. In fact, both obstacles have been weakened to an extent where bypassing ASLR through information leakage becomes not only realistic but also practical. Since the memory addresses of all shared libraries and code, despite the presence of ASLR, are identical among all Android app processes running concurrently on the same device, the memory layout information of one app can be easily inferred from that of any other app or a different run of the same app. This cross-app and cross-run sharing of critical memory layout paves the way to bypass the aforementioned obstacles and further facilitates the crafting of exploits that streamline ASLR compromise and control-flow hijacking.

First of all, the search for the required vulnerabilities no longer need to be confined within a single app. The relaxed search criteria dramatically increases the chance of satisfying the exploit prerequisites, to the level of finding apps that contain *either* address information leaks *or* control-flow hijack vulnerabilities—both types are retrievable from vulnerability disclosure lists. Furthermore, an attacker who already managed to gain a footprint on the victim's device (e.g., installed a trojan or controlled an app) can launch a control-flow hijacking attack at any vulnerable app without having to find any address information leakage. Second, the exploitation process can now span several temporally separated stages, which obviate the need to craft a single master exploit that has to drive the entire attack continuously. Therefore, rather than taking the complicated path that involves chaining two or more individual exploits together and finding an explicit communication channel within the victim process, attackers now can proceed progressively by executing disconnected exploits within different vulnerable apps and coordinating them off-site. Finally, the uniform memory layout among apps allows for multiple exploit attempts as well as reuse of the leaked information. Less stable exploits are given multiple chances to succeed, even at the cost of crashing the targets. Moreover, once the address information is obtained, it can be reused by future exploits (as long as the device does not reboot) to resume an attack.

We also observed that the side effect of the Zygote process creation model, in addition to directly undermining ASLR security, has created a rich source of ROP gadgets in every app process on Android. Figure 2 shows the size of the `.text` section (i.e., executable code) in each shared library loaded by the Zygote process (tested on Android 4.2 with Galaxy Nexus), which translates into a total of 27 `MB` of executable code that is identically located in all app processes.

B. Attacks on Real Apps

We now propose two attack scenarios where Android's negatively impacted ASLR can be exploited to carry out ROP under separate threat models. For each scenario, we first

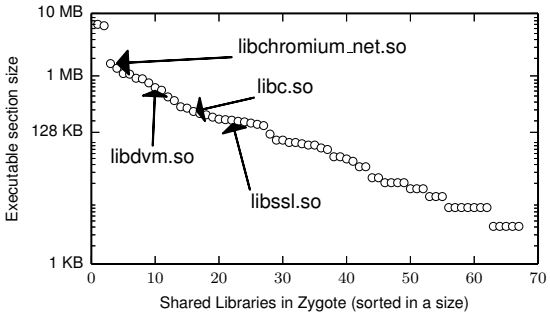


Fig. 2. The size of each shared library's executable section in the Zygote process. The x-axis represents the indices of the shared libraries and the y-axis represents the `.text` section size, at logarithmic scale.

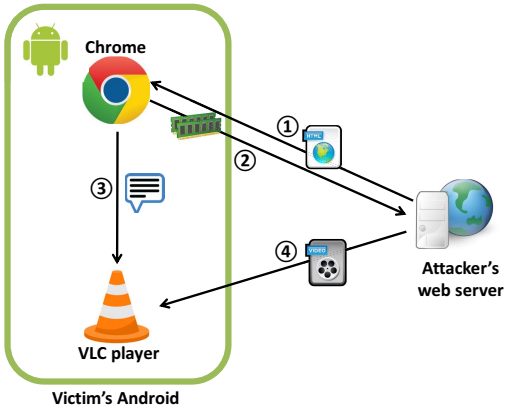


Fig. 3. A remote coordinated attack for the Chrome browser and VLC Media Player on Android. Each numbered step represents: 1) a malformed html file exploiting the information leak vulnerability, 2) leaked memory layout information, 3) URI intent, and 4) a malformed video file exploiting the control-flow hijack vulnerability which bypasses ASLR.

discuss its assumptions, setup, and attack workflow. We then demonstrate a concrete attack against popular apps that are found to be vulnerable, emphasizing the power of the attack and the realistic nature of the corresponding scenario.

Remote Coordinated Attacks: In this scenario, attackers do not have any prior presence on the targeted device, that is, the entire attack is carried out remotely. Two vulnerabilities are required: one memory address information leakage and one control-flow hijack. These can exist in different apps and must be remotely exploitable. The attacker needs to either actively or passively provide input, serving as exploits, to the vulnerable apps.

The general workflow of attacks in this category resembles that of the existing efforts to bypass ASLR on other platforms, but is significantly easier to realize for the reasons discussed in Section III-A. The workflow starts with exploiting an address information leakage in an app. Once the first exploit succeeds and the conditions to initiate the second exploit are met, the attacks can then hijack the control-flow of another app. Unlike exploiting ASLR on other platforms, this attack lowers the bar for exploiting the vulnerabilities, lasts through disconnected stages, and can survive unstable exploits.

To verify the feasibility of this attack scenario, we set out to design and execute an attack on real world apps. We used two popular Android apps: Chrome Browser (`com.android.chrome, version 25.0.1364.123`) and VLC media player (`com.overoz.vlc, version 0.1.0`), which contain the vulnerabilities that satisfy the requirements of our attack: an information leak on the Chrome Browser and a control-flow hijack on VLC. In particular, we found a known vulnerability in Chrome Browser (CVE-2013-0912 [4]) and built an exploit to retrieve part of the memory layout of the browser process. Our exploit triggers a type confusion error inside Chrome's SVG parser, leading to an out-of-bound memory read that leaks the memory addresses of the loaded libraries. This exploit is embedded inside a piece of JavaScript code that is served to vulnerable browsers when they connect to a web server under our control, which follows a typical and effective setup of remote exploits. The second vulnerability was found in the `H.263` decoder in VLC Media Player [16], through which our exploit triggers a buffer overflow with a malformed `.swf` file and hijacks the control-flow of the media player in a ROP fashion thereafter.

Figure 3 illustrates the detailed workflow of our attack, which is similar to that of drive-by download attacks on desktops. When a vulnerable Chrome Browser requests a webpage from a compromised server, the attacker pushes the JavaScript exploit to the browser as part of the requested HTML page (①). The exploit coerces the browser's SVG parser to mistakenly convert the type of an attacker-supplied data object into one with a larger size than the original type. As a result, the effective scope of the newly converted object expands into the subsequent memory region, which contains a pointer to a core system library that is preloaded by the Zygote process and shared by all app processes (`libchromium_net.so`). The exploit then simply reads the raw address in its new type object and sends the leaked address information back to the remote server (②). Next, a second exploit is dynamically generated at the server-side with the address information as an input. It can carry out an ROP attack using the VLC Media Player using the gadgets constructed from `libchromium_net.so`. The JavaScript exploit running inside the browser now initiates the second stage of the attack by issuing a request for opening the remote `.swf` file, which is dispatched to the VLC Media Player by the OS through an `Intent` (③). In this step, the VLC Media Player will be automatically launched by the `Intent` even if it was not running on the device. Finally, the media player downloads and starts rendering the exploit-loaded media file (④), during which the call stack is smashed with return addresses to ROP gadgets and the program counter is then hijacked.

Since designing sophisticated ROP attacks is out of the scope of this work, our proof-of-concept attack stops when the call stack is compromised and control-flow is hijacked, which represents an ideal starting point for launching meaningful ROP attacks. Given a huge `.text` section in `libchromium_net.so` (1.57 `MB` of executable code), we estimate the task of searching for ROP gadgets to be fairly easy, even if a large variety and amount of gadgets are desired.

As shown in the above proof-of-concept, remote coordinated attacks do not involve any user interaction except for the initial contact and can stay stealthy during the entire lifecycle,

a process similar to drive-by download attacks. Leveraging the significantly reduced difficulty of bypassing ASLR, our attacks demonstrate that launching purely remote attacks on Android platforms are not as hard as assumed by previously proposed attacks on mobile devices, most of which require a locally installed app to begin with. Our extended search for other suitable target apps for this attack suggests that the vulnerable apps are not uncommon on users' devices. A large number of apps either use old legacy code that have known vulnerabilities or implement error-prone features (e.g., media decoders and input parsers) using native code.

Note that Android's permission model may limit the capability of this ASLR bypass attack. If the permissions are enforced appropriately for an app, no meaningful malicious activities can be done given the limited permissions of the app. In practice, however, it is known that the least privilege principle is difficult to enforce as we have seen from over-privileged apps in Android [20]. Furthermore, low-level code executions in control-flow hijacking attacks could allow attackers to bypass permission checks by exploiting kernel vulnerabilities.

Local Trojan Attacks: Unlike the first scenario, local trojan attacks bypass ASLR by obtaining the memory address information via an unprivileged trojan app, rather than exploiting an information leakage remotely. The trojan app needs to be installed on the victim's device beforehand, most likely through social engineering means. To be appealing to the targeted user or to infect a large number of users, the trojan app may provide bogus features and does not ask for any permissions. In this case, the goal of the attackers is to bypass ASLR and hijack the control of another app, eventually escalating its privilege (e.g., the target app is privileged) or stealing protected data (e.g., the target app manages private or confidential data).

The workflow of this type of an attack is fairly straightforward. The local trojan app reads the address layout information using a simple native function. It can then attack any neighbor apps through locally exploitable control-flow hijacking vulnerabilities. In principle, the attack surface in this case is much larger than that of the previous attack scenario, due to the fact that not only network sockets but also all means of performing IPC on Android can be used for delivering exploits. Finally, the exploit uses a memory corruption error as a window through which the attack-supplied logic can be executed using ROP under the identity of the target app.

We built a simple trojan app for demonstration. It first finds out the base address of `libc.so` in its own process through a JNI call that returns a function pointer value. It then exploits the aforementioned vulnerability in the VLC Media Player through two separate attack vectors—an `Intent` and a `Binder` message, both requesting VLC to render a crafted media file. Once the ROP code starts executing in the process context of VLC, it effectively gives the trojan app (and the attacker) access to not only all granted permissions of the player, but also the victim app's private data in the file system, database, and even memory (where sensitive data exists in decrypted form).

In general, because of the uniform memory layout across all apps, local trojan attacks on Android can easily bypass ASLR to perform malicious activities, e.g., stealing permissions, peeking into another app's data, and even exploiting the system apps or the OS.

C. Quantitative Analysis of Attackers' Advantage

Without using our attack, bypassing fully effective ASLR requires two independent vulnerabilities, namely, a control-flow hijacking and an address information leak, in the same app. Therefore, the difficulty of exploiting ASLR via an app x can be represented by the probability of finding these two vulnerabilities simultaneously inside x:

$$\Pr_{\text{exploit}}(x) = \Pr_{\alpha}(x) * \Pr_{\beta}(x),$$

where $\Pr_{\alpha}(x)$ is the probability of finding a control-flow hijack vulnerability in x, and $\Pr_{\beta}(x)$ is the probability of finding an address information leak vulnerability in x.

Next, we derive the difficulty of bypassing ASLR for both of the attack scenarios we discussed earlier, in terms of the probability of finding for the exploitable vulnerabilities.

Remote Coordinated Attack Analysis: Since the two independent vulnerabilities can now exist in separated apps, the difficulty of exploiting the app x becomes:

$$\Pr^{\text{coordinated}}_{\text{exploit}}(x) = \Pr_{\alpha}(x) * \sum_{y \in \mathbf{S}} \Pr_{\beta}(y),$$

where $\mathbf{S}$ is the set of apps running on the same device as x. Note that apps in $\mathbf{S}$ must have the proper permissions to interact with the app x so that the information leak and control-flow hijacking attacks can be chained together (i.e., using `Intent` in our shown attack).

Thus, the advantage of the adversary A who chooses remote coordinated attacks over the conventional ASLR exploits can be defined as the difference between the difficulty of each exploitation:

$$\begin{aligned}\text{Adv}^{\text{coordinated}}(A) &= \Pr^{\text{coordinated}}_{\text{exploit}}(x) - \Pr_{\text{exploit}}(x) \\ &= \Pr_{\alpha}(x) * \left\{ \sum_{y \in \mathbf{S}} \Pr_{\beta}(y) - \Pr_{\beta}(x) \right\}.\end{aligned}$$

Obviously, $x \in \mathbf{S}$, thus $\text{Adv}^{\text{coordinated}}(\text{A}) \geq 0$, showing that remote coordinated attacks always have a non-negative advantage. $\sum_{y \in \mathbf{S}} \Pr_{\beta}(y)$ captures the fact that any app in $\mathbf{S}$ can be used as address information leak vector, and it grows as more apps are included in $\mathbf{S}$. The advantage can be quite large on a user device where many popular and complex apps are installed, due to the fact that such apps often contain vulnerability-prone code and allow interactions with other apps for better usabilities.

Local Trojan Attack Analysis: Having a local trojan app installed on the victim's devices obviates the need for finding and exploiting an address information leakage vulnerability. Therefore, the difficulty of launching a local trojan attack can be represented by:

$$\Pr^{\text{trojan}}_{\text{exploit}}(x) = \Pr_{\alpha}(x) * \Pr_{\gamma}(y),$$

where $\Pr_\gamma(y)$ is the probability of a user voluntarily installing the trojan app y.

Accordingly, the advantage of the adversary B who chooses local trojan attacks over the conventional ASLR exploits is:

$$\begin{aligned}\mathrm{Adv}^{\text{trojan}}(\mathrm{B}) &= \Pr^{\text{trojan}}_{\text{exploit}}(x) - \Pr_{\text{exploit}}(x) \\ &= \Pr_\alpha(x) * \{\Pr_\gamma(y) - \Pr_\beta(x)\}\,.\end{aligned}$$

The attacker's odds in this case are determined by the differences between $\Pr_\gamma(y)$ and $\Pr_\beta(x)$, and we argue that $\Pr_\gamma(y)$ would be bigger than $\Pr_\beta(x)$, given that human users are often considered as the weakest link in security and can be easily fooled by skilled attackers—in this case, well-disguised and attractive-looking trojan apps. Taking the well-known DroidDream trojan for an example, 250,000 Android users downloaded the app within three months [36].

IV. Morula: Effective and Practical Mitigation

Motivated by our identification and analysis of the weakened ASLR on Android, we propose Morula, a security-enhanced process creation model with simple design and optimized performance. Morula mitigates the negative side effects on ASLR by the current Zygote model that (accidentally) uniforms the layout of critical memory regions across all running apps. We now discuss the design of Morula, starting with an intuitive yet impractical idea, and then an effective and performant solution to reinforce Android ASLR.

A. An Intuitive Idea and Limitations

The simplest approach to removing the uniform memory layout from Android processes is to create each app process independently from scratch without using the Zygote as a template. This approach essentially reverts the process creation model back to that of Linux and abandons the efficient design choice made in the early days of Android that employs a pre-built and pre-initialized template process to speed up the launching for every app. We implemented and evaluated this intuitive approach in order to examine its feasibility, and indirectly, the design rationales of the Zygote process creation model on today's much improved mobile hardware.

We found an OS debugging feature in Android, namely `process_wrap`, that allows us to carry out this intuitive idea easily. `process_wrap` provides a hook into the Zygote process creation model, which is invoked immediately after each process is forked from Zygote and before an app image and any app-specific data are loaded into the forked process. Using this hook, we implemented a so-called Wrap process creation model, which forces every process forked from Zygote to regenerate their memory layouts through a call to `exec()` placed by the hook. Since `exec()` reloads the process image and the shared libraries, ASLR is now able to arrange the memory layout individually for each process as part of the `exec()` invocation. Figure 4 illustrates the workflow of this new model. At first, Activity Manager sends an app creation request to the Zygote process (①). Next, Zygote forks a new process, which then invokes `exec()` to load the master app process image (`/system/bin/app_process`), as a result of our `process_wrap` hook (②). A cold process initialization follows (②), which is not needed in the Zygote model (see Figure 1). Finally, the new process goes through the specialization step, where the process attributes are set and the control flow jumps to the target application's entry point (④). Now that Android follows the typical *fork-then-exec* process creation model, each process has a uniquely randomized memory layout, where shared libraries are no longer identically located in memory, as shown in the shaded area in Figure 4.

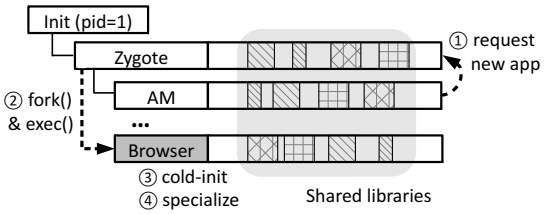


Fig. 4. The Wrap process management model when launching a browser app. Since a new process is invoked on every request of launching new applications in the Wrap scheme, all running processes locate shared libraries differently in ASLR-enabled Android.

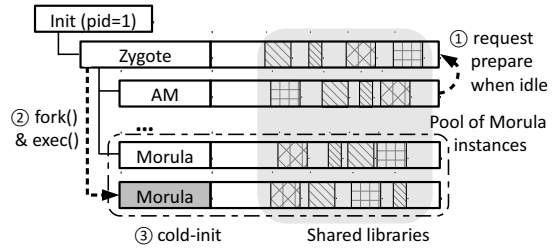
As expected, our experiment shows that, even on devices with the recent generation of hardware, replacing the Zygote process creation model with the Wrap model causes slowdown in app launches. Since the Wrap model goes through a full cycle of DVM creation and initialization, which is avoided in the Zygote model, it results in an average app launch time of 4.34 seconds, adding a 3.52-second wait to the current app-launching user experience, which we deem as unacceptable. In addition, the device boot-time suffers a 190% increase and amounts to 37.80 seconds.

Due to the prohibitive performance overhead, this intuitive approach is of little potential to be used as a practical solution. However, we gained useful insights into the OS while implementing this approach. The insights, especially those into the `process_wrap` hooks and app process creation, contributed to an efficient and easy-to-adopt design of Morula. More importantly, exploring this approach helped us understand that, despite its side effect in weakening ASLR, the Zygote process creation model is a performance-critical design in Android that cannot be simply removed. Security solutions that aim to eliminate its negative side effects should be able to largely conserve its performance benefits.

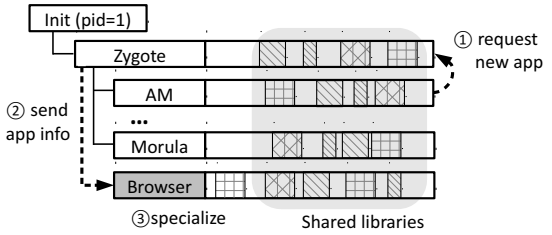
B. Performant Process Creation without Damaging ASLR

The goal of Morula is to enable app processes to have individually randomized memory layouts, as what the Wrap process creation model produced, while maintaining the app launch time at a similar level as that of the Zygote model. We observed that the optimal app launch time of the Zygote model is achieved by having the template process performing the common and time-consuming initialization tasks beforehand. We designed Morula to perform a similar task—speeding up app launches by keeping the time-consuming task out of the critical path—but in a different approach that does not reduce the effectiveness of ASLR or other security mechanisms.

Specifically, our Morula process creation model revamps the Zygote model by upshifting the role of the Zygote process into an abstract process template, which no longer directly



(a) A preparation phase



(b) A transition phase

Fig. 5. The Morula process creation model when launching a browser app. Morula, surrounded above by a dashed line, represents a template for any Android app. It is created when the device is in idle states (a), and then transformed into the browser (b). Since Morula prepares a pool of template instances having random memory layouts ahead of time, application launch on request is not only fast but also takes full advantage of ASLR.

spawns app processes but forks intermediate process templates, called Morula processes. Created ahead of time, Morula processes serve as initialized app execution hosts, which are allocated to start and host individual apps. Upon receiving an app launch request, a Morula process instantly loads the app and starts it without repeating the time-consuming initialization tasks. Therefore, the app launch time remains optimized. Moreover, each Morula process has to reload an independent memory image during the ahead-of-time initialization phase, which guarantees that no memory layout is shared among different app processes.

Figure 5 demonstrates the two phases where a Morula process is first prepared in advance and later used to start an app process, a browser in this case. The *preparation phase*, shown in Figure 5-(a), is initiated by the Activity Manager via a preparation request to the Zygote process, when the system is idle or lightly occupied (①). In turn, the Zygote process forks a child, which immediately makes a call to `exec()` to establish a new memory image with a freshly randomized layout (②). At the cold-init step (③), the new process constructs a DVM instance and loads all shared libraries and common Android classes, which would tremendously prolong the app launch time if not done in advance, as indicated by the significant slowdown caused by the Wrap model. At the end of this step, a Morula process is fully created, waiting for the request to start an app. Note that multiple Morula processes may be created when conditions permit, in order to accommodate an uncommon demand for starting several apps in relatively short time intervals. As shown in Figure 5-(a), each Morula process has a distinct memory layout, unlike the processes created under the Zygote model. Since Morula processes are created asynchronously to upcoming app launch events, they will enter sleep mode if not instantly needed and then move to the next phase when wakened.

A Morula process enters the *transition phase*, as depicted in Figure 5-(b), only when requested by the Activity Manager to start a new app (①). The request is routed to the Zygote process first, where a decision is made regarding if the app should be started in a Morula process or in a fork of the Zygote process. Having this option allows the Morula model to be backward compatible with the Zygote model, in order to carry out an optimization strategy called "selective randomization" (explained shortly). When a Morula process is chosen, the Zygote process forwards the app launch request through a `pipe` (②). With the concrete app information, the Morula process then starts specializing itself, loading the app package and setting the appropriate UID, GID, debugging flags, thread capabilities, etc. Finally, the Morula process hands over the control-flow to the app and transitions into a browser process (③). Unlike the preparation phase, the transition phase stands on the critical path of app launches, whose delay, if noticeable, can hurt the responsiveness of the apps and the launch-time performance in general. Thanks to the minimized workload that has to be executed in the transition phase, which is almost the same as what is required under the Zygote model when launching apps, the Morula model does not incur additional noticeable launch-time delays in most cases when compared to the Zygote model, as shown in Section V.

Morula's design can be viewed as a hybrid that combines the Zygote model and the Wrap model: similar to the Zygote model, it carries out in advance the time-consuming and commonly required tasks involved in creating app processes; learning from the Wrap model, it enforces a memory layout refresh when pre-creating the template processes. As a result, the Morula model achieves its goal of bringing individually randomized memory layout to Android apps while maintaining an optimal app launch time. However, if the design stays at this stage, our Morula model would have to pay non-negligible penalties on device boot time and memory usage efficiency.

We found that the only time when app launch requests can briefly outnumber the prepared Morula processes is during device boot-time. This is because multiple apps may be started simultaneously at boot-time, whereas a booted mobile device does not experience such dense requests for launching apps. This bottleneck results in a half-minute boot time, which can be unsatisfactory to some users. Additionally, with each process explicitly reloading the shared libraries and creating a private instance of the DVM, global sharing of common memory pages is not possible and therefore physical memory usage efficiency is reduced. However, thanks to copy-on-write memory pages enabled by the Linux kernel, the shared libraries, though loaded individually by each process, only have a single copy in physical memory, which does not cause additional space overhead. In fact, Morula only poses a moderate 13 `MB` overhead to the physical memory for each running application, which we believe can be digested by many modern mobile devices.

Nevertheless, we retrofitted our basic design of Morula with two optimization strategies, which brings the boot time and memory usages on par with the Zygote model. We now explain both strategies.

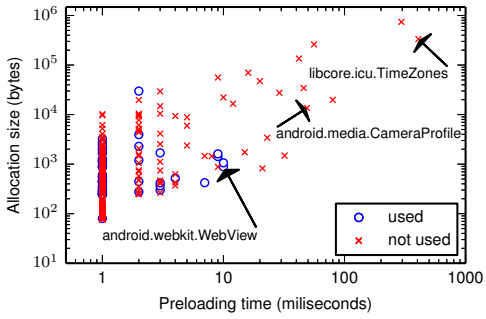


Fig. 6. An example of preloaded Dalvik classes and their run-time usages in Android Browser. The blue circles (204 in total) represent the Dalvik classes in-use while the red crosses (2,337 in total) represent the unused ones. The x-axis (class loading time) and the y-axis (memory image size) are both on a logarithmic scale.

C. Further Optimizations Exploiting System Characteristics

While Morula provides fast app launching time to the user, the performance of Morula can be further optimized using two optimization techniques: on-demand preloading and selective randomization.

On-demand Loading: This optimization strategy was inspired by our observation on the use-load ratio of Dalvik classes, which we acquired during our study of the time-consuming initialization tasks carried out when an app process is being created. We found that the construction and initialization of the DVM represent the single most resource-consuming task involved in app process creation. This task requires, among other things, preloading a list of 2,541 Dalvik classes, which are deemed to be commonly used by regular Android apps and thus worth preloading into all app processes. However, our analysis of a set of 110,014 free apps from the Google Play Market suggests otherwise: on average, each app only makes use of a small fraction, around 5%, of the preloaded Dalvik classes (i.e., 122 out of 2,541). Taking the Android Browser app as an example (`com.android.browser`), despite its heavy dependence on a broad range of classes, it only requires 204 out of the 2,541 preloaded classes, as shown in Figure 6, resulting in a waste of load time and memory space.

This extremely low use-load ratio of the preloaded classes introduces a large amount of unnecessary delay to the preparation phase in our Morula model, which happens at the creation of each Morula process. This adverse effect on performance is amplified when a rare shortage of prepared Morula processes is encountered (due to the boot-time burst of app launch requests), contributing to the previous discussed issue of the prolonged boot time under the Morula model. It is worth noting that under the Zygote model, boot-time is not affected as much because the preloading of classes and the entire DVM initialization only happens once per device boot.

Given that the overly broad class preloading is not suitable to Morula, our optimization strategy, on-demand loading, simply removes the workload of loading the 2,541 classes from the preparation phase, and instead relies on the DVM's dynamic class loading feature that dynamically loads a class when needed. As a result, on-demand loading addresses a performance bottleneck of the preparation phase, which enables a much higher throughput of Morula processes and significantly reduces the likelihood of having Morula process shortages even during device boot. As shown in Section V, the boot time is eventually improved by 51% on average. Additionally, the memory usage efficiency of apps is also improved due to the exclusion of unnecessary classes.

On the other hand, this optimization strategy may incur slight launch-time or runtime delays when on-demand dynamic loading is triggered for those classes that would have been preloaded otherwise. In average cases, based on our aforementioned statistics, disabling the preloading only leaves 122 classes to be loaded dynamically by a regular app, whose negative impact on performance is too small to measure. However, we have seen some real apps that have a large dependency on the preloaded classes, and they tend to be delayed by less than 0.5 seconds when launched (because these classes now need to be loaded dynamically).

Recognizing its strength and the potential drawback, we designed and implemented this optimization to be dynamically controllable through a new Android kernel property. We recommend enabling this optimization by default because it significantly shortens device boot time and reduces app memory footprints. However, it can be disabled when preparing Morula processes for launching and hosting potential apps that require a large portion of the preloaded classes.

Selective Randomization. The key security benefit of Morula that fortifies the weakened ASLR on Android—allowing individually randomized memory layout for each process—is also the main reason why Morula causes additional time and space overhead compared to the Zygote model. This optimization strategy, namely selective randomization, aims to strike a balance between security gain and performance penalty, especially for low-end devices with restricted computing resources. For instance, suppose the device can only support a limited number of apps under the Morula model due to the performance overheads. In this case, it would be better to run the apps that are more likely to be vulnerable to ASLR exploits under the Morula model, and run the rest under the Zygote model.

A simple and efficient approach to classify apps into one of the two groups described above can be done based on whether an app is distributed with a native code component (i.e., built with Android NDK). In fact, Android apps without the native code component are less likely to be exploited to leak memory layout information compared to the apps with the native code component, because the native code distributed with each app can be an easy target for attackers. Apps without the native code components only run the default libraries in Android, and these libraries are well-maintained by either device vendors or Google, as they are widely used. However, apps with native code components may load uncommon third-party native code, and many of them are not well-maintained and out dated. Note that this approach sacrifices security for performance benefits, and thus it should be applied with the proper understanding of its limitations (Section VI).

Our selective randomization strategy simply reroutes app launch requests for a non-NDK app to the old Zygote process creation model, saving the prepared Morula processes for those apps that may be a concern for leaking address layout or evading ASLR. In general, checking if an app contains or

uses native code can be easily achieved by searching for binary executables in the package or JNI declarations in the bytecode. However, we are aware of techniques, such as binary file obfuscations, remote bytecode, Java reflections, etc., that can be used to hide the use of native code, and in turn, abuse this optimization to bypass Morula. A simple solution is to instrument the JNI binding component inside the DVM and reject stealthy 3rd party native code. We leave this as future work.

D. System Implementation

We used Android 4.2 (Jelly Bean) as the reference platform to prototype Morula. The implementation is generic to all Android versions, including the latest 4.4[2], and can be ported to them without changes. To minimize the changes introduced to the OS and maintain backward compatibility, we implemented Morula on top of the current Android process manager and confined Morula's code within two existing modules: the Activity Manager and the Zygote daemon. The Activity Manager is extended to dynamically maintain a pool of Morula processes and fulfill app launch requests using either a prepared Morula process or a forked Zygote process, depending on the selective randomization strategy. The code added to the Zygote daemon handles Morula preparation requests and creates Morula processes accordingly. The new daemon also checks the system property that indicates whether on-demand loading is enabled. When possible, our code reuses the existing initialization and specialization routines in Zygote, which have been peer-reviewed and deployed in the real world for years. All inter-module requests are sent and received through `pipes` or `sockets`. The sender's UID is always checked to make sure that only system modules can interact with Morula, avoiding abuses or attacks from malicious apps.

Morula only adds 548 lines of Java code and 197 lines of C code to the Android OS, which we hope can be promptly reviewed by the security community and soon merged into the Android Open Source Project. We believe this simple and effective countermeasure can be easily adopted by vendors without any technical hurdles. The implementation is generic to all versions of Android and free of compatibility issues with vendor-specific OS customizations, which are implemented at a much higher level than Morula.

V. Evaluation

In this section we report of our evaluation of Morula and its comparison with the original Android system. First, we evaluated the effectiveness of ASLR by determining whether Morula can provide different memory layout among Android processes and prevent our proposed attacks. Second, we measured end-to-end device boot performance to see whether Morula can be used by real-world users without noticeable overheads. Lastly, we conducted Android compatibility tests to check whether Morula causes compatibility issues with either Android apps or the OS. All experiments were conducted on the Galaxy Nexus, which has a dual-core 1200 MHz CPU (ARM Cortex-A9), 1 `GB` RAM, and 32 `GB` built-in storage.

[2] Android 4.4 (KitKat) introduced an experimental feature, `ART`, which pre-compiles an app's bytecode into native code at installation-time [6]. This feature changes various aspects of the Android OS and DVM, but `ART` is also relying on the Zygote process creation model to speed up the app launches.

A. ASLR Effectiveness

The goal of Morula is to make ASLR effective without degrading performance so that a commodity Android system can prevent the attacks described in Section III. We measured the effectiveness of ASLR in two ways: examining randomness of the memory layout of the entire Android system, and estimating how much effort is required to bypass ASLR by an attacker.

System-wide Randomness. Based on our experience of breaking the ASLR scheme, we define a measure of a memory layout's randomness and use this measure to check whether the address space layout in the entire Android system is sufficiently random. If it is, no attacker can guess exploitable target addresses in other applications from leaked memory addresses.

First, we measure the address space layout randomness of each shared library using the notion of entropy [10]. Since entropy captures the uncertainty of a given random variable, we can apply this to measure the address space layout randomness by treating the possible addresses as a random variable.

To be specific, let $\mathbf{X}_m$ be a discrete random variable with base addresses $\{x_1, x_2, ..., x_n\}$ for a shared library m, and $p(x_i)$ is a probability mass function (pmf). Then $H(\mathbf{X}_m)$, the normalized address space layout entropy of the shared library m, is defined as

$$H(\mathbf{X}_m) = -\sum_{i=1}^{n} p(x_i) \frac{\ln p(x_i)}{\ln n},$$

and $0 \leq H(\mathbf{X}_m) \leq 1$ due to the normalization factor $\ln n$. $H(\mathbf{X}_m)$ becomes zero when the shared library m is mapped to the same address for all different apps, and becomes one when it is mapped to all different and unique addresses.

For example, suppose `libc.so` is loaded by four Android apps at the same address, `0x1000`. Then the output of pmf will be $p(\text{0x1000}) = 1$, and the address space layout entropy of `libc.so` is computed as $H(\mathbf{X}_{\text{libc.so}}) = 0$. However, suppose `libssl.so` is loaded by four Android apps at four different addresses {0x1000, 0x2000, 0x3000, 0x4000}. In this case, the outputs of pmf will be always 0.25 because each address is uniformly distributed, and $H(\mathbf{X}_{\text{libssl.so}}) = 1$.

Based on the randomness on each shared library above, the address space layout randomness of an entire Android system in the device D is defined as

$$R(D) = \frac{\sum_{m \in M} H(\mathbf{X}_m)}{|M|},$$

where M is a set of shared libraries running on the device D and $|M|$ is a size of M. Thus, $R(D)$ shows the averaged address space layout randomness for all shared libraries running in the device D.

We measured $R(D)$ on our Android device after booting is finished, and different process creation models were applied for the boot procedure. As shown in Figure 7, the current Android (labeled as Zygote) has 0.127 entropy, which means the current shared libraries among Android applications mostly share their address space layouts. Therefore, it is possible for attackers to exploit vulnerable applications by guessing target addresses

Mode	$R(D)$	$T(D)$
Zygote	0.127	1
Wrap	0.993	19,373
Morula	0.992	18,360

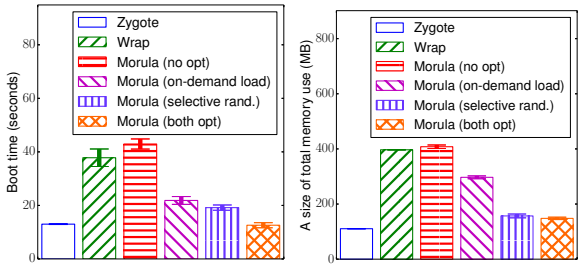
Fig. 7. ASLR effectiveness; R(D) denotes the averaged address space layout randomness and T(D) denotes the averaged number of trials for successful ASLR bypassing.

based on leaked addresses. Note that entropy for Zygote is not zero because a few shared libraries are independently loaded by apps, which results in different base addresses for such libraries. Morula along with Wrap, however, has more than 0.990 entropy. This suggests that an Android system with Morula deployed has heterogeneously different address space layouts for apps, and thus it is difficult for attackers to correctly guess the address space layouts.

Number of Trials to Bypass ASLR. To see how Morula helps an Android system prevent remote coordinated or local trojan attacks, we first design a cross-ASLR attack model and use this model to measure the attacker's required efforts to bypass ASLR. In this model, it is assumed that the attacker already leaked a single piece of address information: the address x for a certain application a_i. Based on such leaked information, the attacker tries to further infer a semantically equivalent addresses in other applications. The semantically equivalent address here refers to the address with the identical memory values or footprints in other applications. For example, two addresses in separate apps are semantically equivalent address if those two point to the same library with the same offset. It is also assumed that the attacker has prior knowledge of the memory layout in that semantically equivalent addresses are in similar locations for two apps[3], and the attacker has access to a decision oracle to test whether the given address is semantically equivalent in the other application. An example of how to query an oracle, in practice, is exploiting the target application with the guessed address and then making a decision based on the result: *accept* if the exploitation was successful, and *reject* if the exploitation failed (e.g., an application crashed).

Based on the assumptions described above, a cross-ASLR attack is described in Algorithm 1. Given the leaked address x, the attacker tries to find a semantically equivalent address for the other target application. The attacker queries the decision oracle repeatedly, starting from the leaked address x and stepping up/down by adding/subtracting the current address with a page offset. This is because the base addresses in the library are aligned at page boundaries and the semantically equivalent addresses are located in similar areas, as per the attacker's prior knowledge.

[3] This prior knowledge is based on widely used techniques to break ASLR deployed systems. For example, to bypass stack address randomization, attackers usually get the rough stack base addresses from other systems running similar execution environments (i.e., an operating system with same distribution and version) [42]. The other reasoning behind this is that the base address for each library is in the order of its loading due to the functions of mmap() system call implementations and such a loading order is deterministic [21].



(a) Booting time (b) Memory usage

Fig. 8. End-to-end boot-time performance overheads of Morula compared to the default Zygote process creation model. Morula increased boot time 230% and memory usage 269%. After applying both optimizations, however, Morula decreased boot time 3% and increased memory usage 34%.. We measured boot time and total memory usage immediately after Android becomes ready to use. Performance benefits from selective randomization need careful interpretations (Section VI).

Algorithm 1 Cross-ASLR Attack

```
// In : an address ptr leaked by an attacker
// Out: a semantically equivalent address in
//      the other target application

// walk away from ptr, by PAGE_SIZE step
for (offset = 0;; offset += PAGE_SIZE) {
  for (target in {ptr-offset, ptr+offset}) {
    // found semantically equivalent address
    if (ORACLE(target) == ACCEPT)
      return target;
  }
}
```

Thus, since each oracle access involves additional costs (e.g., observing crashes), the attacker's required efforts to bypass ASLR is directly related to the number of oracle accesses. We call this number of oracle accesses the number of trials, and count the number of trials using the address space layout immediately following device boot. This is shown as $T(D)$ in Figure 7. The count under the Zygote model is one, which indicates that the attacker would be successful on the first try. However, under the Morula model, the attacker would face immense challenges in that they need to try more than 18,000 times. This suggests that the attacker needs to search more than 18,000 memory pages or crash the app more than 18,000 times to correctly guess the address. We believe that this huge number of trials makes the attack meaningless and shows that Morula significantly raises the bar for attackers to bypass ASLR.

B. Performance Overheads

End-to-end Boot-time Performance Overheads. To compare the performance overhead, we evaluated each process creation model independently. For each model, once the booting process has finished, we measured the elapsed time for booting and the size of used memory. The elapsed time for booting is measured by capturing the timestamps of logs that signal the completion of device boot. The size of used memory spaces is measured by parsing `/proc/meminfo`, which shows the system memory information maintained by the Linux kernel.

This process repeats five times before the average and the standard deviation are computed.

These measurements are illustrated in Figure 8 with error bars representing the standard deviation. We can see that Morula has a larger overhead than Zygote before the optimization techniques. This is expected because many default system apps are launched at once during the booting process to support the basic features of Android. For example, `com.android.phone` is executed to support phone calling features and `com.android.nfc` is launched to enable NFC features. On our evaluation environment (Android 4.2), 19 apps are started during the booting process.

The device booting time under Morula is 230% (29 seconds) slower than that under Zygote (Figure 8-(a)). This is because Morula needs more time to execute each app for DVM initializations and address relocations, which are not required in the Zygote model. While Morula can take advantage of the device's idle states to prepare Morula processes with initialized DVM instances, this is not possible when the device is fully occupied to process many workloads during the booting time. Moreover, without optimization, Morula is even slower than the Wrap model because it needs to prepare extra Morula processes and manage communications through pipes.

For memory usage overhead (Figure 8-(b)), Morula used 269% (297 `MB`) more memory to boot the device than the Zygote model. This is because Morula does not allow sharing of resources such as relocated sections or heaps allocated by DVM. Compared to the Wrap model, Morula shows slightly more memory usage due to the extra Morula process.

However, once we have applied the optimization strategies, Morula's performance improved significantly. For boot time, on-demand loading and selective randomization reduce the additional delay down to 9 seconds and 6 seconds, respectively, compared with the boot time under the Zygote model. This implies that each optimization technique is effective at improving the boot time of a device using Morula. Especially for selective randomization, a total of 19 default system apps are executed at boot-time but only two require Morula processes for individually randomized memory layouts. When both optimization techniques are applied simultaneously, a device boot under Morula outpaces a boot under Zygote by 0.4 seconds. Though this improvement would be negligible in practice, it shows Morula with full optimizations imposes no additional overhead compared to Zygote. In regards to memory use, on-demand loading and selective randomization curtail the boot-time memory space overhead to 186 `MB` (168%) and 46 `MB` (41%) overheads, respectively, compared to the Zygote model, and are lower than the overhead from native Morula. When we apply both optimizations together, Morula used 37 `MB` (33%) more memory space than the Zygote model.

App Execution Performance Overheads. To see performance impacts on executing each app on Android, we first selected five popular apps from Google Play. Figure 9 shows a list of apps that we selected, including apps for social networks, messaging, and web browsing. Each app is executed as follows. First, an Android device is booted up with the Zygote process creation model. Next, a system property is set to specify which process creation model (Zygote/Wrap/Morula) will be used for executing the app. Then, the app is automatically executed by sending an `Intent` to the main activity class via adb terminal interfaces, which has the same effect as a user clicking an app icon to launch the app. Note that each process creation model is specified after booting up the device to avoid any system-wide effects during the booting processes and fairly compare the app execution performance overhead across different process creation models.

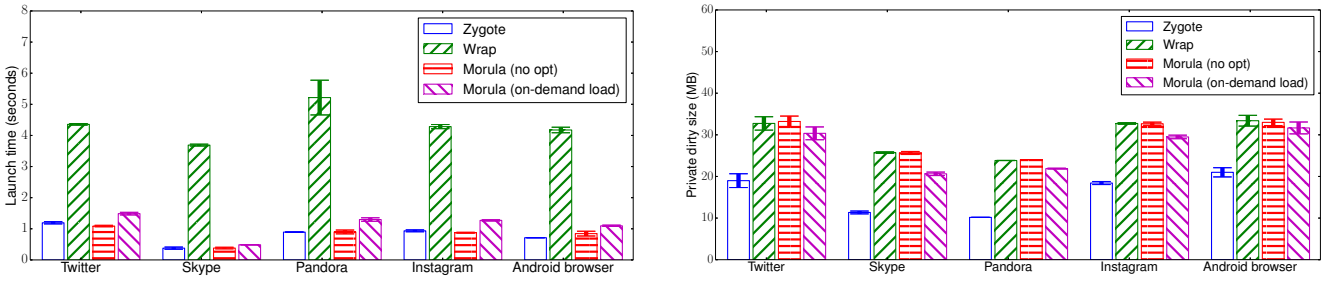
Package name		Ver.	Size (`MB`) .apk/.dex
Twitter	com.twitter.android	4.1.2	6.1/2.3
Skype	com.skype.android.access	1.3.0.2	4.4/0.8
Pandora	com.pandora.android	4.4	5.8/3.7
Instagram	com.instagram.android	4.0.2	15.7/4.1
Android browser	com.android.browser	4.2	2.4/0.8

Fig. 9. A list of apps to evaluate app execution performance overheads

We measured two primary performance overheads in executing apps: 1) launch time and 2) memory use. Launch time is defined as an elapsed time between the time when the activity manager receives the app creation request and the time when the target app is displayed on the screen. These two timing events were measured using log messages captured via adb terminal interfaces.

Morula aims to provide a similar quality of launch time as the Zygote model, and Figure 10-(a) shows that Morula successfully met this goal. For all five apps we evaluated, Morula and Zygote show a similar launch time. Interestingly, Morula was slightly faster than Zygote (2%), which was a negligible trade off between the extra pipeline communication time (in Morula) and invoking `fork()` system call (in Zygote). This also implies that Morula successfully precomputed the resource initializations and address relocations ahead of time, and these computations were not included in launch time. This suggests that a mobile user under Morula can enjoy equivalent launch times compared to Zygote while additionally having randomized memory layouts. When on-demand loading is applied on Morula, the launch time is increased 37% compared to Zygote. This slowdown compared to Morula without the on-demand optimization stems from the fact that Dalvik classes need to be loaded during launch time. Launch time under the Wrap model increases 3.52 seconds (427%) on average. This increase was deemed unacceptable for users.

To see how much memory space is required to execute an app, we measured `private dirty` for the app, which shows the amount of RAM space that is not shared with any other app. This `private dirty` would show actual memory space solely responsible for executing an app. This information was collected by aggregating `private dirty` sizes located in `/proc/pid/smaps`. We captured this information after executing each app, which is shown in Figure 10-(b). On average, Morula uses 13.7 `MB` more memory compared to the Zygote model. These memory costs were mostly due to: 1) address relocations and 2) DVM's private heap. Since Morula randomizes address layouts, all relative-addressing pointers should be relocated. Thus, memory pages with such pointers cannot be shared with other apps, and these will be counted as `private dirty` pages. In addition, Morula always creates a new DVM instance for executing an app, and this new DVM instance requires allocating its own private heap to maintain



(a) A launch time (b) Memory uses

Fig. 10. Per application execution performance overhead: (a) A launch time to execute an app using different process creation models. The launch time shows how long a user needs to wait until the app is displayed on the screen after executing the app. Compared to the Zygote model, on average Morula is 5 milliseconds (0.7%) faster than Zygote while Wrap slows down 3.52 seconds (427%). After applying the on-demand loading optimization, it slows down 0.30 seconds (37%); (b) Memory uses measured with the private dirty size. Morula uses 13.7 MB (85%) more memory spaces compared to Zygote. After on-demand loading optimization, 10.8 MB (68%) more memory spaces were used.

VM execution contexts and other resources. However, since the Zygote model simply forks the existing Zygote process to execute an app, it does not have these memory costs. When we apply the on-demand loading optimization, Morula uses 10.8 MB more memory on average, which is 2.9 MB less than Morula before the optimization. This memory saving results from not loading unnecessary Dalvik classes. Compared with the Wrap model, Morula shows similar memory costs because these two models are the same in terms of memory usage per app.

Note that the memory usage difference between Zygote and Morula stays the same across all five apps. This is because the memory difference between Morula and Zygote are due to additional the process initialization steps in Morula. This suggests that Morula's memory use overhead is independent of a specific app's features, and devices deploying Morula would have fixed memory costs around 13.7 MB to execute each app[4]. Considering recent mobile device trends providing more than 2 GB RAM [23], this static memory cost for running an app should not be of significant concern to guarantee ASLR security. For low-end devices having less than 512 MB RAM, however, this cost would not be acceptable. In this case, selective randomization can be applied with proper understanding of its security trade-offs (see Section VI).

Battery Consumption. Battery life is an important resource for mobile devices. Since Morula clearly performs more computations than the Zygote process creation model, Morula should consume more battery power than the Zygote model. We made the following measurements to determine how much more power is consumed by Morula. First, we booted our device with the Zygote process creation model and fully charged the battery. Then, the power cable was detached and the Android web browser (`com.android.browser`) was executed every 10 seconds. We then booted our device using the Morula process creation model and performed the same executions of the Android web browser under this system. For both measurements, we gathered current battery capacity over time by reading the contents of `/sys/class/power_supply/battery/capacity`.

[4]It is possible to further reduce this fixed memory cost by having apps share the VM or heap, but it should be carefully done as sharing such heaps may not be secure (e.g., OpenSSL's PRNG states initialized by Zygote [27]).

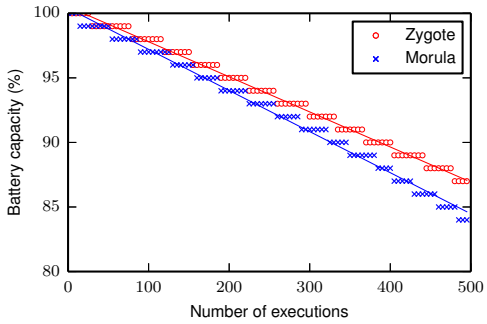


Fig. 11. Remaining battery life over the number of application executions. Morula requires additional computations compared to Zygote. We executed the Android web browser every 10 seconds and estimated the battery life for both Zygote and Morula. Morula imposes 0.5% extra battery consumption in the most active use case, which we consider negligible overhead.

Figure 11 shows battery capacity vs. the number of executions while plotting linear fit as a solid line. The fit line indicates that there will be 0.5% more power consumption if the user executes an app 100 times. We believe that most users would not launch an app more than 100 times between charges, thus they should not notice the increase in power consumption using Morula.

C. Compatibility Tests

To show Morula still meets the requirements of a compatible Android platform, we ran the Android Compatibility Test Suite (CTS) distributed by Google [5]. CTS offers comprehensive unit tests on various Android modules and functions. We ran system and security related test packages as shown in Figure 12. Among the 4,693 test cases we ran using CTS (version 4.2 r4), Morula passed 4,686 cases. Both Zygote and Morula failed on the same 7 test cases, and these were not related to either of the process creation models; `app` failed two cases because of the incompatibility of camera and GPS device drivers we installed, `permission2` failed three cases because our testing device cannot send SMS, `security` and `appsecurity` failed one case, respectively, because testing units had bugs in packaging signatures [7, 8]. This implies that

modifications to Android to implement Morula do not break Android compatibility, and we believe Morula is quite ready to be used for Android end-users.

Test package	#Tests	#Failed	
		Zygote	Morula
accessibility	25	0	0
accounts	28	0	0
admin	12	0	0
app	297	2	2
dalvik	51	0	0
libcore	3510	0	0
database	261	0	0
os	300	0	0
permission	149	0	0
permission2	15	3	3
security	37	1	1
appsecurity	8	1	1

Fig. 12. Android Compatibility Test Suite (CTS) results. Morula shows the same compatibility test results as Zygote, and the failed test cases in both models were not related to either of the process creation models. The prefixes on test package names were removed for simple representations.

VI. Discussion

Security Implications for Other Zygote-Based Systems. The concept of the Zygote process creation model is not only used for Android, but also used for other systems including the Chromium OS and the Chromium browser. The reason for using the Zygote model varies depending on the environment. For example, Chromium in Linux uses it to keep a reference to original binaries/libraries. As a result, after being updated in runtime, Chromium can keep running without version compatibility issues with newly updated binaries/libraries [48]. As we have shown, the Zygote model can be a possible attack surface because all child processes inherit resources, including memory layouts, from their parent process. Although Chromium only relies on the Zygote model for rendering processes, it is possible to have cross attacks involving multiple rendering processes or security related resource leaks initialized by the parent process.

Limitations of Selective Randomization. Selective randomization aims to balance the security gain and performance penalty. Thus, selective randomization does not protect Android from the attacks described in Section III because it provides the unique address space layouts only for selective apps. This indicates that a system deploying selective randomization can still be vulnerable to our ASLR attacks. For example, even if the app is distributed without native code components, it can be still exposed to address leak vulnerabilities while running the default system libraries loaded by Zygote. Although the possibility of having such vulnerabilities should be low, we recommend device vendors deploy selective randomization only if the Morula design cannot be applied for the whole system (i.e., low-end devices with limited computing resources). The security limitation of selective randomization stems from the fact that an app still runs native code in default system libraries even if it does not contain its own native code component. To handle this limitation, control-flow analysis can be performed on the app to see whether it actually runs (or heavily depends) on native code.

VII. Related Work

ASLR has been considered an effective defense mechanism for mitigating exploitation of security bugs by increasing diversity in address space layout of a program [39]. ASLR-enabled systems, combined with DEP, have successfully mitigated attack techniques such as arbitrary code execution or return-oriented programming (ROP).

ASLR Attacks/defenses. As modern commodity OSes provide ASLR/DEP defense mechanisms by default [24, 43], attack techniques also try to evolve to bypass ASLR/DEP. One example is to brute-force insufficient randomness in memory layout [30, 39]. Another example is to generate exploits based on memory layout extrapolated from leaked pointers, type confusion (heap overflow), and use-after-free bugs [35, 37]. Furthermore, by repeatedly abusing memory disclosures, an attacker can learn the entire memory layout of a system and chain ROP gadgets on the fly [41]. Moreover, it is possible for attackers to target non-randomized components of an application. For example, Flash, Java, and the .NET runtime in IE8/9/10 [42, 44] are well-known targets for ROP-gadgets to break ASLR/DEP in Windows.

With increasing use of bytecode interpreters [13], JIT compilation, which optimizes performance by compiling byte code to native instructions, opens other threats for breaking ASLR/DEP [11, 15]. Many commonly utilized attack vectors, such as JavaScript, are being compiled to native code via JIT compilation, providing attackers the means to convert their bytecode to a native executable. However, these attacks are only effective in breaking specific applications, so mitigation mechanisms such as anomaly detectors [14], are already deployed and used in commercial products [34].

Researchers also explored interesting ways to estimate target addresses for attacks by using cache or hash collision, both in the OS kernel [25] and in web browsers [2]. However, we believe ASLR-enabled systems raise a high bar for attackers to compromise servers [47] and mobile devices, which encounter attacks via small sets of interfaces like HTTP or media streaming.

Attacks/defenses on Android. We classify common attacks on Android into two types: the first is an exploit of underlying system components [45, 46, 51], which are privileged and separated processes in Linux. The other is abuse of over-privileged application permissions [9, 19, 20, 26, 49]. According to Lookout's Android Threat Report [36], one emerging problem on Android is the repackaged trojan attack, which prevents users from distinguishing an official and legitimate application from a trojan [18, 50]. For example, 250,000 users downloaded a disguised trojan application, DroidDream [36]. Therefore, we believe our local trojan attacks will become more critical to mobile users in the near future, and the ASLR solution in Morula can mitigate the threats effectively.

Although Android already ships with a variety of security features, such as UID separation and digitally signed applications [38, 40], we found that individual security bugs in applications can seriously threaten the entire system if Android does not fix the ASLR problem in the default Zygote process creation model.

Adopting ASLR/DEP in Mobile Devices. As exploits against

mobile platforms have increased, vendors outside the Android platform have also begun to include defensive mechanisms even at the cost of performance. For example, iOS 4 [29] supports ASLR of applications and the kernel.

Mitigating ASLR problems on Android. The most relevant work to Morula is Retouching [12], a mechanism that randomizes prelinked code when deploying Android applications. Since Retouching randomizes at the time of deployment or update, it does not require direct kernel changes. Therefore, Retouching provides differing memory layouts across different devices. However, Retouching has uniform address space layouts for all running apps in a device as it uses the Zygote process creation model, and thus it is still vulnerable to remote coordinated and local trojan attacks.

VIII. Conclusion

In this paper, we presented a new security threat to Android's ASLR and proposed Morula as a countermeasure. We showed that Zygote, Android's low-level process creation system, can severely weaken the effectiveness of ASLR, an existing and standard security mechanism. We demonstrated two attack scenarios where either a remote attacker or a local trojan app can exploit the weakened ASLR and execute code by means of return-oriented programming. As a replacement for the insecure Zygote, Morula fortifies the weakened ASLR on Android using three key designs and optimizations: the Morula process creation model, on-demand loading of Dalvik classes, and selective randomization of app memory layouts. We conducted a thorough evaluation, showing that Morula restores the effectiveness of ASLR on Android to the same level as on Linux, and at the same time maintains app launch time on par with or even better than Zygote. Morula pays an acceptable cost of increased app memory usage for much improved security, and imposes no obvious overhead to other resources, including battery power. Morula's design yields an easy-to-adopt and backward-compatible implementation, which is ready to be merged into the open source branch of Android OS as well as vendor-customized branches.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments. We thank William Enck for the thoughtful feedback that guided the final version of this paper. We also thank the various members of our operations staff who provided proofreading of this paper. This material is based upon work supported in part by the National Science Foundation under Grants No. CNS-1017265, CNS-0831300, and CNS-1149051, by the Office of Naval Research under Grant No. N000140911042, by the Department of Homeland Security under contract No. N66001-12-C-0133, and by the United States Air Force under Contract No. FA8650-10-C-7025. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Office of Naval Research, the Department of Homeland Security, or the United States Air Force.

References

[1] Android Web Browser GIF File Heap-Based Buffer Overflow. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0985.

[2] Leaking information with timing attacks on hashtables. http://gdtr.wordpress.com/2012/08/07/leaking-information-with-timing-attacks-on-hashtables-part-1.

[3] SRS One Click Root for Android. http://www.srsroot.com/.

[4] Type confusion in WebKit. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0912, 2013.

[5] Android Compatibility Program. Android Compatibility. http://source.android.com/compatibility.

[6] Android Open Source Project. Introducing ART. http://source.android.com/devices/tech/dalvik/art.html.

[7] Android Open Source Project Issue Tracker. android.security.cts.PackageSignatureTest failuer in android CTS R4. https://code.google.com/p/android/issues/detail?id=19030.

[8] Android Open Source Project Issue Tracker. CTS com.android.cts.appsecurity.AppSecurityTests testPermissionDiffCert FAIL. https://code.google.com/p/android/issues/detail?id=53532.

[9] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: Analyzing the Android Permission Specification. In *ACM conference on Computer and communications security (CCS '12)*, 2012.

[10] C. M. Bishop and N. M. Nasrabadi. *Pattern recognition and machine learning*. springer New York, 2006.

[11] D. Blazakis. Interpreter exploitation. In *USENIX conference on Offensive technologies (WOOT '10)*, 2010.

[12] H. Bojinov, D. Boneh, R. Cannings, and I. Malchev. Address space randomization for mobile devices. In *ACM conference on Wireless Network Security (WiSec '11)*, 2011.

[13] H. Chen, C. Cutler, T. Kim, Y. Mao, X. Wang, N. Zeldovich, and M. F. Kaashoek. Security bugs in embedded interpreters. In *ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '13)*, 2013.

[14] P. Chen, Y. Fang, B. Mao, and L. Xie. JITDefender: A defense against JIT spraying attacks. In *Future Challenges in Security and Privacy for Academia and Industry*. Springer, 2011.

[15] Y. I. Chris Rohlf. Attacking Clientside JIT Compilers. In *Black Hat USA*, 2011.

[16] coolkaveh. VLC media player 2.0.4 suffers from buffer overflow. https://trac.videolan.org/vlc/ticket/7860.

[17] J. Easton-Ellett. Android 4.1 Jelly Bean Features ASLR, Making It Much Harder To Exploit. http://www.ijailbreak.com/android/android-4-1-jelly-bean-features-alsr.

[18] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *ACM conference on Computer and communications security (CCS '09)*, 2009.

[19] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, 2010.

[20] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *USENIX Security Symposium (Security '11)*, 2011.

[21] C. Evans. Some Random Observations on Linux ASLR. http://scarybeastsecurity.blogspot.com/2012/03/some-random-observations-on-linux-aslr.html, 2012.

[22] D. Fisher. Android 4.1 Jelly Bean Includes Full ASLR Implementation. http://threatpost.com/android-41-jelly-bean-includes-full-aslr-implementation-071612, 2012.

[23] M. Flores. Google Nexus 5 Review. http://www.techradar.com/us/reviews/phones/mobile-phones/google-nexus-5-1194974/review.

[24] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium (Security '12)*, 2012.

[25] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space aslr. In *IEEE Symposium on Security and Privacy (Oakland '13)*, 2013.

[26] S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints. In *IEEE Symposium on Security and Privacy (Oakland '12)*, 2012.

[27] S. H. Kim, D. Han, and D. H. Lee. Predictability of android openssl's pseudo random number generator. In *ACM conference on Computer and communications security (CCS '13)*, 2013.

[28] N. Kralevich. Address Space Layout Randomization in Android. https://groups.google.com/forum/#!msg/android-security-discuss/Af71Z2QYdMo/u1miB1A9UOwJ.

[29] T. Mandt. Attacking the iOS Kernel: A Look at 'evasi0n'. http://blog.azimuthsecurity.com/2013/03/attacking-ios-kernel-look-at-evasi0n.html.

[30] T. Muller. ASLR Smack & Laugh Reference. http://www-users.rwth-aachen.de/Tilo.Mueller/ASLRpaper.pdf, Feb. 2008.

[31] J. Oberheide. A look at ASLR in Android Ice Cream Sandwich 4.0. https://www.duosecurity.com/blog/a-look-at-aslr-in-android-ice-cream-sandwich-4-0, .

[32] J. Oberheide. Exploit Mitigations in Android Jelly Bean 4.1. https://www.duosecurity.com/blog/exploit-mitigations-in-android-jelly-bean-4-1, .

[33] L. Page. Google I/O 2013 Keynote. http://www.google.com/events/io/2013/, 2013.

[34] P. Ratanaworabhan, V. B. Livshits, and B. G. Zorn. NOZZLE: A defense against heap-spraying code injection attacks. In *USENIX Security Symposium (Security '09)*, 2009.

[35] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *Annual Computer Security Applications Conference (ACSAC '09)*, 2009.

[36] L. M. Security. Lookout Mobile Threat Report. https://www.lookout.com/_downloads/lookout-mobile-threat-report-2011.pdf.

[37] F. J. Serna. The info leak era on software exploitation. In *Black Hat USA*, 2012.

[38] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google Android: A comprehensive security assessment. *Security & Privacy, IEEE*, 8(2), 2010.

[39] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM conference on Computer and communications security (CCS '04)*, 2004.

[40] S. Smalley. The Case for Security Enhanced (SE) Android. https://events.linuxfoundation.org/images/stories/pdf/lf_abs12_smalley.pdf.

[41] 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 *IEEE Symposium on Security and Privacy (Oakland '13)*, 2013.

[42] A. Sotirov and M. Dowd. Bypassing browser memory protections in windows vista. In *Black Hat USA*, 2008.

[43] The PaX Team. PaX. http://pax.grsecurity.net.

[44] S. J. Vaughan-Nichols. Pwn2Own: Down go all the browsers. http://www.zdnet.com/pwn2own-down-go-all-the-browsers-7000012283.

[45] T. Vidas, D. Votipka, and N. Christin. All your droid are belong to us: a survey of current Android attacks. In *USENIX conference on Offensive technologies (WOOT '11)*, 2011.

[46] T. Vidas, D. Votipka, and N. Christin. All your droid are belong to us: A survey of current Android attacks. In *USENIX conference on Offensive technologies (WOOT '11)*, 2011.

[47] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *ACM conference on Computer and communications security (CCS '12)*, 2012.

[48] C. Wiki. The use of zygotes on Linux. https://code.google.com/p/chromium/wiki/LinuxZygote.

[49] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The impact of vendor customizations on Android security. In *ACM conference on Computer and communications security (CCS '13)*, 2013.

[50] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt. Identity, location, disease and more: Inferring your secrets from Android public resources. In *ACM conference on Computer and communications security (CCS '13)*, 2013.

[51] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy (Oakland '12)*, 2012.