Address Space Randomization for Mobile Devices

Hristo Bojinov
Stanford University
Stanford, CA, USA
hristo@cs.stanford.edu

Dan Boneh
Stanford University
Stanford, CA, USA
dabo@cs.stanford.edu

Rich Cannings
Google, Inc.
Mountain View, CA, USA
richc@google.com

Iliyan Malchev
Google, Inc.
Mountain View, CA, USA
malchev@google.com

ABSTRACT

Address Space Layout Randomization (ASLR) is a defensive technique supported by many desktop and server operating systems. While smartphone vendors wish to make it available on their platforms, there are technical challenges in implementing ASLR on these devices. Pre-linking, limited processing power and restrictive update processes make it difficult to use existing ASLR implementation strategies even on the latest generation of smartphones. In this paper we introduce *retouching*, a mechanism for executable ASLR that requires no kernel modifications and is suitable for mobile devices. We have implemented ASLR for the Android operating system and evaluated its effectiveness and performance. In addition, we introduce *crash stack analysis*, a technique that uses crash reports locally on the device, or in aggregate in the cloud to reliably detect attempts to brute-force ASLR protection. We expect that retouching and crash stack analysis will become standard techniques in mobile ASLR implementations.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security, Experimentation, Performance

Keywords

ASLR, control flow hijacking, return-to-libc, mobile devices, smartphones, Android

1. INTRODUCTION

Over the last few years Address-Space Layout Randomization (ASLR) has become mainstream, with various levels of support in Linux [25], Windows [20], and Mac OS X [19]. ASLR randomizes the base points of the stack, heap, shared libraries, and base executables. The goal of ASLR is to make certain classes of control-hijacking attacks more difficult: with executable code residing at unknown locations, garden variety buffer or stack overflow attacks are made significantly harder to develop and execute [14]. In conjunction with OS mechanisms that only allow writing to non-executable memory (e.g. DEP in Windows), ASLR prevents many network-based native code control hijacking attacks from completing [24].

Implementation challenges.

Although there has been much work on implementing and evaluating ASLR for general-purpose PCs, none of the major smartphones currently use it. In principle, the same ASLR techniques should carry over to mobile devices, however there are several practical obstacles that make this difficult. Smartphone operating systems spend considerable effort to minimize boot and application launch time, power consumption, and memory footprint. These optimizations make existing ASLR implementation strategies insufficient. We give two examples for challenges in implementing ASLR in Android:

- The Android OS prelinks shared libraries to speed up the boot process. Prelinking takes place during the build process and results in hard-coded memory addresses written in the library code. This prevents relocating these libraries in process memory. Android also uses a custom dynamic linker that cannot self-relocate at run-time (unlike *ld.so*). Recent attack techniques against ASLR clearly demonstrate the need to randomize the whole process address space, including base executables and shared libraries [18].
- During normal operation the filesystem on the device is mounted read-only for security reasons. This prevents binary editing tools [12] from modifying images on the device or in file-backed memory.

Our contributions.

We propose *retouching*, a novel mechanism for randomizing prelinked code for deployment on mobile devices. Retouching can randomize all executable code including libraries (and prelinked libraries), base executables, and the linker. Unlike traditional ASLR implementations, retouching requires no kernel modifications. We implement the

WiSec'11, June14–17, 2011, Hamburg, Germany.

mechanism for Android, evaluate its effectiveness, and measure its impact on performance at build and runtime and on memory footprint. Our conclusion is that retouching is an effective approach to ASLR and is particularly well-suited in situations where performance is an issue, or when there are incentives to avoid kernel changes.

Our second contribution is a cloud-based approach to detecting and preventing ASLR brute-forcing [23]. We introduce *crash stack analysis*, a technique that analyzes crash reports from mobile devices and reliably detects attempts to bypass ASLR by guessing the random offset used on each device. We evaluate crash stack analysis using real crash data as well as simulated attacks, and conclude that the approach can effectively detect attacks and, in addition, can help pinpoint the OS code being targeted.

Brute forcing mobile ASLR can be very effective and difficult to detect locally. By making a *single* attempt on every mobile user the attacker can compromise 1/256 of mobile devices (assuming 8 bits for ASLR randomness as in Windows). Given the billions of phones in use, this fraction gives the attacker control of a large number of devices.

In the rest of the paper, Sections 2 and 3 give some background information on ASLR and the Android OS, Section 4 presents the threat model that we address with our design and implementation in Section 5. Section 6 evaluates the implementation and discusses it in the context of other related and future work. Section 7 introduces crash stack analysis and evaluates its effectiveness. Sections 8 and 9 discuss future and related work, and Section 10 concludes.

2. OVERVIEW OF ADDRESS-SPACE RANDOMIZATION

Before discussing our system we first survey traditional strategies for implementing ASLR and their limitations on Android. The first ASLR implementation, PaX [25], was designed for Linux. Subsequently, ASLR was implemented in Windows Vista and Mac OS X. We briefly describe these implementations focusing primarily on user-space randomization (as opposed to kernel randomization, which is a separate topic).

2.1 PaX

PaX implements ASLR by generating three different random offsets (delta values) that apply to different areas of the address space:

- **delta_mmap** controls the randomization of areas allocated via mmap(), which includes shared libraries, as well as the main executable when compiled and linked as an ET_DYN ELF file
- **delta_exec** is the offset of the base executable, followed by the heap, when the base executable is of type ET_EXEC (not position-independent)
- **delta_stack** is the offset for the user-space stack

PaX ASLR is complemented by data and stack execution prevention logic. While several elements of PaX are applicable to Android, the technique for randomizing the location of shared libraries and the dynamic linker is not: most shared libraries are prelinked and mapped to specific locations when built, and the dynamic linker is unable to self-relocate.

2.2 Windows

Following the implementation of DEP (Data Execution Prevention: marking the stack and heap as non-execute) in Windows XP SP2, Microsoft implemented ASLR in Windows Vista. The two mechanisms work together to prevent control-flow hijacking attacks (such as return-to-libc), as well as injected code from being executed on the stack or heap. In the Windows ASLR implementation, executable code randomization happens on every reboot, when a global image offset is selected randomly out of 256 possibilities. Additionally, every process is launched with an individually randomized stack, heap, and Process Environment Block (PEB) [26]. The 8 bits of entropy used for selecting the offsets renders the Windows ASLR implementation vulnerable to guessing attacks [23], but is still better than no randomization at all.

Adopting the Windows ASLR approach directly in Android would increase boot time of the device substantially by eliminating library prelinking.

2.3 Mac OS X

Apple introduced ASLR in the Leopard release of Mac OS X. Currently, OS X only randomizes the offsets of shared libraries. This randomization is performed at the time libraries are prelinked, effectively prelinking them at a different address on each system. In addition to ASLR, the operating system protects stack and heap data from being executed (heap protection is only available for 64-bit binaries) [19, 15].

Retouching, the technique we have developed, is conceptually similar to randomization during prelinking. The additional benefits of retouching are in significantly reducing the amount of work performed on the target device by performing the prelinking during the build process and retaining the minimum information needed for randomization. Additionally, no ELF manipulation code needs to be installed on the target.

3. OVERVIEW OF ANDROID

Android [1] is an operating system for mobile devices developed by Google, Inc. While Android borrows much platform code from other open-source operating systems, its security model was built from the start with the assumption that the device will be running a variety of untrusted (or partially trusted) applications. A manifestation of this approach is the execution of each installed application in a separate process running under a unique user identifier (UID): any damage that the application can cause will be contained within the resources that are dedicated to this UID—disjoint from those of any other UID or application.

Android is built on top of the Linux kernel. The system includes many device drivers and native system libraries, including a customized implementation of libc. Applications in Android are written in Java and execute in a virtual machine called Dalvik, in the form of Dalvik bytecode. After boot, the system runs many services (such as the media service, telephony service) each in its own process and having a unique UID.

An important Android process, called *zygote*, is used to speed-up application launch. The zygote is initialized at boot time with commonly used shared libraries, application frameworks, and the Dalvik virtual machine. When the user launches an application the zygote forks and the requested

Relocation Type	Count
local	139837
external	28480

Table 1: Local vs. external relocations in a typical release build of the platform (all prelinked libraries). Removing local relocations by pre-linking saves space and reduces library load time.

Prelinking	Trial 1	Trial 2	Trial 3
enabled	57.2s	57.6s	57.5s
disabled	60.7s	60.9s	60.3s

Table 2: The effect of prelinking on boot time of the HTC Magic (about 3 seconds, or 5% on average).

application runs in the forked process. Since most resources are already loaded in the zygote, the application can immediately begin executing. The zygote architecture implies that with stack and heap randomization during process launch all applications launched on the phone inherit the same randomization parameters.

Prelinking using apriori.

One notable extension at the platform level is the prelinking mechanism implemented by the *apriori* tool. In dynamic linking, relocations are contents (addresses) in a binary object file which need to be adjusted upon loading the binary in memory. Apriori is a Google-built prelinker which resolves local relocations (relocations that refer to code in the same object) in native shared libraries, and pins the libraries to specific memory offsets. The prelinking happens after library objects are compiled and linked, but before they are stripped of unnecessary sections. Apriori looks at the relocations listed for each prelinked library, and resolves those that are local (i.e. not referencing other libraries)—removing them from the relocation section of the library.

Table 1 shows that local relocations comprise the majority of relocation entries in prelinked libraries, and Table 2 demonstrates the impact of prelinking on boot time of the device—a 5% improvement (the Eclair branch of the code base was used for this comparison). While removing relocations also contributes to a reduced filesystem image and relieves demand for main memory, the main goal of prelinking was to speed up the boot process. The impact of prelinking on individual application launch is smaller, because much of that cost is absorbed when the zygote is started.

Prelinking in Android offers clear benefits, but at the same time it prevents standard implementations of ASLR which rely exclusively on randomizing library locations at load time. Prelinked libraries contain hard coded memory addresses and cannot be relocated. When the dynamic linker loads a prelinked file, it uses a provided hard coded address as the location of the library in memory instead of selecting an available address in the regular shared library load area (Table 4). Android does not use the standard Linux dynamic linker (*ld.so*), and instead has a simpler implementation that is mapped to a fixed location in memory.

Software updates.

The Android platform has a built-in mechanism for over-the-air (OTA) software updates, comprising the following components:

- Scripts for packaging over-the-air updates, invoked from *build/ tools/ releasetools/ ota_from_target_files*; the resulting package includes a list of instructions in the Edify language—these instructions are executed during the update on the target device;
- The *updater* binary, which is statically linked, and executed on the handset while in recovery mode; the source code is located under *bootable/recovery/updater*.

4. THREAT MODEL

The primary goal of ASLR is to make remote exploitation difficult. ASLR is not designed to protect against a malicious application already on the phone. To see why, recall that shared libraries are loaded at the same memory location for all processes in the system. Hence, a malicious application can determine the memory location of libc, and use that information to mount a return-to-libc attack on another process. ASLR cannot prevent this. Consequently, in evaluating the security of our proposal we only consider remote attackers who do not already have a foothold on the phone. More precisely, we use the following threat model.

In-scope threats.

Our goal is to prevent network attackers from exploiting vulnerable network-facing services.

- **Network attackers** have the ability to send arbitrary packets to any open port on the device, as well as receive responses. A malicious website and a nearby rogue access point are potential network attackers.
- **Network-facing services** can have exploitable vulnerabilities such as buffer and stack overruns. These can result in either code injection or return-to-libc exploits, and ASLR aims to prevent the latter. Examples of such vulnerabilities would include a rogue SMS packet [16] or a malicious video that targets a codec flaw.

Out-of-scope threats.

In the context of Android we do not address the case of malicious applications (executed by the Dalvik VM) attempting to attack other processes on the system. The UID-based compartmentalization mechanism in Android is specifically intended to sandbox applications and limit the impact they can have on the system overall. We point out that Dalvik applications have access to native libraries (via JNI), and thus to the randomization offsets that have been applied in the system.

5. DESIGN AND IMPLEMENTATION

The Android environment prevents existing approaches to ASLR. Our goal is to design a new light-weight ASLR strategy that is well suited for constrained environments of this type. Our approach applies equally well to other mobile operating systems.

5.1 Background

Modern compilers like GCC can generate code which is position-independent (PIC). PIC object files have all of their location-sensitive offsets listed in *relocation sections*: these lists are later used to "fix" the library to a location at load time. Shared libraries built with PIC code can be linked as ET_DYN ELF objects, which means that they can be loaded at arbitrary addresses. In Linux, *ld.so* is a special shared library (an ET_DYN object itself) which is responsible for dynamically linking any other libraries that must be loaded into a process. Notably *ld.so* is able to relocate itself, which is not trivial to implement.

In contrast to shared libraries, base executable files are often built as ET_EXEC objects, which must be loaded at a specific location known during the link process. On some platforms base executables can also be linked as ET_DYN, which makes it possible to load them at an arbitrary location in the process address space. Such executable objects are referred to as PIE (position-independent executables).

Effects of prelinking.

In the absence of any prelinking, position-independent (ET_DYN) ELF objects can be loaded at arbitrary locations in the process address space: this is the general idea in PaX. In contrast, for Android:

- Shared libraries are PIC, but the majority of them are prelinked to specific addresses.
- Base executables are compiled as PIC, but not linked as PIE.
- The dynamic linker is linked at a fixed address because it is simpler than *ld.so* and is not able to relocate itself.

On the one hand, the extensive use of PIC code in the platform comes at a minimal cost to performance due to the extensive prelinking performed. On the other hand, PIC code allows for easier patching of software: only the modules affected by a bug fix need to be recompiled, and all the rest can simply be prelinked again, lowering the risk of introducing new bugs in an incremental update.

5.2 Design Idea

For a base executable object that was compiled to be PIC, linking to a fixed base address consists of resolving primarily internal relocations (usually in the GOT section) using that fixed address and removing the entries from the relocation table; for shared libraries, prelinking has a similar effect (Figure 1), outlined in Section 3.

We now make three important observations. First, PIC binaries can be rebased even after prelinking. At prelinking time we can save the address of all locations where apriori inserted hardcoded addresses (this is exactly the set of local relocations). Then, to shift the binary to a new location we can loop over the list of addresses and add the ASLR random offset to the contents at each of them.

Second, binaries can be trivially reverted to their original state to support software updates. This is necessary in the case of incremental updates, where the hashes of patched files are checked to ensure the device being updated has the expected build. To revert randomization, we simply overwrite the contents at the known locations in each file with their known, build-time contents.

Third, randomization is possible at software update time (rather than on every boot or process restart). This enables a light-weight, user-space implementation, and in Section 6 we argue that the loss in terms of security is small.

Retouching design.

Based on the observations made earlier, the process of *retouching* is spread over all stages of building and deploying a software update. During a build, we retain some of the relocation data that is normally lost after the build completes (file offsets and, for convenience, a copy of the original contents at those offsets). The data is retained in a separate area at the end of each binary. When packaging OTA updates, the retained relocation data is still in the executable files; in addition, the OTA update script now includes a command which explicitly applies randomization (or derandomization if desired) to all relevant files. Finally, when the OTA update script is run on each target device, randomization (or derandomization) is executed.

During normal device boot there are no execution flow changes: the affected binary objects are simply loaded to their randomized addresses instead of the nominal addresses that were used in the original build.

5.3 Implementation

In order to assess the feasibility of retouching we have implemented ASLR for the Android platform, and contributed the code to the Android Open Source Project (AOSP). We start with a discussion of shared libraries, and then expand to base executables and the dynamic linker.

Shared libraries.

Randomizing libraries during software updates is preferable because it does not eliminate the performance gains offered by prelinking in the first place. In addition, update time randomization is performed in recovery mode in which the main operating system image is not locked and can be safely modified.

The randomization process consists of the following four steps:

Step 1: Keeping track of relocation lists.

Prelinking in the Android platforms involves resolving internal relocations for each shared library (Table 1). Subsequently, during randomization these previously resolved relocations must be adjusted (retouched) by the difference

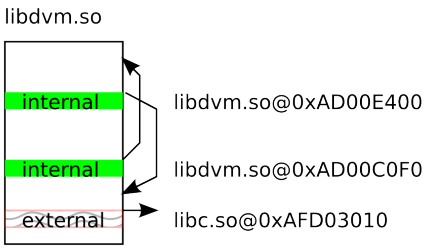

Figure 1: Prelinking resolves internal relocations while leaving external ones intact.

Record Type	Format	Description
2 bytes	$1S_2C_{13}$	2-bit offset (4, 8, 12, 16) 13-bit content delta (signed)
3 bytes	$01S_2C_{20}$	2-bit offset (see above) 20-bit content delta (signed)
8 bytes	$00S_{30}C_{32}$	absolute offset, max $2^{30}-1$ absolute contents (4 bytes)

Table 3: The three record formats used in retouch file compression.

between the "default" prelink location of the library and the new, randomized location.

By the end of the platform build process, relocations that have been prelinked are stripped from the final shared library files. In order for retouching to succeed during a device update, we must have the list of relocations available at that time. We achieve this by modifying *apriori*, the Android prelinker, to output a list of file offsets that have been prelinked. For each library, this list is stored at the end of the target binary file.

Step 2: Compressing retouch data.

Even though we only need to keep a small amount of data for each prelinked relocation (a file offset, and contents at that offset), the aggregate size of retouch data ends up being substantial. We came up with a simple variable record size encoding in order to minimize the size of the OTA update package and the amount of additional space required on the device.

Each prelinked relocation is nominally eight bytes in size: a four-byte file offset, and four bytes of original contents. In our compression scheme, each relocation corresponds to a 2, 3, or 8-byte record, essentially implementing a form of Huffman encoding based on the following observations:

- Relocation offsets and contents are 4-byte aligned.
- Relocation offsets are output in order and tend to be clustered closely together.
- Relocation contents also tend to exhibit proximity, but to a lesser degree.

Specifically a 2-byte record will be used for a relocation if it is located within 4 to 16 bytes from the previous one, and has contents within $2^{12}-1$ in absolute value. A 3-byte record will be used if the relocation is located within 4 to 16 bytes, with contents that differ by no more than $2^{19}-1$. As a fallback, the relocation can use up a full 8 bytes, with the most significant two bits of the offset used to indicate this type of record (limiting the file size of shared libraries to $2^{30}-1$ bytes). Record formats are specified in Table 3. Compression reduces the amount of space needed for retained relocation data by approximately 60%.

Step 3: OTA update file generation.

After the build is complete, the OTA update file is generated, and we ensure that an instruction to retouch all binaries is always included. Alternatively, the instruction can be to undo retouching (in case derandomization is required for some reason).

Step 4: OTA deployment on target device.

OTA updates are executed on Android phones in the following steps: reboot into recovery mode, check filesystem digest (only in incremental updates), extract files from update package (zip), reboot into the updated main image.

In our implementation, randomization involves on-device modification of all shared library files. As a consequence, for incremental updates we have to mask the randomization so that the filesystem digest check will succeed. This is why every list of retouch entries contains the prelinked relocation offsets **and** original contents at those locations: before computing a digest of the software image the update script can restore in memory each binary to its original state. Another benefit of this approach is evident during randomization: retouch data is never modified, and should randomization be interrupted in the middle, the process can simply be rerun, generating a new randomization offset and overwriting any modified shared library contents. (In reality, this process is a bit more complicated due to flash filesystem unreliability that goes beyond what is normally seen with a hard-disk based filesystems. For brevity, we skip the details here.)

With ASLR, software update proceeds as follows (randomization-related steps are in bold):

- The device is booted in recovery mode.
- (Incremental updates only) **After masking randomization in memory,** a digest of the existing contents of each patched file is checked against the digest included in the update. The update proceeds only if there is a match: this ensures that an incremental update will only be applied to the appropriate build it was created against.
- Files are extracted from the update and copied to their destinations, for example in the */system* directory. (This includes shared libraries about to be retouched.)
- **Retouch data are used to randomize the prelinked relocations in specified binaries (all prelinked shared libraries in the first release of retouching, and eventually all binaries). First, a random offset is generated, then used to shift all the binaries.**
- The device can now be rebooted into the new, updated software build.

Randomizing the base executables and dynamic linker.

Base executable offset randomization is possible by porting existing functionality e.g. from the PaX project, however this is challenging for a number of practical reasons.

Firstly, Android executables are not readily linked as ET_DYN ELF objects, but rather are of the ET_EXEC type (in other words, even though the code is position independent via the "-pic" compiler option, the resulting executables are not position independent); retouching allows us to navigate around this hurdle.

Second, implementing the PaX shadow copy technique for ET_EXEC binaries would require kernel changes difficult to open-source for the ARM architecture, while all the changes

we introduce are in user-space and in the platform build system.

The third, and last obstacle is that the Linux dynamic linker *ld.so* is randomized in PaX via changing the mmap base. In contrast, the Android linker is much simpler and must exist at a predefined address in memory, fixed at build time. Here once again retouching comes to the rescue.

Our implementation of retouching for base executable and linker address randomization requires two separate builds of each executable, and proceeds in the following steps:

- Perform a build to the default base address (0x8000 for executables, and 0xB0000100 for the linker).
- Save the output binary.
- Perform a build to a new base address (e.g. 0xFF8000, and 0xB0FF0100 for the linker).
- Run the *retouch-bindiff* tool on the two builds of the same binary, which outputs a list of retouch entries that can be appended to the binary.
- (During OTA update) Retouch using the binary file generated at build time. Same approach as the one used for libraries.

The *retouch-bindiff* tool simply takes two input files and finds the 4-byte file records that differ, outputting their offsets and contents. The tool also performs a sanity check to ensure that the provided base offset difference (0xFF0000 for the examples above) is exactly equal to the difference at each record. In our experiments, out of all the executables present on the Eclair branch of AOSP the sanity check failed only for the *debuggerd* binary, which has some base offset-dependent data compiled in. This executable serves to gather crash data from the device, and can easily be excluded from randomization without introducing vulnerability in the system.

In its current form *retouch-bindiff* cab be impacted by changes in compilation flags or implementation. For example, if different types of relocations are created during compilation, the tool may not be able to generate the correct type of retouch entry. Ideally, base executable and dynamic linker randomization should be implemented by retaining relocation data during the linking stage and converting that to retouch entries, eliminating any guesswork and the inconvenience of double compilation introduced by *retouch-bindiff*.

5.4 Sources of Randomness

When we generate a random offset during the update process, we use two sources of randomness. The first one is /dev/random, which contains random bits saved across reboots. Note that randomization happens during software update, which means that the device has been operational for some period of time and has been able to collect some entropy.

The second source of randomness we use is system time. While time is not truly random, the clock reading during updates will tend to be random across the population of devices (Android phones will not automatically reboot when an update is being deployed by the carrier). This means, that without prior knowledge of the attacked device, the low-order bits of the system time during the last update will look random to the attacker.

The general problem of gathering entropy on mobile devices is a topic that has received attention on its own, and [13] evaluates several approaches. Randomization during device manufacture is also a possibility which can be explored in the future.

6. EVALUATION

To evaluate our new approach to ASLR we discuss the amount of work to mount a brute-force guessing attack, the additional storage requirements on the device, and the negligible impact on performance.

6.1 Guessing Attacks

Existing ASLR implementations randomize offsets at boot or run-time. Our retouching approach randomizes offsets only at system update time. We briefly argue that from the point of view of the attacker this difference has minimal impact — it only makes a brute force guessing attack easier by a factor of 2 in expectation.

Consider a randomization space of size N (in other words, the number of different offsets for the target executable code post-randomization is N). If the randomization offset is constantly updated at boot or run-time then an attacker making random guesses at the randomization value will need N random attempts in expectation before making a correct guess.

With randomization at install time and during software update (which is less frequent than boot-time randomization), the attacker is **guaranteed** to succeed after N attempts, and is expected to succeed in $(N+1)/2$ attempts. This factor of 2 is the result of the difference between *sampling with replacement* and *sampling without replacement*.

Randomizing during software updates has a number of advantages over run-time randomization. First, it is less likely to corrupt the device because it happens in a simple "recovery boot" environment in which platform executables can be safely written. In addition, the kernel need not get involved in this process, and the boot time savings that prelinking affords can be preserved (i.e., no additional run-time relocation is necessary).

Entropy.

In our initial implementation we limit ourselves to 10 bits of entropy for each base address (executable, prelinked libraries), and 8 bits for the dynamic linker. We do this because of space constraints and to ensure system stability; the number of bits used can be revised in the future. Table 4 shows the default address space layout of an Android process, along with the maximum randomization offset that we add or subtract. In Table 5 we show the significant bits of the offsets generated by the randomization code over several uploads of an ASLR-enabled package. In the actual retouching, these offsets are multiplied by 4096 for shared libraries and base binaries and 256 for the dynamic linker.

6.2 Storage Impact

We require 426KB of space to store randomization data for shared libraries in /system/lib (averaging about 3.3 bytes per relocation entry). For comparison, the actual libraries take up 24.63MB on disk, and thus the storage overhead is less than 2%. Table 6 summarizes these numbers against a non-prelinked, non-retouched baseline build.

Area Purpose	Location	Randomization
executable	0x00000000	+ 0x003FF000
stacks	0x10000000	
mmap	0x40000000	
shared libraries	0x80000000	
prelinked libraries	0x9A100000	- 0x003FF000
linker	0xB0000100	+ 0x0000FF00
thread 0 stack	0xB0100000	
kernel	0xC0000000	

Table 4: Default address space layout in Android. Note that the prelinked library area is offset downward, to avoid overlapping with the linker (we can do this because addresses from 0x90000000 to 0xB0000000 are dedicated to prelinked libraries).

Run #	Value (10 bits)	Run #	Value (10 bits)
1	0011000011	7	1001101001
2	1010010101	8	1000011000
3	1100011100	9	1110100000
4	1101100100	10	1001001011
5	1010000001	11	1010010001
6	0011111101	12	0011110110

Table 5: Randomization offsets generated by using `/dev/random` and the current time.

The impact of retouching on OTA update package size is smaller than that on the filesystem (250KB added), and incremental updates (which are typically used) will be substantially less impacted because only changed binaries need to have their retouch data patched, and that can also be done incrementally.

We estimate that if base executables and the dynamic linker are to be retouched, there will be about 52K additional retouch entries, which would add approximately 170KB to the space required in the OTA update and the filesystem, bringing the total storage overhead to 0.8MB.

6.3 Performance Impact

The impact of retouch file generation on build time is negligible. OTA updates take 10 additional seconds to retouch (randomize) shared libraries, and after that there is no performance impact at boot or run-time.

7. DETECTING ASLR ATTACKS: CRASH STACK ANALYSIS

The difficulty of brute-forcing ASLR implementations has been studied extensively: Shacham et al. [23] demonstrate

Build Type	`/system/lib` Size (MB)	OTA Size (MB)
regular	25.63	40.96
prelinked (default)	24.63	40.75
prelinked and retouched	25.35	41.01

Table 6: Storage space impact of prelinking and retouching vs. a regular, non-prelinked build.

No ASLR (@-0)		ASLR (@744)	
Offset	Result	Offset	Result
0	correct	744	correct
1	crash (pc@-28)	745	crash (pc@-28)
2	crash (pc@-1)	746	crash (pc@-1)
3	crash (fr#2@6)	747	crash (fr#2@6)
4	crash (lr@4)	748	crash (lr@4)
5	infinite loop	749	infinite loop
6	crash (lr@-6)	750	crash (lr@-6)
7	stack corrupt	751	stack corrupt
8	crash (pc@-1)	752	crash (pc@-1)
9	crash (pc@-1)	753	crash (pc@-1)
10	crash (pc@9)	754	crash (pc@9)

Table 7: Results of simulated attack attempting to call `exit()`. The nominal (non-randomized) location was `0xAFD1977D`. The crash results indicate the reported location of the PC, LR, or stack frame #2 return address compared to the target address for the `exec()` function, thus for example "pc@-28" means that at the time of the crash the PC contained the attempted jump address minus 28.

how a process can be derandomized relatively quickly on a 32-bit OS with PaX enabled. While 64-bit address spaces make it harder to crack randomization, the fundamental concern about how much each element of the address space is randomized remains valid. In addition, 64-bit addresses are simply not available on many platforms that are of interest today and in the foreseeable future, including most that run the Android OS.

For massively deployed, networked platforms there is an additional risk. An attacker who chooses to keep a low profile can try to guess the randomization offset of a target device only once. While most of the time the guess will be unsuccessful, about $1/2^n$ of the targets will be compromised, where n is the number of bits of randomness introduced. For example if $n = 8$, one out of every 256 devices will be compromised, which can be a significant number.

We chose to tackle the problem of ASLR brute-forcing by focusing on detection by the OS. In related work, `segvguard` [21] attempts to do this in a very basic way by throttling the rate at which a process can be restarted on a single machine. The key observation is that brute-forcing inevitably leads to a significant number of process crashes before the attack succeeds. Importantly, we expect that the crash patterns are mostly invariant to relocation: library code crashes in a similar way regardless of where the target library is relocated. This insight leads to a detection algorithm which is much more reliable than the one used by `segvguard`, and which can also be applied in a centralized manner to detect low-profile brute-forcing attempts.

When a process crashes, it does so at a specific address of the program counter (PC). It is reasonable to expect that during unsuccessful ASLR brute-forcing attempts the address of the crash will be closely related to the guessed address of the jump. For example, if the function being exploited is at the non-randomized address `0xAD000000` and we guess incorrectly that it has been randomized to `0xAD002000`, we expect the crash to happen with a PC value close to `0xAD002000`. The reasoning is that we can only execute a

few instructions at a random position in memory without triggering a segmentation violation. Even more frequently the crash will be immediate due to an attempt to execute instructions from a non-executable memory page. It turns out that this idea is applicable to the ARM architecture if we also take into account the link register (LR) which commonly holds the return address when leaf functions are executed (this optimization avoids having to access the stack when making most leaf function calls). In our tests, during most ASLR brute-forcing crashes one of the two registers contains an address which is very close to the one that was guessed (Table 7).

Crash address traces.

In order to detect ASLR brute-forcing, we collected crash reports, and grouped crash addresses (PC and LR values) by their least significant 12 bits into *traces*. For every trace we counted the number of *distinct* addresses with the intuition that large traces will be present exactly when an attack is in progress. Our experiments confirm this (Section 7.2).

Android tombstones.

The Android runtime environment creates a crash dump file (called a *tombstone*) each time a process in the system exits abnormally. In addition, customer devices can report crashes to central servers, making a limited amount of information available for analysis (this information is retained only for a short time). We use these mechanisms to evaluate our crash stack analysis technique.

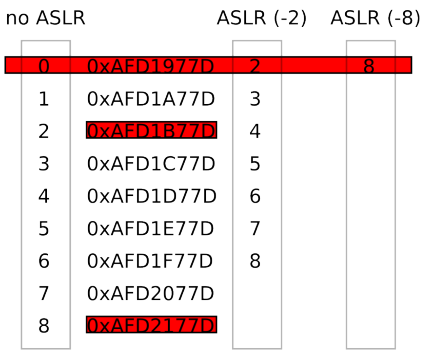

Figure 2: An attack guessing the same address over multiple randomized devices will manifest itself as crashes at the same offset on multiple pages: one long trace due to the frequency of immediate crashes at the called address. In this case, the attacker jumps to address 0xAFD1977D, and crashes at 0xAFD1B77C and 0xAFD2177C are observed on the two randomized devices on the right, after the PC in each report is derandomized. An attack making different guesses will create multiple long traces, each matching a particular crash behavior such as "pc@-28" or "lr@4" (not shown), in addition to the most frequent "pc@-1" behavior.

7.1 Evaluation via Simulated Attacks

We wrote a small piece of simulated attack code which attempts to execute the `exit()` function from `libc` by guessing its randomization offset. A successful run is one which produces the supplied exit code. We executed the attack code on a non-randomized system as well as on several randomized instances. Representative results are shown in Table 7. Clearly crash address patterns are retained across randomization, yielding identical crash offsets relative to the guessed address during brute-forcing. The pigeonhole principle implies that if a sufficient number of devices are attacked (e.g. a multiple of the size of the randomization space—several thousand in our case), we are guaranteed to see a long crash trace for at least one page offset: an attacker has no apriori information about the randomization at each device, so his guess about the address of the target code will be spread over the whole space of randomization offsets, *when taken relative to the randomization of each device* (see the example in Figure 2, based on the data in Table 7). This holds even when the attacker is making exactly the same guess across all target devices (a "normal" process crash will rarely, if ever, behave this way: the crash location will be consistently offset from the randomization base).

7.2 Evaluation Using Real Crash Reports

In order to estimate the likelihood of false positives generated by our crash analysis algorithm, we used the set of all 6805 crash reports (from close to 5000 different devices), generated by the `system_server` process on a specific build of the Android operating system. These reports did not contain any identifying information—in fact, the only data available was the program counter (PC), link register (LR), and return addresses in all stack frames at the time of the crash.

Our main goal was to confirm that in normal execution (without ASLR or brute-forcing attacks) crash address traces are short, and thus easily distinguished from those expected in attack scenarios. The longest trace we found had a length of 4 (Table 8), while any successful attack that attempts to brute-force the current ASLR implementation will inevitably create a trace of size close to 1024 (the number of different randomization offsets used in our implementation) over a relatively small number of attacked devices. Thus, crash address trace size is an excellent indicator that can be used to detect ASLR derandomization attempts.

7.3 Implementation Notes

In our experiments we have evaluated crash stack analysis deployed as a cloud service which can continually monitor crash reports from user devices, grouping the reports by device ID, process name, and build number, and looking for telltale crash address traces.

Local detection.

Crash stack analysis can be also run locally on a device, in order to detect and immediately block attempted ASLR brute-forcing. Such attempts carry a signature which is distinct from that of a process repeatedly crashing on some error condition. In this context, blocking the attack can involve preventing the automatic restart of the crashing process.

In order to implement local detection, the algorithm for building crash address traces needs to be modified. While it may be sufficient to look for process tombstones that match exactly on their 12 least-significant bits but differ on the next 10, looking at *similar* offsets might yield more accurate

Offset (bits 0-11)	Address	Library
0xCF4	0xAC04CCF4	libskia.so
	0xAD012CF4	libdvm.so
	0xAD035CF4	libdvm.so
	0xAD214CF4	libnativehelper.so
0xCB8	0xAD00ECB8	libdvm.so
	0xAD018CB8	libdvm.so
	0xAD041CB8	libdvm.so
0x95C	0xAD00F95C	libdvm-ARM.so
	0xAD01395C	libdvm.so
	0xAD3EB95C	libandroid_runtime.so
0x260	0xAC072260	libOpenVG_CM.so
	0xAC08A260	libOpenVG_CM.so
	0xAF90B260	libcutils.so

Table 8: Largest crash address traces, listed by tag, obtained from 6805 actual device crash reports for `system_server`. Small trace size indicates no ASLR brute-forcing (as expected).

detection. The reason for this is that not all crashes happen at exactly the same distance from the guessed function address (Table 7), and at the same time on a single device it is unlikely that a "regular" crash will exhibit a crash pattern with similar page offsets in a number of different memory pages.

Protecting user privacy.

Central reporting of device information such as crash data, always has the potential of violating user privacy. In Android, there are several safeguards: first, crash reports contain only a minimum amount of data necessary to identify the location of the problem: register and stack contents, and minimal memory contents pointed to by *instruction* registers such as LR and PC in ARM. Second, reports are only retained for a small amount of time, on the order of days. Finally, access to reports is highly restricted even within the Android team. Crash stack analysis can work within the existing privacy safeguards, without the need to disclose any additional device or user information.

Reacting to attacks.

The primary use of crash stack analysis is to identify attacks that are in progress—almost in real time. There could be a variety of responses to such attacks: from quickly finding and patching the root cause (the vulnerability which made the brute-force ASLR attack possible in the first place), to restricting device access at the network level with the cooperation of carriers, or even alerting potentially affected users.

8. EXTENSIONS AND LIMITATIONS

We briefly mention a few extensions of the retouch approach and some limitations which may encourage further work.

Same random offset across processes.

A limitation of all major ASLR implementations to date [17] including ours, is that all processes on a single device have the same shared library layout. This is necessary to not effect system performance. To exploit this limitation, however, the attacker must already have a foothold on the device, which is not the intended threat model for ASLR.

Not using ELF utilities.

Retouching can be performed by retaining all relocation information for shared libraries and prelinking at update time. At that time base executables could also be linked at a randomized address. However, the space overhead of such implementation would be substantial, due to the larger (uncompressed) size of relocation sections and the need to include ELF libraries and a linker in the updater binary. The added complexity would also be significant as there would have to be a method for undoing the randomization to accommodate future incremental updates.

Non-prelinked libraries and mmap randomization.

While our retouching technique is applicable to non-prelinked shared libraries, randomization for this area of memory is best achieved via the PaX approach: by randomizing the `mmap` base. This protection is already in place, and also extends to file data mapped by processes. Additionally, the majority of shared libraries in Android are already prelinked, and the expectation is that with time non-prelinked libraries will become increasingly rare.

Recent advances in attack techniques, such as JIT spraying [4], have cast new doubts about the effectiveness of ASLR in preventing exploits. We note that randomization of the `mmap` region in the process address space effectively neutralizes such concerns: the Dalvik VM uses a small mmap'ed area to store executable JIT output, and thus inherits the benefits of randomization.

8.1 Heap and Stack Randomization

So far we focused on randomization of executable memory because the heap and stack areas of a process are already randomized via traditional techniques. For example, the Android kernel already performs stack randomization for each process. Heap randomization is performed by either randomizing the location of `brk` or by modifying *malloc()* to allocate space randomly. Additional pointer protection features have been available for several years, both in allocator implementations and as compile-time options [6]. Android uses `dlmalloc` which offers some overflow protection for allocated chunks, and ProPolice [9] which compiles the use of stack canaries into native binaries.

Stack randomization in userspace.

Since retouching requires only userspace OS modifications, we also explored userspace techniques for stack randomization. One approach is to modify the code in `bionic/linker/arch/arm/begin.S` used as a prologue in the linker binary. This code normally invokes *linker_init()*, which in turn loads the base executable and returns its address. We modified this prologue code in `begin.S` to add a random number of harmless additional lines to the process environment strings before invoking the dynamic linker. The resulting executable prologue looks as follows:

```
_start:
        /* BEGIN RANDOMIZATION CODE */
        mov     r0, sp
        mov     r2, sp
```

```
        sub     r2, r2, <RAND>
aslr_args:
        ldr     r1, [r0]
        str     r1, [r2]
        add     r0, r0, #4
        add     r2, r2, #4
        cmp     r1, #0
        bne     aslr_args
        sub     sp, sp, <RAND>
        /* add more env[] strings */
        adr     r1, ASLR_ENV_PAD
aslr_pad:
        str     r1, [r2]
        add     r2, r2, #4
        cmp     r2, r0
        bne     aslr_pad
       /* END RANDOMIZATION CODE */

        /* original code */
        mov     r0, sp
        mov     r1, #0
        bl      __linker_init

        /* linker init returns the
           _entry address in the
           main image */
        mov     pc, r0

        .globl ASLR_ENV_PAD
ASLR_ENV_PAD:
        .ascii "ASLR_ENV_PAD=1\0"
```

Since the environment is on the stack, this has the effect of shifting the process stack by a small, random number of bytes (note that here we don't specify the source of randomness; `<RAND>` stands for a register that holds a random value, perhaps based on the current time and/or stack contents). We verified that with this change, using an arbitrary 4-byte aligned random offset, the system boots successfully and process stacks are shifted down as expected. The drawback of performing this type of stack randomization is that the added offset can only be relatively small—on the order of hundreds or thousands of bytes, and thus may not prevent some buffer-overrun attacks. The existing kernel implementation is more robust as it randomizes the more significant bits in the stack location while it also does not waste physical memory.

Applicability to other platforms.

Since most code running on Android devices is written in Java, and thus not vulnerable to buffer-overflow attacks, mobile ASLR is even more applicable to platforms that run primarily native applications, such as iOS. While we have used Android to develop and demonstrate our approach, we expect broader adoption across the different smartphone ecosystems.

9. RELATED WORK

For real impact on security, ASLR must be deployed in conjunction with protections against injecting and executing code on the system. Our work is complementary to current work on write-protecting executable pages in Android.

In spirit, retouching is related to Address Space Layout Permutation (ASLP), proposed in [12]. ASLP performs modifications on base executable files at launch time by using the retained relocation section in each executable. However, for shared libraries, ASLP defaults to the standard kernel-based approach of *mmap()* randomization, without performing any fine-grained permutation of code. This approach cannot work in Android due to prelinking of shared libraries. Our retouching approach can randomize shared libraries and works well with prelinking, without requiring any kernel modifications or executable file editing at runtime; in addition, retouching addresses the randomization of the smaller, non-self-relocating dynamic linker in Android called `linker`.

Retouching is also conceptually similar to the Windows utility `rebase` [2], which allows a user to manually move the starting offset of an executable or DLL file by executing relocation in advance. However `rebase` has no facility for supporting software updates, prelinked libraries, or randomization. The relevance of `rebase` has declined once Windows implemented ASLR.

In related work, *address obfuscation* has been proposed as a way to achieve higher levels of randomization, beyond those achievable in ASLR [3]. Similar to address obfuscation, retouching starts during build time and completes at install. In contrast to it, retouching does not involve any complicated transformations on the code or data sections. This should result in lower overall risk at deployment as well as possibly better performance, since shifting the whole executable object by a multiple of the CPU page size is generally expected to have no impact on caching. A similar technique called *code islands* has been proposed, targeting large multithreaded server deployments [27].

Randomization can reach beyond the layout of a single, user-space process. The kernel stack can be also randomized, as well as system calls [5] and even the CPU instruction set [11]. Retouching is complementary to all of these mechanisms, providing an efficient, effective, and simple way to reduce the attack surface of processes.

A different thread of work has investigated control-flow hijacking prevention in embedded devices lacking modern CPU capabilities such as a MMU [10]. We note that in their architecture smartphones are more similar to desktop PCs than simple microcontroller-based devices. In this sense, desktop-grade protection techniques are more relevant to our work.

Privilege escalation techniques have also been explored in the context of Android inter-process communication via the Intent mechanism [7]. Our work is inherently at a lower level in the stack, however ASLR can still help prevent applications from being exploited through native code vulnerabilities, thus closing some of the possible routes to abuse.

Finally, ASLR has been evaluated in the past, and often found to have limited effectiveness [23], or to be poorly implemented [15, 8]. Shacham's Return-Oriented Programming technique [22] demonstrates that preventing code injection offers little protection on its own; in addition, return-oriented programming can be used when some executable pages are left non-randomized—this highlights the need to randomize all binary code in the system. In Section 7 we show that crash stack analysis can be used to rapidly detect and block ASLR brute-forcing. At the same time, control over all the shipping native code in Android makes complete randomization of the process address space feasible; this will

help avoid many of the mistakes or omissions made by early ASLR implementations for the desktop.

10. CONCLUSION

This paper introduces a new technique for implementing ASLR, which is particularly well-suited to the constraints imposed by modern consumer-oriented mobile devices. Our approach, called retouching, can randomize the location of all native executable code without kernel modifications, and without erasing the savings in boot time afforded by prelinking. We implemented retouching-based ASLR on the Android platform, and evaluated its impact on the system: from building to OTA updates and execution. We also developed and evaluated crash stack analysis, a technique for detecting ASLR brute-forcing attempts which is the only one we are aware of that uses crash address information to reliably detect targeted attacks. We conclude that retouching, in combination with crash stack analysis, is a robust ASLR implementation, resilient to brute-force derandomization.

11. REFERENCES

[1] Android. `www.android.com`.

[2] Ruediger R. Asche. Rebasing win32 dlls: The whole story, 1995. `http://msdn.microsoft.com/en-us/library/ms810432.aspx`.

[3] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *In Proceedings of the 12th USENIX Security Symposium*, pages 105–120, 2003.

[4] Dion Blazakis. Interpreter exploitation: Pointer inference and jit spraying, 2010. `http://www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf`.

[5] Monica Chew and Dawn Song. Mitigating buffer overflows by operating system randomization. Technical report, UC Berkeley, 2002.

[6] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard^{TM}: Protecting pointers from buffer overflow vulnerabilities. In *In Proc. of the 12th Usenix Security Symposium*, 2003.

[7] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In *ISC*, pages 346–360, 2010.

[8] Jake Edge. Linux aslr vulnerabilities, 2009. `http://lwn.net/Articles/330866/`.

[9] Hiroaki Etoh. Gcc extension for protecting applications from stack-smashing attacks, 2005. `http://www.research.ibm.com/trl/projects/security/ssp/`.

[10] Aurélien Francillon, Daniele Perito, and Claude Castelluccia. Defending embedded systems against control flow attacks. In *SecuCode '09: Proceedings of the first ACM workshop on Secure execution of untrusted code*, pages 19–26, New York, NY, USA, 2009. ACM.

[11] Gaurav S. Kc. Countering code-injection attacks with instruction-set randomization. In *In Proceedings of the ACM Computer and Communications Security (CCS) Conference*, pages 272–280. ACM Press, 2003.

[12] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 339–348, Washington, DC, USA, 2006. IEEE Computer Society.

[13] J. Krhovjak, V. Matyas, and J. Zizkovsky. *Generating Random and Pseudorandom Sequences in Mobile Devices*, pages 122–+. Springer, 2009.

[14] David Litchfield. Buffer underruns, dep, aslr and improving the exploitation prevention mechanisms (xpms) on the windows platform, 2005. `http://www.ngssoftware.com/papers/xpms.pdf`.

[15] Charlie Miller. Owning the fanboys: Hacking mac os x, 2008. `http://www.blackhat.com/presentations/bh-jp-08/bh-jp-08-Miller/BlackHat-Japan-08-Miller-Hacking-OSX.pdf`.

[16] Charlie Miller. Fuzzing the phone in your phone, 2009. `http://www.blackhat.com/presentations/bh-usa-09/MILLER/BHUSA09-Miller-FuzzingPhone-PAPER.pdf`.

[17] John Moser. Prelink and address space randomization, 2006. `http://lwn.net/Articles/190139/`.

[18] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib(c). In *ACSAC '09: Proceedings of the 2009 Annual Computer Security Applications Conference*, pages 60–69, Washington, DC, USA, 2009. IEEE Computer Society.

[19] Clint Ruoho. Aslr: Leopard versus vista, 2008. `http://www.laconicsecurity.com/aslr-leopard-versus-vista.html`.

[20] Mark Russinovich. Inside the windows vista kernel: Part 3, 2007. `http://technet.microsoft.com/en-us/magazine/2007.04.vistakernel.aspx`.

[21] segvguard. `http://www.daemon-systems.org/man/security.8.html`.

[22] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86. In *In Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[23] Hovav Shacham, Eu jin Goh, Nagendra Modadugu, Ben Pfaff, and Dan Boneh. On the effectiveness of address-space randomization. In *In CCS'04: Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 298–307. ACM Press, 2004.

[24] Brad Spengler. Pax: The guaranteed end of arbitrary code execution, 2003. `http://grsecurity.net/PaX-presentation_files/frame.htm`.

[25] The PaX Team. Homepage of the pax team, 2008. `http://pax.grsecurity.net/`.

[26] Ollie Whitehouse. An analysis of address space layout randomization on windows vista, 2007. `http://www.symantec.com/avcenter/reference/Address_Space_Layout_Randomization.pdf`.

[27] Haizhi Xu and Steve J. Chapin. Improving address space randomization with a dynamic offset randomization technique. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 384–391, New York, NY, USA, 2006. ACM.