ASLR and ROP Attack Mitigations for ARM-based Android Devices

Vivek Parikh and Prabhaker Mateti
Amrita Center for Cybersecurity Systems and Networks
Amrita School of Engineering, Amritapuri,
Amrita Vishwa Vidyapeetham, Amrita University, India
Wright State University, Dayton, OH 45435, USA
viv0411.parikh@gmail.com pmateti@wright.edu

Abstract

ASLR (address space layout randomization) and ROP (return oriented programming) attacks have been happening for years on the PC platform. Android devices are ripe for these same attacks. Android has made mitigation efforts, mostly in the Zygote (mother of all Java processes), which is presently exposed to a vast number of ASLR bypassing exploits. We carefully re-analyzed the Zygote process creation model. We include mitigations not only for ASLR but also for ROP attacks. We demonstrate that Android becomes robust against most of the ROP exploits by running such attacks on the device, in the presence of our solution. We compare our solution with existing solutions and show that ours is a more effective approach to mitigate ASLR and ROP attacks on ARM based Android devices. Our changes do not interfere with the normal functioning of the Android device and can be easily incorporated as a secure replacement for the existing Zygote that is presently exposed to a vast number of ASLR bypassing vulnerabilities.

Keywords:

Android; Android Framework Services, Crowd Sourcing; Verifying Trust; Security Exploits, ROP, ASLR, ARM, Zygote, Morula.

1 Introduction

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. One such technique that has recently been gaining popularity is the class of attack that bypass software defense mechanisms such as non-executable memory, address randomization etc. Return oriented programming (ROP) has emerged as one such prime exploitation technique.

The objective of the work reported here is to secure Android devices from ASLR and ROP attacks. The existing (2017) Zygote model permits these attacks. The Stagefright [Drake, 2015] exploit is based on ROP attacks.

1.1 Organization

Background needed for this paper is provided in Section 2. Section 3 describes tools and techniques that we use in our design. Sections 6 are about related work and evaluation. We conclude the paper with Section 7.

2 Background

2.1 ASLR

Address-Space Layout Randomization (ASLR) is a technique to thwart exploits which rely on knowing the location of the target code or data. ASLR randomizes addresses of methods and data as executable is loaded into virtual memory. When implemented correctly, ASLR makes it impossible to infer the location of the code and data of a program.

Android 4.0 Ice Cream Sandwich introduced ASLR. Library load ordering randomization was accepted into the Android open-source project in 2015, and was included in the Android 7.0 release.

The address space layout randomization also has vulnerabilities. Shacham et al. [2004] points out that the ASLR on 32-bit architectures is limited by the number of bits available for address randomization. Only 16 of the 32 address bits are available for randomization, and 16 bits of address randomization can be defeated by brute force attack in minutes. For 64-bit architectures, 40 bits of 64 are available for randomization. In 2016, brute force attack for 40-bits randomization are possible, but it is unlikely to go unnoticed.

2.2 ROP

Return oriented programming is a growing class of exploits in which the exploit consists of re-using code that is already present in the virtual memory of the system.

Return oriented programming was discovered in 2007 by Shacham [2007]. The authors used it to bypass the limitations of executable stack such as DEP, Stack Canaries, etc. in jumping to valid code regions avoiding the need to place shellcode on the stack. They bypassed the hardware enforced NX (non-executable protection) in Intel, XN in ARM and software enforced DEP (Data execution prevention) with the ROP technique. ROP was also shown to be applicable on ARM architecture (Kornau [2010]).

The most common type of attack is attack is Return-to-LibC (Ret2LibC), where an attacker exploits a buffer overflow to redirect control flow to a library function already present in the system. To perform a Ret2LibC attack, the attacker must overwrite a return address on the stack with the address of a library function. Additionally, the attacker must place the arguments of the library function on the stack in order to control the execution of the library function.

2.2.1 Basic Idea of an ROP Attack

First the attacker places the payload consisting of return oriented instructions in the memory area. This payload is not the actual exploit code but merely contains the pointers to gadget chains in memory. After that the attacker exploits a heap/stack corruption vulnerability and the stack is pivoted to the attacker region. Now the instructions (generally) ending with `ret` will redirect the execution flow to ROP gadgets that are placed in memory. Note that this is just one of the many scenarios that make use of ROP gadgets.

2.2.2 ROP Gadgets

Short sequences of instructions that end with a transferring instruction (e.g., `ret` on x86, blx on ARM) are called gadgets. Shacham [2007] showed that ROP gadgets are Turing complete, given a binary of sufficiently large size. Even when the program is not large enough, assuming the attacker has already injected code in a writable program area, the following steps can be taken to induce a Turing complete behaviour:

1. Compose minimum ROP gadgets to make a call to allocate memory (Using alloc for instance)

2. Copy shellcode which is in writable memory area to the newly allocated memory region

3. Run shellcode from the newly allocated region

There is yet another approach if the attacker has already access to memory modification routines such as VirtualProtect on Windows and mprotect on Linux. Compose ROP shellcode that makes a call to memory modification routines such as mprotect

We ran ARM shellcode on Android and found that ROP shellcode/payload is perfectly possible on ARM.

Load/Store (i) Loading a Constant (ii) Loading from Memory (iii) Storing to Memory

Control Flow Unconditional Jump, Conditional Jumps

Arithmetic & Logic Add, Exclusive OR (XOR), And, Or, Not, Shift and Rotate

The ropper[1] tool discovered on the ARM `/bin/ls` binary a staggering 1112 ROP gadgets. This shows that the incentive for attackers is quite the same on ARM architecture as on x86(-64) architecture.

```
0x0000fee4: adc.w r0, r0, r3; bx
 lr;
0x0001540e: adc.w r1, r1, r4, lsl
 #20; orr.w r1, r1, r5; pop {r4,
 r5, pc};
0x0001529e: adc.w r2, r2, r2; it
 hs; subhs.w r0, r0, r1; mov r0,
 r2; bx lr;
0x0001540a: adcs r0, r0,#0; adc.w
 r1, r1, r4, lsl #20; orr.w r1, r1
 , r5; pop {r4, r5, pc};
0x0001551a: adcs r1, r1; it hs;
 orrhs r1, r1, #0x80000000; pop {
 r4,r5,pc};
```

Listing 1: An ROP Gadget Found in ARM `/bin/ls`

2.2.3 Defenses Against ROP

Several defensive approaches have been suggested: ROPDefender [Davi et al., 2011], kBouncer Pappas et al. [2013], dynamic binary instrumentation (DBI) based ROP protection (see Section 3), and instruction set randomization. Defences like ROPDefender and kbouncer do not require any modifications to the binary. These techniques often take into consideration ret based ROP gadgets only. But Checkoway et al. [2010] showed that ROP is also possible when ret gadgets are not used.

[1] https://github.com/sashs/Ropper

2.3 Position Independent Executables

Position Independent Executables are binaries constructed so that that their entire code base can be loaded at a random location in memory.

Position-independent executable support was added in Android 4.1. Android 5.0 dropped non-PIE support and requires all dynamically linked binaries to be position independent.

Full ASLR is achieved when applications are compiled with PIE (`-fpie -pie` flags).

2.4 ART Format

ART was introduced with Android 5.0 and is now the default runtime. ART compiles all the code into `oat` format, translating a `dex` file with ahead-of-time (AOT) version. The `oat` format is close to ELF format that Linux uses. The opcodes from the `oat` file can be utilized as a ROP gadgets. In our opinion, the new ART format has increased the attack surface in Android.

3 Binary Instrumentation

Dynamic Binary Instrumentation (DBI)[Backes et al., 2016] is an introspection technique in which a binary is re-built just before being run with added hooks that invoke certain callback routines. These callback functions help identify events that happen during execution. These hooks can be called at various points in the program execution to facilitate advanced tracing and analysis of binary. DBI does not require access to source code and is therefore perfectly suitable for third party programs whose source code cannot be accessed. It is also useful for legacy applications which are no longer maintained. DBI does not incur significant performance overhead if implemented properly. The dynamic compilation feature offered by DBI can be used in our case to detect ROP attacks. We plan to use selective DBI techniques described below. Selective Monitoring takes constant feedback from the cloud and gathers information about the current exploitation trends(both from the user's device as well as from national security advisories). It monitors apps whose critical

factor exceeds a certain threshold. Critical factor is defined as the probability of the app getting exploited in wild by the attackers. This probability will be constantly updated through inputs from crowd sources.

3.1 Android Dynamic Binary Instrumentation

ADBI[2] is a tool for dynamically tracing Android native layer. Using this tool you can insert tracepoints (and a set of corresponding handlers) dynamically into the process address space of a running Android system. When the tracepoint is hit your custom handler (which can be written in C) is executed. You can deliver your own code through the handlers. It is possible to access process variables and memory. Host side tool written in Python communicates with the native adbiserver process (which resembles the gdbserver in its operation) and translates source level symbols into addresses within the final binaries.

3.2 Valgrind

Valgrind[3] is a Dynamic Binary Analysis (DBA) tool that uses DBI framework to check memory allocation, to detect deadlocks and to profile the applications. Valgrind tool loads the application into a process, disassembles the application code, add the instrumentation code for analysis, assembles it back and executes the application. It uses Just In time Compiler (JIT) to embed the application with the instrumentation code.

3.3 DynamoRIO

DynamoRIO[4] is also a DBI tool. It is supported on Linux, Windows and Android.

4 Architecture and Design

We describe a method that can monitor the critical points at the time of execution of an Android binary by leveraging DBI.

4.1 Pre-Exploitation Phase Module

- Trace execution flow using DynamoRIO to see last n branches executed.
- Check to see if the branches contain a return instruction. If they do, verify(from the stack) if the address taken is genuine.
- Keep last N branches' history in a remote/local database

4.2 Post-Exploitation Phase Module

We use a DynamoRIO based plugin to detect any post exploitation privilege escalation. E.g., `java.lang.Runtime.exec("su")` is generally used for getting `su` privileges using `setuid(0)` system call. We detect such functions using a global hook. Another example is use of network system call after `mprotect/mmap`. We intend to detect such sequences which might be harmful through our framework.

The following modules work concurrently to ensure that the ASLR is not violated by any of the running apps.

4.2.1 ROP Gadget Database

The database contains all the latest ROP gadgets for all the apps that are installed on the device. We plan to store ROP gadgets of all Android libraries and store the gadgets in a cloud database. This way, during instrumentation, we would we able to download the ROP database and compare the ROP instructions with the binaries. We plan to use Firebase for cloud storage and retrieval. This database will be stored

4.3 Instruction Analyzer DBI Module

The conditional branch logger can verify whether or not an indirect branch has been taken. It further inspects the return instruction at the end of a basic block. It then inspects the contents of LR register to find out which address is being returned to. The module will then analyze the instruction present at

[2] https://github.com/Samsung/ADBI
[3] http://valgrind.org/
[4] http://www.dynamorio.org/

that address to find out if its an instruction preceding by a CALL instruction (Intel) or a BL(X) LR instruction in ARM. If they are, then its execution would be stopped. It will be protecting against active threats by continuously monitoring execution of each process. This tool will trigger an alert whenever a `ret` instruction returns to an address which is not preceded by a call instruction

The monitor is able to notify the background daemon in case there happens to be a violation of integrity of app behaviour. If the monitor is able to detect any live exploits that are taking place it will immediately notify the background service which will terminate the application in context.

The following indirect branches (on ARM) are inspected:

- A load instruction (LDR or LDM) with the PC as one of the destination registers
- A data operation (MOV or ADD, for example) with the PC as the destination register
- A BX instruction, that moves a register into the PC
- A SVC instruction or an Undefined Instruction exception
- All other exceptions, such as interrupt, abort, and processor reset.

4.4 Vulnerable App Database

This database contains information about all the apps that are getting exploited, as gathered through crowd sources. Such apps will be blacklisted and will not be allowed to run.

4.5 Crowd Sourcing

We use crowd sources to collect trust in Android apps. We have a remote MD5/SHA1 database which is periodically updated. We plan to improve the database by regularly adding the latest exploits that are prevalent in the Android ecosystem. As far as we know, this is the first implementation that leverages

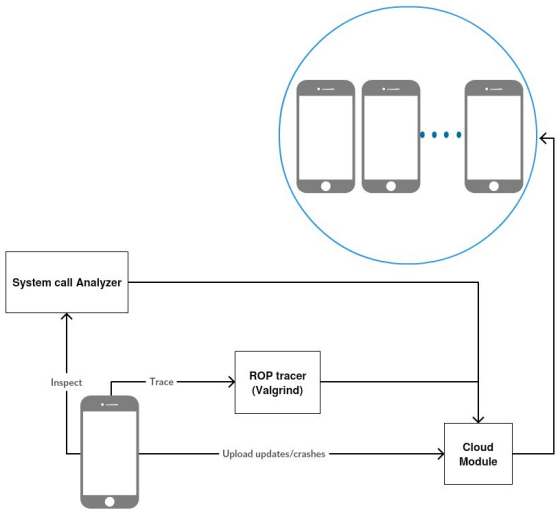

Figure 1: ROP Protection Tool Outline

the crowd sourcing philosophy to improve the existing condition of ASLR in Zygote module and provides strong defence against current ROP exploits. Granted that crowd sourcing will not protect against latest 0-day exploits in the wild. Against such exploits we plan to introduce further randomization in the Android source code (AOSP).

4.5.1 Suspicious System Call Sequence Analyzer/ Database

The end goal of an attacker is generally to take control of the system after a successful exploit attempt. In order to do so he/she must execute a series of system calls which help him/her do exactly that. A suspicious system call is executed in which case the information will be sent to the crowd source module immediately.

This will feature a (nearly) comprehensive list of system calls that can be taken by the ROP shellcode. This analyzer will constantly monitor an app for a dangerous sequence of system calls being executed. This will work post infection after a shellcode is able to detect the protection offered by the conditional branch logger.

4.6 Algorithm 1

The following algorithm is adapted[5]

```
for each IMAGE
  for each BLOCK in IMAGE
    insert BLOCK in BLOCKLIST
      for each INSTRUCTION in BLOCK
        if INSTRUCTION  is RETURN or
    BRANCH
          insert retrieve SAVED_EIP
   from stack
          insert CALL to
   ROP_VALIDATE(SAVED_EIP) before
   INSTRUCTION
          ROP VALIDATE
          if SAVED_EIP not in
   BLOCKLIST
             exit with error warning
```

Listing 2: Instrument Program

Notes: (i) The type of branch has not been specified. It has to be an indirect branch. (ii) Code to retrieve saved EIP might not be easy to construct.

4.7 Algorithm 2

The following algorithm is adapted[6]

```
PRE-PROCESSING STEP
for_each image in process
    for_each bb in image
       Bblist.push_back(BBInfo(bb))

DETECTION STEP
for_each ins in Program
    if IsIndirectBranchOrCall(ins)
        if BranchDest(ins) is
   InsideLoadedModules()
            if BranchDest(ins) not
   in Bblist.Instructions()
                ShellcodeDetected()
```

Listing 3: Pre-Processing and Detection

[5] http://www.talosintelligence.com/
[6] http://public.avast.com/caro2011/

This algorithm works for all types of branches and not just return instructions. But, as a side-effect, there is greater performance overhead.

4.8 Algorithm 3

1. Find out all valid call sites in a binary (Parse Export Add all valid call sites to a block list.
2. If the BB ends with a call or ret instruction then (insert code to) check the starting address of the next BB. It should either be a valid call site (from the block list) or should be an address just below a valid call site. In ARM this would be PC+4
3. Else Continue
4. Note: BB is computed before the image begins execution

4.9 Algorithm 4

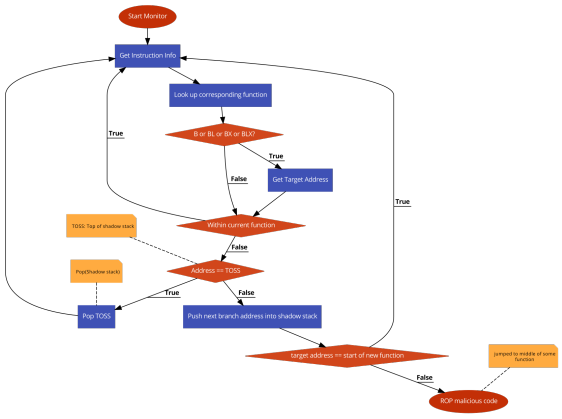

Figure 2: ROP Detection Algorithm [from Huang et al. [2012]]

Criticism: Need to trace each instruction

4.10 Algorithm 5

1. Do instruction tracing (i.e., instrument each instruction to figure out the return instruction(usually at then end of BB)

2. Find out where it is jumping (Intel: Return address is on stack, ARM: Return address is in LR)

Criticisms: 1. Tracing each instruction will be slow. 2. Only checks return instruction.

4.11 Algorithm 6

Algorithm of shadow stack[7].

4.12 Algorithm 7

1. At the end of each basic block, we check whether a `ret` (return) instruction is taken (In ARM the equivalent is the POP PC or MOV PC, LR) by analyzing the branch instruction at the end of basic block.
2. Check the top of the stack (or LR register in case of ARM) in case a return instruction is taken. Let the address be A.
3. If (Disassemble(A-4) == call instruction) then it is a genuine function returning back from where it was called.
4. Else alert the cloud module about an ROP attack.
5. Note:A basic block is always ended by a direct/indirect branch

Comparison with Algorithm 2: This algorithm does not need to calculate the valid call sites Might not work if lr register is updated inside a Basic Block(For such a scenario you may need to execute the instrumentation function after the Basic block is executed and not before) Performance is yet to be compared between these two algorithms.

4.13 Suggested Performance Improvements

For making DBI faster we can use a technique of selective binary instrumentation that only targets critical applications in the device which may have a higher probability of getting exploited in the wild. We are certainly aware of the performance hit. That is why we will plan to have selective instrumentation (ie. the binary will not be instrumented every time it runs. e.g. if a binary runs for 100 times then at 101th time it will not be instrumented, or instrument at random times). Also researchers have used the technique of dynamic binary instrumentation in the past to detect ROP attacks.Specifically for ARM/Android the following paper has used Valgrind based instrumentation to detect ROP.

[7] https://github.com/benwaffle/DynamoRIO-shadow-stack

5 Implementation

5.1 Analysis of Zygote

Whenever a new app has to be launched, Zygote forks off its own process. The base zygote process has already loaded most of the framework core libraries. After forking, the process inherits all the libraries associated with parent zygote process. After creating a process, the Zygote does several things like assigning group id, etc to the apps. In `init.rc` file there is a file service Zygote `/system/bin/app-process -X Zygote /system/bin --Zygote --start-system-server` The aim is to improve the existing security of current Zygote implementation. We will be testing existing Android ASLR exploits on the current security implementations and demonstrate that they are not nearly enough for security of the device. Then we will demonstrate how Zygote4 can stop those exploits owing to its new and improved defenses against ROP init runs `/system/bin/app-process` ROP is actually possible on Android ARM.

Due to Zygote design the effectiveness of ASLR is undermined as all forked processes from Zygote inherit the same memory layout of the shared libraries. Also multiple copies of the same process share the exact same code layout. This greatly reduces the security provided by ASLR and makes the device vulnerable to several kinds of attacks against randomization.

5.2 Creating Our Own ROP Sample

We were able to create a self executing ROP executable. It behaves exactly like a ROP payload. We developed it for X86, ARM and Android(X86 and ARM). This executable scans in the memory for `/bin/sh` (`/system/bin/sh` on Android) and also locates the address of the `system()` function. It then spawns a shell (with the same privileges as user) by pushing arguments to system() on the stack (or on register in case of x86-64 and ARM). We use the inline assembling capabilities of gcc (clang). For Android, we use JNI to invoke the native ROP code.

Our POC is perhaps the simplest way of demonstrating a ROP execution without incurring any complications that may arise due to change in addresses due to ASLR. The project is located at https://github.com/techvoltage/addr-info. The pseudo code for the same sample is shown below.

```
  function find_func(char *func,
   void *ptr) {
  do {
    iterate modules in virtual
   memory;
    void *handle = dlopen (module);
    ptr = dlsym(handle, func);
  } while(module or !ptr);
  leave;
}

function find_pattern(char *pattern,
    char *arg){
  arg=memmem(pattern);
  call find_func("system", ptr);
  call find_pattern("/bin/sh", arg);
  `mov r0, arg`; //on x86: push
   arguments to register
  `blx ptr`; // call system("/bin/sh
   ")
  }
```

Listing 4: Self ROP Sample: Pseudo Code

5.3 Continuous Monitoring of App Execution on Android

We have a native Android self exploiting executable. Hello ROP is an Android sample that uses JNI to execute code in a ROP-like manner from an Android Java Activity. It locates system() and `/system/bin/sh` at runtime to execute a shell (using inlined ARM assembly), in a ROP like manner.

We also have a DynamoRIO plugin for detecting ROP attacks on the Android architecture. We run the executable under our DynamoRIO plugin. We are able to run the app with our plugin with the help of setprop command which acts as a wrapper for the app. Our DynamoRIO plugin runs in the background and continuously monitors the executable. Now normally the app should be able to execute its code in a ROP like fashion pretty easily but due to our plugin verifying all the calls and returns, an assertion error is throwed when we detect that a ROP like execution (meaning the x86(ARM) calling conventions are most certainly violated) As soon as the attack is detected, DynamoRIO immediately stops the executable.

DBI plugin monitors all the return instructions and checks to see whether there is a discrepancy at the call site. If the target instruction of the return instruction does not immediately precede a call instruction. If the heuristic fails, the application is deemed to have been exploited. We also plan on introducing an impact factor metric to reduce the number of false positives.

6 Related Work

6.1 Zygote to Morula Enhancements

Lee et al. [2014] demonstrate the weakness of the existing Android Zygote implementation and tries to introduce a new replacement, Morula instead of the weak(from security perspective) Zygote model. Morula promises full ASLR support for the processes spawned. Unlike Zygote which only uses fork() to create new processes , Morula uses fork() along with exec() to bring full ASLR support on Android platforms. In existing Android systems the apps have the same code base address even when ASLR is present.

In Morula there is a pool of Zygote processes which are maintained at all times. When a new app starts, the process will inherit from any one of these Zygote processes Legacy apps tend to have a heavy usage of native code. This code is loaded through JNI-like interfaces The authors further highlight the weakened ASLR model of Android by devising two real exploits on Android apps. These exploits break aslr and achieve ROP on current systems They then designed Morula as a countermeasure and implemented it as an extension to the Android OS. By leaking an address in his/her own process , an attacker can relate the address to another app which needs too be exploited , thus providing a memory disclosure vulnerability. This works because Zygote causes two child process to have the same memory and thus revealing memory of one sibling process will also let us find the corresponding memory location in another sibling process.

Shetti [2015] did a further enhancement to the Morula framework. ASLR in 32 bits is weaker even when best randomization practices are followed. This is due to the fact that sufficient entropy is simply unattainable in 32 bit architecture. However, the same cannot be said for 64 bit architectures as a wide address size (8 bytes) offers a considerable advantage over 32 bit architecture. Due to the huge virtual memory it becomes relatively easy improving the existing ASLR techniques on 64 bits.The idea of dynamic offset randomization is quite effective against de-randomization exploits against Android Zygote. Enhanced Morula's process creation model and Randomization highlights shortcomings with Morula framework. It proposes Zygote3, an enhancement of Morula which features Dynamic offset randomization and base pointer randomization.

6.2 Kbouncer and Related ROP Mitigations

Kbouncer(Carlini and Wagner [2014]) uses the last branch tracing functionality provided by the intel architecture. Kbouncer uses hardware support for tracing indirect branches. It inspects the history of indirect branches taken at every system call. Their implementation consists of three components: An offline gadget extraction and analysis toolkit, A userpace layer and a kernel module to modulate all the system calls being passed to the kernel and also to log all the indirect branches being taken by the application.

7 Conclusion

In this report, we how the Android Zygote is still not secure and proposed our new framework as a countermeasure. Our framework promises to be an enhancement over the past work involving Zygote. The design is based on an open source ideology and can serve as a better alternative to the traditional ROP protections. One of the possible challenges that we face is the process slowdown introduced by dynamic binary instrumentation. We expect to solve such issues as we further make progress in our research. For making DBI faster we can use a technique of selective binary instrumentation that only targets critical applications in the device which may have a higher probability of getting exploited in the wild.

References

Joshua Drake. Stagefright: Scary code in the heart of Android. *BlackHat USA*, 8 2015. slides: https://www.blackhat.com/ docs/us-15/materials/us-15-Drake- Stagefright-Scary-Code- In-The-Heart-Of-Android.pdf, video: https://www.youtube.com/watch?v=71YP65UANP0.

Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004. http://www.hovav.net/dist/asrandom.pdf.

Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

Tim Kornau. Return oriented programming for the ARM architecture. Master's thesis, Ruhr-Universitat Bochum, Germany, 2010. http://zynamics.com/downloads/kornau-tim--diplomarbeit--rop.pdf.

Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on Android. In *Information Security*, pages 346–360. Springer, 2011.

Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 447–462, 2013.

Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572. ACM, 2010. http://cseweb.ucsd.edu/~hovav/dist/noret-ccs.pdf.

Michael Backes, Sven Bugiel, Oliver Schranz, Philipp von Styp-Rekowsky, and Sebastian Weisgerber. Artist: The android runtime instrumentation and security toolkit. *arXiv preprint arXiv:1607.06619*, 2016. https://arxiv.org/pdf/1607.06619.pdf.

ZhiJun Huang, Tao Zheng, and Jia Liu. A dynamic detection method against ROP attack on ARM platform. In *Proceedings of the Second International Workshop on Software Engineering for Embedded Systems*, pages 51–57. IEEE Press, 2012.

Byoungyoung Lee, Long Lu, Tielei Wang, Taesoo Kim, and Wenke Lee. From Zygote to Morula: Fortifying weakened ASLR on Android. In *IEEE Symposium on Security and Privacy*, 2014.

Priyanka Shetti. Enhancing the security of Zygote/Morula in Android Lollipop. Master's thesis, Amrita Vishwa Vidyapeetham, Ettimadai, Tamil Nadu 641112, India, 6 2015. Advisor: Prabhaker Mateti; http://cecs.wright.edu/~pmateti/Students/.

Nicholas Carlini and David Wagner. Rop is still dangerous: Breaking modern defenses. In *USENIX Security*, volume 14, 2014.