On Return Oriented Programming Threats in Android Runtime

Akshaya Venkateswara Raja, Jehyun Lee, Debin Gao
Singapore Management University
{akshayavr, jehyunlee, dbgao}@smu.edu.sg

Abstract—**Android has taken a large share of operating systems for smart devices including smartphones, and has been an attractive target to the attackers. The arms race between attackers and defenders typically occurs on two front lines — the latest attacking technology and the latest updates to the operating system (including defense mechanisms deployed). In terms of attacking technology, Return-Oriented Programming (ROP) is one of the most sophisticated attack methods on Android devices. In terms of the operating system updates, Android Runtime (ART) was the latest and biggest change to the Android family. In this paper, we investigate the extent to which Android Runtime (ART) makes Return-Oriented Programming (ROP) attacks easier or more difficulty. In particular, we show that by updating system libraries and adopting Ahead-of-Time compiling instead of Just-in-Time compiling in the ART architecture, a larger number and more diverse gadgets are disclosed to ROP attackers, which serve as direct ingredients to ROP attacks. We show that between three and six times more gadgets are found on the ART adopted versions of Android due to the new ART runtime. Moreover, in constrained situations where an attacker requires specific instructions and target registers, Android running ART provides up to 30% more conditional coverage than pre-ART Android does. We additionally demonstrate a sample ROP attack on post-ART Android that would not have been possible on pre-ART Android.**

Keywords: Android Runtime, Return-Oriented Programming, Software attacks

I. INTRODUCTION

The Android operating system, which has taken substantial market share on the mobile platform, recently introduced Android Run Time (ART) [1] to replace its previous runtime environment Dalvik Virtual Machine (DVM). Introduction of the ART brought significant overall performance enhancement [2] by replacing the Just-in-Time (JIT) compilation used in DVM with Ahead-of-Time (AOT) compilation. AOT performs almost all compilation of the Android applications at installation time before they start executing. AOT compiled applications take more space and installation time but less runtime overhead than JIT compiled ones.

Meanwhile, improvements on runtime performance by ART may also give chances for malicious activities. In particular, AOT compilation of Android applications makes the fully/partially compiled versions of application code available before program execution, which might also be taken advantage by a sophisticated attacker. One of the threats to which we give special attention on the ART environment is Return-Oriented Programming (ROP). ROP attacks introduced by Shacham et. al. [3] utilize existing code sequences in the libraries and executables that end with a return instruction. These instruction sequences are called gadgets, and chaining the gadgets by an injected malicious code enables arbitrary executions. The attack has been shown available on ARM instructions used by many Android devices as well [4], [5]. Intuitively, executable (machine) code is a necessary ingredient to ROP attacks, and ART runtime is making such code more readily available because such machine code, if generated at program installation time compared to program execution time, significantly improves user experience.

In this paper, we perform the first detailed analysis on gadget availability in the two runtime environments (ART and DVM). Our detailed analysis is a comprehensive study in the following three aspects. First, we cover all potential sources of gadgets an attacker could use, including the system libraries and the application code. We extend our analysis to underlying reasons of the changes to system libraries to shed light on contributing factors to the different number of instruction sequences found. Second, we classify instruction sequences into different types according to their functionality and present detailed statistical results according to these different types. Instruction sequences are classified from the perspective of attack construction so that our results show direct implication of attack capability and difficulty. Third, our analysis takes into consideration the programming restrictions an attack construction would face, in particular, the different set of registers available to the attacker. Note that although we try to consider all practical limitations and requirements an ROP attack has, our analysis is independent of any specific attack objective.

Through empirical experiments using the stock Android OS images released by Google and Android applications from the official app market [6], we show that the number of available instruction sequences for ROP attacks on ART is approximately three to six times of that on DVM in their vanilla state. Through our feasibility analysis from an attacker perspective, we show that ART has up to 30% more coverage on constrained attacking scenarios than DVM with diverse instruction sequences. Higher coverage on constrained attacking scenarios indicates that it is easier to find suitable gadgets under constraints an attacker possibly faces. As a concrete example of our analysis results, we also present a sample ROP attack on ART which are not feasible in DVM due to the absence of necessary instruction sequences.

The rest of this paper is organized as follows. In Section II, we briefly cover the background of ROP attacks and the difference between ART and DVM runtime environments. As the main finding of our analysis, we present the increased number of useful gadgets and its diversity in ART in Sec-

tion III. In Section IV, we show how the diverse gadgets contribute to an actual attack case which differentiates the ROP attack feasibility. After a brief discussion on our limitation in Section V, we conclude in Section VI.

II. Background and Related Work

Since our analysis is about ROP on Android ART runtime, we first briefly discuss the background of ROP and ART. We also present the related work in this topic and how our analysis differs from them.

A. ROP Attack in Android

ROP is an attacking technique utilizing existing code on the victim side by hijacking the control flow directed by its return address. After ROP was introduced by Shacham et. al. [3], it has been shown that most of the computer systems and instruction sets, including IA32, ARM, and MIPS, are potential victims. ROP attacks generally exploit the epilogue mechanism of function calls that use return addresses stored on the stack for returning to the callee. By modifying the return addresses on the stack, an attacker can control the address where the current call returns.

In addition to exploiting the epilogue mechanism, free branches and jumps can also be utilized for ROP attacks. Jump-Oriented Programming (JOP) [7] hijacks the original control by modifying the destination address of a jump or branch instruction to the address of a desired gadget. Data Oriented Programming (DOP) [8] utilizes a normal control flow but modifies the data handled by the control flow to make the victim routine perform requested computation.

Snow et al. introduced another ROP attack called Just-In-Time ROP, which avoids even the fine-grained address space layout randomization (ASLR) [9]. The basic mechanism of JIT ROP attack is finding and chaining gadgets in runtime on exploitation code. This attack defeats ASLR by collecting required addresses and information on runtime, although it requires heavier exploitation code than traditional code reuse techniques.

Behind the many technical variants, a general work flow for such an attack is as follows [10]:

- Disassembling the vulnerable application and system libraries.
- Finding useful instruction sequences from the vulnerable application and disassembled system libraries.
- Chaining together the instruction sequences to formulate the intended attack.
- Finding a vulnerability in the program.
- Exploiting the vulnerability with the gadget sequences.

All the five steps are necessary for an ROP attack to be successful. In this paper, we specifically focus on the second step, i.e., finding instruction sequences for gadget construction. Instruction sequences are the very basic resources commonly utilized by various code reuse attacking techniques and gadget construction tools, e.g., [11]. Note that although the locations of libraries and functions are typically randomized through ASLR techniques, advanced ROP attacks have been shown to bypass ASLR and successfully use the instruction sequences for ROP attacks [9], [12], [13].

The most important ingredient of an ROP attack is *gadgets* available. Gadgets are instruction or chained instruction sequences that satisfy the following two conditions [14]:

- The instruction has to divert the control flow.
- The target of the control flow must be a register.

Putting this into the context of Android running on ARM devices, such gadgets typically include instructions allowing one to specify the target of a control transfer, e.g., `POP` with `R14` or `R15` as its operands where `R14` is the link register (`LR`) for the return address and `R15` is the program counter (`PC`). In this example, the `POP` instruction writes the popped value from the stack to the registers in its operands, which subsequently are used as the targets of control transfers. Branching instructions `BL`, `BX`, and `BLX` can also be used, as they write the address stored in their operand register to `R14` and then make a control transfer to that address. For example, `BLX R6` copies the value in `R6` to `R14`.

Gadgets (or instruction sequences) are typically found in the target application to be exploited or the system libraries to perform some basic operation, which might be moving values between registers and memory, adding values together, performing control transfers, etc. Attackers first locate all the available gadgets for various basic operations, and then carefully chain them together to perform the specific attack.

B. Android Runtime (ART)

ART is the runtime architecture of Android OS which was experimentally adopted in 4.4 (Kitkat) and officially replaced the Dalvik runtime (DVM) from Android 5.0 (Lollipop) onward. The introduction of ART brought about two main changes: how the application is executed and garbage collection [1]. By replacing Just-In-Time (JIT) compilation with Ahead-Of-Time (AOT) compilation, ART brings performance enhancement to application execution.

On DVM, Java classes in an application are represented in the form of Dalvik Executable (`.dex`) format. Each time when the application is executed, the JIT compiler translates the smali bytecode in the `.dex` file into machine instructions. On the other hand, ART uses AOT compilation to compile the `.dex` code into native code in `.oat` files when an application is installed. When the application is executed, the machine instructions from the `.oat` file are directly executed and no more compilation is needed.

C. Related Work

Many efforts have been made to make ROP attacks more difficult. For example, ROP prevention mechanisms, such as changing memory addresses of basic blocks [15] and instruction sequences [16], [17], and detecting abnormal change of control flow [18], [19], were proposed. These mechanisms have shown their effectiveness, but memory address leakage [12], [13], [20] and JIT-ROP [9] attacks still make a system vulnerable.

Android systems have been shown to be more vulnerable to ROP attacks due to the limited computing and battery resources that limit fine-grained and strong ASLR techniques to be adopted. The application runtime architecture that forks every user application from the zygote process has been addressed as a weakness of Android system because every application has uniform memory layout to the zygote [21], [22]. Sun et. al. also introduced return-to-art attack which is similar to return-to-libc attack but is newly available in ART architecture [22]

As countermeasures to ROP attacks on Android, Retouching [23] and LR^2 [24] proposed code and link randomization methods which make the memory layout different from what an attacker may know. However, these static methods have limitations solving the uniform layout problem. Morula [21] and Blender [22] support randomized layout for every application forked by modifying the Android framework and patching the class linker and `boot.oat`.

Other approaches focus on the ingredients of an ROP attack, i.e., the target vulnerable binaries and their gadgets, which are closer to what we do in this paper. The number of and the diversity of gadgets have been shown to be critical factors to the feasibility of ROP attacks. For example, in Microgadgets [25], it is shown that the susceptibility to ROP attacks highly depends on the size of the binaries and the number and diversity of gadgets available. Joshi et. al. also claimed that obfuscated software has more diverse gadgets, which may increase the susceptibility to ROP attacks [26]. Onarlioglu et. al. proposed a practical defense mechanism named G-Free [27] which modifies the instruction sequences inside a library so that ROP attackers cannot find available gadgets while maintaining their original functionality.

Acknowledging the importance of gadget availability in assessing feasibility of an ROP attack as in these previous work, our paper performs a detailed comparison between gadget availability in Android pre-ART (version 4.4) and post-ART (version 5.1, 6.0, and 7.1) to find out where we stand with the latest Android runtime in terms of restricting ROP or making ROP attacks easier.

III. Analysis on ROP Feasibility

As discussed in Section II-A, typical ROP attacks require finding useful instruction sequences and chaining them together to formulate the intended attack. Such instruction sequences are to be located in Android system libraries, Android system `.oat` files, application libraries, and application `.oat` files. Intuitively, the Android system `.oat` files and application `.oat` files only exist on Android 5.0 onward where ART replaces DVM as the default runtime environment and uses Ahead-Of-Time (AOT) compilation, which add to the pool of instruction sequences that are potential ingredients of ROP attacks. This suggests that ROP may become easier because of higher availability of useful instruction sequences and ROP gadgets.

In this section, we analyze the extent to which this intuition is correct. More specifically, we perform a detailed comparison between Android 4.4 (with the last DVM) and major versions of Android post-ART, i.e., 5.1, 6.0, and 7.1. Our comparison will focus on analysis from the attackers' perspective; in particular, we analyze 1) all potential sources of attack gadgets and reasons behind the different gadget availability from these sources across different Android versions; 2) instruction sequences with different functionality and their corresponding availability; and 3) special instruction sequences that are useful to attackers under various programming constraints.

A. Target of analysis

We target four different versions of Android, 4.4, 5.1, 6.0 and 7.1, which are the last release of major updates to the Android OS. We obtain the corresponding Android OS images from the Google repository for Android developers [28] and install them on Android virtual devices supported by Android Studio.

For each of the Android OS targets, we extract and disassemble all binary files that contain machine instructions that are ready for execution, which include the system libraries and `boot.oat`. Since ROP typically uses machine instructions, these form the pool of instruction sequences for an ROP attack to locate its ingredients from the operating system.

Besides machine instructions from the OS, ROP attacks can also make use of instruction sequences from the application code and library. We target the top 50 Android applications published on Google Play [6] as in July 2016, and extract the `.odex` file of these applications after their installation to our virtual device. These `.odex` files are the output of the AOT compiler, `dex2oat`, which contains the native code of the applications.

After obtaining the machine instructions (from three sources: system libraries, `boot.oat`, and application `.odex` files), we search for special sequences ending with a control transfer instruction (e.g., `BL`, `BX`, and `BLX`) using a Python script that implements the gadget searching algorithm [29]. Note that although application libraries also contribute to ROP attacks, they are independent of Android versions and therefore are excluded from our analysis. We present our searching and analysis results in the following subsections.

B. Instruction sequences from Android OS

The most important source of useful instruction sequences from the ROP attackers' perspective is system libraries, which exist in two forms — `.so` files and `boot.oat`. The legacy Linux shared libraries (`.so` files) exist in both Android DVM and Android ART systems. These system libraries are mainly built on C/C++ and are shared with all applications on demand. `boot.oat` only exists in Android post-ART versions. With the introduction of Ahead-Of-Time (AOT) compilation in ART, Java core libraries are converted into native code and stored in `boot.oat`. Since `boot.oat` also consists of native machine code, attackers could make use of instruction sequences from it to construct ROP attacks.

Fig. 1 shows detailed breakdown of the number of useful instruction sequences found in the four Android versions, in which we categorize instruction sequences according to their functionality (i.e., arithmetic, logical, system call, data transfer), and report results for `.so` system libraries and the combination of `.so` system libraries and `boot.oat`. Note that Android 4.4 has only `.so` libraries since system's `boot.oat` only exists post-ART.

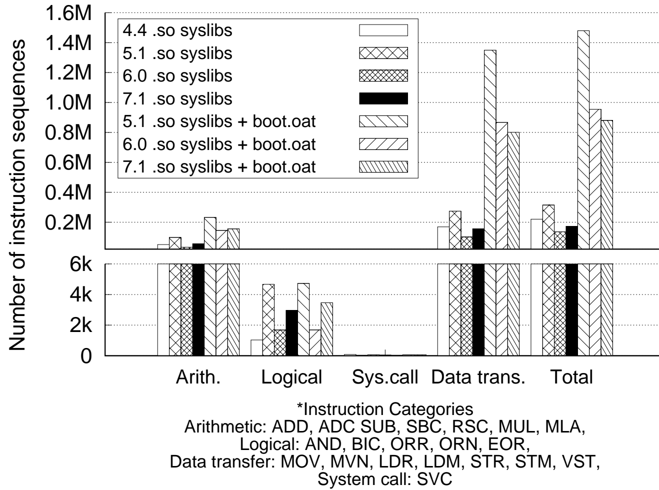

Fig. 1. Number of useful instruction sequences for ROP attacks among Android OS versions

It has been a general increase of the number of `.so` system libraries from Android 4.4 to Android 5.1. The number of libraries increases from 172 to 209, with 51 libraries newly added. Fig. 2a clearly shows that most libraries (78%) have grown bigger in terms of the number of useful instruction sequences for ROP attacks. An interesting observation is that such increase is not uniformly distributed among all libraries. In particular, two libraries, `libwebviewchromium` and `libgoogle_recognizer_jni_l` contributed 65% of all the additional instruction sequences useful for ROP attacks.

From Android 5.1 to Android 6.0, the number of system libraries continues to increase from 209 to 215. Fig. 2b demonstrates a similar statistical pattern with that in Fig. 2a, suggesting that most libraries have more instruction sequences for ROP attacks. However, the larger decrease of useful instruction sequences in a few libraries coupled with the depreciation of two big libraries (`libwebviewchromium` and `libgoogle_recognizer_jni_l`) from Android 5.1 change the overall trend and result in the total number of useful instruction sequences being smaller in Android 6.0 than Android 5.1 (see Fig. 1). Upon further investigation, we realize that `libwebviewchromium` (part of a large webview project *Chromium*) was moved to the user space as an application. The official reason of the change is to enhance flexibility and security of the library by independent updates [30].

Android 7.1 also sees an increase in the number of libraries from 215 in Android 6.0 to 244. However, their increases and decreases of useful instruction sequences for ROP attacks more or less cancel out, which results in only a small increase on the overall statistics.

We further look into the reasons behind such changes on the number of useful instruction sequences for ROP attacks in the system libraries. The change in file size of the libraries and the change of number of useful instruction sequences for ROP attacks demonstrate some correlation, which seems to suggest that bigger libraries contribute more to the instruction sequences for ROP attack. To verify this intuition, we perform a more thorough analysis on such correlation; see Fig. 3, which

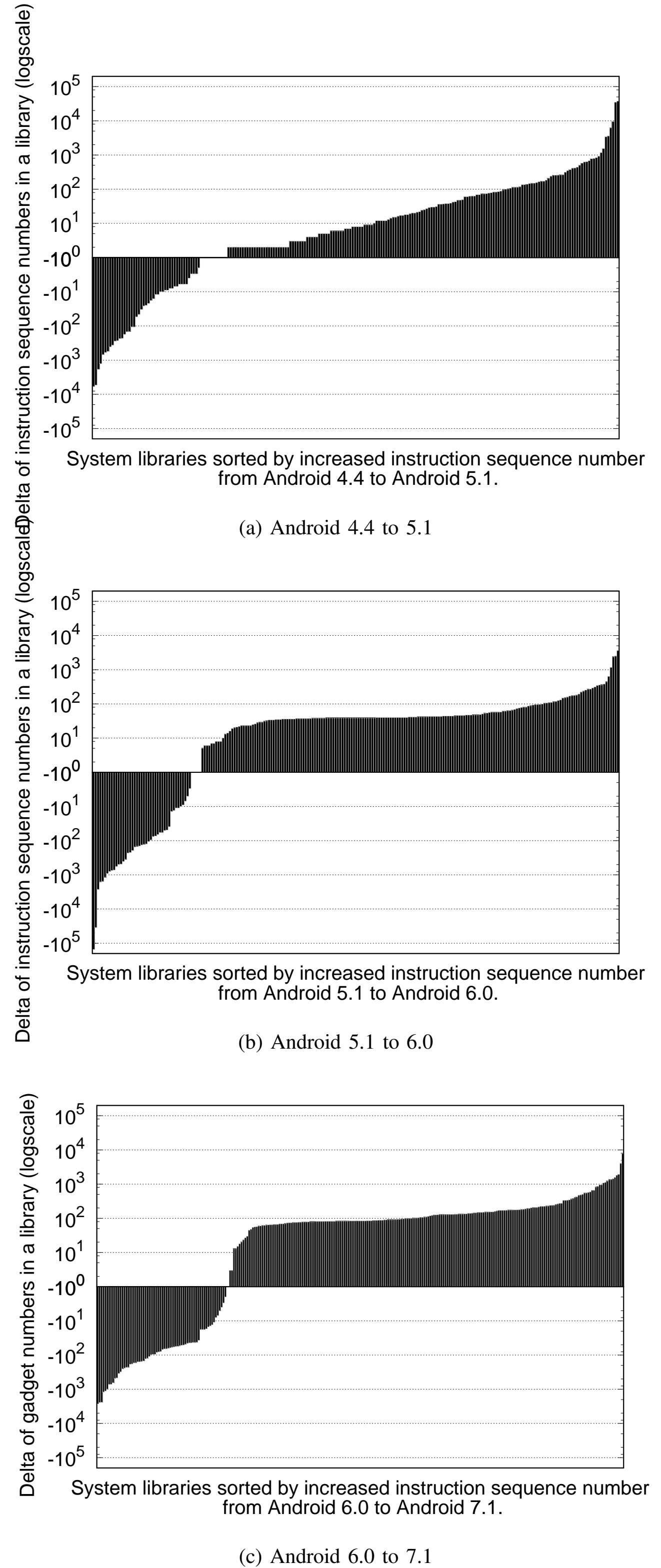

Fig. 2. Changes on the number of instruction sequences in each system library between each version update.

shows two statistics — the number of instructions and the number of useful instruction sequences for ROP attacks — for every `.so` library in various Android versions[1].

[1]A couple of outliers are excluded in the plot as they do not significantly change the pattern.

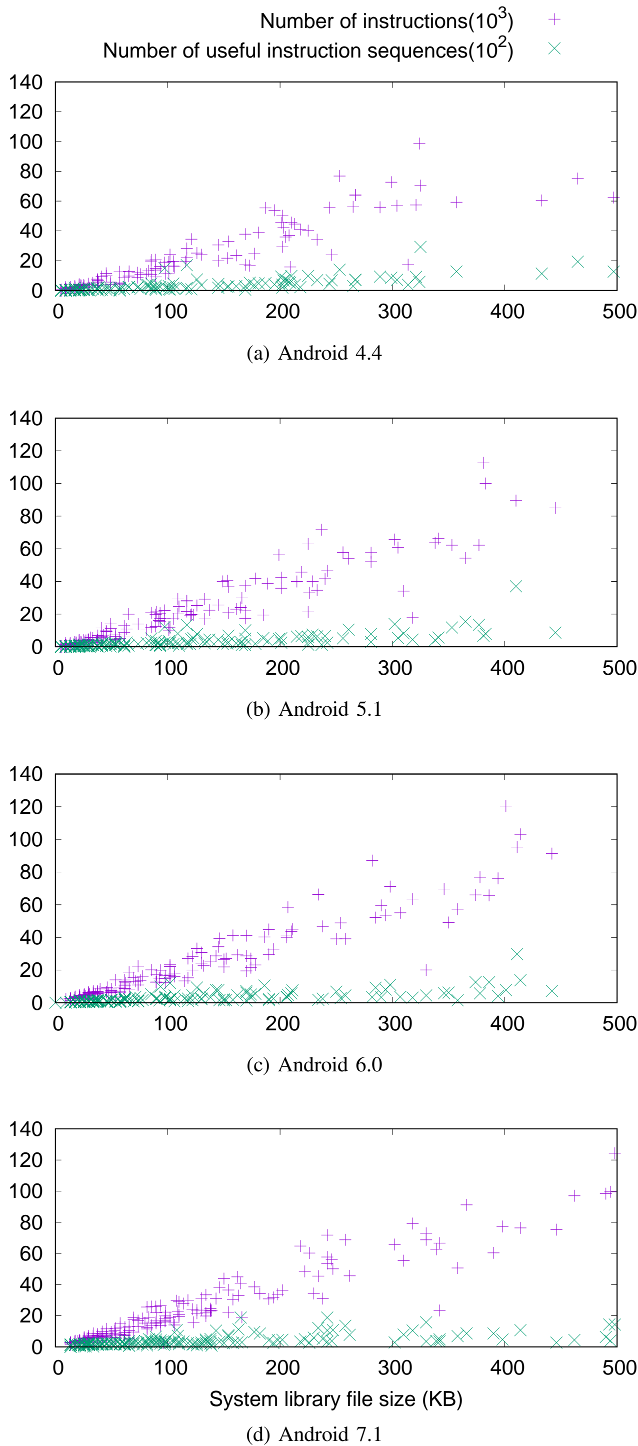

Fig. 3. Number of instructions and useful gadgets by file size from system libraries in Android 4.4-7.1

Fig. 3 shows that although the number of instructions demonstrate clear correlation with the library file size (which is true as long as the library consists of mostly executable code), the number of useful instruction sequences for ROP attacks is not. Instead, the number of useful instruction sequences per one thousand instruction ranges from less than 1 to more than 80 among all the libraries. For instances, `libglesv1_enc` has 1200-1700 useful instruction sequences in 4.4-6.1, while `libstagefright_soft_mpeg4dec` which has a similar number of instructions has only 100-130 useful instruction sequences for ROP attacks in the same versions. This is a contrasting feature of `.so` system libraries compared to that of application `.odex` files discussed in the next subsection. Application `.odex` files in Fig. 4 show a much clearer correlation with between 70 and 90 of instruction sequences per one thousand instructions. One of the key factors which affect the number of instruction sequences is the number of returns and branches. Even though the two libraries have the same number of instructions, the number of useful instruction sequences can be different along with the number of condition branches, methods, and method calls which make branch instructions in their compiled code.

In contrast to the `.so` system libraries which differ quite substantially across multiple Android versions, `boot.oat` has almost the same structure and contain 12 (in Android 6.0) or 13 (in Android 5.1 and 7.1) Java core libraries. Fig. 1 shows that `boot.oat` contributes significantly to the ROP attacks by introducing a large number of useful instruction sequences for ROP attacks, especially on sequences for data transfer. We stress again that unlike `.so` system libraries which exist in pre- and post-ART Android, `boot.oat` exists only in post-ART Android versions. In other words, the introduction of ART newly added all these instruction sequences for attackers to use in constructing an ROP attack.

One interesting observation, though, is that this damage introduced by ART (in terms of new instruction sequences for ROP attacks) is on the declining trend from Android 5.1 to 6.0 and to Android 7.1. In particular, the number of such instruction sequences decreased in a few large sized libraries and packages in `boot.oat` by removal and significant down-scale from Android 5.1 to Android 6.0. We leave it our future work to investigate the underlying reason for this change in each library, be it (ROP-specific) security awareness among the Android development team, or just coincidence of other considerations.

C. Instruction sequences from application .odex files

One of the most significant differences between DVM and ART is the existence of AOT compiled files, which exist in the form of both systems `boot.oat` as well as application `.odex` files. With the introduction of AOT compilation in ART, `.dex` files of all the applications are compiled into `.odex` files after installation, which contain native instructions. This subsection focuses on the analysis of such `.odex` files for the Android applications to find out the extent to which they contribute instruction sequences for ROP attacks. We stress that these `.odex` files are optimized DEX in DVM, and are in a new file format in Android ART. They experience negligible differences among various versions post-ART (Android 5.1, 6.0, and 7.1) though, and therefore, we present only the result of our analysis on Android 6.0 in this subsection. We install the top 50 applications on Google Play (as of Jun 2016) on our virtual device running Android 6.0, extract the resulting `.odex` file for each application, and then run our Python scripts to locate all useful instruction sequences for ROP attacks.

When we investigate the correlation between the number of useful instruction sequences for ROP attacks and the total

number of instructions as well as the file size, we find that unlike the `.so` system libraries, these `.odex` files are much "better behaved". Fig. 4 shows that for the top 50 Android applications, both numbers (that of useful instruction sequences for ROP attacks and that of all instructions) are proportional to the file size. We attribute this difference to the different languages and compilers used — application `.odex` files are compiled from Java while `.so` system libraries are compiled from C/C++.

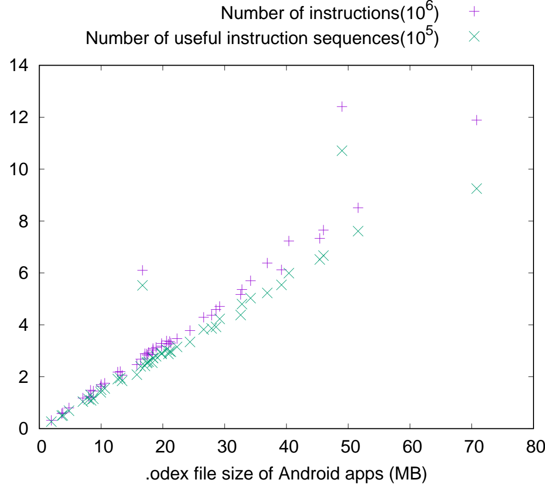

Fig. 4. Number of instructions and useful instruction sequences for ROP attacks from application .odex files for top 50 apps

It is quite intuitive that the larger `.odex` files contain more useful instruction sequences for ROP attacks. Besides that, another factor determining the number of such instruction sequences could be the average size of methods inside the applications, since every such instruction sequence has to end with a control-transfer instruction. Fig. 5 demonstrates this relationship. In general, applications written with smaller methods tend to have more useful instruction sequences for ROP attacks — this confirms our intuition since smaller methods tend to have (on average) more control transfers. This result suggests that compilers could use more function inlining opportunities to generate more secure software, from the perspective of reducing instruction sequences for ROP attacks.

D. ROP construction in constrained scenarios

So far, our analysis has been focusing on available instruction sequences for ROP attack construction, and we had considered instruction sequences for different functionality (see Fig. 1). In this subsection, we go into more fine-grained categorization of these instruction sequences and consider more realistic attacking scenarios where attackers are facing various constraints.

While performing an ROP attack, an attacker may have various constraints in addition to the specific functionality of instruction sequences. For instance, the attacker could only write to a small subset of the registers (probably due to the fact that other registers are containing critical information that ensures continuous normal execution of the application). In such a scenario, not all available instruction sequences for the needed functionality might help — the attacker needs one that operates on some specific registers.

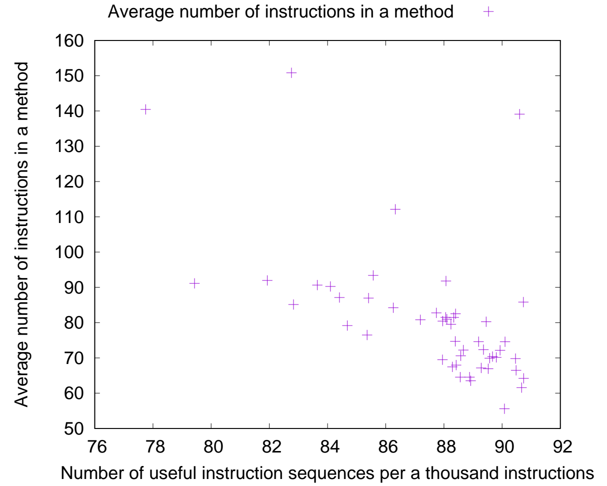

Fig. 5. Relation between normalized number of useful instruction sequences for ROP attacks and the average size of methods in application .oat files of top 50 Android apps

To analyze such scenarios, we perform a finer-grained categorization on the instruction sequences found from the Android OS (including those from `.so` system libraries and `boot.oat`) at two dimensions — one along their specific functionality (e.g., the `add` function or the `and` function) and the other along the specific registers used (i.e., `R0`, `R1`, etc.). We perform this analysis on the four different Android versions 4.4, 5.1, 6.0, and 7.1, and present the results in Table I. Dark-gray-shaded cells indicate the scenario in which the instruction sequence only exists in ART versions, and light-gray-shaded cells indicate the opposite. There are a lot more dark-gray cells than light-gray cells in Table I.

Results show that, with certain constraints, only specific versions of the Android OS are vulnerable to a certain type of ROP attack, and ART versions generally are much more vulnerable (provide more instruction sequence opportunities) than the DVM version.

For more straight-forward comparison, we introduce a measurement metric *constraint coverage* which represents the extent to which an Android version is vulnerable. It is measured by the proportion of unique constraints out of the 352 conditions (16 registers by 22 functionality) under which the corresponding Android OS provides available instruction sequences. 100% coverage means that an attacker can find at least one instruction sequence under all the 352 different types of constraints from system libraries. Results of such a measurement are presented in Fig. 6.

We can see the huge increase on this constraint coverage from Android 4.4 to Android 5.1. Although it drops significantly in Android 6.0 and Android 7.1, the values are still greater than that in Android 4.4. This result is consistent with our earlier findings in a more coarse-grained categorization (see Fig. 1).

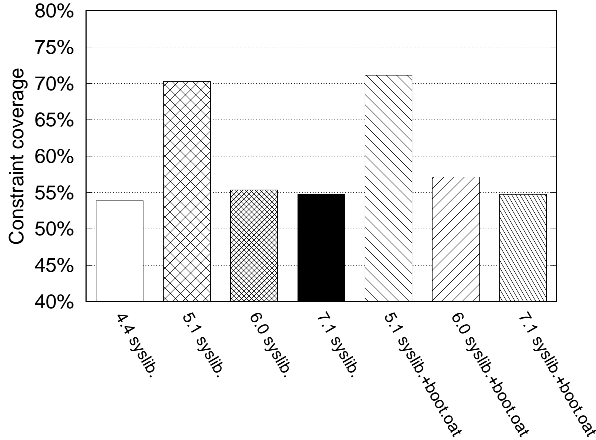

Fig. 6. Constraint coverage for various Android OS versions

IV. Empirical Evaluation

Our analysis so far in Section III shows that post-ART Android versions suffer from more ROP opportunities due to the more readily available instruction sequences an ROP attacker can use to construct an attack, even at various constrained scenarios. In this section, we verify this analysis result by performing an actual ROP attack which is feasible on Android 6.0 but not on Android 4.4 due to the lack of available instruction sequences.

A. Vulnerability and exploit strategy

In this sample ROP attack, we utilize a well-known vulnerability introduced earlier [10]; see Source Code 1. We insert this buffer-overflow vulnerability into the native library of a simple Android app *Standup Timer*, an open-source android application that acts as a stop watch. Due to the vulnerability introduced, we are able to overflow a local buffer and overwrite the adjacent control flow information to execute the attack.

Source Code 1 Buffer overflow vulnerable code for test ROP attack scenario

Params: JNIEnv* env, jobject obj
Return: jstring
1: struct foo {
    char buffer [200];
    jmp_buf jb;
    };
2: **procedure** Java_com_example_vulnapp_NewActivity_stringFromJNICPP
3:     // A binary file is opened (not depicted)
4:     . . .
5:     foo *f = new foo;
6:     $i = setjmp(f->jb)$;
7:     **if** $(i != 0)$ **then**
8:         return 0;
9:     **end if**
10:     $fgets(f->buffer, sizeOfFile, sFile)$;
11:     $longjmp(f->jb, 2)$;
12: **end procedure**

In Source Code 1, function `setjmp` on the sixth line creates a special data structure `jmp_buf` to store the register values from `R4` to `R15`, including the stack pointer, link register, and program counter. Therefore, when we overwrite `jmp_buf` before calling `longjmp`, the program counter (PC) value in `R15` is changed and control is then transferred to the desired address without corrupting the return address. Through this buffer-overflow vulnerability, we attempt to execute a system call with a manipulated parameter through a `syscall` routine implemented in `libc` — a pretty standard and general exploit that many attacks attempt to perform. For performing a malicious action, we multiply the value in `R0` by a given scale. For demonstration purposes, we further narrow down the constraint that the attacker faces to a smaller range, `R12` to `R15`.

B. Instruction sequences needed for gadget construction

The following is an example of the attack gadget which consists of instruction sequences satisfying the requirement.

Sequence A2: **MUL R0, R0, R12; BX R14**
Sequence A1: **system()**

Because `R15` (PC) is used for the control flow hijacking, the only available registers for the addresses and the argument are `R12`, `R13`, and `R14`. To make a control transfer to Sequence A1 which consists of the target function to be called, we need to find an instruction sequence which performs a MUL type of actions and stores its result to `R0`. Sequence A2 satisfies this requirement since `R12` is among the registers that the vulnerable code could overwrite (from `R12` to `R15`) and the source register for branch instruction `R14` is also within the constrained range.

This example demonstrates a realistic attacking scenario in which instruction sequences satisfying certain constraints (in this example, the constraints are about a range of registers as the source, since those are the only ones that the vulnerable code is able to overwrite) are needed. We manage to find these instruction sequences from `libstagefright_soft_aacenc.so` system library in Android 6.0, but could not find them (including variations of them satisfying our requirements) from any system libraries in Android 4.4. The reasons are clearly demonstrated in Table I where we see many more instruction sequences targeting specific registers found only in Android post-ART.

For the same functionality, we may consider another way by using three instruction sequences as follows.

Sequence B3: **MOV R3, R0; BLX R13**
Sequence B2: **MUL R0, R12, R3; BX R14**
Sequence B1: **system()**

Sequence B2 is the only instruction sequence available in Android 4.4 which potentially satisfies our attacking requirement. To utilize Sequence B2 for the same purpose, the value of `R3` must be the same as (or a multiple of) `R0`. Therefore, we need an additional sequence like Sequence B3 (hypothetical); however, such an instruction sequence is, again, unavailable in Android 4.4 libraries.

TABLE I. THE NUMBER OF INSTRUCTION SEQUENCES FROM SYSTEM LIBRARIES BY WRITE-TARGET REGISTERS AND ACTIONS

		ADD	ADC	SUB	SBC	RSB	RSC	MUL	MLA	AND	BIC	ORR	ORN	EOR	MOV	MVN	LDR	LDM	STR	STM	VST	SVC
R0	4.4	10046	1310	625	1	164	0	79	152	297	105	347	2	185	93053	818	9841	0	7703	11	1	0
	5.1	11955	1440	903	0	291	0	108	194	430	198	1591	5	312	311799	1353	265843	2	8367	27	0	0
	6.0	5627	393	435	0	138	0	65	86	255	79	813	1	103	219002	1013	279610	1	4726	42	0	0
	7.1	27116	83	1136	7	94	0	91	103	991	143	1250	5	349	163014	900	411027	0	19126	62	0	0
R1	4.4	2332	17	59	6	46	0	8	31	12	20	52	0	10	11748	26	19784	0	5725	1	0	0
	5.1	8469	56	229	41	56	9	31	51	120	61	286	0	30	273867	89	32669	4	10972	12	0	0
	6.0	1782	135	190	23	18	26	5	15	9	3	125	0	4	84557	205	18330	4	4370	2	0	0
	7.1	6216	257	332	171	11	33	13	22	104	65	488	0	36	32860	261	33699	0	41546	10	0	0
R2	4.4	1199	3	43	0	52	0	11	6	10	13	25	0	5	4200	8	22418	1	5430	0	0	0
	5.1	3173	6	236	1	84	0	34	25	31	48	211	0	10	136916	82	33180	1	10943	4	0	0
	6.0	415	1	49	0	12	0	11	5	15	5	19	0	3	117577	42	13361	0	3279	7	0	0
	7.1	1356	1	135	0	1	0	21	5	33	11	30	0	4	5809	8	17886	1	35113	3	0	0
R3	4.4	1084	25	38	1	55	0	7	3	20	5	44	0	6	3196	9	27669	1	8872	4	0	0
	5.1	3675	35	130	8	88	0	18	7	57	37	125	0	13	64328	187	49513	1	10842	23	0	0
	6.0	333	21	19	6	13	0	161	5	8	2	53	0	3	43260	92	13477	0	3914	8	0	0
	7.1	850	6	27	0	0	0	105	3	28	12	39	0	2	2972	16	8104	5	23267	3	0	0
R4	4.4	103	0	40	0	2	0	0	1	2	1	1	0	0	1695	28	4930	1	2632	6	0	0
	5.1	962	1	136	1	24	2	6	10	9	6	43	0	6	2457	14	9463	11	4604	106	0	0
	6.0	82	1	25	1	0	2	4	0	2	4	8	0	0	1095	7	3813	2	1966	41	0	0
	7.1	127	0	19	2	1	0	8	1	3	2	6	0	3	882	16	999	2	5010	8	0	0
R5	4.4	183	4	14	0	1	0	0	0	4	1	1	0	2	1397	23	3071	1	4080	3	0	0
	5.1	1764	7	119	0	30	0	7	6	31	49	36	0	9	2002	13	8778	2	6152	26	0	0
	6.0	51	0	12	0	4	0	1	0	4	2	1	0	2	813	16	3397	0	1752	10	0	0
	7.1	78	1	19	1	0	0	6	3	1	3	4	0	0	1304	19	993	0	1555	8	0	0
R6	4.4	136	0	11	0	0	0	1	0	3	1	2	0	2	652	3	2802	0	1366	1	0	0
	5.1	1216	3	81	1	11	0	11	5	38	8	20	0	6	4351	10	9530	2	3671	47	0	0
	6.0	56	0	9	0	4	0	0	0	4	3	0	0	0	1184	10	2953	0	1082	5	0	0
	7.1	52	2	19	0	0	0	8	3	1	2	15	0	2	2618	8	1551	0	766	0	0	0
R7	4.4	135	2	10	0	0	0	2	0	1	1	0	0	1	605	2	2529	0	782	0	0	0
	5.1	801	0	68	0	16	0	15	9	74	4	987	0	5	2368	12	8271	0	2048	9	0	0
	6.0	28	3	19	0	0	0	1	1	0	1	1	0	0	1021	4	3809	1	679	2	0	0
	7.1	48	2	19	0	0	0	3	0	2	1	9	0	0	1237	2	3238	0	457	1	0	0
R8	4.4	45	0	12	0	3	0	0	0	0	0	0	0	0	320	0	183	1	263	2	0	0
	5.1	292	2	36	0	10	0	4	5	6	0	3	0	3	1404	10	656	1	514	18	0	0
	6.0	40	1	1	0	0	0	0	1	0	0	216	0	0	679	2	322	1	288	0	0	0
	7.1	14	0	1	0	0	0	1	0	0	1	1	0	0	771	1	204	0	365	4	0	0
R9	4.4	47	0	7	0	1	0	0	0	0	0	0	0	0	244	0	147	0	124	0	0	0
	5.1	711	3	11	0	5	0	0	2	12	2	2	0	0	442	4	479	2	567	2	0	0
	6.0	27	1	4	0	0	0	0	0	1	0	120	0	1	150	1	483	0	230	2	0	0
	7.1	9	0	0	0	0	0	0	1	0	0	0	0	0	54	0	90	1	300	8	0	0
R10	4.4	23	0	2	0	2	0	1	0	0	2	0	0	0	198	0	70	2	68	0	0	0
	5.1	203	0	33	0	11	0	0	7	5	2	3	0	2	1517	1	290	0	363	2	0	0
	6.0	23	0	5	0	0	0	0	1	1	2	1	0	1	717	0	101	1	199	0	0	0
	7.1	13	0	2	0	0	0	0	1	0	2	2	0	2	454	0	94	0	213	6	0	0
R11	4.4	29	1	2	0	1	0	0	0	0	0	0	0	1	113	0	62	0	57	0	0	0
	5.1	183	4	14	0	1	0	0	3	8	1	7	0	2	719	3	237	0	317	1	0	0
	6.0	786	0	1	0	1	0	0	0	1	0	0	0	0	291	0	77	0	224	0	0	0
	7.1	5	0	0	0	0	0	1	0	0	4	0	0	0	396	0	46	0	232	0	0	0
R12	4.4	18	0	5	0	125	0	0	1	1	5	35	0	1	11	4	50	0	554	2	0	0
	5.1	222	3	21	0	112	0	5	14	8	17	39	0	14	160	1	32626	2	1768	37	0	0
	6.0	19	0	7	0	28	0	0	0	2	0	11	0	0	65	2	16707	4	831	20	0	0
	7.1	69	0	4	0	0	0	0	0	6	0	7	0	0	93	2	17833	1	822	8	0	0
R13	4.4	33224	0	182	0	0	0	0	0	2	0	0	0	0	6	0	1	0	1	0	0	0
	5.1	194447	1	837	0	0	0	0	1	3	0	0	0	2	53	0	3	3	1	10	0	0
	6.0	134067	0	1082	0	0	0	0	0	0	0	0	0	0	41	0	22	0	2	27	0	0
	7.1	119229	0	736	0	0	0	0	0	0	0	0	0	0	8	0	25	0	0	23	0	0
R14	4.4	13	0	6	0	24	1	0	0	2	0	0	0	0	3	2	43	0	150	2	0	0
	5.1	120	1	10	0	11	0	0	0	2	0	1	0	2	22	2	586244	1	389	1	0	0
	6.0	7	0	4	0	1	0	0	0	1	0	3	0	0	22	3	633733	5	349	12	0	0
	7.1	7	0	1	0	4	0	0	0	1	0	0	0	0	37	2	581864	0	150	1	0	0
R15	4.4	1	0	0	0	0	0	0	0	4	0	0	0	0	2	0	1	0	3	0	0	0
	5.1	0	0	5	0	2	2	0	0	3	0	1	0	3	1	1	9	0	0	0	0	0
	6.0	0	4	1	0	1	5	0	0	0	0	0	0	0	1	1	12	1	0	0	0	0
	7.1	19	0	1	0	1	1	0	0	5	0	0	0	1	2	0	0	0	0	0	0	0

Note: Functionality of instruction types on column header
ADD: Add, ADC: Add with carry, SUB: Subtract, SBC: Subtract with carry, RSB: Reverse subtract, RSC: Reverse subtract with carry, MUL: Multiply, MLA: Multiply accumulate
AND: Logical AND, BIC: Bit clear by logical AND NOT, ORR: Logical OR, ORN: Logical OR NOT, EOR: Logical Exclusive OR
MOV: Move, MVN: Move NOT, LDR: Load, LDM: Load multiple registers, STR: Store, STM: Store multiple registers, VST: Vector store
SVC: Supervisor calls

V. LIMITATIONS AND FUTURE WORK

Useful instruction sequences which are the source of ROP gadgets are an important factor for ROP feasibility in a system. However, the practicality of each instruction sequence is difficult to be fully presented only with the numbers and diversity of them demonstrated in our analysis. Some additional factors, such as the distance between the instruction sequences, being able to read and write to the heap space, static values inside a sequence, etc., can also be critical to their usefulness when an attacker tries to construct an attack gadget. The measurement of ROP feasibility considering those practical factors from the attacker perspective may give more precise and abundant insight. Figuring out the conditions and characteristics of a victim system that make an attack difficult to attack can be one of our future work.

VI. CONCLUSION

In this paper, we analyze and show the ROP feasibility exacerbated by the change of runtime architecture from DVM to ART. Not only more gadgets caused by larger system libraries but also AOT compiled applications contribute to a larger number and more diverse and useful gadgets than the

DVM architecture. In particular, an attacker can find specific types of gadgets performing a certain action to a required register only from ART adopted versions. We show that some attacks which are not possible in previous versions of Android are now becoming possible in new versions. We believe that the intuitions from our analysis results are helpful to designing future systems and countermeasures.

REFERENCES

[1] A. Frumusanu, "A closer look at android runtime (art) in android l," *AnandTech. Retrieved July*, vol. 5, 2014.

[2] R. Yadav and R. S. Bhadoria, "Performance analysis for android runtime environment," in *Communication Systems and Network Technologies (CSNT), 2015 Fifth International Conference on*. IEEE, 2015, pp. 1076–1079.

[3] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*. ACM, 2004, pp. 298–307.

[4] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2011, pp. 121–141.

[5] M. Prandini and M. Ramilli, "Return-oriented programming," *IEEE Security & Privacy*, vol. 10, no. 6, pp. 84–87, 2012.

[6] Google Inc., "Google Play," https://play.google.com.

[7] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 2011, pp. 30–40.

[8] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 969–986.

[9] 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 *Security and Privacy (S&P), 2013 IEEE Symposium on*. IEEE, 2013, pp. 574–588.

[10] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 559–572.

[11] Jonathan Salwan, "ROPgadget," http://shell-storm.org/project/ROPgadget/.

[12] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses." in *USENIX Security*, vol. 14, 2014.

[13] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "Aslr-guard: Stopping address space leakage for code reuse attacks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 280–291.

[14] T. Kornau, "Return oriented programming for the arm architecture," Ph.D. dissertation, Masters thesis, Ruhr-Universität Bochum, 2010.

[15] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 157–168.

[16] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd my gadgets go?" in *Security and Privacy (S&P), 2012 IEEE Symposium on*. IEEE, 2012, pp. 571–585.

[17] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Security and Privacy (S&P), 2012 IEEE Symposium on*. IEEE, 2012, pp. 601–615.

[18] ——, "Transparent ROP exploit mitigation using indirect branch tracing," in *USENIX Security*, vol. 30, 2013, p. 38.

[19] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Security and Privacy (S&P), 2013 IEEE Symposium on*. IEEE, 2013, pp. 559–573.

[20] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection." in *USENIX Security*, vol. 14, 2014.

[21] B. Lee, L. Lu, T. Wang, T. Kim, and W. Lee, "From zygote to morula: Fortifying weakened aslr on android," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 424–439.

[22] M. Sun, J. C. Lui, and Y. Zhou, "Blender: Self-randomizing address space layout for android apps," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016, pp. 457–480.

[23] H. Bojinov, D. Boneh, R. Cannings, and I. Malchev, "Address space randomization for mobile devices," in *Proceedings of the fourth ACM conference on Wireless network security*. ACM, 2011, pp. 127–138.

[24] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi, "Leakage-resilient layout randomization for mobile devices," in *Network and Distributed Systems Security Symposium (NDSS)*, 2016.

[25] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz, "Microgadgets: size does matter in turing-complete return-oriented programming," in *Proceedings of the 6th USENIX conference on Offensive Technologies*. USENIX Association, 2012, pp. 7–7.

[26] H. P. Joshi, A. Dhanasekaran, and R. Dutta, "Impact of software obfuscation on susceptibility to return-oriented programming attacks," in *Sarnoff Symposium, 2015 36th IEEE*. IEEE, 2015, pp. 161–166.

[27] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-free: defeating return-oriented programming through gadget-less binaries," in *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 2010, pp. 49–58.

[28] Google Inc., "Android system image repository," https://dl.google.com/android/repository/sys-img/google_apis/.

[29] H. 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*. ACM, 2007, pp. 552–561.

[30] Chromium project group, "WebView for Android," https://developer.chrome.com/multidevice/webview/overview/.