Mitigating ROP Attacks via ARM-Specific In-Place Instruction Randomization

Yu Liang[1,2], Guojun Peng[1,2]*, Yuan Luo[1,2], Huanguo Zhang[1,2]

[1] Key Laboratory of Aerospace Information Security and Trust Computing of Ministry of Education, Wuhan 430079, China;
[2] Computer School of Wuhan University, Wuhan 430079, China

Abstract: Defending against return-oriented programing (ROP) attacks is extremely challenging for modern operating systems. As the most popular mobile OS running on ARM, Android is even more vulnerable to ROP attacks due to its weak implementation of ASLR and the absence of effective control-flow integrity enforcement. In this paper, leveraging specific ARM features, an instruction randomization strategy to mitigate ROP attacks in Android even with the threat of single pointer leakage vulnerabilities is proposed. By popping out more registers in functions' epilogue instructions and reallocating registers in function scopes, branch targets in all (direct and indirect) branch instructions potential to be ROP gadgets are changed randomly. Without the knowledge of binaries' runtime instructions layout, adversary's repeated control flow transfer in ROP exploits will be subverted. Furthermore, this instruction randomization idea has been implemented in both Android Dalvik runtime and ART. Corresponding evaluations proved it is capable to introduce enough randomness for more than 99% discovered functions and thwart about 95% ROP gadgets in application's shared libraries and oat file compiled from Dalvik bytecode. Besides, evaluations on real-world exploits also confirmed its effectiveness on mitigating ROP attacks within acceptable performance overhead.

Keywords: software security; ROP mitigation; instruction randomization; ARM architecture

I. INTRODUCTION

Return-oriented programming is an advanced system security exploitation technique[1,2]. With a control-flow hijacking vulnerability, it can repeatedly transfer control flow among existing executable code snippets (a.k.a gadgets) in memory space and represent some unintended malicious behaviors without code injection. This kind of code reuse allows an attacker achieving some malicious or sophisticated functionalities even in the presence of modern security defenses, such as non-executable (NX) memory and dynamic code signing, in commodity operating systems[2–6].

However, a successful ROP exploitation has two prerequisites: 1) the presence of proper gadgets that can perform control flow transfers among gadgets, and 2) the capability to correctly locate corresponding gadget address in application memory, which would be used as the transfer target[7]. Therefore, approaches including control flow transfer enforcement and memory layout randomization are employed to defend against ROP by breaking these two prerequisites in recent years.

For the first prerequisite, both ROP prevention and detection approaches based on control flow integrity (CFI) are proposed. CFI-based

A novel ARM-specific ROP mitigation strategy and its implementation in Android are presented. The evaluation on real-world applications and ROP attacks also proves the proposed randomization design could introduce enough randomness on branch instructions to thwart the majority of gadgets and mitigate ROP attacks practically within acceptable performance overhead.

prevention approaches thwart ROP execution by examining the control-flow transfer target around branch instructions in runtime, and enforcing the transfer target, the transfer source and even the execution context are legitimate[8–13]. ROPdefender and RCAP-stack allocate shadow stacks to store the intended branch target and examine them before issuing control-flow transfers by instruction instrumentation[10,11]. CCFIR and binCFI enforces the legality of indirect branches via binary rewriting[12,13]. Related detection approaches, such as CFIMon, kBouncer, DCFI-Checker and ROPecker, heuristically discover ROP attacks by examining the control-flow transfer frequency and gadgets length with the assistance of hardware support[14–16].

Successful ROP attacks greatly depend on a good understanding of instructions layout. In terms of the second prerequisite, randomization-based approaches defeat ROP by lowering adversaries' knowledge about runtime instructions layout and making adversaries fail to locate necessary gadgets engaged in ROP attacks[17–20]. Address space layout randomization mechanism (ASLR) introduces randomness to the base of modules, heaps and stacks and widely deployed in commodity OSes[17]. However, with the presence of information leakage vulnerability, ASLR can be easily defeated. ASLR-Guard presented an security-enhanced address randomization solution[7]. Besides, ILR [14], Binary Stirring [15] and Smashing the gadgets [16] provide some fine-grained randomization strategies, such as instruction reordering, instruction substitution and basic block randomization, to mitigate the threat introduced by information leakage vulnerability and enhance the capability of ROP defense.

Though these approaches work well on specific platform, neither of them can be conveniently applied to Android running on ARM, the most popular mobile platform. Specifically, reasons are as following.

- Firstly, ARM is a totally different architecture from x86/x86-64, and it doesn't support performance monitor unit (e.g., LBR or BTS), so those hardware-assisted approaches (kBouncer, ROPecker and etc.) cannot be transplanted.
- Secondly, approaches based on instruction randomization, such as ILR, Smashing the gadgets and CCFIR, rely on particular instructions and function invocation patterns on x86, which are different from ARM. It's difficult to port these ideas to ARM.
- Furthermore, these randomization-based approaches are implemented via binary rewriting. This kind of binary rewriting usually introduces extra instructions and needs shift unrelated instructions, which are some complicated work.

In contrast, a much more practical, efficient and achievable ROP mitigation approach for applications and binaries in Android on ARM is presented in this paper. It can thwart ROP attacks even with the threat of single pointer leakage vulnerabilities. The main idea is to randomize *both* direct branch instructions (*b*, *bl*, *blx* etc.) and indirect branch instructions (*pop {…,pc}*) in applications' binaries via minimal ARM-specific in-place rewriting. Note that, leveraging specific features of ARM, this in-place rewriting will not introduce any extra instructions and be much efficient to achieve.

A prototype system is implemented and employed to evaluate the randomization idea. It consists of 1) an offline binary analyzer to generate randomization info, 2) and a customized Android binary loader with the capability to perform coin-flip randomizations and instruction transformation during binaries mapping. Branch instructions in both shared libraries and oat files compiled from Dalvik bytecode (*.dex) can be randomized.

The evaluation result on top 20 free apps in Google Play shows that, the proposed randomization strategy can be applied to 99.87% ARM functions and 92.33% Thumb functions in shared libraries, and 99.86% functions in oat files (device-compiled files on Android ART). The average amounts of randomness granted to ARM and Thumb functions in shared libraries and Thumb functions in oat files are 4, 7 and 5 bits, respectively. It thwart-

ed the majority of gadgets and greatly reduced the possibility of successful ROP exploitation.

In summary, this paper's contributions are as follows:

- Proposed a novel ARM-specific ROP mitigation strategy that can randomize branch instructions in potential gadgets and immune from the threat of single pointer leakage vulnerabilities.
- Implemented the proposed randomization design in both Android 4.4.4 with Dalvik runtime and Android 5.1.1 with ART runtime. The evaluation result confirmed its acceptable performance overhead and effectiveness on mitigating ROP attacks.
- Explored the possibility of mounting ROP attacks by employing device-compiled binary code (in oat files) as gadgets, and applied the randomization idea to device-compiled code in oat files.

The remaining of this paper is organized as follows. In section 2, related background of ARM instructions and Android applications are presented together with the basics of ROP on ARM. In section 3, the two-stage randomization idea is illustrated. The implementation details are explained in section 4. The performance overhead and the effectiveness of this randomization approach are evaluated in section 5. Besides, section 5 also complementarily discusses the security and limitations of this randomization idea. In the end, we make a conclusion.

II. ROP on Android

In this paper, Android running on ARM is adopted as the target platform to implement and exercise the proposed randomization approach. Therefore, in this section, some basics of android and ARM instructions are introduced firstly, and then how ROP works on Android is presented. Particularly, the possibility of mounting ROP attacks with gadgets in oat files compiled from Dalvik bytecode is also explored.

2.1 The basics of android and ARM

Android is now the most popular mobile OS running on ARM. Its applications usually contain two kinds of code: the native code in shared libraries (*.so) and the Dalvik bytecode in dex files (*.dex). The Dalvik bytecode in Android apps is interpreted by Dalvik virtual machine in Dalvik runtime (the default runtime in Android 4.4.4 and early versions). However, when it comes to ART (Android runtime, the sole runtime in Android 5.0 and later versions), the bytecode are compiled to native code and stored in a newly generated elf binary file (oat file, /data/dalvik-cache/data@app@*.dex) while installing applications. The other kinds of code in apps is shared libraries (*.so), which is compiled to native code before distribution and mainly used for computation-intensive tasks. All kinds of native code, either compiled while app installation or existing in shared libraries, are executable in memory at runtime, so they are potential to be gadgets for ROP attacks. In this paper, we are trying to randomize branch instructions in native code that is either compiled from Dalvik bytecode or residing in shared libraries.

Native code in Android may contain both ARM and Thumb-2 instructions. ARM is a 32-bit fixed-length instruction set. In contrast, Thumb-2 instructions are 16-bit and 32-bit intermixed. Though this enables flexibility and performance, the various instruction length makes binary analysis more difficult. ARM architecture provides 16 core registers with 32-bit length for ARM and Thumb-2 instructions. These registers are labeled from *r0* to *r15*. Specifically, registers from *r12* to *r15* are assigned for specific usage, known as *ip*, *sp*, *lr*, and the *pc* register. In function invocation, registers from *r0* to *r3* are used to store parameters if needed, *lr* is used to store the return address, and *r0* is used to keep the return value. Besides, in native code methods compiled from Dalvik bytecode, *r0* is the pointer of *ArtMethod* class (a class used to maintain all loaded methods), *r4* is used to store suspend check interval, and *r9* is the pointer of current

thread context. Fig. 1 shows a disassembled method compiled from app's Dalvik bytecode, the last two instructions in snippet *sub_35adbc* examine whether *r4*-1 is zero, if it is true, then the suspend check method will be invoked via *r9*, the thread context pointer. In summary, only partial of core registers have specific usage, and there are some free to use registers, which can be employed to perform our randomization.

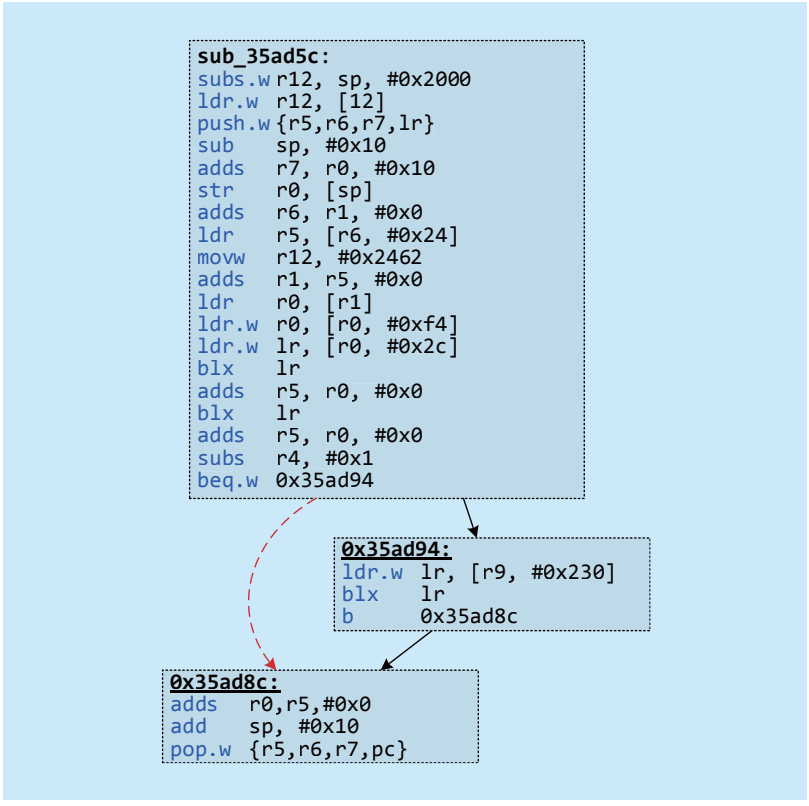

Fig.1 *A native method in oat binary compiled from Dalvik bytecode*

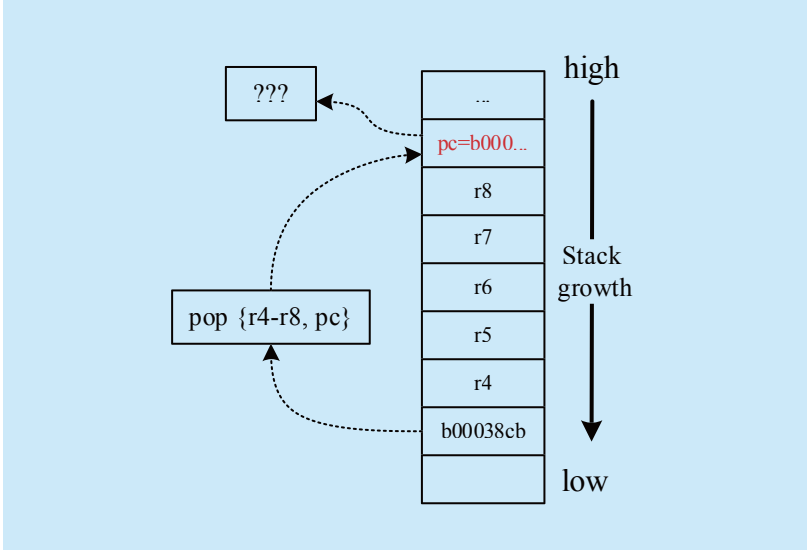

Fig.2 *A simple POP-ROP chain*

In an ARM binary, there are two kinds of branch instructions for control-flow transfer. One is direct branch instruction, such as *blx*, *bl*, *bx* and *b*. The other one is indirect branch instruction, which operates *pc* register directly, e.g., *pop {r3, pc}*. Usually, indirect branch instructions reside at methods' epilogue, and work equivalently with *ret* instructions in *x86*/*x86-64*; the direct branch instructions reside inside methods, and are equivalent with *call* and *jump* instructions. Both of them can be used to transfer control flow in ROP, even in "ROP without return"[5].

2.2 ROP on Android

Without the support of *ret*-like return instructions in ARM, adversary can mount ROP attacks by reusing other branch instructions instead. Indirect branch instructions on ARM, like *pop {r#, ...,r#, pc},* can retrieve branch target and necessary parameters from a forged stack. Therefore, it's possible to construct a gadgets chain for ROP attacks. Fig.2 shows a simple POP-ROP chain: how the instruction *pop {r4-r8, pc}* transfer the control flow to a target address stored in stack[21]. Besides, Checkowa also showed it's possible to craft ROP chain without indirect branch instructions, named ROP on ARM without return[4,5]. He proved the Turing-completeness of ROP gadgets consisted of *blx*-like branch instructions. However, in contrast, indirect branch instructions are more preferable to construct ROP chains in practice due to the direct data retrieving capability from stack.

ROP exploits with branch instructions in oat files

Since Android 5.0, ART become the one and only runtime. During app installation, it compiles app's Dalvik bytecode to executable native code, stored in newly generated oat files. Therefore, a rather straightforward question comes. Can native code in oat files be gadgets for ROP attacks?

We answered this question by conducting

offline analysis on compiled native code in oat files and runtime examination of oat gadgets execution. We found that, native code in oat files can be parsed correctly (showing in Fig. 1) and mapped into executable memory areas. Moreover, native code compiled from Dalvik bytecode contain a large amount of branch instructions that are potential to be ROP gadgets. E.g., as shown in Fig. 1, if adversaries have a chance to compromise *r0*, then they can direct the control flow to somewhere they want via the gadget "*ldr.w lr, {r0, #0x2c};blx lr*", or retrieve a transfer target from the forged stack via the gadget "*add sp, #0x10; pop.w {r5, r6, r7, pc}*".

In general, ART runtime provides much more gadgets for adversaries. In this paper, we also take ART compiled native code into consideration for the first time, which makes ROP defense more complete.

III. ARM-Specific Instruction Randomization

As discussed in Section 2.2, both indirect and direction branch instructions in executable memory are potential gadgets for ROP attacks. In this section, we will introduce our two-stage randomization idea to mitigate ROP attacks on Android by thwarting all branch instructions.

3.1 Stage-1: indirect branch instruction randomization

Fig.3 presents an example of native code execution of a function in an Android shared library (*.so). The function firstly pushes data in *r*4, *r*5, *r*6, and *lr* into stack, then executes its function body, and finally restores saved stack data into *r*4, *r*5, *r*6, and *pc*, redirecting the control flow to its caller. In this example, the epilogue instruction, *pop* {*r*4, *r*5, *r*6, *pc*}, can be a gadget for ROP attacks.

Our objective is to randomize branch instructions that are potential ROP gadgets. More specifically, we try to transform branch targets randomly and transfer the control flow to unintended destinations. Our initial idea is to put more random amount of registers into

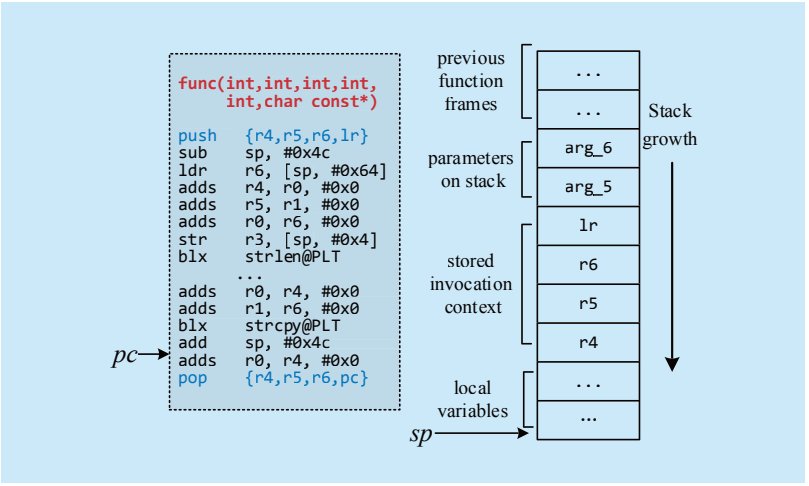

Fig.3 *Native code execution for a function*

functions' prologue and corresponding epilogue instructions, such as "*push {r4-r6,lr}*" and "*pop {r4-r6,pc}*". When adversaries use this epilogue *pop* instruction to jump to next gadget, such incomplete function invocation would make incorrect data restored into *pc* and fail to jump to designed target for the lack of knowledge about the *pop*-like branch instruction and its corresponding stack operation. Particularly, adversaries fail to know how many registers are restored from the forged stack and which registers would be restored with specific pieces of stack data. Fig. 4 shows an example of how our instruction randomization approach works on defending against ROP. From an adversary's perspective, he can only infer a stack layout shown in the left of Fig. 4 based on the knowledge of the original instruction *pop {r4-6, pc}*, and fill corresponding crafted gadgets data into the forged stack. However, after a runtime instruction randomization, the original instruction is transformed to *pop {r3-r7 pc}*. When the randomized instruction is executed, expected pieces of data in the compromised stack won't be loaded into exact registers, especially the *pc* register. Then, the designed gadgets chain will not execute smoothly, and the ROP attacks is stopped.

3.2 Stage-2: register reallocation

Recall the objective that defeating ROP attacks comprehensively, all branch instructions

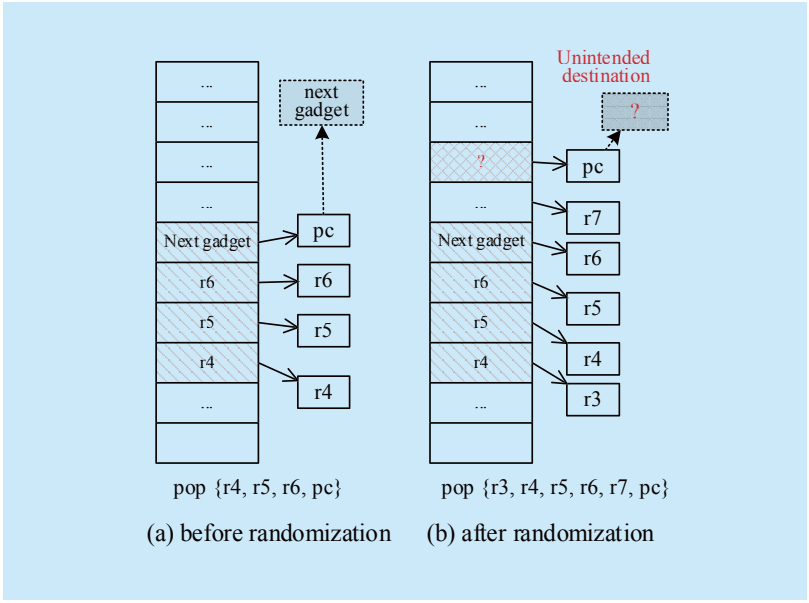

Fig.4 *Example of how instruction randomization works on defending against ROP*

```
;Java_com_***_util_***LibLoader_LibraryUsable
0000b970      push        {r4, r5, r6, lr}
0000b972      sub         sp, #0x14
0000b974      ldr         r4, = 0x1247ac
... ...       ... ...
0000b99e      ldr         r2, [r5]
0000b9a0      strh.w      r1, [sp, #0x8]
0000b9a4      mov         r1, r5
0000b9a6      ldr.w       r6, [r2, #0x340]
0000b9aa      movs        r2, #0x0
0000b9ac      blx         r6
0000b9ae      ldr         r1, = 0xef28a
0000b9b0      mov         r6, r5
0000b9b2      add         r1, r6, pc
0000b9b4      bl          0xaa3e8
0000b9b8      ldr         r2, [sp, #0xc]
0000b9ba      ldr         r3, [r4]
0000b9bc      cmp         r2, r3
0000b9be      beq         0xb9c4
0000b9c0      blx         __stack_chk_fail@PLT
0000b9c4      add         sp, #0x14
0000b9c6      pop         {r4, r5, r6, pc}
                  ; endp
```

Fig.5 *Example of direct branch instructions*

in ARM binaries should be transformed. Therefore, we conduct a register reallocation as the second stage to randomize direct branch instructions inside functions.

ARM uses *blx*, *bl*, *bx*, *b* and other *b*-like direct branch instructions to invoke subroutines or jump to specific code blocks[22]. However, only direct instructions whose branch targets (operands) are registers are potential ROP gadgets. Those direct branch instructions whose branch targets are fixed addresses or constant-specific offsets are ignored while transforming direct branch instructions. As Fig. 5 shows, if adversaries can manipulate *r6* directly or set *r6* by manipulating *r2* via "*ldr.w r6, [r2, #0x340]*" at 0x0000b9a6, then the branch target of *"blx r6"* at 0x0000b9ac can be arbitrary exploitable address.

A straightforward way to transform this kind of direct branch instructions like *blx r6* is to reallocate branch target registers with other free to use registers that we have pushed into stack in the first stage. We can randomly choose a register from the newly pushed registers to replace the branch target registers for each distinguishing direct branch instruction.

Besides, registers in instructions residing in the scope of starting from the assignment and ending before the next assignmentand should be replaced. In Fig. 5, the first assignment of *r6* is "*ldr.w r6, [r2, #0x340]*" at 0x0000b9a6, and the last instruction before the second assignment is *ldr r1, =0xef28a* at 0x0000b9ae, so all instructions operating *r6* and residing in the scope of two assignments ([0x0000b9a6, 0x0000b9b0)) should transform *r6* to a newly allocated register. Note that, instructions at 0x0000b9b0 and 0x0000b9b2 needn't to be updated since they belong to another assignment scope. In this example, if *r7* is the randomly pushed register in the first stage, then *r6* can be replaced with *r7*.

Due to the instruction length-fixed feature of ARM, all instruction transformations in these two randomization stages can be achieved without introducing extra instructions or changing the binary/memory layout. Therefore, this approach is very practical and achievable for randomizing all branch instructions.

IV. Implementation

4.1 System overview

The objective of our randomization approach is to lower adversary's knowledge about runtime instruction layout and make crafted ROP

gadgets fail to execute. For Android system running on ARM, we have implemented a prototype that is compatible with Android 4.4.4 (KTU8P) and Android 5.1.1 (LMY48M), and capable to thwart ROP attacks by randomizing branch instructions in shared libraries and oat binaries. As shown in Fig. 6, the system consists of a static analyzer and a customized image loader, which correspond to the offline static analysis procedure and runtime instruction transformation respectively.

The static analyzer takes shared libraries (*.so) and oat files (/data/dalvik-cach/data@app@*.dex) compiled from Dalvik bytecode as input, and generates corresponding description files for instruction transformation. In a binary's description file, each line describes how a specific function can be transformed. As shown in Table I, each line contains five categories information for a function. In Section 4.2, we will illustrate how we collect these data via offline static analysis in detail.

We put the instruction transformation component into a customized Android image loader[1]. Table II shows the details about source code we patched into Android linker and oat file class. Note that, we need only patch linker for Android 4.4.4, but need additionally patch oat file class for Android 5.1.1 to make oat files transformed as well. Once the original image loader maps a shared library or an oat file into application's memory space, then this newly patched component continually performs instruction randomization and transformation according binary's description file. We will illustrate how we perform instruction transformation in section 4.3.

4.2 Static analysis

In the offline static analysis procedure, we need figure out all information listed in Table I for each discovered function. More specifically, we should 1) discover all functions that are candidates for applying our randomization, 2) find out all candidate registers for each discovered function, 3) identify all instructions for which the offsets need to be updated, and 4) search all reallocatable registers inside a function and corresponding instructions.

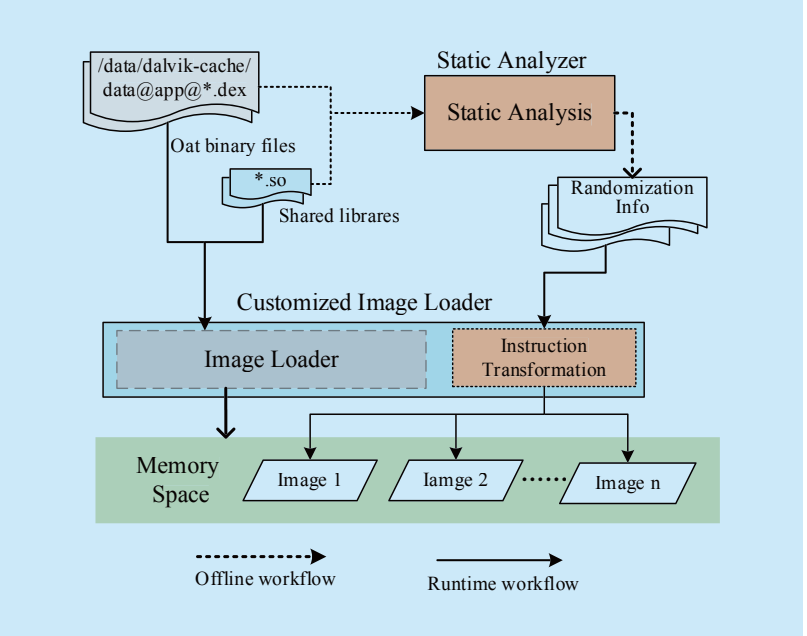

Fig.6 *System overview and its workflow*

We use Hopper[2], a powerful disassemble tool, to disassemble the binary file. However, the mixture of ARM instructions (32-bit) and Thumb instructions (16-bit and 32-bit) and the existence of embedded constants between instructions make the analysis complicated. Like other disassembling tools, Hopper is not reliable in correctly disassembling all instructions. Therefore, we use analysis results from Hopper as a reference and conduct more in-depth analysis with self-developed scripts to ensure the completeness and correctness of static analysis.

4.2.1 Discovery Functions

1) Parse Compiled Functions in Oat Files

Although oat files follow the elf file format, when we tried to analyze oat file's functions compiled from Dalvik bytecode, we found that there is no existing disassembler with the capability to process oat file.

Fortunately, according source code of oat file format, we can conveniently locate all compiled functions correctly and completely by parsing oat file. The compiled functions' file offsets are listed in OatClassHeader structures, which are used to describe classes for

1 Android ART runtime have two backends, Quick and Portable. Different backends have different image loaders. Since Quick is the default backend, so the customized image loader in our prototype system works for Quick backend. Making it compatible with Portable backend only needs some engineering efforts.

2 Hopper Disassembler: http://www.hopperapp.com

Table I *A function's transformation info fields*

Cat. #	Field Type	REMARK
0	32-bit address	Function address
1	16-bit short value	Candidate Registers (each single bit indicates a register from r0 to r15)
2	32-bit address	Address of push instruction
3	32-bit address	Address of pop instruction.
4	64-bit data (a 32-bit address and a 32-bit offset)	Addresses and offsets for instructions that need update their immediate offset.
5	A register number (16-bit), and multiple 32-bit addresses	Reallocated register and address of affected instructions in a register reallocation scope.

(NOTE: Cat. #3, #4, #5 may appear more than once. E.g., a function may have multiple epilogues or multiple register reallocation scopes.)

Table II *Details of patched source code*

Component	Objective	lines of C code	FILE
Android linker	Transform shared libraries	About 320	bionic/linker/linker_phdr.cpp
Oat file class	Transform oat binaries	About 260	art/runtime/elf_file.cc

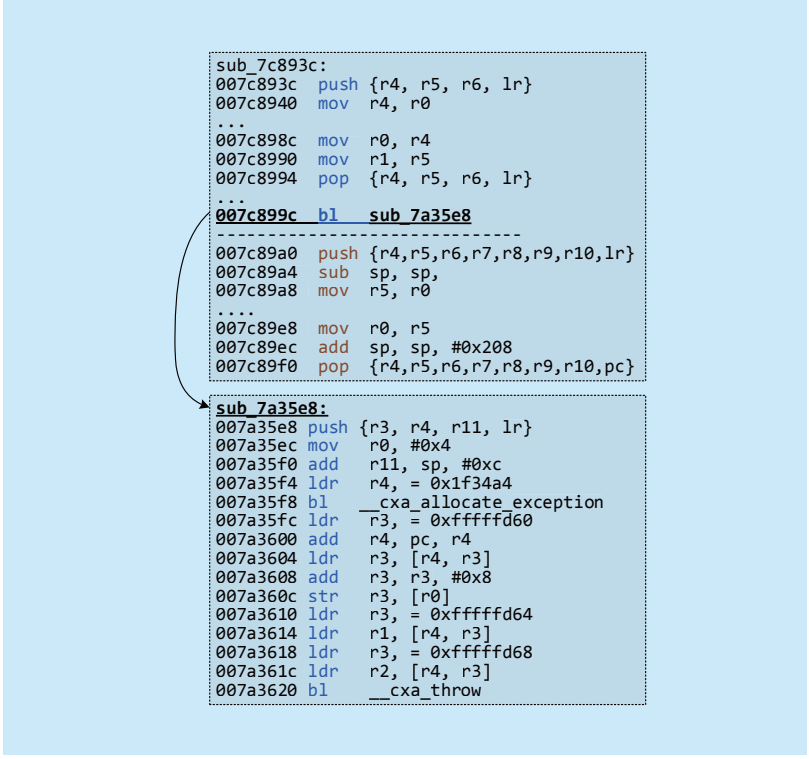

Fig.7 *Failure in discovering blx/bl proceeded functions*

dex file[23]. Therefore, we developed a plugin for Hopper to discover all compiled functions' offsets, and enable Hopper to disassemble functions from these offsets.

Besides, in oat file's OatFileHeader, the instruction set of compiled native code is also specified. We found that Thumb-2 instruction set (kThumb-2) is the default value for Android running on ARM. This eliminates the complexity of inter-mixture of ARM and Thumb-2 instruction set in a binary file.

2) Recognize Functions in Shared Libraries

The Hopper-assisted static analysis can naturally process Android shared libraries that are generic elf files. It starts with functions listed in the exported function table and recursively disassembles instructions and functions by tracing control flow targets of *blx* and *bl*. By paying more attentions on instructions that make transitions between ARM and Thumb-2 instructions set during analysis, we can successfully figure out whether a function is ARM or Thumb-2.

We found that Hopper fails to recognize *blx* and *bl* proceeded functions when the target of *blx*/*bl* is a function that never returns to its caller, e.g., the target is an exception handler. This leads to the missed recognition of new functions. In the example shown in Fig. 7, the jump target sub_7a35e8 is an exception handler that does not return as normal functions would do, and Hopper fails in recognizing the *bl* proceeded function at 0x7c89a0.

As regular functions never have multiple entry points, we solve this problem by recognizing multiple prologue *pop* instructions in functions recognized by Hopper, which indicates the identification of new functions.

4.2.2 Retrieve Candidate Registers

For each function discovered, we should retrieve all potential candidate registers from function's prologue and epilogue instructions. These candidate registers will be used for the runtime coin-flip randomization illustrated in Section 4.3.

The candidate set of registers from which we can choose includes *r1* to *r11* for 32-bit (ARM and Thumb32) instructions and *r1* to *r7* for 16-bit (Thumb16) instructions excluding those in the original prologue and epilogue. For oat files, *r4* and *r9* also should be exclud-

ed from candidate registers for their exclusive usage discussed in Section 2.1.

In order to ensure the correctness of execution and equivalence of semantic, we have to find all epilogue instructions to determine function's candidate registers. All functions discovered in oat files are compiled from Dalvik bytecode's methods, so they are well formatted and share a similar epilogue as the function shown in Fig. 1. The push and pop instructions in oat functions are one-to-one paired. However, for functions in shared libraries, there are some cases should be processed with extra efforts.

CASE 1: Multiple Epilogue Instructions

It is common for a function to have multiple returns and corresponding epilogue *pop* instructions. In the example shown in Fig. 8, there are three return-like *pop* instructions (0x28804, 0x28810, and 0x2881c) pairing with the prologue *push* instruction (0x287bc). It is necessary to identify all these return-like epilogue instructions that pair with the function's only prologue *push* instruction, so that a same randomization result can be applied on all of these epilogue *pop* instructions to make all possible execution path correct and maintain the stack balance. We make sure not to miss any epilogue instructions by extracting the intra-procedure control flow graph (CFG) for each function with the assistance of Hopper and examining all leaf nodes in the CFG.

CASE 2: push-proceeded Prologue

Fig.9 presents a snippet of disassembled code in which there is another push instruction before function's prologue instruction that pushes lr register. Correspondingly, the last three instructions first pop out whatever was pushed at 0x45f62a, adjust sp to offload whatever was pushed at 0x45f628, and, in the end, use a direct branch instruction bx lr to return back to its caller. For simplicity, we only identify push instructions involving register lr as the only push prologue instruction. Note that, we have ensured there is only one exclusive prologue instruction.

CASE 3: Unmatched push and pop

There are scenarios in which the set of

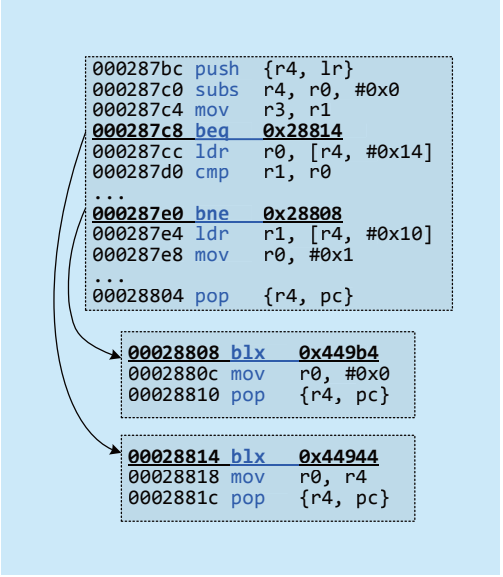

Fig.8 *An example of function with multiple returns*

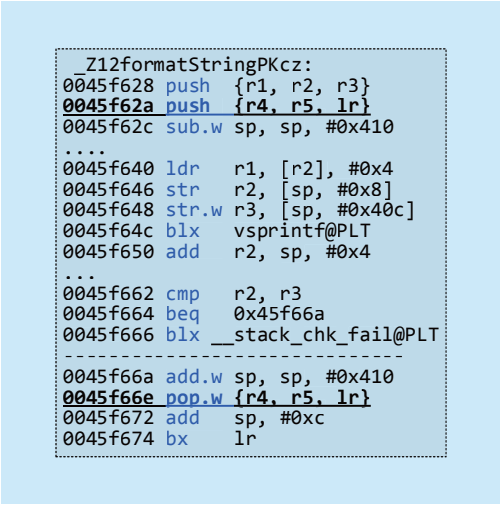

Fig.9 *An example of push-proceeded prologue*

registers pushed and popped in a function prologue and epilogue does not exactly match. Code snippet in Fig. 10 shows there are prologue *push* instruction (0x45fc68) and epilogue *pop* instruction (0x4616d4) involving a same number of registers, but involved registers are different. Fig. 11 presents another example in which different numbers of registers are involved at 0x4616d4 and 0x46171e.

To exercise the maximum of randomization opportunities, we go ahead and figure out candidate registers for our randomization in both cases. We pays additional care in these cases to make sure the sequence (not just the

```
VTestURadio10cellCreateEi:
0045fc68  push {r0, r4, r5, lr}
0045fc6a  adds  r1, #0x1
...
0045fc84  add    r0, sp, #0x4
0045fc86  blx    0x7d3ca4
0045fc8a  mov    r0, r4
0045fc8c  pop  {r2, r4, r5, pc}
```

Fig.10 *Different registers in prologue and epilogue*

```
sub_46724:
004616d4  push {r0,r1,r4,r5,lr}
004616d6  mov    r4, r0
004616d8  ldrb.w r3,[r0,#0x1a8]
004616dc  cbz    r3, 0x46171c
...
0046171c  add    sp, #0x8
0046171e  pop  {r2, r4, r5,  pc}
```

Fig.11 *Different number of registers in prologue and epilogue*

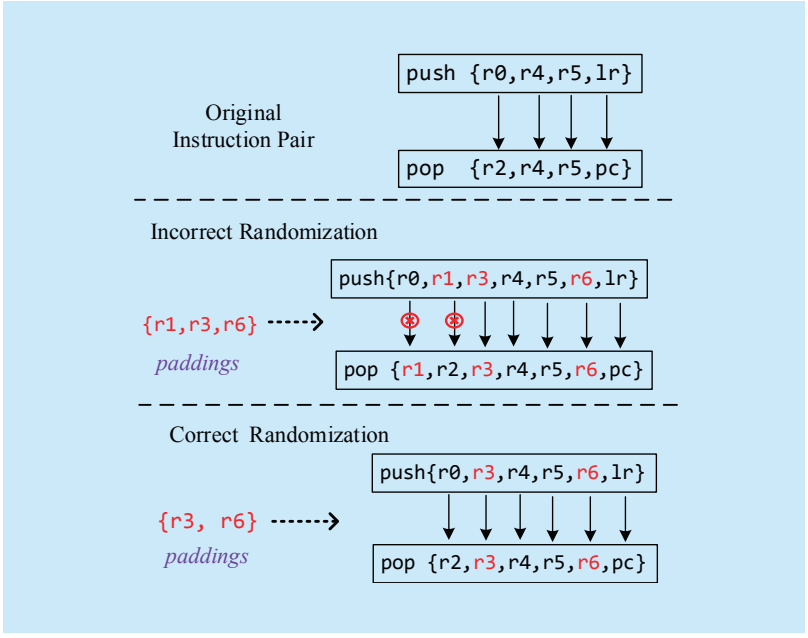

Fig.12 *Correct and incorrect randomization examples for unmatched push and pop*

set) of registers pushed and popped maintain the same one-to-one mapping before and after randomization. Fig. 12 presents examples of correct and incorrect randomization results for the function shown in Fig. 10.

4.2.3 Identify Offset-changed Instructions

Since our randomization inserts extra registers into stack, the offset between data objects on stack may change. Therefore, instructions that address stack data objects with the stack pointer *sp* (or directly via *r11*) should be identified in static analysis procedure, and offsets involved in these instructions should be updated as well by adding the padded registers' size in instruction transformation procedure.

Fig.13 shows how data objects residing in four different stack regions obtain new offsets. With the exception of local variables, instructions accessing data in all other regions, including stored invocation context, function parameters, and data from the previous stack frames, need to be identified for further instruction transformation.

4.2.4 Search Reallocatable Registers

As illustrated in section 3.2, in order to randomize direct branch instructions like *blx r6*, we can reallocate the branch target *r6* with other free to use registers that we have pushed into stack, so *r6* is a reallocatable register. If a function has candidate registers, we will continuously traversal its instructions to search *bl(x)*-like direct branch instructions involving reallocatable registers and determine affected instructions in reallocatable registers' assignment scope. These affected instructions involving the reallocatable registers also should be transformed in the runtime instruction transformation procedure.

4.3 Instruction transformation

We perform the two-stage instruction transformation with static analysis result as soon as binary files are mapped into memory.

In the first stage, we employ a coin-flip randomization to determine the set of registers from candidate registers identified in static analysis, and then encoded these randomly determined registers into function's prologue *push* instruction and epilogue *pop* instructions. Benefiting from the ARM instruction encoding rules that every possible register's presence

in *push/pop* instructions are indicated with a single bit, so we can involve these randomly determined registers into instructions by setting corresponding bits to 1, without inserting or shifting any instructions.

The new offsets involved in instructions addressing relocated stack data objects (shown in Fig. 13) depend on the amount of registers additionally put into stack. More specifically, the increased amount of offset is the size of a register value (4 bytes) multiplied by the number of pushed registers.

The second stage is to reallocate target registers in direct branch instructions, which have already identified in the static analysis procedure. The set of registers for reallocation consists of identified reallocatable registers and registers additionally pushed into stack. We randomly choose a new register from the set of registers for reallocation to replace the reallocatable register in original function. These corresponding affected instructions involving the replaced registers are transformed as well.

After this two-stage randomization, all branch instructions inside functions are transformed with randomness. Without the knowledge of instruction details, adversaries would fail to mount ROP attacks.

V. Evaluation and Discussions

The evaluation focuses on the performance overhead and effectiveness of proposed randomization approach on defending against ROP attacks. To put the evaluation into the context of real-world applications and ROP attacks, we used the top 20 free Android applications in Google Play as evaluation data set. Details of these applications are listed as an appendix.

5.1 Performance overhead

In the prototype, the performance overhead comes from the offline static analysis and runtime instruction transformation. We evaluated these two aspect performance overhead as following.

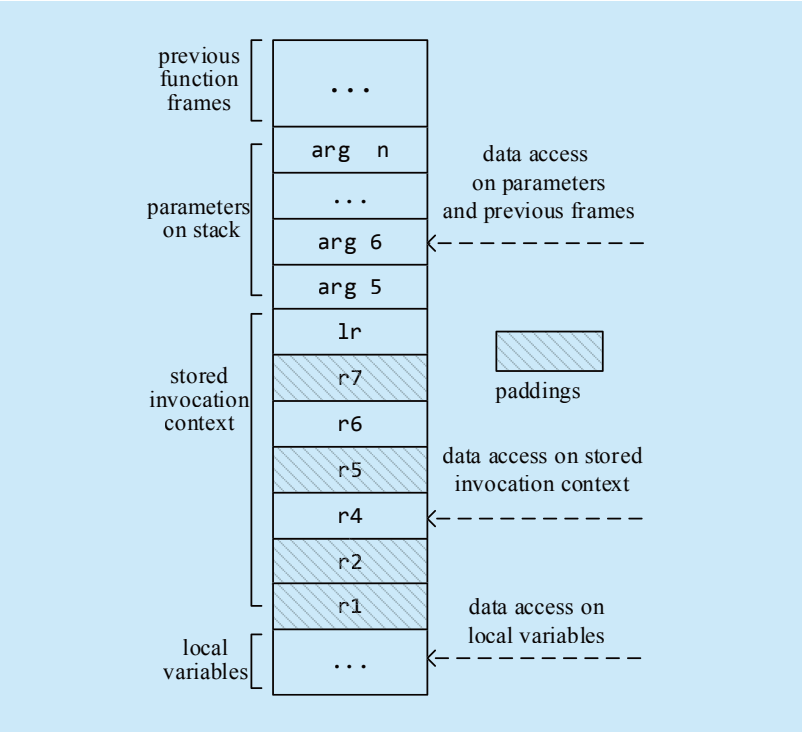

Fig.13 *Data objects on stack with new offsets*

5.1.1 Static Analysis Performance

We conducted the performance overhead evaluation of static analysis on Linux (Ubuntu 14.04-amd64) with i7-5500U processer and 8GB memory.

Fig.14 shows the time consumed by static analysis of shared libraries and oat files. Besides, the number of functions in analyzed binaries are also labeled. Android applications may contain multiple shared libraries, Fig. 16-(a) shows the total time for analyzing all shared libraries in each application. There are 137 libraries in these applications analyzed. Since our static analysis focuses on every individual function inside binaries, the analysis time should positively correlated with the number of functions, which is also confirmed in Fig. 14-(a) and Fig. 14-(b). The time for analyzing a function in shared libraries and oat files are 2.09ms and 14.42ms, respectively. The Hopper plugin for static analysis firstly parsed oat files to retrieve all compiled functions' addresses from OatHeader, thus the average static analysis time for oat functions is larger than the corresponding time for shared

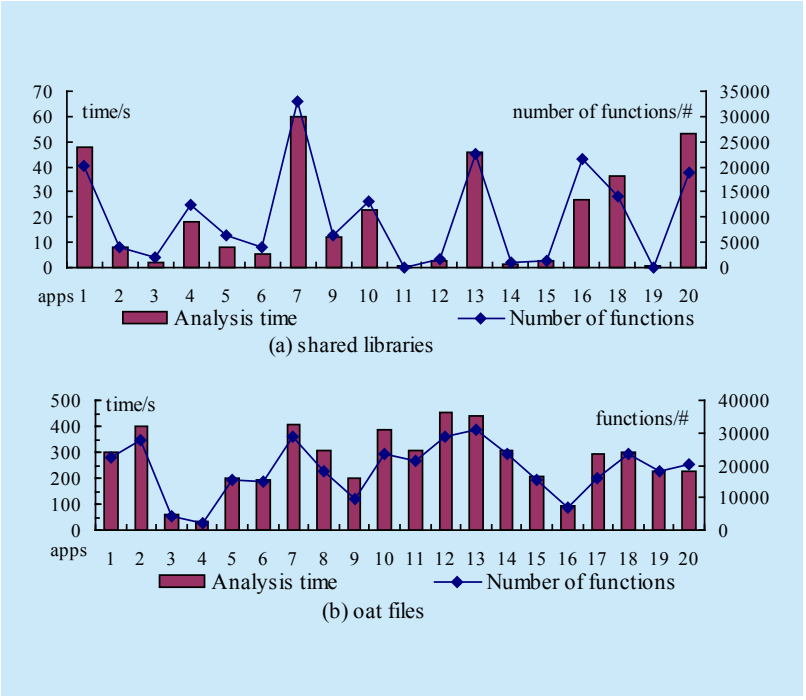

Fig.14 *Static analysis time and number of functions for shared libraries and oat files*

libraries. In addition, the huge amount of functions in oat files also made the analysis of cross-reference more complicate.

Considering the redundant analysis of object cross-reference conducted by Hopper, it's promising to make the static analysis much more efficient by adopting a specific and optimized disassembler. Moreover, the static analysis is a one-off offline analysis and can be achieved ahead of applications' distribution. Therefore, we suppose the performance of static analysis is reasonable and acceptable for practical deployment.

5.1.2 Runtime performance overhead

Both the instruction transformation procedure in the customized image loader and the execution of transformed instructions may introduce some runtime performance overhead. We evaluated the binary mapping time on Google Nexus 5, running with two comparative Android systems compiled from AOSP (Android open source code project) respectively. One was the original system, and the other one adopted our customized loader for instruction randomization.

Fig.15 depicts the time for mapping binaries and corresponding performance overhead ratio. More specifically, bars in Fig. 15-(a) indicate the average mapping time for one shared library in the original system and the randomization-enabled system. The overhead ratio curve in Fig. 15-(a) indicates the ratio value ranges from 24% to 45%. Likewise, as shown in Fig. 15-(b), the overhead ratio for mapping oat files ranges from 19% to 31%. The average overhead ratio for mapping shared library and oat file are 34.9% and 25% respectively. Note that, we only compared the transformation overhead with binary's mapping time, which was in hundreds of microsecond (μs).

When it comes to the overhead of the runtime execution of transformed instructions, there is no observable performance overhead. This is contributed by the minimal in-place instruction transformation that leverages ARM-specific features and doesn't insert any extra instructions into binaries.

Android applications can stay in background if there is enough memory and none manually cleanup. When user try to launch an application, Android prefer switching the application to foreground from background without loading shared libraries and oat files again. That is to say, file mapping is not very frequency for applications.

We also compared the performance with another known approach, CCFIR[13]. Both CCFIR and our prototype share a similar implementation that consists of static analysis and runtime instruction randomization even though CCFIR's objective is to enforce the control flow integrity on x86. As Table III shows, comparing with CCFIR, our prototype provided much better load-time and runtime performance by moving all binary analysis stuffs to the one-off offline static analysis procedure and omitting code blocks shifting.

Therefore, the performance overhead of our prototype system is reasonable and applicable for practical use.

5.2 Effectiveness on Mitigating ROP attacks

5.2.1 Randomization capability

As our objective is to mitigate ROP attacks by randomizing branch instructions residing in functions, we firstly evaluated the percentage of functions that can be randomized and the amount of randomness obtained with the proposed randomization design.

Most applications' shared libraries contain Thumb and ARM functions, but compiled oat files only contain Thumb functions by default. When a function's amount of registers for randomization becomes zero, it means this function cannot be randomized. If recognized functions' prologue and epilogue instructions already involve all alternative registers, they will obtain no randomness. For instance, e.g., if an ARM function pushes/pops *r0-r11*, we cannot randomize branch instructions inside this function.

Fig.16 show various amounts of registers for randomization for ARM and Thumb functions in shared libraries (*.so), and Thumb functions in oat files compiled from applications' Dalvik bytecode (*.dex).

Evaluation results in Fig. 16-(a) and Fig. 16-(b) show that only 0.13% ARM functions and 1.30% Thumb functions in shared libraries have zero register for randomization. In other words, the majority of functions in applications' shared libraries can be randomized. For compiled oat files, the percentage of randomized functions is about 99.86%. Therefore, our randomization idea has the capability to contribute a huge function coverage.

Besides, from figures in Fig. 16, we note that many functions have large amount registers for randomization. More specifically, the average amount of registers for randomization is shown in Fig. 17. For ARM functions and Thumb functions in shared libraries, and Thumb functions in compiled oat files, the average amount of registers for randomization are 7, 4 and 5, respectively. The number of available registers for randomization indicates

Table III *Performance comparison with CCFIR*

Performances	CCFIR	Our Prototype
Static analysis time	Dozens of seconds	Several minutes
Load-time performance	Hundreds of milliseconds(ms)	Hundreds of microseconds(μs)
Instruction execution	3.6% overhead on average	No observerable performance overhead

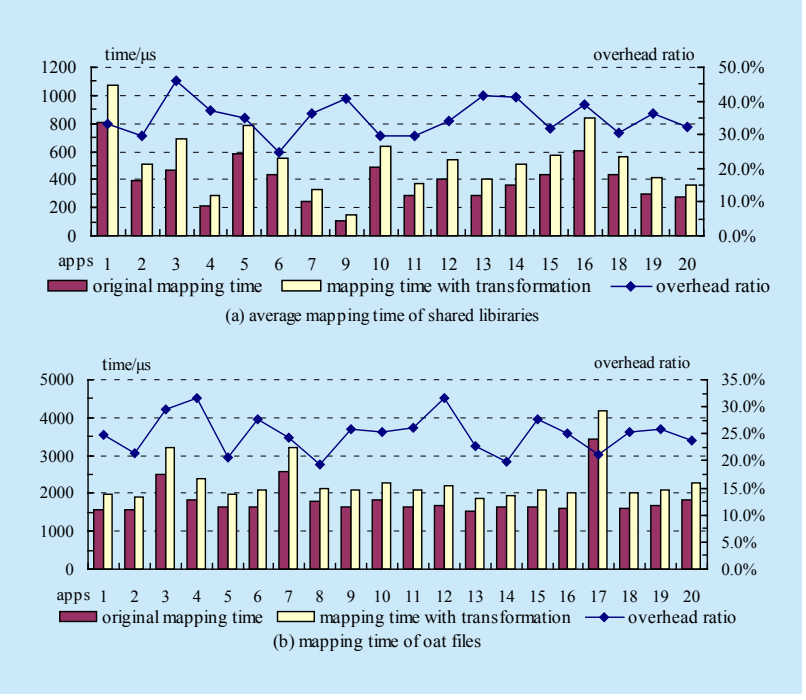

Fig.15 *Runtime performance overhead on mapping shared libraries and oat*

the number of bits in randomness exactly. Note that this is the amount of randomness applied to each individual function (independently). In terms of gadgets chain, the randomness would be more.

Thumb functions in shared libraries are 16-bit and 32-bit instruction intermixed, and 16-bit *push*/*pop* instructions can only operate registers *r0-r7*. In contrast, *push*/*pop* instructions in shared libraries' in ARM functions can operate *r0-r11*. For functions in oat files, all *push*/*pop* instructions are 32-bit Thumb instructions and have 12 operable registers theoretically. However, *r4* and *r9* have exclusive usage for storing suspend check interval and thread object in Android ART, thus the actual amount of operable registers is 10. Consequently, ARM functions in shared libraries obtain the largest

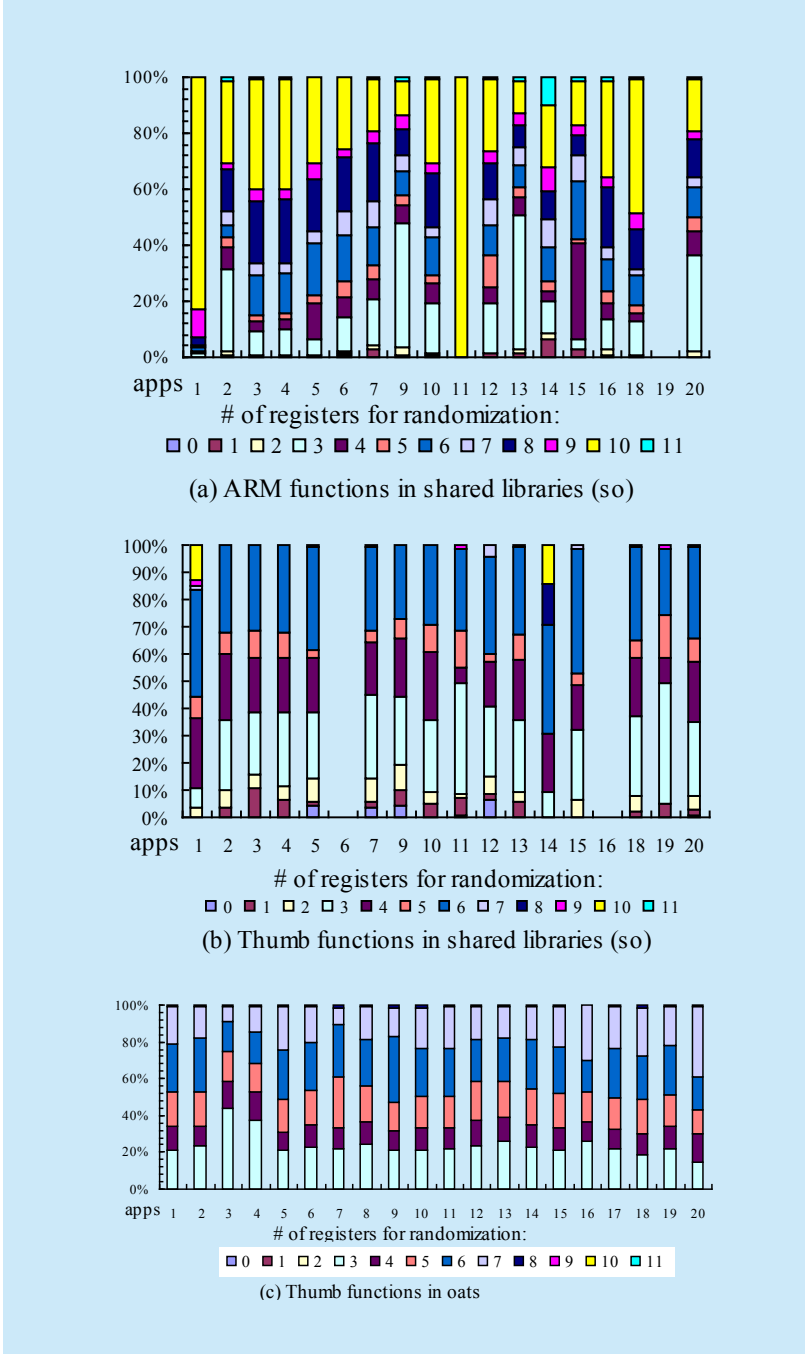

Fig.16 *Percentage of various amount (#) of registers for randomization in shared libraries and compiled oat files*

3 ROPgadget is python script tool for searching gadgets in specified binaries to facilitate ROP exploitation. It supports ELF/PE/Mach-O format on x86, x64, ARM, ARM64, PowerPC, SPARC and MIPS architectures.
4 ROPgadget tool may treat one 4-byte ARM instruction as two 2-byte unintended Thumb instructions.

bits of randomness and Thumb functions in shared libraries obtain least randomness.

5.2.2 Effectiveness on thwarting gadgets

We evaluated the effectiveness on thwarting ROP gadgets by comparing the change of gadgets discovered by ROPgadget[3] in original binaries and randomized copies. We modified ROPgadget tool by adding about 30 lines python code to make it compatible with the compiled Android oat files, which are naturally unsupported[24]. In order to harvest gadgets from randomization result via ROPgadget in the evaluation, we developed an auxiliary binary rewriter with a litter effort on the implementation of runtime randomization to enable offline access of randomized contents.

We counted gadgets in original binaries and 10 randomized copies for every evaluated application. In ROPgadget's ARM mode, the average percentage of randomized ARM gadgets in shared libraries' randomized copies is 94.39%, and this average value for oat files is 95.82%. Similarly, in ROPgadget's Thumb mode, the average percentage of randomized Thumb gadgets for shared libraries and oat files are 96.81% and 97.92%, respectively. Considering ROP attacks need multiple gadgets, we suppose these randomized gadgets would be employed into ROP-chains with much probability to breakdown ROP-chains' execution.

Besides, we also note the percentage of randomized gadgets is higher in Thumb mode than that in ARM mode. One possible reason is that, the gadget search pattern in Thumb mode has two more two-byte aligned instruction patterns that would enable the discovery of gadgets that are two-byte aligned unintended instructions[4]. In contrast, all search patterns in ARM mode are four-byte aligned.

5.2.3 Capability on Mitigating Real-world ROP Attacks

From the perspective of mitigating state-of-the-art ROP-based exploitation techniques, we use two instances to demonstrate the capability of the proposed ARM-specific instruction randomization approach on preventing ROP attacks.

Real-world ROP Exploit for CVE-2014-7911

CVE-2014-7911 is a critical systems vulnerability that resides in all Android versions before 5.0. Without verifying whether the object that is being deserialized is serializable, the system may execute a crafted finalize ()

method for a serialized object, and potentially lead to privilege escalation[25,26]. Here we use an open-sourced exploit that has been widely adopted to gain device's root privilege[27]. In this exploit, various techniques (heap spraying, stack pivoting and ROP) are engaged to perform unexpected behaviors after taking over the control flow. Fig. 18 shows the memory layout of the forged stack crafted by attackers and expected execution traces. However, by randomizing the branch instruction in employed gadgets, our approach made the crafted ROP chain fail to execute. As Fig. 18 depicts, execution trace 2 and 3 were broken down in each exploits respectively.

Oat-gadget-involved Exploit for Manually Crafted Vulnerability

We assumed that Android system already adopted fine-grained randomization on shared libraries, making adversaries fail to retrieve all expected gadget addresses via information leakage; adversaries had to use gadgets residing in application's compiled oat file.

To examine the capability of mitigating oat-gadget-involved exploit, we manually developed an Android application with buffer overflow vulnerability in its shared library. As shown in Fig. 19, when application invoked the vulnerable function *vulfunc* to process data in the specified file via JNI, it might cause buffer overflow and lead code execution by crafting malicious file content. Note that, in order to overwrite the stack in this example, we temporally disabled the stack protector temporarily by specifying the compilation flag **"**-fno-stack-protector**".**

As Fig. 20 shows, we exploited three oat gadgets to set up the command string and redirect the control flow to the system function. During evaluation, we launched this ROP attack for ten times in our customized Android system deployed with randomization implementation. We found that all these attacks were successfully defeated. In other words, our ARM-specific instruction randomization implementation works well for mitigating ROP attacks.

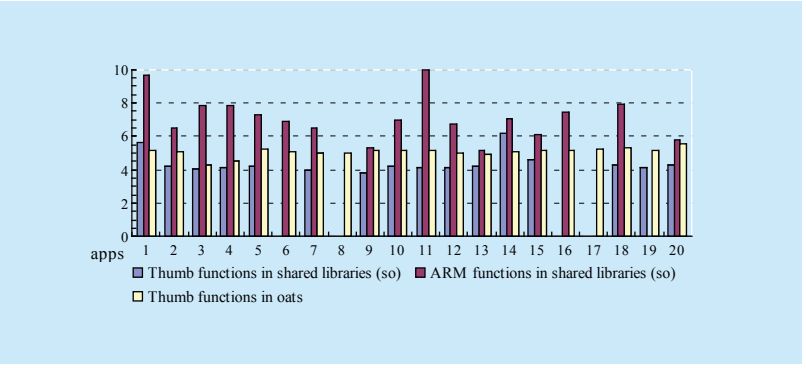

Fig.17 *The average amount of registers for randomization in shared libraries (*.so) and compiled oat files*

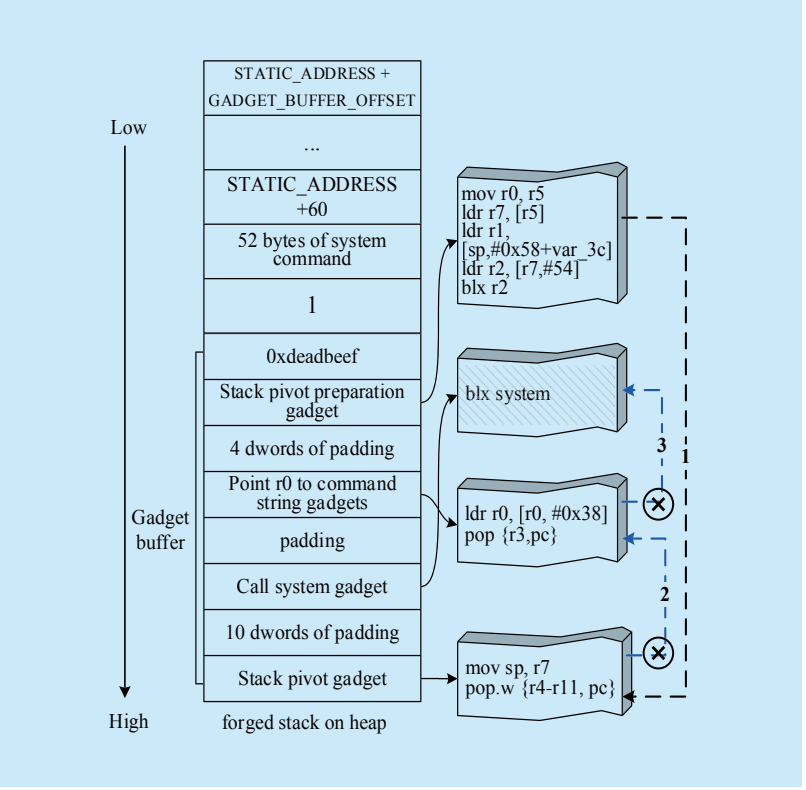

Fig.18 *Memory layout of the forged stack and execution traces CVE-2014-7911*

5.3 Discussions

5.3.1 Security analysis

In terms of the security of the randomization approach itself, we may wonder that whether this kind of instruction transformation introduces new gadgets. Recall that our instruction transformation is an in-place rewriting strategy that only affects original branch instructions

```
char* vulfunc(const char* path){
    char DeadBeef[1024]={0};
    int f_size,bytes_readed;
    FILE* f = fopen(path, "rb");
    if (f==NULL) {
        LOGD("Open File Failed!--[%s]\n", path);
        return err_msg;
    }

    fseek(f, 0, SEEK_END); f_size = ftell(f);
    fseek(f, 0, SEEK_SET);
    bytes_readed = fread(DeadBeef, 1, f_size, f);
    LOGD("read [%d] bytes from file", bytes_readed);
    fclose(f);
    ...
    return err_msg;
}
```

Fig.19 *Manually constructed vulnerable function in application's shared library*

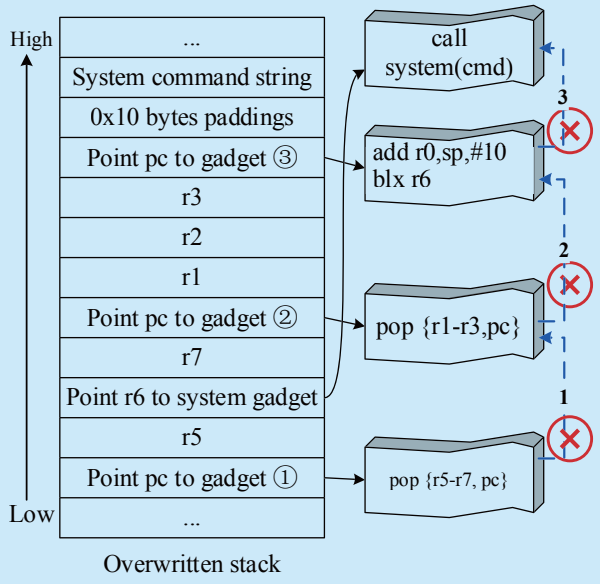

Fig.20 *Stack memory layout and its gadgets*

and applies corresponding changes to instructions accessing randomized stack objects. In consequence, there is no new gadgets introduced.

Comparing with the most wildly deployed randomization approach—ASLR, our randomization design introduces another security property, a fine-grained ARM-specific randomization strategy. ASLR focuses on the base address of mapped objects; in contrast, the proposed randomization approach pays more attention on contents layout of mapped objects. Therefore, our randomization design is not only compatible with ASLR, but also brings complementary security enhancement for Android.

Another issue is that adversaries may question whether it is possible to mount attacks by overwriting particular stack areas with same value, making sure the correct value is restored to *ip* register. The answer is negative. In real-world scenarios, multiple gadgets are necessary while achieving a malicious objective; but it is almost impossible to continuously hijack the control flow transfer via excess stack overwriting.

5.3.2 Protection Capability

In this paper, the implementation and corresponding evaluation of instruction randomization design adopt Android system running on ARM as the target platform.

However, the randomization design itself is an ARM-specific technique, which means it's applicative to the majority of systems on ARM architecture. Therefore, it is promising to apply this instruction randomization idea to iOS, Linux, and other systems running on ARM.

Besides, our randomization strategy is also capable to introduce randomness into the stack by inserting paddings into function frames. By putting more registers into *push* instructions' operands while randomizing branch instructions, the offset of stack object change. Thus, it is reasonable to defense more stack-based attacks, such as stack objects leakage and overwritten.

5.3.3 Limitations

Although our randomization approach is effective in many aspects, there are potential attacks that could circumvent our randomization. More particularly, *massive* memory leakage is the most challenging threat for fine-grained instruction randomization. Adversaries may exploit memory leakage vulnerability to figure out code memory layout and addresses of intended instructions on the fly. The recently

proposed idea, Just-In-Time code reuse, seems to be able to achieve the object of massive memory leakage[28]. However, to the best of our knowledge, this kind of massive memory leakage is only possible for applications that support continuously interactive environment, e.g., action script in flash and java script in internet browser. In contrast, most Android applications are immune to massive memory leakage attack for the absence of continuously interactive script environment.

The static analysis of binaries is always a challenging issue that we also suffered. Some irregular functions that have no *push*-like prologue instructions or *pop*-like epilogue instructions in binaries, especially in shared libraries. In our prototype, these irregular functions are skipped for the absence of indirect branch instructions like *pop {..., pc}*. However, adversaries prefer using indirect branch instructions to construct ROP exploits rather than direction branch instruction for convenience of manipulating registers. In the future, we would like to work on these irregular functions.

VI. Conclusion

In this paper, a novel ARM-specific ROP mitigation strategy and its implementation in Android are presented. By performing a coin-flip randomization on the presence of free to use registers in functions' *push* prologue instructions and *pop* epilogue instructions, indirect branch instructions like *pop {...,pc}* are randomly transformed. With those free to use registers previously introduced into indirect branch instructions, a register reallocation can be performed with randomness to update the branch target for each direct branch instructions in the function scope. Consequently, all branch instructions that are potential to be gadgets for ROP attacks are randomly transformed without introducing extra instructions, leveraging ARM features. Moreover, the implemented prototype is also capable to randomize native code in oat files compiled from Dalvik bytecode. The evaluation on real-world applications and ROP attacks also proves the proposed randomization design could introduce enough randomness on branch instructions to thwart the majority of gadgets and mitigate ROP attacks practically within acceptable performance overhead.

Acknowledgements

The authors would like to thank the reviewers for their detailed reviews and constructive comments, which have helped improve the quality of this paper. This work was supported by the National Natural Science Foundation of China (Grant No. 61202387, 61332019 and 61373168.) and the National Basic Research Program of China ("973" Program) (Grant No. 2014CB340600).

References

[1] Return-oriented programming[EB/OL]. (2015-07-03)[2015-07-03]. https://en.wikipedia.org/w/index.php?title=Return-oriented_programming&oldid=669727753.

[2] BUCHANAN E, ROEMER R, SHACHAM H, *et al.* When Good Instructions Go Bad: Generalizing Return-oriented Programming to RISC[C]// Proceedings of the 15th ACM Conference on Computer and Communications Security. New York, NY, USA: ACM, 2008: 27–38.

[3] KORNAU T. Return oriented programming for the ARM architecture[D]. Ruhr-Universität Bochum, 2010.

[4] CHECKOWAY S, DAVI L, DMITRIENKO A, *et al.* Return-oriented Programming Without Returns[C]//Proceedings of the 17th ACM Conference on Computer and Communications Security. New York, NY, USA: ACM, 2010: 559–572.

[5] DAVI L, DMITRIENKO A, SADEGHI A-R, *et al.* Return-oriented programming without returns on ARM[R]. Ruhr University Bochum, Germany, 2010.

[6] GÖKTAŞ E, ATHANASOPOULOS E, POLYCHRONAKIS M, *et al.* Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard[C]//23rd USENIX Security Symposium (USENIX Security 14). San Diego, CA: USENIX Association, 2014: 417–432.

[7] LU K, SONG C, LEE B, *et al.* ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks[C]//Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security. New York, NY, USA: ACM, 2015: 280–291.

[8] ABADI M, BUDIU M, ERLINGSSON Ú, *et al.* Control-flow Integrity[C]//Proceedings of the 12th

ACM Conference on Computer and Communications Security. New York, NY, USA: ACM, 2005: 340–353.
[9] MASHTIZADEH A J, BITTAU A, BONEH D, *et al.* CCFI: Cryptographically Enforced Control Flow Integrity[C]//Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. New York, NY, USA: ACM, 2015: 941–951.
[10] QIAO R, ZHANG M, SEKAR R. A Principled Approach for ROP Defense[C]//Proceedings of the 31st Annual Computer Security Applications Conference. New York, NY, USA: ACM, 2015: 101–110.
[11] DAVI L, SADEGHI A-R, WINANDY M. ROPdefender: A Detection Tool to Defend Against Return-oriented Programming Attacks[C]//Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. New York, NY, USA: ACM, 2011: 40–51.
[12] ZHANG M, SEKAR R. Control Flow Integrity for COTS Binaries[C]//Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13). Washington, D.C.: USENIX, 2013: 337–352.
[13] ZHANG C, WEI T, CHEN Z, *et al.* Practical Control Flow Integrity and Randomization for Binary Executables[C]//2013 IEEE Symposium on Security and Privacy (SP). 2013 IEEE Symposium on Security and Privacy (SP), 2013: 559–573.
[14] XIA Y, LIU Y, CHEN H, *et al.* CFIMon: Detecting violation of control flow integrity using performance counters[C]//Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on. Boston, Massachusetts USA: IEEE, 2012: 1–12.
[15] PAPPAS V, POLYCHRONAKIS M, KEROMYTIS A D. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing[C]//Proceedings of the 22Nd USENIX Conference on Security. Berkeley, CA, USA: USENIX Association, 2013: 447–462.
[16] SHI W, ZHOU H, YUAN J, *et al.* DCFI-Checker: Checking kernel dynamic control flow integrity with performance monitoring counter[J]. China Communications, 2014, 11(9): 31–46.
[17] PAX TEAM. Address Space Layout Raondomization(ASLR)[J]. 2001.
[18] HISER J, NGUYEN-TUONG A, CO M, *et al.* ILR: Where'd My Gadgets Go?[C]//2012 IEEE Symposium on Security and Privacy (SP). 2012 IEEE Symposium on Security and Privacy (SP), 2012: 571–585.
[19] WARTELL R, MOHAN V, HAMLEN K W, *et al.* Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code[C]//Proceedings of the 2012 ACM Conference on Computer and Communications Security. New York, NY, USA: ACM, 2012: 157–168.
[20] PAPPAS V, POLYCHRONAKIS M, KEROMYTIS A D. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization[C]//2012 IEEE Symposium on Security and Privacy (SP). 2012 IEEE Symposium on Security and Privacy (SP), San Francisco, CA: IEEE, 2012: 601–615.
[21] DRAKE J J, LANIER Z, MULLINER C, *et al.* Android Hacker's Handbook[M]. 1st edition. Indianapolis, IN: Wiley, 2014.
[22] ARM and Thumb-2 Instruction Set Quick Reference Card[EB/OL]. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.qrc0001m/index.html.
[23] SABANAL P. Hiding Behind Android Runtime (ART)[EB/OL]. Blackhat Asia, Singapore: (2015-03-24)[2015-04-21]. http://www.blackhat.com/docs/asia-15/materials/asia-15-Sabanal-Hiding-Behind-ART.pdf.
[24] SALWAN J. ROPgadget[EB/OL]. GitHub, [2015-12-19]. https://github.com/JonathanSalwan/ROPgadget.
[25] CVE-2014-7911[EB/OL]. [2015-04-24]. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-7911.
[26] HORN J. CVE-2014-7911: Android <5.0 Privilege Escalation using ObjectInputStream[EB/OL]. (2014-11-19)[2015-01-19]. http://seclists.org/fulldisclosure/2014/Nov/51.
[27] retme7. Local root exploit for Nexus5 Android 4.4.4(KTU84P)[EB/OL]. [2015-04-24]. https://github.com/retme7/CVE-2014-7911_poc.
[28] SNOW K Z, MONROSE F, DAVI L, *et al.* Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization[C]//2013 IEEE Symposium on Security and Privacy (SP). 2013 IEEE Symposium on Security and Privacy (SP), 2013: 574–588.

Appendix

The evaluated top 20 free apps from Google Play are listed as follows.

App#	Package Name
1	com.cmplay.tiles2
2	com.whatsapp
3	com.facebook.katana
4	com.facebook.orca
5	com.qihoo.security
6	com.wb.goog.ellen.psych
7	com.tencent.mm
8	com.grabtaxi.passenger
9	com.imo.android.imoim
10	com.instagram.android
11	com.dianxinos.optimizer.duplay
12	com.cleanmaster.mguard
13	com.viber.voip
14	com.thecarousell.Carousell

App#	Package Name
15	com.lenovo.anyshare.gps
16	com.king.candycrushjellysaga
17	com.classdojo.android
18	com.ubercab
19	com.dianxinos.dxbs
20	jp.naver.line.android

Biographies

Yu Liang, is currently a Ph.D. candidate at Wuhan University in the major of information security. His research interests include software security, mobile security and trust computing. Email: liangyu@whu.edu.cn.

Guojun Peng, the corresponding author, email: guojpeng@whu.edu.cn. He is currently an associate professor and Ph.D. supervisor in Computer School in Wuhan University. His research interests include software security, malware detection and computer forensics.

Yuan Luo, is currently a graduate student at Wuhan University in the major of information security. His research interests include mobile security and software security.

Huanguo Zhang, is a professor and Ph.D. supervisor in Computer School in Wuhan University. His research interests include cryptography, trust computing, and information security. Email: liss@whu.edu.cn.