11-2015

Stack Layout Randomization with Minimal Rewriting of Android Binaries

Yu Liang

Xinjie Ma

Daoyuan Wu

Xiaoxiao Tang

Debin GAO
Singapore Management University, dbgao@smu.edu.sg

See next page for additional authors

Stack Layout Randomization with Minimal Rewriting of Android Binaries

Yu Liang[1], Xinjie Ma[2], Daoyuan Wu[3], Xiaoxiao Tang[3], Debin Gao[3], Guojun Peng[1], Chunfu Jia[2], and Huanguo Zhang[1]

[1] Wuhan University, Wuhan, China
{liangyu,guojpeng,liss}@whu.edu.cn
[2] Nankai University, Tianjin, China
{mxjnkcs,cfjia}@nankai.edu.cn
[3] Singapore Management University, Singapore, Singapore
{dywu.2015,xxtang.2013,dbgao}@smu.edu.sg

Abstract. Stack-based attacks typically require that attackers have a good understanding of the stack layout of the victim program. In this paper, we leverage specific features on ARM architecture and propose a practical technique that introduces randomness to the stack layout when an Android application executes. We employ *minimal* binary rewriting on the Android app that produces randomized executable of the same size which can be executed on an unmodified Android operating system. Our experiments on applying this randomization on the most popular 20 free Android apps on Google Play show that the randomization coverage of functions increases from 65% (by a state-of-the-art randomization approach) to 97.6% with, on average, 4 and 7 bits of randomness applied to each 16-bit and 32-bit function, respectively. We also show that it is effective in defending against stack-based memory vulnerabilities and real-world ROP attacks.

Keywords: Memory layout randomization, Android security

1 Introduction

Stack plays an essential part in maintaining and managing runtime data of an execution, e.g., context of function invocation, parameters, and local variables. Many attacks are based on disclosure or modification of such information on the stack. Examples include traditional code injection attacks that overwrite sensitive data, e.g., return addresses and function pointers, to execute the injected malicious code [1], and more recent code reuse attacks that chain existing code gadgets together to perform malicious activities [2–5].

A common requirement of such stack-based attacks is a good understanding of the stack layout by attackers. Applications with predictable stack layout are typically exposed to the high risks of such attacks. This requirement of knowing the stack layout becomes more critical in recent Return-Oriented Programming (ROP) attacks because an attacker needs to put more efforts in arranging data on the stack to chain various code gadgets together [2, 6–9].

Randomizing the stack layout is a natural response to make it more difficult for attackers to locate critical data. Modifications have been proposed to operating systems to introduce such randomness. For example, Address Space Layout Randomization (ASLR) randomizes the base address of many code/data segments, and is widely used in both x86 and mobile platforms [10–12]. However, researchers have been questioning such randomization techniques with modified operating systems in their effectiveness (or amount of randomness) [13,14], completeness [15–17], and many claim that they can be circumvented with advanced attacking techniques like Return-Oriented Programming [5,8].

Randomness could also be introduced to the application alone without modifications to the operating system [18–20]. However, this has since been considered as a less favorable solution mainly due to the difficulty in binary rewriting the application as well as the relatively low applicability and amount of randomness [21]. Binary rewriting an executable could be problematic especially when the size of a function in the resulting binary increases, which means that all instructions in the subsequent functions have to be shifted and all jump targets affected have to be recalculated. This can sometimes be avoided with tricks like re-ordering of functions [13], but is in general an unsolved problem that puts heavy stress on its applicability.

In this paper, we explore how far we can go in terms of introducing randomness into the stack layout with some *minimal* rewriting to the executable binary without any operating system support. By *minimal* binary rewriting, we exercise the restriction that no insertion or deletion of instructions is allowed, which also implies that the program size will remain unchanged.

We show that a reasonable amount of randomness of up to 7 bits to many functions is possible by leveraging special features of the ARM architecture when binary rewriting Android applications. Our solution does not require any modification to the Android operating system. The main idea is to randomize the set of registers to be pushed onto the stack at prologue of a function (and the corresponding registers to be popped). For example, a function might be surrounded with `push {r3,r4,r5,lr}` and `pop {r3,r4,r5,pc}` to store registers used in the caller function. Our technique randomly chooses a superset of the registers, e.g., `{r1,r3,r4,r5,r8,r9,lr}`, to be pushed onto (and popped off) the stack. This change effectively adds a random amount of data on the stack and shifts all other data on the stack frame by a random offset. The intuition behind such a design is that this change requires a simple mutation to the `push` and `pop` instructions which change neither the length of the instructions nor the overall size of the app on ARM architecture.

We implement a proof-of-concept binary rewriter to automatically apply this randomization to Android apps. We show that many existing code reuse attacks no longer work with our randomized Android apps. Our experiments with the most popular 20 free Android apps on Google Play also show that the randomization successfully applies to more than 97.6% of the functions, a noticeable increase from 65% achievable by previous state-of-the-art randomization techniques [15]. Every function receives, on average, 7 bits of randomness to their

location on the stack when it uses 32-bit ARM/Thumb instructions or 4 bits of randomness if it uses 16-bit Thumb instructions. Experiments on a format-string vulnerability and a real-world ROP attack show that our proposed randomization is effective in defending against real-world attacks.

2 Background and Threat Model

As our proposed technique deals with binary rewriting of Android apps to be executed on ARM devices, we first briefly present some necessary background of ARM instructions and registers. We also present the threat model under which our proposed solution works.

Any ARM binary (containing native code from shared libraries or Dalvik bytecode compilation) may contain both ARM and Thumb-2 instructions. ARM is a 32-bit fixed-length instruction set. Thumb-2, developed from 16-bit Thumb instructions, constitutes an instruction set with 16-bit and 32-bit instructions intermixed. This brings flexibility and performance; however, the difference in instruction length also makes binary analysis and rewriting more difficult.

ARM architecture provides 16 core registers of 32-bit length for ARM and Thumb-2 instructions. These registers are labeled `r0` to `r15`. Registers from `r12` to `r15` are also known as the `ip`, `sp`, `lr`, and `pc` register. During a function call, 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. A function typically uses some but not all registers.

We assume a threat model in which the adversary has a copy of the original application (without randomization) and understands the full details of our randomization algorithm. The adversary may also have multiple copies of the randomized app; although he/she does not have the specific randomized copy that the victim is using. We also assume that the app might contain some exploitable vulnerabilities that the adversary is aware of.

3 Randomizing Stack Layout and Application Scenarios

Recall that our objective is to introduce randomness to the stack layout when an Android app executes, and to do so with *minimal* binary rewriting without operating system support. In this section, we present the high-level idea of our design and a few scenarios in which our proposed solution might be applied.

3.1 Randomization Design

Fig. 1 shows the native code of a function in an Android app and the corresponding stack layout when it executes. The function first pushes registers `r4`, `r5`, `r6`, and `lr` onto the stack, performs its execution during which `r4`, `r5`, and `r6` are used as temporary storage, and finally pops data out of the stack into `r4`, `r5`, `r6`, and `pc`. Randomizing this stack layout is to make the location of data on the

stack frame unpredictable from the attacker. Considering possible ways of doing so with binary rewriting only (recall that we do not want to modify the operating system), one could introduce random padding to the base (as done in one of the previous state-of-the-art randomization techniques [10]), or to introduce the random padding among data objects in the stack frame [16, 22].

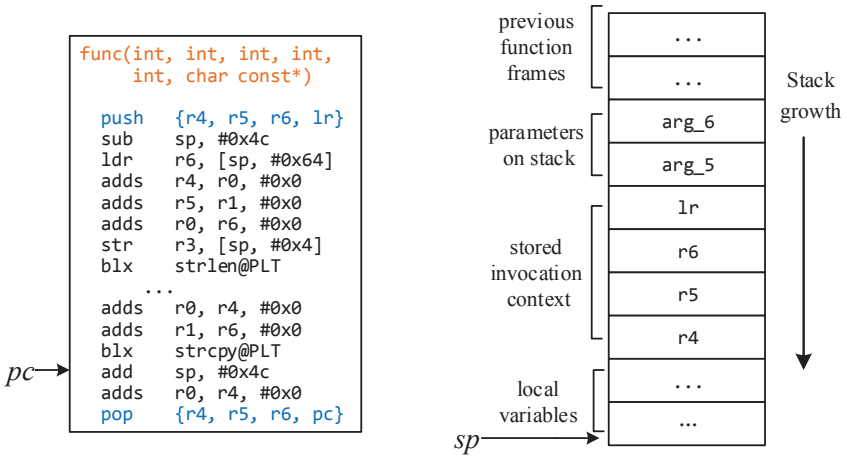

Fig. 1: An example of native code execution

However, we will not be able to introduce padding among the various data objects due to our requirement of doing *minimal* binary rewriting without addition or deletion of instructions. What we could do, though, is to modify the `push` instruction to have a random set of additional registers pushed, as shown in Fig. 2, effectively randomizing the base of the stack frame. In this example, we additionally push `r2` and `r7`. In general, for 16-bit Thumb instructions, the set of general registers that can be pushed and popped includes 8 registers `r0` to `r7`. For ARM and 32-bit Thumb instructions, the set contains 13 registers `r0` to `r12`. Besides them, `lr` and `pc` can also appear on the list.

This design of pushing and popping a random set of registers satisfies our requirement of *minimal* binary rewriting because ARM architecture uses a single `push` or `pop` instruction to push or pop any number of registers, and the instructions to push/pop different sets of registers are of the same length — a feature that is very different from the x86 platform.

To maintain semantic equivalence with the original app for proper execution, there are a few things we have to take note. First, the same set of registers are to be pushed and popped; otherwise our modification could have modified the execution context of the caller function. Second, any references to memory locations on the stack frame between the `push` and `pop` instructions are to be updated with the modified offsets. Lastly, note that `r0` cannot be added to the

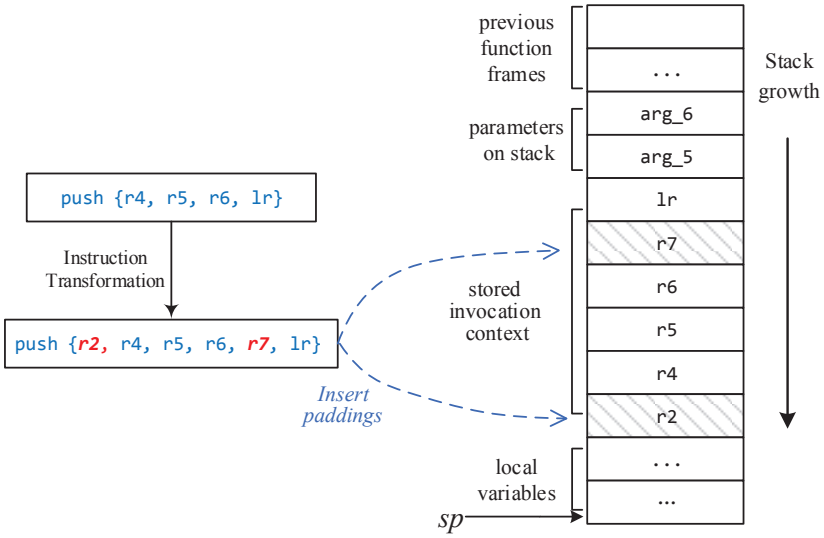

Fig. 2: Randomized set of registers pushed onto the stack

set of padding registers when it is used to store the return value, since popping `r0` would overwrite the return value stored.

3.2 Application Scenarios

Our system can be implemented as a third-party Android app to introduce randomization to the subject app (and potentially re-package it and re-sign it). This satisfies our requirement of not modifying the operating system while achieving the objective of randomizing stack layout. However, there are a few ways in which we can gain better user experience and security.

We can perform the binary rewriting right after app installation. The binary to be rewritten could be the original native code included in the installation package or the `oat` file compiled during installation (when ART runtime is used). This would not require re-packaging of the rewritten app.

We could also perform the binary rewriting every time the app starts its execution. This has the advantage that a new and different randomization is used every time the app is loaded, making it even harder for an attacker to predict the stack layout.

In either case, minimal changes are needed to the Android application installer or loader, and the binary rewriting becomes completely transparent to the end user. Also note that our proposed randomization can be used in conjunction with other existing security mechanisms, e.g., ASLR on Android.

4 Implementation

As a proof-of-concept implementation, our automatic binary rewriting has been implemented in Python with about 2000 LOC. It takes as input any Android app and outputs the randomized app ready for installation and execution on Android 4.0 or later. We first extract the binary files from the Android app, disassemble them, and discover functions and the `push` and `pop` instruction pairs. For each pair identified, we flip a coin and apply our randomization design to change the set of registers accordingly. After that, we update the offset of operands in affected instructions to be consistent with the randomization applied. In the rest of this section, we present details of our implementation in each step and show the complexity involved.

4.1 Static Analysis

In this step, we discover all functions that are candidates of applying our randomization, find out all `push` and `pop` instruction pairs, and discover all instructions for which the offsets need to be updated. We use Hopper[4], a powerful disassembler, 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 sometimes make Hopper incorrect in disassembling all instructions. We use analysis results from Hopper as a reference and conduct more in-depth analysis to ensure the completeness and correctness of static analysis.

Function Discovery Our analysis performs recursive disassembling of instructions and functions by starting with functions listed in the exported function table and tracing control flow targets of `blx` and `bl`. Hopper fails to recognize `blx` and `bl` proceeded functions when the target of `blx` or `bl` is a function that never returns back to its caller, e.g., when the target is an exception handler. Listing 1 in Appendix A shows an example of this case.

We solve this problem by recognizing the multiple prologue instructions in a function recognized by Hopper which signals the identification of a new function.

Push/Pop Instructions For each function discovered, we need to find *all* epilogue instructions to apply the randomization to ensure correct execution and semantic equivalence. Here are some cases that require special attention.

CASE 1: Multiple Epilogue Instructions It is not uncommon for a function to have multiple returns and the corresponding (multiple) epilogue instructions (Listing 2 in Appendix B shows an example). It is important that we identify and change *all* these epilogue instructions to maintain balance of the stack. We make sure not to miss any epilogue instructions by constructing the intra-procedure control flow graph for each function and identifying all leaf nodes.

[4] Hopper Disassembler: `http://www.hopperapp.com`

CASE 2: `push`-proceeded Prologue There could be additional `push` instructions before a function prologue (Listing 3 in Appendix B shows an example). For simplicity, we randomize only the `push` and `pop` instructions pairs involving register `lr`.

CASE 3: Unmatched `push` and `pop` There are scenarios in which the set of registers pushed and popped in a function prologue and epilogue does not match (Listing 4 and Listing 5 in Appendix B are two examples). To maximize the opportunities of randomization, we go ahead and apply our binary rewriting in both cases. We exercise extra care in these cases, though, to make sure that the *sequence* (not just the set) of registers pushed and popped maintain the same one-to-one mapping before and after randomization (see the examples in Appendix B).

4.2 Randomization and Updating Offsets

We flip a coin here to determine the set of registers to be pushed in addition to those included in the original prologue and epilogue. The candidate set of registers from which we can choose includes `r1` to `r11` for ARM instructions and `r1` to `r7` for Thumb (16-bit and 32-bit) instructions excluding those in the original prologue and epilogue. Exception goes to some ARM instructions that use a special constant encoding as explained below.

As discussed in Section 3.1, we have to modify instructions in the randomized function to make sure that they can access the correct data on the stack which is now shifted by a random offset. This typically applies to data addressed with the stack pointer `sp` (or directly via `r11`). Figure 8 in Appendix B shows data in four different regions that require different treatment in our updating.

With the exception of local variables, instructions involving access to data including stored invocation context, function parameters, and data from the previous stack frames need to be updated by adding to the original offset the randomized amount of padding introduced. A complexity arises when trying to update the offset to a number that cannot be properly represented in the ARM instruction. This is due to the design of using only 12 bits of instruction space to represent a useful set of 32-bit constants. If the new offset cannot be properly represented in the 12-bit "rotation" format, we simply exclude the corresponding set of registers from the randomization candidate. Note that the required offset could potentially be represented with (multiple) alternative instructions; we do not explore this option due to the *minimal* binary rewriting requirement.

5 Evaluation and Discussions

Our evaluation focuses on the security effectiveness in defending against stack-based memory attacks. In this section, we present our analysis on the function coverage, amount of randomness introduced, and demonstrate our capability in mitigating real-world attacks. To put our analysis into the context of real-world

applications, we pick the top 20 free Android apps on Google Play as in Jan 2015 and apply our randomization on the native code included in these application packages. One of these twenty apps, QRCode Reader, does not include any native code in its application package, and we therefore did not include it in our analysis; however, as discussed in Section 3, our randomization could also be applied to the native code compiled at load time or installation time with some engineering effort. Our experiments include some widely used native libraries, e.g., `libffmpeg.so` and `libcocos2dcpp.so`. We directly execute every randomized app on a Google Nexus 5 phone with Android 4.4.4 to make sure that our modification maintains the semantics and correctness of execution.

In terms of performance (not the focus of our evaluation in this section), since there is no extra instruction inserted while performing the instruction randomization and *minimal* binary rewriting, there is no observable performance overhead at runtime.

5.1 Function Coverage and Amount of Randomness

Our first evaluation focuses on the number of functions that can be randomized and the amount of randomness obtained with our proposed scheme. Functions that cannot be randomized are those with their prologue and epilogue originally covering all candidate registers, i.e., when `r0-r7` were all pushed/popped in a 16-bit Thumb function or when `r0-r11` were all pushed/popped in an ARM or 32-bit Thumb function.

Fig. 3 shows the percentage of functions that have various numbers of registers for randomization (0 means that the function cannot be randomized). Our evaluation shows that the percentage of functions that cannot be randomized is 0.8% and 2.4% for 16-bit and 32-bit functions, respectively, which are both small. We also notice that many functions have large (≥ 6 for 16-bit functions and ≥ 10 for 32-bit functions) randomization opportunities, average of which account for 32.75% and 30.28% of all 16-bit and 32-bit functions, respectively.

Here we compare our function coverage with another state-of-the-art stack layout randomization technique that does not require operating system support as well. Bhatkar et al. proposed to introduce a randomized padding between the base of stack frame and the local variables by modifying instructions that create the space for local variables, typically `sub esp, #0x100` for example [15]. They reported a function coverage of 65% – 80%. We apply Bhatkar's idea on the 19 Android apps in our experiment and obtain even worse results with an average function coverage of 9.94% function. This relatively low coverage is mainly because only functions with at least one local variable would have instructions like `sub esp, #immediate`. Android, however, has more general-purpose registers and applications typically favors using them rather than local variables. With many functions not using local variable, the applicability of Bhatkar's approach on Android applications is low.

We also count the number of available registers for randomization as it tells us the number of bits of the randomness we introduce for a function frame. Our evaluation results show that 16-bit and 32-bit functions enjoy an average

(a) 16-bit Thumb functions (b) ARM and 32-bit Thumb functions

Fig. 3: Percentage of functions with various number of registers for randomization

of 4 bits and 7 bits of randomness, respectively, out of the maximum amount of randomness of 8 (for the entire set of registers `r0` to `r7`) and 12 (for the entire set of registers `r0` to `r11`) for 16-bit and 32-bit functions, respectively[5]. Note that this is the amount of randomness applied to *each individual function* (independently). Functions usually use only a small subset of the registers with the rest being available for our introducing randomness.

5.2 Randomness among objects inside a function

The previous subsection evaluates the function coverage of our scheme and the amount of randomness introduced. In this subsection, we walk *inside* each function and see the amount of randomness applied to *various objects* inside a function. In particular, we count the distribution of data objects over four different stack regions in Fig. 4. We find that most of data objects that are accessed by the current function residing in regions that can be randomized. These include invocation context, parameters, and previous function frames. Only 4.83% of data objects on average reside in non-randomized invocation context and location variable regions.

5.3 Defending against Stack-based Vulnerabilities

As shown earlier, our approach can randomize stack data objects with a wide randomness coverage to defend against stack-based memory vulnerabilities (e.g.,

[5] We utilized one fewer bit as we chose not to include r0 for simplicity since it usually carries the return value; however, it could be included if the function does not return anything.

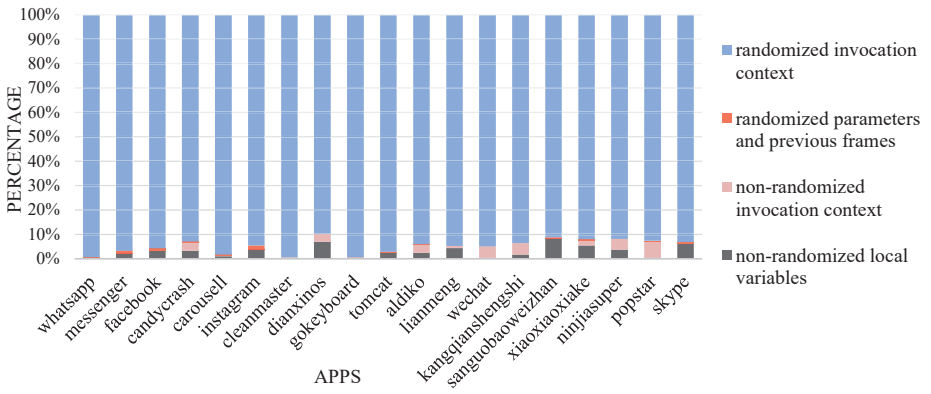

Fig. 4: How data objects are randomized and distributed in stack regions

buffer overflows). Here we further demonstrate this capability using a concrete example. Fig. 5 presents a self-designed format string vulnerability that causes stack data leakage. `sprintf` in function `vulnerable(char* fmt)` enables the attacker to insert an evil format-control string (e.g., `"%s"+4×"%p"`) to retrieve security-critical data `key` by supplying four more `"%p"`. Our experiment demonstrates that such a working exploit fails to succeed with our randomized app. This is because our approach inserts random padding between objects on the stack and changes the relative distance as shown in Fig. 5. These random padding `r7,r2,r3` successfully relocate the previous function frames in stack and randomize locate the security-critical data `key`.

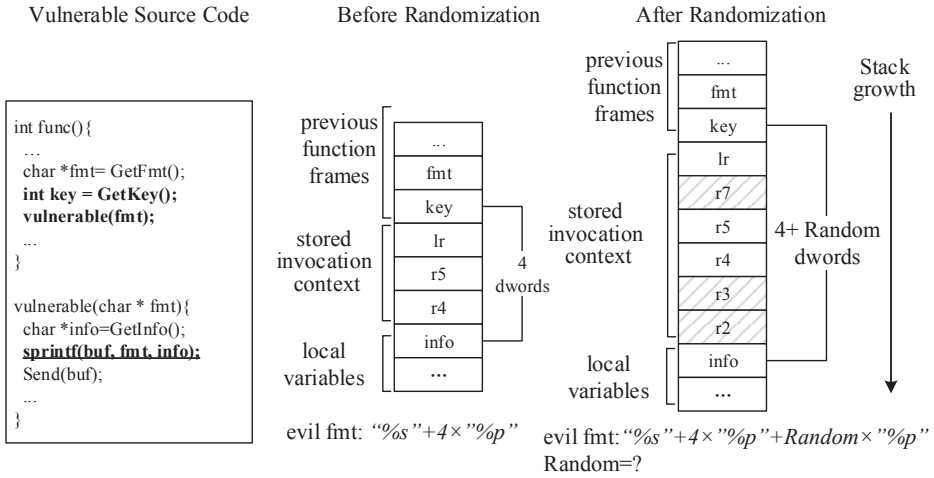

Fig. 5: Our randomization in defending against a format string vulnerability

5.4 Mitigating ROP-based Attacks

Our approach also randomizes gadgets that are potentially employed in building the execution path of an ROP attack. ROP attacks prefer using indirect branch instructions residing in functions' epilogue as gadgets to construct and drive the malicious control flow. Our approach randomizes functions' epilogue, effectively lowering the attacker's knowledge about gadgets and making ROP attacks more difficult. We use a recent famous Android system vulnerability, CVE-2014-7911 [23, 24], as an example. This vulnerability can lead to arbitrary code execution and be exploited to obtain the system privilege [25, 26].

We test and analyze its publicly available exploit code [26]. In Fig. 6, we show the gadgets used by this exploit and the pivoted stack constructed by attackers (using the stack pivoting technique [27]).

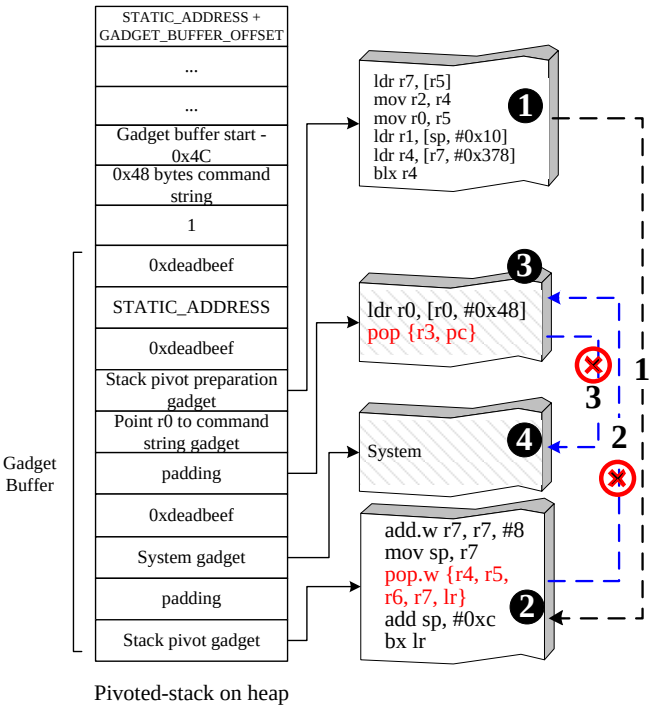

Fig. 6: The real-world exploit [26] for CVE-2014-7911, and it major gadgets.

By analyzing the four major gadgets shown in Fig. 6, we can find that two of them are randomized by our approach. More specifically, the `pop` instructions (marked in red color) in gadget ❷ and ❸ are added with random registers. Consequently, the attacker-intended stack layout (for entering gadget ❹) is changed, and the original control flow from gadget ❸ to ❹ will be disrupted. The exploit

code thus fails to invoke the `system` function. It is worth noting that besides `pop`-based instructions, some `sp`-based data-addressing instructions are randomized (see Section 4.2).

5.5 Limitations

Although our proposed technique is simple and effective in many aspects, there are potential attacks that could circumvent our randomization. In particular, an attacker might want to make use of memory leakage vulnerabilities to find out the randomized set of registers pushed onto the stack, or the effective additional offset introduced. The recently proposed Just-In-Time code reuse attack [5] might seem to be a reasonable strategy in achieving this. However, to the best of our knowledge, this type of attacks have only shown to be possible on applications that support scripting environment to which most Android applications are immune.

6 Related Work

Address space layout randomization (ASLR) is probably the most widely deployed randomization technique to make memory attacks more difficult. The traditional coarse-grained ASLR [10] randomizes the base address of data/code segments for each program, providing relatively small entropy for randomization especially on a 32-bit platform [13]. Fine-grained ASLR techniques [18, 21, 28] proposed recently focus on randomizing code segments to defend against code reuse technique.

In-place code randomization [21] permutes and substitutes instructions for basic blocks. Our technique, instead, substitutes instructions to randomize the memory layout rather than randomizing the code. STIR [18] randomizes addresses of basic blocks at load-time with a focus on the code segment. Our technique, on the other hand, randomizes more fine-grained elements in memory layout of programs and focuses on the data segment. In addition to that, our work is much easier to implement and has a higher chance to get user acceptance due to the *minimal* binary rewriting by leveraging the fixed instruction length on ARM architecture.

One of the advanced ASLR techniques, stack frame padding proposed by Bhatkar et al. [15], is probably the closest to our work. Bhatkar et al. introduce padding within a stack frame to randomize the base address by inserting additional code into the original binary. Our approach achieves the same objective without inserting new code or deleting existing code while achieving higher function coverage (see Section 5.1).

There are other randomization techniques proposed for improving security, e.g., instruction set randomization [29, 30] and control flow randomization [31]. There is also a wide body of research that defend against stack disclosure and modification without randomization [10, 32–34]. Being very different from these techniques, our work reallocates different types of data on the stack frame and could defend against more general memory attacks.

7 Conclusion

In this paper, we introduce a novel stack data randomization method which is achieved by a lightweight ARM-specific instruction randomization strategy. By randomly updating the number of registers in the operand of function's prologue `push` and epilogue `pop` instructions, randomized padding is inserted between function's invocation context. Evaluation on real-world applications shows that out technique covers more than 97.6% functions in an application and introduces on average 4 and 7 bits of randomness to 16-bit and 32-bit functions, respectively. More than 95% of objects in functions are randomized with a new address. We also show the effectiveness of our approach in defending against stack-based memory vulnerabilities and real-world ROP attacks.

Acknowledgments We would like to thank the anonymous reviewers for providing valuable feedback on our work. This research was partially supported by the National Science Foundation of China (Grant No. 61202387, 61332019, and 61373168) and the National Key Basic Research Program of China (Grant No. 2014CB340600).

Appendix

A Missing Functions in Static Analysis

Listing 1: Failure in discovering blx and bl proceeded functions

```
1  sub_7c893c:
2  007c893c  push {r4, r5, r6, lr}
3  007c8940  mov  r4, r0
4  ...
5  007c8990  mov  r1, r5
6  007c8994  pop  {r4, r5, r6, lr}
7  ...
8  007c899c  bl   sub_7a35e8
9  --------------------------------
10 007c89a0  push {r4, r5, r6, r7, r8, r9,
           r10, lr}
11 007c89a4  sub  sp, sp,
12 ....
13 007c89e8  mov  r0, r5
14 007c89ec  add  sp, sp, #0x208
15 007c89f0  pop  {r4, r5, r6, r7, r8, r9,
           r10, pc}
16 --------------------------------
17 ...
```

```
18 ...
19 --------------------------------
20 sub_7a35e8:
21 007a35e8 push {r3, r4, r11, lr}
22 007a35ec mov   r0, #0x4
23 007a35f0 add   r11, sp, #0xc
24 007a35f4 ldr   r4, = 0x1f34a4
25 007a35f8 bl   __cxa_allocate_exception
26 007a35fc ldr   r3, = 0xfffffd60
27 007a3600 add   r4, pc, r4
28 007a3604 ldr   r3, [r4, r3]
29 007a3608 add   r3, r3, #0x8
30 007a360c str   r3, [r0]
31 007a3610 ldr   r3, = 0xfffffd64
32 007a3614 ldr   r1, [r4, r3]
33 007a3618 ldr   r3, = 0xfffffd68
34 007a361c ldr   r2, [r4, r3]
35 007a3620 bl    __cxa_throw
```

In this example, jump target `sub_7a35e8` is an exception handler that does not return as a normal function would do, and Hopper fails in recognizing the `bl`-proceeded function at `0x7c89a0`.

B Complexities in Identifying Push/Pop Instructions

Listing 2: A function with multiple returns

```
000287bc push  {r4, lr}
000287c0 subs  r4, r0, #0x0
000287c4 mov   r3, r1
000287c8 beq   0x28814
000287cc ldr   r0, [r4, #0x14]
000287d0 cmp   r1, r0
...
000287e0 bne   0x28808
000287e4 ldr   r1, [r4, #0x10]
000287e8 mov   r0, #0x1
...
00028804 pop   {r4, pc}
------------------------------
00028808 blx   0x449b4
0002880c mov   r0, #0x0
00028810 pop   {r4, pc}
------------------------------
00028814 blx   0x44944
00028818 mov   r0, r4
0002881c pop   {r4, pc}
```

Listing 3: A function with push-proceeded prologue

```
 _Z12formatStringPKcz:
0045f628 push  {r1, r2, r3}
0045f62a push  {r4, r5, lr}
0045f62c sub.w sp, sp, #0x410
....
0045f640 ldr   r1, [r2], #0x4
0045f646 str   r2, [sp, #0x8]
0045f648 str.w r3, [sp, #0x40c]
0045f64c blx   vsprintf@PLT
0045f650 add   r2, sp, #0x4
...
0045f662 cmp   r2, r3
0045f664 beq   0x45f66a
0045f666 blx __stack_chk_fail@PLT
------------------------------
0045f66a add.w sp, sp, #0x410
0045f66e pop.w {r4, r5, lr}
0045f672 add   sp, #0xc
0045f674 bx    lr
```

In Listing 2, instructions at `0x28804`, `0x28810`, and `0x2881c` are epilogue instructions corresponding to the prologue instruction at `0x287bc`.

Listing 3 shows an example in which there is another `push` instruction before the prologue instruction that pushes register `lr`. 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.

Listing 4: Different registers in prologue and epilogue

```
VTestURadio10cellCreateEi:
0045fc68  push {r0, r1, r4, lr}
0045fc6a  adds  r1, #0x1
...
0045fc84  add    r0, sp, #0x4
0045fc86  blx    0x7d3ca4
0045fc8a  mov    r0, r4
0045fc8c  pop  {r2, r3, r4, pc}
```

Listing 5: Different number of 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  {r4, r5,  pc}
```

Listing 4 shows an example where the same number of registers are pushed and popped, but they are of different registers. Listing 5 shows another example where different numbers of registers are pushed and popped.

Fig. 7 presents examples of correct and incorrect randomization results for the original function which is similar with the function shown in Listing 5.

References

1. One, A.: Smashing the stack for fun and profit. Phrack magazine (1996)

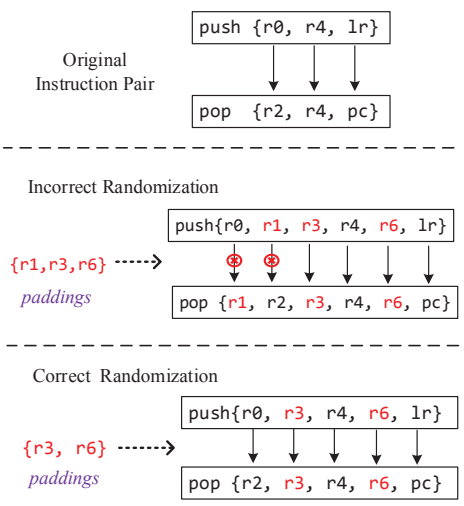

Fig. 7: Correct and incorrect randomization examples for unmatched push and pop

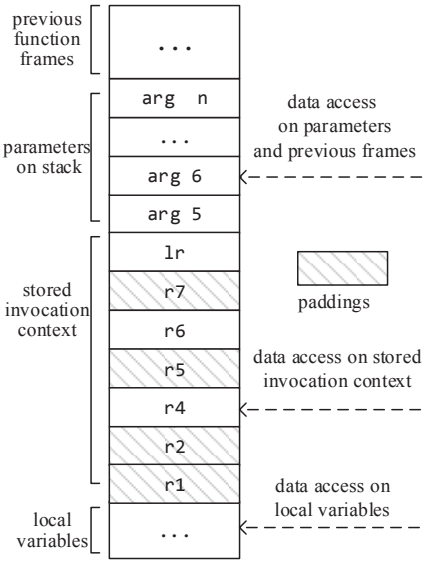

Fig. 8: Data on stack with new offsets

2. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: Proc. ACM CCS. (2007)
3. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: Proc. ACM CCS. (2010)
4. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: Proc. ACM AsiaCCS. (2011)
5. Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., Sadeghi, A.R.: Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In: Proc. IEEE Symposium on Security and Privacy. (2013)
6. Davi, L., Sadeghi, A.R., Lehmann, D., Monrose, F.: Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In: Proc. USENIX Security. (2014)
7. Carlini, N., Wagner, D.: Rop is still dangerous: Breaking modern defenses. In: Proc. USENIX Security. (2014)
8. Buchanan, E., Roemer, R., Shacham, H., Savage, S.: When good instructions go bad: Generalizing return-oriented programming to RISC. In: Proc. ACM CCS. (2008)
9. Francillon, A., Castelluccia, C.: Code injection attacks on harvard-architecture devices. In: Proc. ACM CCS. (2008)
10. Team, P.: Pax address space layout randomization (ASLR). `https://pax.grsecurity.net/docs/aslr.txt` (2003)
11. Apple: iOS Security Guide. `https://www.apple.com/business/docs/iOS_Security_Guide.pdf` (2014)
12. Google: Security Enhancements in Android 1.5 through 4.1. `https://source.android.com/devices/tech/security/enhancements/enhancements41.html`

13. Shacham, H., Page, M., Pfaff, B., Goh, E.J., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: Proc. ACM CCS. (2004)
14. Durden, T.: Bypassing pax ALSR protection. Phrack Magazine (2002)
15. Bhatkar, S., DuVarney, D.C., Sekar, R.: Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In: Proc. USENIX Security. (2003)
16. Chen, X., Slowinska, A., Andriesse, D., Bos, H., Giuffrida, C.: StackArmor: Comprehensive Protection from Stack-based Memory Error Vulnerabilities for Binaries. In: Proc. ISOC NDSS. (2015)
17. Bhatkar, S., Sekar, R., DuVarney, D.C.: Efficient techniques for comprehensive protection from memory error exploits. In: Proc. USENIX Security. (2005)
18. Wartell, R., Mohan, V., Hamlen, K.W., Lin, Z.: Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In: Proc. ACM CCS. (2012)
19. Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., Zou, W.: Practical control flow integrity and randomization for binary executables. In: Proc. IEEE Symposium on Security and Privacy. (2013)
20. OSullivan, P., Anand, K., Kotha, A., Smithson, M., Barua, R., Keromytis, A.: Retrofitting security in cots software with binary rewriting. In: Proc. Springer IFIP SEC. (2011)
21. Pappas, V., Polychronakis, M., Keromytis, A.: Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In: Proc. IEEE Symposium on Security and Privacy. (2012)
22. Bhatkar, S., DuVarney, D.C., Sekar, R.: Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In: Proc. USENIX Security. (2003)
23. Horn, J.: CVE-2014-7911. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-7911` (2014)
24. Horn, J.: CVE-2014-7911: Android <5.0 Privilege Escalation using ObjectInputStream. `http://seclists.org/fulldisclosure/2014/Nov/51` (2014)
25. Lavi, Y., Markus, N.: CVE-2014-7911: A Deep Dive Analysis of Android System Service Vulnerability and Exploitation. `http://goo.gl/XMCM2J` (2015)
26. retme7: Local root exploit for Nexus5 Android 4.4.4 (KTU84p). `https://github.com/retme7/CVE-2014-7911_poc` (2015)
27. Li, X.: Emerging stack pivoting exploits bypass common security. `https://goo.gl/4FbVlF` (2013)
28. Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., Davidson, J.W.: Ilr: Where'd my gadgets go? In: Proc. IEEE Symposium on Security and Privacy. (2012)
29. Kc, G.S., Keromytis, A.D., Prevelakis, V.: Countering code-injection attacks with instruction-set randomization. In: Proc. ACM CCS. (2003)
30. Barrantes, E.G., Ackley, D.H., Palmer, T.S., Stefanovic, D., Zovi, D.D.: Randomized instruction set emulation to disrupt binary code injection attacks. In: Proc. ACM CCS. (2003)
31. Davi, L., Liebchen, C., Sadeghi, A.R., Snow, K.Z., Monrose, F.: Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In: Proc. ISOC NDSS. (2015)
32. Microsoft: /GS (buffer security check). `https://msdn.microsoft.com/en-us/library/8dbf701c.aspx`
33. Cowan, C., Beattie, S., Johansen, J., Wagle, P.: Pointguard tm: protecting pointers from buffer overflow vulnerabilities. In: Proc. USENIX Security. (2003)
34. Vendicator: Stack shield. `http://www.angelfire.com/sk/stackshield/` (2000)