IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2024.0429000

Comments on: RIO: Return Instruction Obfuscation for Bare-Metal IoT Devices with Binary Analysis

KAI LEHNIGER[1], PETER LANGENDÖRFER[1,2]
[1]IHP - Leibniz-Institut für innovative Mikroelektronik, 15236 Frankfurt (Oder), Germany ({lehniger, langendoerfer}@ihp-microelectronics.com)
[2]Brandenburgische Technische Universität Cottbus-Senftenberg, 03046 Cottbus, Germany (e-mail: peter.langendoerfer@b-tu.de)
Corresponding author: Kai Lehniger (e-mail: lehniger@ihp-microelectronics.com).

ABSTRACT This is a comment on "RIO: Return Instruction Obfuscation for Bare-Metal IoT Devices with Binary Analysis". RIO prevents finding gadgets for Return-Oriented Programming attacks by encrypting return instructions. This paper shows flaws in the design of RIO that allow for the easy retrieval of the plaintext return instructions without decrypting them. Additionally, changes are proposed to improve upon the original idea.

INDEX TERMS ARM, Internet of Things, Return-Oriented Programming, Security

I. INTRODUCTION

Return-oriented programming (ROP) [**?**] is a popular attack method for applications written in unsafe languages. By utilizing some kind of memory vulnerability, an attacker can overwrite a return address with his payload, jumping to a gadget, a small code snippet that ends with a return instruction. Each return of a gadget will pop the next gadget address from the stack, creating a gadget chain that allows arbitrary computations.

Several countermeasures against ROP attacks have been proposed, one of it being Return Instruction Obfuscation (RIO) [**?**]. The basic presumption RIO works with is that an attacker needs to know the return instructions in order to find gadgets with binary analysis. Without gadgets, no ROP attack can be performed. This paper will describe a method which allows to recover the unencrypted return instruction with high precision without the need of breaking the encryption. Afterwards, possible improvements will be discussed.

The rest of this paper is structured as follows. Section II briefly summarizes the function return mechanic in ARM, Section III gives an overview of RIO, and Section IV discusses possibilities to break its protection. Section V shows possible improvements of the original design and Section VI concludes the paper.

II. FUNCTION RETURNS IN ARM

ARM processors have a dedicated register that holds return addresses, `lr`, the link register. When `bl` or `blx` instructions are used to perform a function call, the return address is automatically being put in the `lr` register. A function return can be performed with `bx lr`. In such a case, the return address is never exposed to an attacker that has access to the stack. When more function calls happen, `lr` needs to be stored in the stack. Instead of restoring its value in the end, the return address can be popped directly from the stack into the program counter. This is typically done together with other register values in a single `pop {reglist}` instruction. Attackers can use this by controlling not only the return address but also register values that can serve as an input for the gadget.

III. RIO: RETURN INSTRUCTION OBFUSCATION

RIO [**?**] removes the possibility to find gadgets by encrypting all return instructions. Typically, gadget finding algorithms start to search for gadgets with the return instruction. By removing this starting point from the binary, these algorithms need to be adapted in order to work again. Also, by removing the information which registers are popped from the stack, it is no longer possible to construct a payload, as it is unknown how much the stack pointer advances and where in the payload to place the next gadget address.

RIO is implemented by using compiler custom passes of the LLVM compiler. An initialization module is inserted in the beginning of the existing firmware and a Return Control Flow (RCF) module is placed in front of every return instruction [**?**]. The return instructions themselves are encrypted. When running the firmware, the initialization module scans the binary for all RCF modules and decrypts subsequent return instructions. The decrypted instructions are placed in a

Listing 1. Return Control Flow module code [?]

```
ldr     r0, [pc, #12]
adds    r0, #offset
mov     pc, r0
```

table in SRAM.

The RCF module consists of three instructions shown in Listing 1. The first instruction loads the base address of the table into register `r0`, which is placed after the encrypted return instruction. The second instruction adds an offset to the base address to point to the corresponding return instruction in the table. The last instruction jumps into the table, where the return instruction is executed.

IV. ATTACK POINTS

RIO has three major flaws that attackers could exploit, two of them being used to adapt binary analysis in order to find gadgets.

The first flaw is the general assumption that in order to perform ROP attacks, an attacker needs to perform binary analysis. There have been examples in the past of attacking a device without having a copy of the target binary [**?**]. However, without binary analysis, ROP attacks become more difficult and therefore the goal of RIO to hinder this step is still worth pursuing.

The second flaw is the assumption that RIO hides the position of return instructions. The RIO initialization module itself uses the RCF modules in order to find the positions of return instructions. Consequently, any attacker could do the same. Of course only knowing the position of return instructions is not enough for ARM, as the exact registers need to be known in order to prepare the payload.

This can be done by using the last flaw of the approach: RIO keeps the `push` instructions in the prologue unencrypted. Typically, function prologues and epilogues are symmetrical. Registers that are being pushed onto the stack in the prologue are being popped in the epilogue. This can be illustrated by taking the code examples from [**?**] that were used to demonstrate the encryption. Listing 2 shows the `push` and `pop` instructions in the prologue and epilogue of two functions. In the first function *RIOtest* four registers `r4`, `r6`, `r7`, and `lr` are pushed onto the stack in the prologue. In the epilogue, the same registers are popped, with the only difference of `lr` being replaced with `pc`. The same pattern repeats for *main2* with the registers `r7` and `lr`. Even when completely removing the `pop` instructions from the binary, they could easily be derived by looking at what registers where pushed onto the stack and need to be restored. Of course, instead of restoring `lr`, the return address is directly popped into `pc`.

V. POSSIBLE IMPROVEMENTS

While the first two flaws are inherent and cannot be targeted, as the encrypted return addresses must be locatable in order to

Listing 2. Symmetrical push and pop instructions of unencrypted functions taken from [?]

```
RIOtest:
    push    {r4, r6, r7, lr}
    ...
    pop     {r4, r6, r7, pc}

main2:
    push    {r7, lr}
    ...
    pop     {r7, pc}
```

decrypt them, the third flaw can be addressed. By extending the encryption to the `push` instructions as well, using a similar method, the information of what registers are being stored on the stack can be hidden. Doing so would take a bit more effort due to the fact that, after the execution of the decrypted `push` instruction, the control flow needs to jump back to the function.

However, even encrypting both, `push` and `pop`, instructions would probably not suffice. The reason to push and pop registers to the stack in the prologue and epilogue is given by the calling conventions. Callee saved registers are registers that must remain unchanged for the caller when calling a function. In order to meet this guarantee, the callee is required to store and restore all callee saved registers it uses during its execution. Consequently, by analysing which callee saved registers are being used inside a function, the `push` and `pop` instructions can be derived.

The authors found two ways to remove this method of deriving the instructions that can be used in conjunction:

- For each `push` and `pop` pair a random number of additional registers can be added. These additional registers would not be possible to be derived from binary analysis.
- Each decrypted `push` and `pop` pair in the table could be replaced by multiple instructions to randomize the position of the return addresses in each stack frame. Since the table is created at runtime, the positions on the stack would be different with each reset of the IoT device. FIGURE 1 shows a return instruction on the left, as well as possible replacement sequences next to it and how it effects the position of the return address in the stack. The corresponding `push` instructions have to be changed in a similar way to match the layout.

VI. CONCLUSION

This paper showed several severe flaws of the RIO implementation and presented ideas how to use these flaws to gain knowledge of the encrypted return instructions without knowledge of the secret key. Additionally, improvements have been proposed to remove possibilities of binary analysis for attackers.

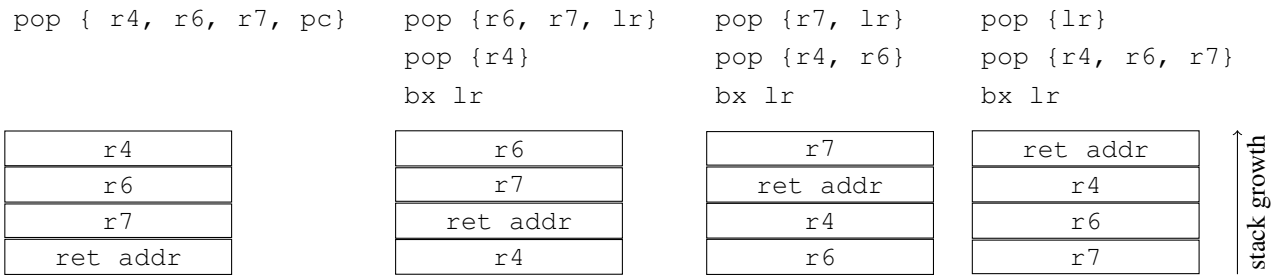

FIGURE 1. **Example return instruction and the corresponding layout in the stack (left) alongside possible replacements with different positions of return addresses**

KAI LEHNIGER received the master's degree in computer science, in 2017. Since 2017 he is with the IHP in Frankfurt (Oder). There, he is working in the wireless systems department as a scientist. He has published more than 10 papers. He currently is interested in efficient security for resource constrained devices.

PETER LANGENDÖRFER received the diploma, in 1995, and the doctorate degree in computer science, in 2001. Since 2000 he is with the IHP in Frankfurt (Oder). There, he is leading the Wireless Systems Department. From 2012 till 2020 he was leading the chair for security in pervasive systems with the Technical University of Cottbus-Senftenberg. Since 2020 he owns the chair wireless systems with the Technical University of Cottbus-Senftenberg. He has published more than 150 refereed technical articles, filed 17 patents of which 11 have been granted already. He is associate editor of IEEE Access, IEEE Internet of Things, Peer-to-Peer Networking and worked as guest editor for many renowned journals e.g., Wireless Communications and Mobile Computing (Wiley) and ACM Transactions on Internet Technology. He is highly interested in security for resource constraint devices, low power protocols, efficient implementations of AI means and resilience. He is member of the "Gesellschaft für Informatik."