IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 28 January 2025, accepted 29 April 2025, date of publication 9 May 2025, date of current version 29 May 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3568598

COMMENTS AND CORRECTIONS

Comment on "RIO: Return Instruction Obfuscation for Bare-Metal IoT Devices"

KAI LEHNIGER AND PETER LANGENDÖRFER
IHP—Leibniz-Institut für Innovative Mikroelektronik, 15236 Frankfurt (Oder), Germany
Brandenburgische Technische Universität Cottbus–Senftenberg, 03046 Cottbus, Germany
Corresponding author: Kai Lehniger (kai.lehniger@b-tu.de)

ABSTRACT This is a comment on "RIO: Return Instruction Obfuscation for Bare-Metal IoT Devices." 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

In the above article [1], the authors propose a novel protection against Return-Oriented Programming (ROP) [2] attacks. 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.

The idea of Return Instruction Obfuscation (RIO) is to prevent ROP attacks by removing the ability of finding gadgets. This is being done by encrypting return instructions in the binary. Without gadgets, no ROP attack can be performed. This article uncovers several flaws with this idea and its implementation. A method is described which allows to recover the unencrypted return instruction with high precision without the need of breaking the encryption. Afterwards, possible improvements of RIO 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.

The associate editor coordinating the review of this manuscript and approving it for publication was Jenny Mahoney.

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

The main idea of RIO [1] is based on a threat scenario that an attacker somehow gains access to the binary of an Internet of Things (IoT) device, for example due to physical access or an unencrypted firmware update. Analyzing the binary reveals gadget positions that could be used for ROP attacks, if a buffer overflow vulnerability exists.

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

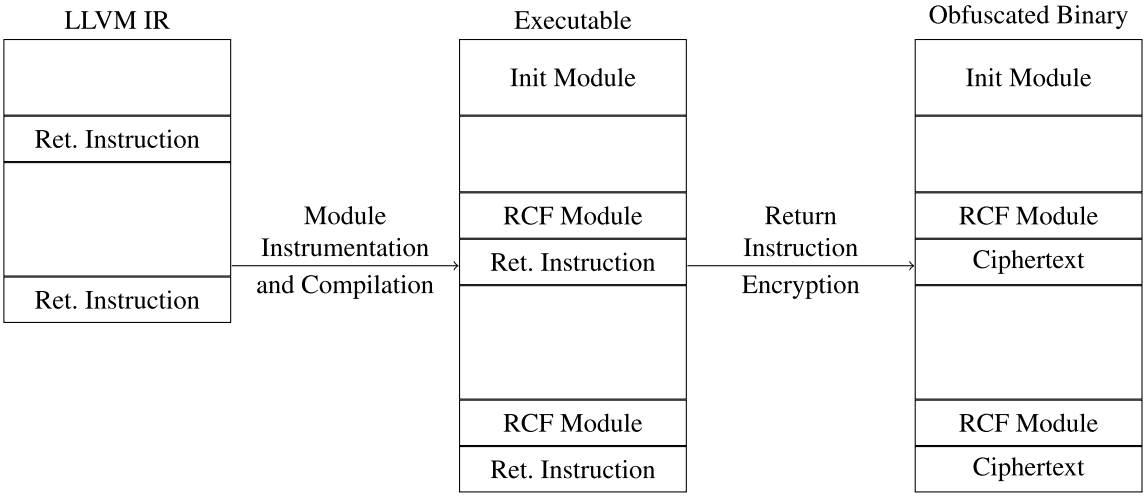

FIGURE 1. RIO firmware binary generation (based on [1]).

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

LISTING 1. Return control flow module code [1].

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. The concept how the LLVM intermediate representation code is modified and compiled into a binary is illustrated in FIGURE 1. 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 [1]. 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 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 [3]. 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. Since each RCF module is placed right in front of an encrypted return instruction, as shown in FIGURE 1, an attacker can simply use the code sequence in LISTING 1 as indicator to find all return instruction positions. 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 [1] that were used to demonstrate the encryption. LISTING 2 shows the `push` and `pop` instructions in the prologue and epilogue of two

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

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

LISTING 2. Symmetrical push and pop instructions of unencrypted functions taken from [1].

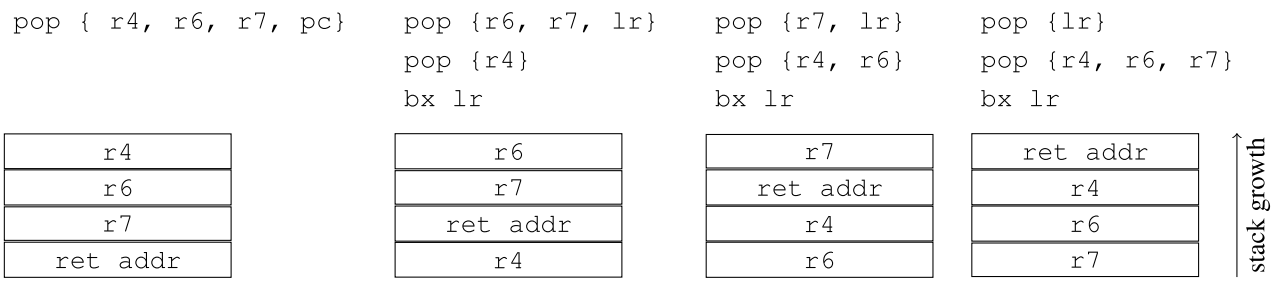

FIGURE 2. Example return instruction and the corresponding layout in the stack (left) alongside possible replacements with different positions of return addresses.

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`.

This flaw of course is specific to the ARM architecture where `pop` instructions act as function returns and can take different form depending on the registers that are being popped. However, the problem is even more severe for other architectures with dedicated return instructions. As soon as the positions of the return instructions are determined, no further analyses for such architectures are required, since the plaintext is already known (the single possible return instruction). While this is not directly a problem for the proposed implementation of RIO, it hinders its applicability to other architectures. However, since ARM is widely used for embedded devices, the authors think a protection mechanism specific to its architecture is still worth investigating.

V. POSSIBLE IMPROVEMENTS

The first flaw is inherent and cannot be targeted, since no amount of binary obfuscation can secure from attacks that do not require access to the binary. The second flaw would require large changes of RIOs design. This is because the encrypted return addresses must be locatable in order to decrypt them, which, no matter how this information is put into the binary, is also accessible to the attacker. Removing this information from the binary and potentially adding it to the secure key storage (if possible) would still require the RCF modules to be removed. This would require a different mechanism to invoke the decrypted return instructions, that leaves no trail in the binary. An alternative to RCF modules could be to use binary rewriting in order to replace encrypted return instructions with their decrypted counterpart once an IoT device is deployed. This approach would be limited to devices where the only point for attackers to collect information of the binary comes before or during the dissemination process. If a deployed device could be analyzed, rewriting the binary is no option. Another idea could be to obfuscate the RCF modules as well. However, if there is a way to obfuscate RCF modules without leaving hints of the RCF module location in the binary, the same approach could probably directly be applied to the return instructions.

However, 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 2 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.

Of course these changes would add more overhead to RIO. While a one-time overhead during the startup phase due to the additional decryption operations might be no concern, the overhead of using decrypted `push` instructions might be significant for IoT devices. While this could already be

true for the original RIO design (with binaries increasing around 29.9% and executing times increasing between 0% and 16.83% [1]), the proposed improvements only add to this problem. While detailed benchmarks would be necessary for exact numbers, it can be estimated that the overhead is at least doubled, due to the fact that for each already encrypted `pop` instruction, a similar overhead is necessary for the corresponding `push` instruction. However, as shown in this paper, RIO does not offer sufficient protection against ROP attacks, therefore changes should be considered necessary.

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.

REFERENCES

[1] B. Kim, K. Lee, W. Park, J. Cho, and B. Lee, "RIO: Return instruction obfuscation for bare-metal IoT devices," *IEEE Access*, vol. 11, pp. 70516–70524, 2023.

[2] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. 14th ACM Conf. Comput. Commun. Secur.*, Oct. 2007, pp. 552–561.

[3] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *Proc. IEEE Symp. Secur. Privacy*, May 2014, pp. 227–242.

KAI LEHNIGER received the master's degree in computer science, in 2017. Since 2017, he has been with the IHP—Leibniz-Institut für Innovative Mikroelektronik, Frankfurt (Oder), where he is currently a Scientist with the Wireless Systems Department. He has published more than ten articles. His research interest includes efficient security for resource constrained devices.

PETER LANGENDÖRFER received the Diploma and Ph.D. degrees in computer science, in 1995 and 2001, respectively. Since 2000, he has been with IHP—Leibniz-Institut für Innovative Mikroelektronik, Frankfurt (Oder), where he is currently leading the Wireless Systems Department. From 2012 to 2020, he was leading the Chair for security in pervasive systems with the Brandenburgische Technische Universität Cottbus–Senftenberg. Since 2020, he has been owning the Chair of Wireless Systems with the Brandenburgische Technische Universität Cottbus–Senftenberg. He is highly interested in security for resource constraint devices, low power protocols, and efficient implementations of AI means and resilience. He has published more than 150 refereed technical articles and filed 17 patents of which 11 have been granted already. He is a member of Gesellschaft für Informatik. He is an Associate Editor of IEEE ACCESS, IEEE INTERNET OF THINGS JOURNAL, and *Peer-to-Peer Networking*, and worked as the Guest Editor of many renowned journals, such as *Wireless Communications and Mobile Computing* (Wiley) and *ACM Transactions on Internet Technology*.