Code Mutation as a mean against ROP Attacks for Embedded Systems

P. Tabatt, J. Jelonek, M. Schölzel
University of Applied Sciences Nordhausen
Nordhausen, Germany

K. Lehniger[+], P. Langendörfer[+*]
[+]IHP, Frankfurt (Oder), Germany
[*]Brandenburg University of Technology, Germany

Abstract—**This paper presents a program-code mutation technique that is applied in-field to embedded systems in order to create diversity in a population of systems that are identical at the time of their deployment. With this diversity, it becomes more difficult for attackers to carry out the very popular Return-Oriented-Programming (ROP) attack in a large scale, since the gadgets in different systems are located at different program addresses after code permutation. In order to prevent the system from a system crash after a failed ROP attack, we further propose the combination of the code mutation with a return address checking. We will report the overhead in time and memory along with a security analysis.**

Keywords-ROP attack, code permutation, embedded systems

I. INTRODUCTION

In the recent years, IoT-devices became more and more present in our daily life, e.g. in the field of smart home or health appliances. Typical for these devices is a wireless interface based on standards like ZigBee or Bluetooth. Therefore, manufacturers of IoT-devices often use off-the-shelve embedded devices with a wireless frontend. The embedded device is responsible for the execution of the protocol stack. Security issues of such devices therefore weaken millions of end-user IoT-devices. Since updating and patching potential security problems in such devices is a challenge in itself, it would be a useful property for these devices to be able to protect themselves against potential attacks. This paper presents an approach inspired by biological systems that use mutation of individuals to create a more diverse population, which is more resilient against pathogens. Our approach creates diversity of devices by randomly mutating the software in-field, creating different variants of the same program. More concrete, the positions of functions inside the memory is changed. We show that this approach can prevent code-reuse attacks and, in combination with a coarse-grained return address check, it is able to detect attacks immediately. The rest of the paper is structured as follows. Section II reviews related work. Section III introduces the concept of permutation in combination with control flow checks. Section IV presents the results.

II. RELATED WORK

Buffer overflows are still one of the most common security risks. They allow injecting code and data into systems, for example into the stack of a program. Modern CPUs can prevent the execution of injected data in non-code segments like the stack. However, attackers can make use of existing code by redirecting the control flow to existing code snippets, creating in this way the desired behavior. One of these code-reuse attacks is Return-Oriented-Programming (ROP). In a vulnerable function, i.e. a function where a buffer overflow can be forced, the return addresses on the stack is overwritten from the attacker with a sequence of addresses to so-called gadgets. A gadget is a short code sequence in the program memory that ends with a return. The regular return of the vulnerable function then returns to the first gadget, whose return will pick-up the next gadget address from the stack to continue the chain. This allows an attacker to execute a sequence of gadgets in an arbitrary order to accomplish his goals.

Modern operating systems are able to increase the difficulty for those attacks by implementing several countermeasures. Address-Space-Layout-Randomization (ASLR) [1] is a coarse-grained randomization of the segment positions in the address space of a process and is applied during the start of a process. This technique is usually not applied in tiny embedded systems, because processes are not created dynamically there. Stack canaries [2] put a guard word on the stack in between local variables that are vulnerable for a buffer flow and the return address, to be able to detect potential buffer overflows. Detours for this protection mechanism try to guess the guard word by running brute-force attacks. A third countermeasure is based on the control flow integrity approach [3] that tries to limit the control flow of a program to the actual intended control flow, either by comparing it dynamically against a pre-computed model of the control flow, or by applying a set of plausibility checks at run time. As gadget addresses are often not a legal branch target, a control flow check can limit an attacker's exploitation space. One of the rules that can be checked in such approaches is that a function must return to a position in the program code that is preceded by a call instruction. Various techniques were published to accomplish this goal. Pre-computed tables are used in [4] to store legal return addresses. Return instructions are then re-written by instructions indexing this table for double-check. In [5] and [6] it is directly checked that the return address is preceded by the opcode of a call instruction. With these checks, it is possible to detect an attack immediately without branching to wrong addresses, which can

This work was supported by the Federal Ministry of Education and Research (BMBF) under research grant number 01IS18065E.

978-1-6654-6828-2/22/$31.00 ©2022 IEEE

2022 11th Mediterranean Conference on Embedded Computing (MECO) | 978-1-6654-6828-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/MECO55406.2022.9797185

prevent the system from a crash. The drawback of these approaches is the introduced run time overhead and a rather weak security, because each call-instruction represent a legal return address for all returns in the program. A fourth countermeasure is the obfuscation of gadet positions by XORing return addresses [8] or even data [9] with randomly generated keys. I.e., before the return address of a sub-routine is pushed onto the stack it is XORed with a secret key that is generated randomly during system startup. Before returning from a sub-routine the return address on the stack is XORed again with same key. This requires an attacker to inject a sequence of gadget addresses that are XORed with the current secret key. Similar to the stack canaries, the detour for this method is a brute force attack that guesses the current secret key. Since this approach obfuscates the gadget positions by XORing their addresses, it is comparable to our approach that obfuscates the gadget positions by moving the functions in the program memory. For that reason we will compare later our technique with the on from [8].

III. SOFTWARE PERMUTATION WITH CONTROL FLOW CHECKS

A. Code Permutation

The knowledge of the gadget positions is vital for a successful attack. Our approach removes this knowledge, as the functions in the program code are permutated randomly. A similar technique was already presented for the MSP430 [13] to repair aging faults in the program memory. Applying this permutation randomly but regularly has the effect that a group of homogenous devices becomes more and more diverse over time, consequently making ROP attacks harder. Even if the attacker manages to compromise one device and leak the gadget positions, the attack will only work for this single device, as the gadget positions are different for each device. Fig. 1 shows this concept. The left side shows the program memory before a permutation, with known gadget positions for the attacker. The right side shows the same memory area after the permutation, where the same three functions are now located at different addresses. When the attacker tries to jump to the known gadget positions on the left side, different code than the intended one is executed, as it can be seen on the right side.

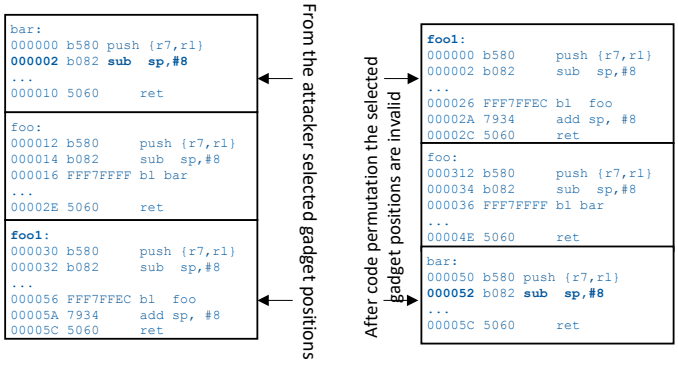

Fig. 1. Attack prevention with code permutation

In order to carry out this permutation in-field, the original program on the microcontroller must be extended by:

- A ***relocation function*** that moves a single function in the program memory to a new position, and
- additional ***relocation information*** about start position and length of each function, as well as the positions of all call-instructions, since these call-instructions must be fixed, if the called functions is moved.

We have implemented the relocation function in C for ARM-based systems. Using the *flash-erase* and *flash-write* functions from the vendor-library, we can pick randomly a single function in the program memory, select a new free position in the program memory, copy the function into the RAM, update all calls inside this function according to the selected new position, and copy the function to the new position. Afterwards, all function calls in the remaining program, pointing to the selected function, are updated, too, and the old position of the function is released. This relocation of a single function is executed periodically at runtime leading to a permutation of the program code. Consequently, an attacker cannot have any knowledge about gadget positions, which makes attacks much more difficult.

However, this permutation may be detoured from attackers, by brute forcing the new gadget positions. Brute forcing gadget positions is a valid approach to determine the positions of gadgets [14, 15] and is also used to detour stack canaries and XOR-based obfuscation. This means the attacker runs the attack by guessing gadget positions, which will succeed after many trials, but also results in lots of program crashes, since most of the trials will jump to random code locations, causing unpredicted behavior, as depicted on the right side in Fig. 1. In tiny embedded systems, without memory protection and operating system, such a crash is usually not handled. With some good luck, a watchdog timer may restart the system, but, we will now present a reliable solution to detect failed trials in a reliable manner and prevent the system from a crash.

B. Return Address Check

We propose to combine the code permutation approach with a coarse-grained control flow integrity check based on the technique presented in [3]. Before returning from a sub-routine we check that the destination instruction is preceded by a function call, i.e. a *bl*- or a *blx*-instruction on ARM processors. Fig. 2 shows the instrumentation of the original program to integrate this check. It is done by rewriting all function returns with a jump to a ***check-routine***. In general, some registers are restored from the stack by a return instruction, including the PC, which causes the return to the calling function. The different ways, how the return from a sub-routine is implemented by the ARM-compiler, are listed in the left column of TABLE I. The right column shows the rewritten code that performs the following steps:

1. All registers but the pc are restored as before.
2. If not already done, the return address is stored in the *lr* register.

3. A jump to the check-routine *_cfi_check_ra* is executed. This check routine verifies that right before the return address stored in *lr*-register a call instruction can be found.

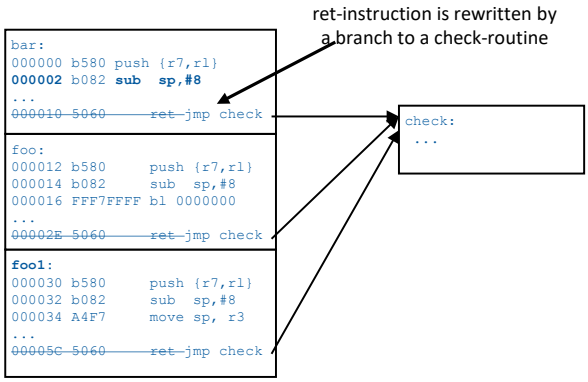

Fig. 2. Code instrumentation for control flow check

TABLE 1
INSTRUMENTATION OF DIFFERENT RETURN INSTRUCTIONS

Before instrumentation	After instrumentation
`bx lr`	`b _cfi_check_ra`
`pop {r3, r5, r6, pc}`	`pop {r3, r5, r6, lr}` `b _cfi_check_ra`
`ldmia.w sp!, {r7, r8, pc}`	`ldmia.w sp!, {r7, r8, lr}` `b _cfi_check_ra`
`ldr.w pc, [sp], #4`	`ldr.w lr, [sp], #4` `b _cfi_check_ra`

Listing 1 shows the check-routine. Before doing the actual verification of the return address, it is checked that *lr* actually points into the code section. This is done to prevent the check-routine itself from causing an exception. The check-routine itself must check for two different instructions, *bl* and *blx*, which can both be used to call a function. Due to the limitation of space, only the check-code for the *bl*-instruction is shown in Listing 1, beginning in line 7.

Listing 1. CHECK-ROUTINE FOR LABEL-CHECK

```
 _cfi_check_ra:
1    push  {r3, r4, r5} @ Save scratch registers
2    mov   r3, #0xFFFF  @ Upper code space address
3    movt  r3, #0x3FFF
4    cmp   lr, r3
5    it    hs           @ Unsigned higher or same
6    bhs   exception_handler

7    mov   r3, lr       @ Copy LR address to r3
8    ldr   r3, [r4,#-5] @ Load opcode of BL
9    mov   r4, #0xF800  @ Building 32-bit AND mask
10   movt  r4, #0xD000
11   and   r5, r3, r4   @ Apply mask on opcode

12   mov   r4, #0xF000  @ Compare with BL label
13   movt  r4, #0xD000
14   cmp   r5, r4       @ Compare r5 with signature
15   itt   eq           @ ARM conditional block
16   popeq {r3, r4, r5}
17   bxeq  lr
18   movt  r4,#0x8000   @ 10 more instructions are
     ...                @ needed for verifying the
27   popeq {r3, r4, r5} @ the BLX-opcode
28   bxeq  lr           @
 exception_handler:
 ...
```

If now an attacker performs a brute force attack, he injects in the vulnerable function randomly chosen return addresses into the stack. If the vulnerable function returns, it executes the presented check-routine that will detect very likely that the return address is not preceded by a function call. Hence, the permutation along with the return address checking provides a reliable mechanism for preventing and detecting ROP attacks. Please note, that moving an instrumented function is still possible with the described relocation function just by additionally adapting the branch to the *_cfi_check_ra*-routine.

IV. EVALUATION

The presented techniques were implemented for an ARM-processor. In order to provide the relocation information, we can automatically derive them from the ELF-object files, since these are the information used by the linker in the compiler tool chain. The relocation function itself is provided in a separate source-code file, which is compiled and linked with the original application. The instrumentation of the code for return address checking is done at assembler code level. I.e., we compile the application to assembler code, perform instrumentation there, and run the assembler and linker steps afterwards. In this way, the original application can be instrumented automatically.

The overhead of the code instrumentation and the additional routines for code-mutation is depicted in TABLE 2. The binary overhead is calculated in bytes (B), while the runtime overhead is calculated in additional instructions (I) that need to be executed or in milliseconds (ms). In most cases these numbers depend on the number of instrumented or executed functions (/F).

TABLE 2
OVERHEAD OF THE CODE INSTRUMENTATION

		Binary overhead	Runtime overhead
Return check instrumentation	**Bx**	2 B/F	0 I/F
	Pop	4 B/F	1 I/F
	Ldmia	4 B/F	1 I/F
	ldr.w	4 B/F	1 I/F
	Check routine	82 B	28 I/F
Relocation routine		1458 B	~260 ms / F

It can be noted that the runtime overhead introduced by the instrumentation for each function is in total 29 instructions. This can be more than 100% for very short functions, but also much less than 1% for long running functions that execute loops for example. The runtime overhead for moving a single function that fits into a single flash-page is about 260 ms, which is the case for most functions. This overhead occurs in the system only sporadically (e.g. once every hour), and it is therefore acceptable from our side.

A. Security Discussion

Like all obfuscation approaches, the security of the presented method is probabilistic. The gadget addresses are obfuscated by permutating the functions positions in the memory. A similar

obfuscation is achieved with the approach presented in [8],where the return address is XORed it with a randomly generated key. Most approaches have randomization built-in at load time, meaning new random values for keys are generated whenever the process/system is started, ensuring new randomization after every failed attack attempt, assuming the process crashed due to side-effects of the attack. With that assumption, the success of brute-force attacks depends on the probability of guessing correctly the current randomized value. We will compare our approach with the approach in [8], since both techniques perform an obfuscation. In both approaches, the attacker has to either brute-force, or leak the necessary information to overcome the defense measure. We will discuss the probability of brute-forcing the right gadget positions. For the XOR-approach the attacker must guess a 32-bit value, that is used for XOR-operation on ARM processors. With the right XOR-value, the ROP-attack will run perfectly, no matter how many gadgets g are used in the gadget chain. Hence, the probability of guessing the right value is $1/2^{32}$. The exact probability in our approach depends on many parameters like size of the program memory, function size, and number of needed gadgets. Let p denote the size of the program memory in bytes. Since only even addresses denote a legal instruction position, the possible number of start addresses of a function of size f is $(p - f)/2$, and the probability of guessing a gadget address in this function right is

$$\frac{1}{(p-f)/2}$$

Since p is usually much larger than f, we will neglect for simplicity the size of a function, and just assume each gadget can be located at any of the $p/2$ possible positions. If an attacker needs to guess the position of g many gadgets, the probability of guessing all the gadget positions right is:

$$\frac{1}{(size/2)^g}$$

Considering a 1MB flash memory, as in in our ARM processor, we have the result that for $g > 1$ the probability of a successful ROP attack is much lower in our approach than by XORing the return address. If a single gadget is sufficient, then XORing gives the lower probability. Another benefit of our approach comes from the fact that the obfuscation of the gadget positions does not require additional runtime during the execution of the normal application, since we do not need to instrument the code with XOR-operations. However, in regular periods we have to move functions, which also takes time and consumes power. Moreover, by instrumenting the code with the return address check, we can even prevent the program from crashing, which is not possible with the XOR-approach.

V. CONCLUSION

We have presented a code mutation technique for embedded systems, and demonstrated their usage for protecting a system against ROP attacks. This technique was inspired by biological systems that also use mutations to create diversity, which makes them more robust. For gadget chains longer than 1, our technique provides better protection against ROP-attack than comparable approaches that use rather small secret keys for obfuscation. Additionally we presented the combination of the code mutation with return address checking. This combination allows the early detection of ongoing ROP attacks, and makes the system very robust against this kind of attack.

REFERENCES

[1] PaX Team, *Pax address space layout randomization (aslr).* [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt

[2] C. Cowan *et al.,* "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks," in *USENIX security symposium*, 1998, pp. 63–78.

[3] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-Flow Integrity: Principles, Implementations, and Applications," in *ACM Transactions on Information and System Security (TISSEC)*, pp. 1–40.

[4] J. Li, Z. Wang, T. Bletsch, D. Srinivasan, M. Grace, and X. Jiang, "Comprehensive and Efficient Protection of Kernel Control Data," *IEEE Trans.Inform.Forensic Secur.*, vol. 6, no. 4, pp. 1404–1417, 2011, doi: 10.1109/tifs.2011.2159712.

[5] *ROPGuard: Runtime prevention of return-oriented programming attacks*, 2012. [Online]. Available: https://www.ieee.hr/_download/repository/ivan_fratric.pdf

[6] *ROPecker: A generic and practical approach for defending against ROP attack*, 2014. [Online]. Available: https://ink.library.smu.edu.sg/sis_research/1973/

[7] *kBouncer: Efficient and transparent ROP mitigation*, 2012. [Online]. Available: https://people.csail.mit.edu/hes/rop/readings/kbouncer.pdf

[8] *{PointGuard™}: Protecting Pointers from Buffer Overflow Vulnerabilities*, 2003. [Online]. Available: https://www.usenix.org/event/sec03/tech/full_papers/cowan/cowan_html/

[9] *Data randomization*, 2008. [Online]. Available: http://www.doc.ic.ac.uk/~cristic/papers/data-rand-msr-tr-2008-120.pdf

[10] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," in *Proceedings of the 10th ACM conference on Computer and communication security - CCS '03*, New York, New York, USA, 2003. [Online]. Available: http://dx.doi.org/10.1145/948109.948147

[11] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proceedings of the 10th ACM conference on Computer and communication security - CCS '03*, New York, New York, USA, 2003. [Online]. Available: http://dx.doi.org/10.1145/948109.948146

[12] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits," in *USENIX Security Symposium*, 2003, pp. 291–301.

[13] F. Muhlbauer, L. Schroder, P. Skoncej, and M. Scholzel, "Handling manufacturing and aging faults with software-based techniques in tiny embedded systems," in *2017 18th IEEE Latin American Test Symposium (LATS)*, 2017. [Online]. Available: http://dx.doi.org/10.1109/latw.2017.7906756

[14] *Half-blind attacks: mask ROM bootloaders are dangerous*, 2009. [Online]. Available: https://www.usenix.org/event/woot09/tech/full_papers/goodspeed.pdf

[15] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh, "Hacking Blind," in *Proceedings, 2014 IEEE Symposium on Security and Privacy: SP 2014 : 18-21 May 2014, San Jose, California, USA*, San Jose, CA, 2014, pp. 227–242.