BackFlow: Backward Edge Control Flow Enforcement for Low End ARM Microcontrollers

Cyril Bresch
Univ. Grenoble Alpes
Grenoble INP LCIS
F-26000 Valence France
cyril.bresch@lcis.grenoble-inp.fr

Roman Lysecky
Dept of Electrical and Computer Engineering
University of Arizona
USA
rlysecky@ece.arizona.edu

David Hély
Univ. Grenoble Alpes
Grenoble INP LCIS
F-26000 Valence France
david.hely@lcis.grenoble-inp.fr

***Abstract*—This paper presents BackFlow, a compiler-based toolchain that enforces indirect backward edge control flow integrity for low-end ARM Cortex-M microprocessors. BackFlow is implemented within the Clang/LLVM compiler and supports the ARM instruction set and its subset Thumb. The control flow integrity generated by the compiler relies on a bitmap, where each set bit indicates a valid pointer destination. The efficiency of the framework is benchmarked using an STM32 NUCLEO F446RE microcontroller. The obtained results show that the control flow integrity solution incurs an execution time overhead ranging from 1.5 to 4.5%.**

Keywords—CFI, Memory Safety, Compiler

I. Introduction

Memory corruptions are one of the most common weaknesses in embedded systems. Their exploitation gives an attacker the ability to overwrite control flow information and hijack the execution flow of an application [1].

Many countermeasures following the *control flow integrity* (CFI) [2] principles have been proposed for the ARM architecture. However, only a few of them are well-suited for low-end ARM Cortex-M microcontrollers. Indeed, these tiny processing systems are very constrained. Most of the time, they do not benefit from common countermeasures such as the ASLR [3], non-executable stacks (NX), or stack canaries [4]. In addition, existing ARM control flow enforcement countermeasures, such as MoCFI [5], CFLAT[6], PAC [7] either incur high overhead, require an operating system support or rely on hardware support which is not available for ARM Cortex-M systems.

In this paper, we present BackFlow, an LLVM [8] compiler extension that enforces backward edge integrity. BackFlow focuses on backward edges because they are more frequent in embedded systems than forward edges. In addition, forward edges can be protected by other software extension such as [9]. BackFlow operates at the software level, produces portable code, and can target any ARM Cortex-M microprocessor. The contributions of this work are then:

- BackFlow: an LLVM extension which provides two efficient bitmap-based protections to ensure CFI;
- A benchmark of the code generated by the compiler using an off-the-shelf STM32 NUCLEO F446RE microcontroller.

This paper is organized as follows. In Section 2, the BackFlow concept is presented. Section 3 details the bitmap implementation within the Clang/LLVM compiler. Then, Section 4 initiates a security discussion. Finally, BackFlow is benchmarked in Section 5.

II. BackFlow

A. Concept

In order to enforce indirect backward edges, BackFlow proposes a bitmap-based CFI [10]. During the execution of an application, a bitmap is kept in memory where each set bits refers to a valid control flow transfer address destination. Consequently, before each protected indirect control flow transfer, the bitmap is checked in order to certify that the destination address is valid.

Following this principle, BackFlow protects backward edges using a dynamic bitmap that is updated over the time. The primary approach is displayed on Figure 1. When vertex 1 calls vertex 2, the index corresponding to the return address pointer of vertex 1 is set in the bitmap. Then, when vertex 2 returns, the return address pointer is checked using its index in the bitmap. Finally, the index is unset.

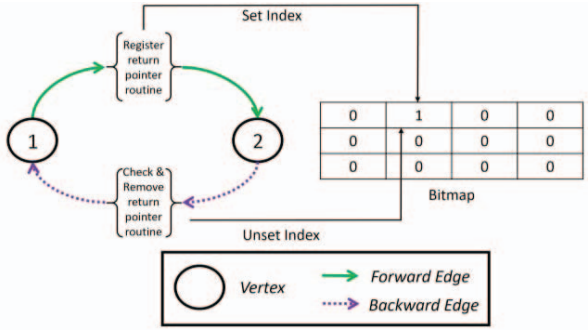

Fig. 1. Bitmap Principle.

B. Security Challenges

The previously presented BackFlows' dynamic bitmap suffers from minor limitations. First, the concept is not compatible with recursion. Indeed, a recursive function may call itself more than twice. As a result, after the first return, the function return pointer's corresponding bit in the bitmap is unset. Second, as the bitmap is updated over the time, it resides in a volatile memory as the unsafe stack. Although the bitmap section can be placed in an unattainable area below the stack, some arbitrary write vulnerabilities could reach the bitmap. Even though these cases are becoming rare, they still have an impact on the security of the concept. To address this, we propose a second relaxed approach based on a static bitmap like [10]. Using this approach, a static bitmap including all return targets is placed in a read-only memory. As a consequence, the static bitmap cannot be tampered by an arbitrary write in memory.

While the two approaches provide different security features, it can be noted that they are both coarse-grained. Indeed, an indirect backward edge can target every activated location in the bitmap. As coarse-grained control flow integrity is vulnerable to call-preceded gadget attacks [11], we propose to randomize the location of each function as well as basic blocks in a program during code generation. Thus, a working exploit for an application A able to bypass the bitmap would not be efficient on A' a mutant instance of A deployed on another device. While this solution imposes recompilation of an application before being deployed, it mitigates the efficiency of deployable call-preceded gadget exploits.

C. Implementation Challenges

Before describing the implementation challenges imposed by BackFlow, we note several essential points concerning the ARM ISA and its calling convention [12]. Both "*bl*" (branch and link) and "*blx*" (branch and link exchange) instructions are used to call a subroutine and put the caller return address pointer in the link register (LR). Furthermore, in a program, each function has a prologue and an epilogue. Function prologues and epilogues have the effect of setting/destroying a new stack frame and spilling/restoring the caller registers. To ensure a function return, we denote two types of instruction patterns. If the return address is previously spilled from the link register to a stack frame during the prologue, the ARM compiler uses a multiple load instruction to both place the return address in the program counter register and restore the spilled registers. Conversely, if a return address handled in the link register is never spilled, the ARM compiler uses a *"bx"* instruction to branch to the return destination target.

From the call procedure standard, it follows that every function saving the link register in a stack frame should be secured using the bitmap. On the other hand, functions that keep return addresses in the link register are not vulnerable to memory attacks. Hence, to protect functions that spill the link register, one challenge is to register the link register in the bitmap in the prologue and check its integrity using the bitmap during the epilogue.

The challenges concerning the static bitmap implementation are similar to the dynamic bitmap ones for the epilogue but not the prologue. Unlike the dynamic bitmap, the functions that require protection cannot register their link register in a read only bitmap. Consequently, one hardship is to statically initialize the bitmap by analyzing all the return address sites after compilation. Finally, the specificity of embedded system programming is the use of interrupts. An interrupt handler can be called from everywhere at runtime leading to a context switch. When an interrupt terminates, the execution flow is restored at the interruption point. Following this principle, it is impossible to statically determine the return destination of such interruptions making them impossible to secure using the static bitmap. To address this issue, one solution would be to prevent the compiler from securing interruption/exception handler when using the static bitmap.

III. Implementation

BackFlow is a C/C++ compiler based on the Clang/LLVM [8] framework which includes the following changes:

- **ARM Backend:** Calling convention changes to ensure that the compiler registers/checks the link register in/using the bitmap each time it is spilled by a function.
- **Static Code Instrumentation:** Implementation of a custom high-level directive handler allowing the compiler's user to annotate specific routines that should not be protected by the bitmap.
- **Function Randomization:** A high-level intermediate representation pass that randomizes functions and basic blocks during compilation.

A. ARM Backend

To ensure the new calling convention supports the bitmap, two custom machine passes and a modified prologue/epilogue inserter feature are introduced in backend pipeline. The resulting code generated by the compiler for the dynamic protection is displayed on the Code Snippet 1.

Code Snippet 1: BackFlow dynamic bitmap code

```
push    {r4, r7,lr}                 ; Saving R4
mov     r4, lr                      ; Moving LR to R4
bl      <__prologue_map_table__>    ; Saving LR in the bitmap
add     r7, sp, #4
...                                 ; assembly code
ldr     r4, [sp, #8]                ; Retrieving LR in the stack
bl      <__bitmap_chk__>            ; Checking LR in the bitmap
pop     {r4, r7, pc}
```

Both the "__prologue_map_table__" and "__bitmap_chk__" routines are a couple of instructions that

register/check the link register in the bitmap using R4 as an argument. After code generation, both approaches have a zero initialized bitmap in memory. While the bitmap of the dynamic approach resides in the read and write section, the bitmap of the static approach is located in the read only section. Thus, the static bitmap indices need to be statically initialized with all valid return address target destinations. To do so, we implemented a post compilation binary patching tool that performs the following tasks:

- Find the location of the zero initialized bitmap in the binary.
- Disassemble the entire binary and find each return address destination of called functions.
- Patch the bitmap index.

B. High Level LLVM passes

In order to hardened the bitmap security, our compiler implements two intermediate representation passes which randomly permute functions and basic blocks. As a consequence, each final binary generated by the compiler has a different bitmap preventing attacks like [11].

Finally, in order to address the static bitmap's interrupt/exception limitation, we implemented a custom directive in the clang front-end. The latter allows developers to annotate functions that should not be protected by the compiler. It's worth mentioning that this custom directive can also be used to overcome the recursive function limitation of the dynamic bitmap.

IV. Security Discussion

This section conducts a security evaluation of BackFlow using SecPump [13], a functional connected open source infusion pump system dedicated to security experiments. As previously explained in this paper, BackFlow only protects indirect backward edges leaving indirect forward edges unprotected. We consider indirect forward edges as a low security risk. Indeed, indirect function calls using pointers are rare in the context of embedded systems. As an example, on the compiled SecPump application we measure that indirect forward edges represent less than 3.5% of indirect branches and less than 1.1% of the total branches. Thus, the likelihood of a working indirect forward edge exploit can be considered low.

Both BackFlow bitmap protections are coarse grained. However, the dynamic concept is more accurate than the static one. Indeed, when a function returns using the dynamic protection, it can only target a valid index in the bitmap that corresponds to all activated return function pointers. Applied to SecPump, we measured that up to 35 return addresses can be active at the same time in the bitmap. In contrast, with the static protection, each function can return to all valid destinations initialized in the bitmap. Applied to SecPump, we measured that 1211 destinations are valid when a function returns. In addition, the static protection does not protect interrupts, leaving them vulnerable. This difference in accuracy shows that the static protection is less robust at mitigating call-preceded gadget attacks than the dynamic protection.

On the other hand, the static bitmap could be considered safer than the dynamic bitmap. Indeed, the latter is statically initialized in a read only memory making it protected from arbitrary writes. Conversely, even if the dynamic bitmap is placed in an unattainable memory area, certain types of buffer overruns may be able to reach and corrupt it. The use of one concept in relation to the other therefore depends on a tradeoff between accuracy or integrity of the bitmap.

Finally, we denote the simplicity with which the proposed CFI can be used. The implementation of a software CFI is not straightforward for non-security experts. With the proposed method, the software security is now managed at the compiler level. The latter makes the security modular over all ARM platform and easy to use.

V. Experimental Results

This section evaluates the static overhead induced by the bitmap protection on an application such as SecPump. Then, the execution time overhead due to the bitmaps is measured using an off-the-shelf STM32 NUCLEO F446RE microcontroller running SecPump and some applications from the MiBench benchmark suite [14].

A. Static Benchmarking with Secpump

After compilation, we determined that the dynamic bitmap increases the total number of instructions by 3.14%. Likewise, the instruction overhead measured for the static bitmap is 1.53%. Indeed, the static implementation requires less instructions per function than the dynamic one.

Despite the optimization of the bitmap, which restricts its indexes to executable memory ranges, its size in memory remains significant. Indeed, for both approaches, one bit in the bitmap refers to one 32-bit pointer in memory. Hence, for any application the size of the bitmap in bytes can be computed as follows:

$$bitmap\ size\ in\ bytes = \frac{size\ of\ the\ executable\ memory\ in\ bytes}{32}$$

As a result, for any application, the size of the bitmap will always require 3.1% of the total size of the available executable memory. Of course, for each protection, it follows that the available space for an application code or stack is reduced. However, we consider that its memory cost is low compared to the security provided. In addition, the size of the bitmap can be easily integrated into the development of an application without too much restriction.

B. Execution Time Overhead Benchmarking

In order to measure the execution time overhead induced by the presented solutions, we use the cycle counter register of

the ARM Cortex-M4 processor. We measured the execution time overhead induced during both the reception of a BLE packet and the transmission of a BLE packet for the SecPump. To extend our evaluation, we also selected some relevant application among the MiBench benchmark suite and adapted them to the STM32 board target.

The results of our measurements are displayed in Figure 2. The average execution time overhead is below 5%. Consequently, both implementations seem to be suitable for tiny embedded applications.

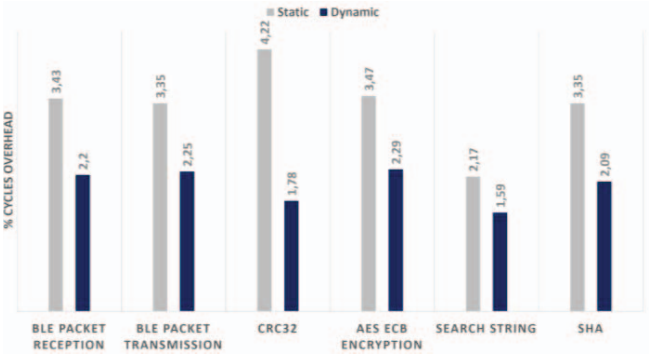

Fig. 2. Execution Time Overhead.

The results revealed that the dynamic bitmap approach is faster than the static approach. This is due to the location of the bitmaps in memory. The dynamic bitmap is included within in a custom "*bdata*" section, which resides in the SRAM memory of the microcontroller. In contrast, the static bitmap is protected in the "*rodata*" section of the application, which is located in the flash memory. As the access to SRAM memory is much faster than flash, the dynamic solution seems to be the more optimal implementation for runtime performance.

VI. Conclusions

This paper introduced BackFlow, a novel compiler based control flow integrity defense for low-end ARM Cortex-M microcontrollers. BackFlow is purely implemented in software, making it an immediately deployable solution for ARM targets. BackFlow relies on a bitmap to keep track of the valid return address pointers at execution time. The experimental results revealed that the execution time overhead induced by the bitmap checks is less than 5%. Consequently, our defense is an acceptable solution for embedded application.

Acknowledgment

This work is carried out under the SERENE-IoT project, a project labelled within the framework of PENTA, the EUREKA cluster for Application and Technology Research in Europe on NanoElectronics.

This work is supported by the French National Research Agency in the framework of the "investissement d'avenir" program (ANR-15-IDEX-02)

This research was partially supported by the National Science Foundation under Grant CNS-1615890.

References

[1] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," *Proc. 17th ACM Conf. Comput. Commun. Secur.*, p. 559, 2010.

[2] M. Abadi, "Control-Flow Integrity Principles, Implementations, Applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, p. 4, 2009.

[3] 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," *Proc. - IEEE Symp. Secur. Priv.*, pp. 574–588, 2013.

[4] P. Wagle and C. Cowa, "Stackguard: Simple stack smash protection for gcc," *Proc. GCC Dev. Summit*, pp. 243–255, 2003.

[5] L. Davi *et al.*, "MoCFI: A framework to mitigate control-flow attacks on smartphones," *Symp. Netw. Distrib. Syst. Secur.*, 2012.

[6] T. Abera *et al.*, "C-FLAT: Control-FLow ATtestation for Embedded Systems Software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 743–754.

[7] H. Liljestrand, T. Nyman, K. Wang, and C. C. Perez, "PAC it up : Towards Pointer Integrity using ARM Pointer Authentication ∗," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 177–194.

[8] "LLVM." [Online]. Available: https://llvm.org/.

[9] C. Tice, T. Roeder, P. Collingbourne, L. Lozano, S. Checkoway, and G. Pike, "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 941–955.

[10] Microsoft, "Control Flow Guard."

[11] N. Carlini, D. Wagner, and N. Carlini, "ROP is Still Dangerous : Breaking Modern Defenses ROP is Still Dangerous : Breaking Modern Defenses," *23rd {USENIX} Secur. Symp. ({USENIX} Secur. 14)*, pp. 385–399, 2014.

[12] ARM, "Procedure Call Standard for the ARM Architecture," 2015. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf.

[13] C. Bresch, D. Hély, and S. Chollet, "SecPump," 2017. [Online]. Available: https://github.com/r3glisss/SecPump.

[14] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench : A free , commercially representative embedded benchmark suite The University of Michigan Electrical Engineering and Computer Science," *Proc. Fourth Annu. IEEE Int. Work. Workload Charact. WWC-4 (Cat. No. 01EX538)*, pp. 3–14, 2001.