"Les Vieux bRISCards" final report from CentraleSupélec / université PARIS-SACLAY Rennes

3rd national RISC-V student contest - 2022-2023

Authors :
Majed Abdennadher
Thomas Chevalier
Ulysse Vincenti

Professors :
Guillaume Hiet
Rubén Salvador

Contents

Rennes, May 12, 2023

Abstract : The 3rd national RISC-V student contest is a hackathon sponsored by Thales, the GDR SOC^2, and the CNFM, whose goal is to improve the security of the CV32A6 RISC-V soft-core. The task is to defeat ten memory corruption attacks taken from the RIPE benchmark with minimal performance overhead. The soft-core is geared with Zephyr RTOS, and the contest allows us to modify the CV32A6 core, the OS and/or the compiler.

Our solution is a comprehensive approach to enhancing the security of a system, consisting of three key components: control-flow enforcement using no-op instructions for the forward-egde and a shadow stack for the back-edge, secure heap memory management, and a non-executable stack.

Control-flow enforcement with a shadow stack helps prevent attackers from hijacking a program's control flow and executing arbitrary code. Secure heap memory management, accomplished through secure versions of a subset of standard C library functions, guards against heap pointer read/write overflow attacks. A non-executable stack further strengthens security by marking stack memory as non-executable and therefore preventing shellcode execution in case of stack injection.

Together, these components work in harmony to provide a robust security solution that addresses a wide range of potential vulnerabilities.

With this implementation, we are able to **counter all the attacks given with a negligible timing overhead**. Eventually, we applied some compilation tricks and tradeoffs to obtain a small performance boost compared to the initial baseline.

I Introduction

The national RISC-V student contest is open to all Master 2 students of a French engineering school or university. Each team can have up to four members and can be coached by teachers or PhD students. For the last 3 years, this contest has proposed hardware challenges to the students; the first year was focused on the FPGA optimization of the core, the second year on power efficiency, and finally this year, **security**. A benchmark of 10 attacks is given to the participants; the goal is to prevent the attacks on a Zephyr RTOS running on the CV32A6. The chosen board is a Zybo-z7 20 which embeds an Artix-7 FPGA. After implementing a solution, neither the memory footprint nor the performance should increase by more than a 100%. The solutions will be evaluated based on their ability to defeat the given set of attacks. In the case of a tie, performance will be used as a tiebreaker, based on the execution time of a provided benchmark.

The CV32A6 is an open-source 6-stage RISC-V 32-bit processor. It implements I, M, and C extensions from the RISC-V instruction set [1]. The goal of this core is to run an OS at a reasonable speed. The chosen core does not have a MMU or PMP, but these features are available on a more recent version of the core.

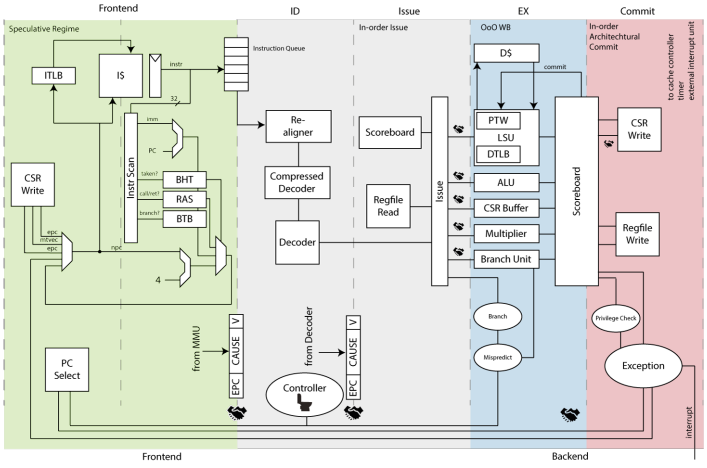

Figure 1: CVA6 core diagram [2]

Zephyr RTOS is an embedded real-time OS that supports a wide variety of microprocessor/microcontrollers architectures. Multiple services and features, like PMP support, single address spaces and stack canaries, are implemented in Zephyr. The OS uses Kconfig and the device tree for its configuration and is compiled through CMake with GNU GCC 12.

The Runtime Intrusion Prevention Evaluator (RIPE) [3] is a security benchmark used to evaluate solutions against buffer overflows attacks. The benchmark was recently ported to RISC-V, and the goal is to protect Zephyr against a selection of 10 attacks.

i	technique	attack code	code ptr	location	function
1	direct	no nop	ret_addr	stack	memcpy
2	direct	no nop	func_ptr	stack	memcpy
3	indirect	no nop	func_ptr	stack	memcpy
4	direct	data	var leak	heap	sprintf
5	direct	ret2libc	ret_addr	stack	memcpy
6	indirect	ret2libc	func_ptr	heap	memcpy
7	indirect	ret2libc	struct_func_ptr	heap	homebrew
8	indirect	ret2libc	longjmp_buf	heap	memcpy
9	direct	rop	ret_addr	stack	memcpy
10	direct	rop	struct_func_ptr	heap	sprintf

Table 1: Selection of RIPE attacks to protect against

I.I Related work

The RIPE benchmark assesses the security of systems against 850 types of buffer overflow attacks through five independent dimensions, which are combined to create over 5000 attack scenarios, 850 of which are practical. However, only a subset of the parameters of the RIPE are taken into account in this competition; we list below the five dimensions with the used parameters:

- Overflow technique: a direct overflow rewrites some variables by sequentially overflowing a buffer, whereas an indirect overflow first overwrites a pointer with the address of the target, then dereferences the pointer and writes the desired value.
- Attack code: the attacks can execute either shellcode without NOP sled, a libc function, or a Return Oriented Programming (ROP) payload.
- Target code pointer: the overflown pointer that will redirect to the attack. It can be the return address, a function pointer, a longjmp buffer, or a function pointer inside a structure that contains the vulnerable buffer.
- Location: the memory location of the vulnerable buffer and data. Either heap or stack.
- Function: the function used to perform the buffer overflow. Memcpy and sprintf are standard libc functions,

and homebrew is an equivalent to memcpy implemented by the vulnerable application.

Notice that, unlike the remainder of the attacks, attack number 4 in Table 1 is not a control-hijacking attack but a data-leak attack, in particular, it is a heap buffer over-read.

The port of the RIPE benchmark to RISC-V and Zephyr RTOS has been simplified in several ways. The libc function used in the ret2libc attack is only a stub function defined in the vulnerable application, and the ROP payload returns to the beginning of a user-defined function. The latter means that it will be more difficult to detect this attack with CFI techniques, as a typical ROP gadget often does not jump at a valid call site. Please note that the RIPE benchmark takes the form of a single binary that attacks itself and thus defeats any countermeasures that rely on the isolation and secrecy of memory locations.

Our solution relies heavily on control flow enforcement, which involves enforcing a set of rules to prevent attackers from hijacking the control flow and jumps of a program. This is typically done by using software and hardware-based protections to ensure that control flow transfers and jumps only occur along permitted paths.

II Implementation

II.I Stack canaries

Stack canaries are a simple and effective way to prevent direct overwriting of the function's return address. A random value is generated at the start of the program and is pushed on the stack for each function that declares a local buffer. When the function returns, the integrity of the canary is verified. A different value from what was stored signals a buffer overflow, which leads to program termination. Implementing this feature in Zephyr only requires changing two lines in the configuration:

```
CONFIG_TEST_RANDOM_GENERATOR=y
CONFIG_STACK_CANARIES=y
```

This technique successfully mitigates attacks 1, 5, and 9 because they directly overwrite the return address. Attacks 2 and 3 do not work because the activation of stack canaries in GCC rearranges the local buffers at the end of the stack frame which is not allowed by the rules (as it erases the vulnerability).

II.II Simple FW & BW control flow

The next technique we implemented is inspired by Intel's CET [4] and consists of defining two custom instructions. The first instruction, "li zero, 2", is used as a landing pad (LP) at the start of a function to enforce the forward edge control flow. The second instruction, "li zero, 1", is used as a LP at the return of a function to enforce the backward edge control flow. By using these custom instructions, we can assert some guarantees about the program's execution. For example, an attacker can no longer jump (call-wise) into the body of a function or return to a different function than intended. By enforcing the forward and backward control flow, we can prevent these types of attacks and improve the overall security of the system.

The result in assembly is:

```
<z_bss_zero>:
li      zero,2
addi    sp,sp,-16
sw      ra,12(sp)
lui     a0,0x8000b
addi    a2,a0,-336 # 8000aeb0
lui     a5,0x8000b
addi    a5,a5,980 # 8000b3d4
sub     a2,a5,a2
li      a1,0
addi    a0,a0,-336
jal     ra,800076e0 # call <z_early_memset>
li      zero,1
lw      ra,12(sp)
addi    sp,sp,16
ret
```

Listing 1: *z_bss_zero* compiled with the landing pads

In RISC-V, the instruction "*nop*" is in reality "*add zero, 0*", which is why we chose the solution of adding the constants 1 and 2 to the zero register. The check of the integrity of the control flow has been implemented in hardware through a simple finite state machine (FSM).

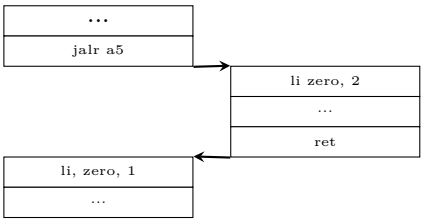

Figure 2: Control Flow Diagram : simple implementation

To further tighten the security of our control flow policy, we sought to identify the function where indirect calls could go and mark them during static analysis using a bit on the first LP. However, static analysis was not feasible due to its complexity and time-consuming nature. As a result, we were inspired by type-based identification and decided to implement an argument-based annotation approach. This was achieved by adding information to the first LP, where the first 2 bits of the immediate were kept unchanged, while the last bit was designated for isVariadic information. The remaining bits of the immediate were used to indicate the number of arguments.

By equipping all functions with this type of annotation, we can inform the hardware (by writing to a custom CSR) that an indirect call is taking place and have it check the argument count to ensure that it matches the function's expected number of arguments. This allows us to detect errors and prevent attackers from exploiting vulnerabilities in the system's control flow.

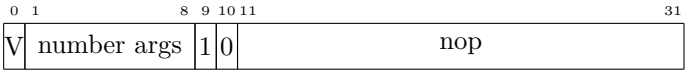

Figure 3: Landing Pad for the FW-CFI

- V : the variadic bit.

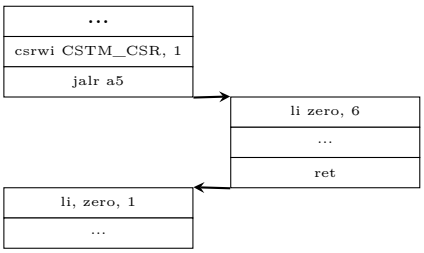

Figure 4: Control Flow Diagram : indirect call

II.III Forward CFI + Shadow Stack

The shadow stack allows us to remove the second LP. A shadow stack is a hardware or software security feature that stores the return addresses of a call in a LIFO secured space. During a return, the return address is checked against the shadow stack return address. If the return address in the shadow stack and the return address in the current flow are not the same, we raise an exception.

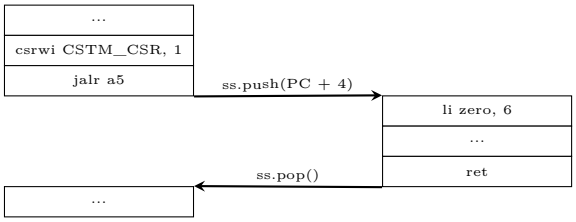

Figure 5: Control Flow Diagram : indirect call with shadow stack

The next three schemes are, in reality, executed in parallel. These schemes were simplified by assuming that there is only one commit per clock cycle. In reality, we can have up to two commits done in one clock cycle, which complicates things. They are implemented through two Finite State Machines (FSM) and two modules, the first for the detection and FSM, the second for the LIFO logic.

We realized later that the real advantage of using a SS is against shell code attacks where the attacker could have used BW-CFI landing pad.

We used an atypical debug method to perform hardware modifications. During the first half of the competition, we did not have access to the simulator, and the ILA components were not functioning correctly. To overcome this, we developed the first elements by linking the outputs to the 9 controllable LEDs of the board as well as emulating the added hardware in QEMU. Once the simulator was up and running, we already had a working solution that only needed verification by the simulator.

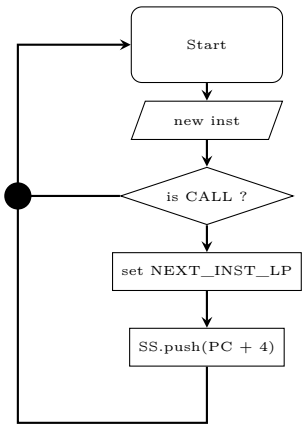

Figure 6: Simplified hardware algorithm implemented in the commit stage (part 1) - detection of calls

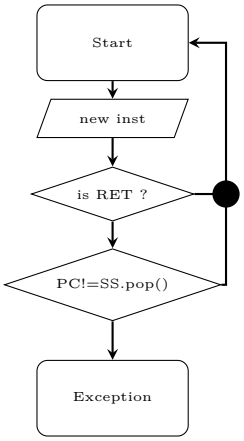

Figure 7: Simplified hardware algorithm implemented in the commit stage (part 2) – SS comparison

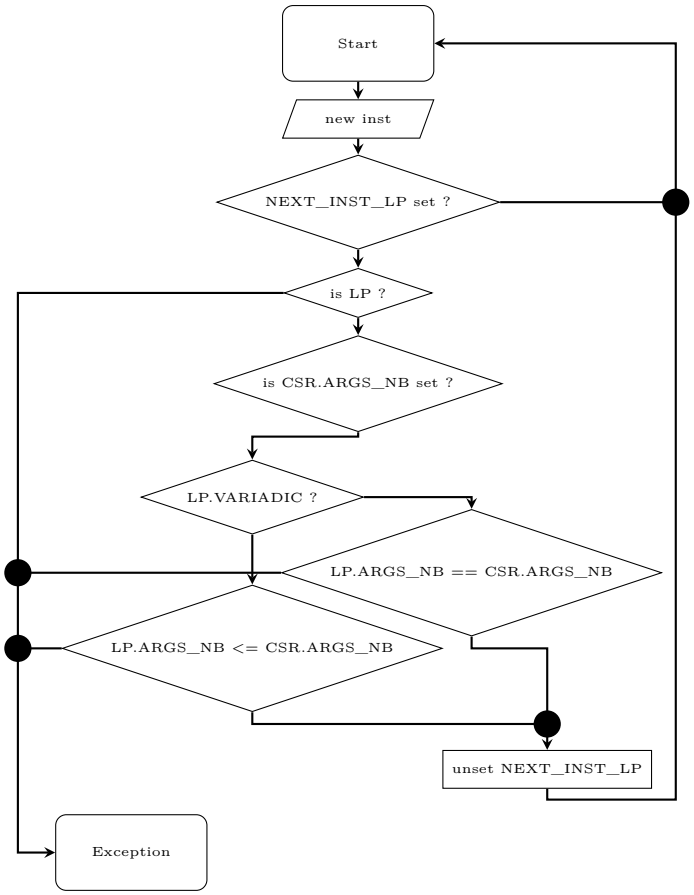

Figure 8: Simplified hardware algorithm implemented in the commit stage (part 3) - landing pad verification

To get better results in the FPGA placement, the SS has a depth of only 100 addresses. When the count of calls gets greater than 100, the SS is deactivated. The SS starts to compare addresses again when the counter goes under 100.

II.IV Heap Protection

Introduction Memory read/write overflow attacks are a common type of vulnerability in C programs. To mitigate this threat, we have modified a subset of common C library standard functions to do an overflow check before committing the results. Our solution provides an added layer of security to existing systems that may not have been designed with buffer overflow protection in mind.

Implementation To integrate this mechanism into the C libraries, we added two weak function hooks: "___ptr_overflow_check___" and "___heap_overflow_check___". The former is a generic hook called by the modified functions to check for buffer overflow, while the latter is provided by the heap implementation to determine if a particular heap chunk is overflowing.

In the case of the simple nano-malloc heap implementation, "___heap_overflow_check___" is a function that maps a pointer to the heap chunk and checks its bounds. This is achieved by using the fact that each heap chunk contains its size and a pointer to the next allocated chunk. By traversing these pointers and comparing them to the requested size, we can determine whether an overflow has occurred.

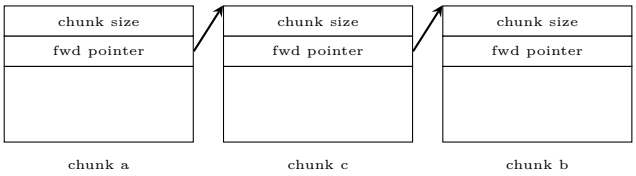

Figure 9: Nano Heap Implementation Scheme

To tailor the solution to our specific needs, we have configured Zephyr to customize "___ptr_overflow_check___" to act only on pointers belonging to the libc heap, then delegating to "___heap_overflow_check___". This approach reduces the protection surface but ensures that we can safeguard the heap without incurring any unnecessary overhead on other parts of the system. This customization is especially relevant for embedded systems that have limited resources, as performance is often a critical factor.

We chose to focus on nano-malloc since it is the default heap implementation used in this competition. However, our solution is not limited to just nano-malloc and can be extended to work with other heap implementations as well.

II.V NX stack protection

Introduction Stack buffer overflow attacks are common attacks that allow an attacker to overwrite the stack frame and execute arbitrary code. A widespread solution to mitigate this attack is the non-executable (NX) bit that can be configured by the memory management unit (MMU) in modern operating systems. The NX bit sets a page-level flag that prevents the execution of code on the corresponding memory page. Inspired by the effectiveness of the NX bit, we have developed a simple hardware-assisted solution using two control and status registers (CSRs) within the RISC-V architecture instead of relying on the MMU.

Implementation To implement our solution, we added two CSRs, CSR_NX_START and CSR_NX_END, to define a memory region as non-executable. Whenever the PC falls into this range, the hardware raises an exception. These values are set by modifying the Zephyr RTOS to write them whenever a thread switch occurs. To tailor this to our needs, we set these values to cover the stack region of the next thread, thereby achieving a non-executable stack.

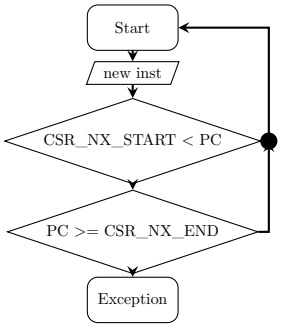

Figure 10: Simplified hardware algorithm implemented in the commit stage (part 4) - stack NX

Our solution is not limited to the stack region and can be used to protect any memory region by adding more CSRs and setting the values correctly.

II.VI Optimization

The first version of the compiled library showed a great lack of optimization; multiple solutions were applicable to solve that, we chose the last one:

1. **Core:** Add an FPU to the core because the performance benchmark is doing floating point operations.
2. **FW-CFI:** Take advantage of instruction pre-fetching to do premature NOP execution.
3. **Heap Protection:** Optimize the chunk search algorithm (example: interval trees).
4. **C Library:** Use an optimized version of the newlib nano C library (-O2 instead of -Os achieves great performance gains for an acceptable space tradeoff).

III Results

Please note that we had to change the '-o0' flag to '-O0' in the perf_baseline/CMakeLists.txt file, as we believe that '-O0' was the intended compile option to disable optimization. The provided default sometimes caused errors that we could not make sense of, nor find an explanation for in the GCC documentation.

III.I Performances

```
*** Booting Zephyr OS build zephyr-v3.2.0-327-
    g869365ab012b  ***
Begining of execution with depth 12, call number 50,
    seed value 63728127.000000
SUCCESS: computed value 868200.000000 - duration:
    25.317176 sec 632929412 cycles
```

Listing 2: Results of the benchmark on the non-modified hardware, OS and compiler

```
*** Booting Zephyr OS build zephyr-v3.2.0-327-
    g869365ab012b  ***
Begining of execution with depth 12, call number 50,
    seed value 63728127.000000
SUCCESS: computed value 868200.000000 - duration:
    24.858038 sec 621450941 cycles
```

Listing 3: Results of the benchmark on the FW CFI + shadow stack + secure heap management + stack NX HW, OS and compiler (using nano -O2)

	number of cycles	time of execution	percentage increase
before modifications	632929412	25.317176 sec	-
stack canaries	633811376	25.352455 sec	0.13%
FW CFI + shadow stack + secure heap management + stack NX	654086284	26.163451 sec	3.34%

Table 2: Results from the benchmark perf_baseline before and after the modifications (using nano -0s)

	number of cycles	time of execution	percentage increase
new baseline in -02	606203781	24.248151 sec	-
FW & BW CFI	628152061	25.126082 sec	3.62%
FW CFI + shadow stack	628152061	25.126082 sec	3.62%
FW CFI + shadow stack + secure heap management	628390584	25.135623 sec	3.66%
FW CFI + shadow stack + secure heap management + stack NX	621450941	24.858038 sec	2.52 %

Table 3: Results from the benchmark perf_baseline before and after the modifications (using nano -O2)

The sudden drop of performance in the last entry of the table (where one would expect an increase) is due to removing the remaining LP for the BW-CFI from the gcc and newlib libraries.

Our testing indicates that using the modified newlib nano compiled with the '-O2' optimization flag, instead of the default '-Os', **results in a performance improvement: -1.81% execution time compared to the initial baseline.**

resources	utilisation after modification (percentage of utilization)	utilisation before modification (percentage of utilization)	percentage increase
LUT	27274 (51.3%)	25765 (48.4%)	5.9%
FF	26147 (24.6%)	22328 (21.0%)	17.1%
BRAM	71 (50.7%)	71 (50.7%)	0%
DSP	4 (1.82%)	4 (1.82%)	0%
maximum frequency	34.2MHz	34.8MHz	-1.72%
perf_baseline size (nano -0s)	65920 B	61936 B	6.4%
perf_baseline size (nano -02)	69696 B	67728 B	2.9%

Table 4: Table of modification's impact

for the calculation of the before given Fmax, we used the formula : $FMAX(MHz) = 1000/(T - WNS)$ *from Xilinx on the clk_out1_xlnx_clk_gen intra clock path which corresponds to the core clock domain.*

Our testing indicates that using the modified newlib nano compiled with the '-O2' optimization flag, instead of the default '-Os', **results in a size overhead of 12.52% compared to the initial baseline**.

III.II Attacks prevented

i	stack canaries	FW&BW CFI	FW CFI + shadow stack	FW CFI + shadow stack + secure heap management	FW CFI + shadow stack + secure heap management + stack NX
1	✅	✅	✅	✅	✅
2	⚠️	⚠️	⚠️	⚠️	✅
3	⚠️	⚠️	⚠️	⚠️	✅
4	❌	❌	❌	✅	✅
5	✅	✅	✅	✅	✅
6	❌	✅	✅	✅	✅
7	❌	✅	✅	✅	✅
8	❌	✅	✅	✅	✅
9	✅	✅	✅	✅	✅
10	❌	✅	✅	✅	✅

Table 5: Table presenting the different prevented attacks with different approaches

IV Conclusion

In conclusion, Forward CFI, Shadow Stack, libC heap protection, and NX stack protection are effective cybersecurity measures that can help protect your digital assets from malicious attacks. While each technique has its own unique advantages, together they can provide a comprehensive and robust security framework that can help safeguard against a wide range of cyber threats. They have contributed to improving the overall security of the system by preventing the RIPE benchmark's 10 chosen attacks while also improving performance by 1.81% (through compilation tricks and space tradeoff).

References

[1] 2. Andrew Waterman1 Krste Asanovi c1, "The risc-v instruction set manual, volume i: User-level isa," CS Division, EECS Department, University of California, Berkeley, Tech. Rep., 2017.

[2] O. GROUP, *Cva6 design document*, https://cva6.readthedocs.io/en/latest/03_cva6_design/, 2023.

[3] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "Ripe: Runtime intrusion prevention evaluator," Dec. 2011, pp. 41–50. DOI: 10.1145/2076732.2076739.

[4] V. Shanbhogue, D. Gupta, and R. Sahita, "Security analysis of processor instruction set architecture for enforcing control-flow integrity," in *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '19, Phoenix, AZ, USA: Association for Computing Machinery, 2019, ISBN: 9781450372268. DOI: 10.1145/3337167.3337175. [Online]. Available: https://doi.org/10.1145/3337167.3337175.