INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Université Fédérale Toulouse Midi-Pyrénées

LAAS

Département de Génie Électrique & Informatique

Project Report

CV32A6 RISC-V Soft-Core Hackathon - INSA Toulouse

May 12, 2023

Students:

Cyprien	Heusse	heusse@insa-toulouse.fr
Emily	Holmes	holmes@insa-toulouse.fr
Arthur	Gautheron	gauthero@insa-toulouse.fr
Maïlis	Dy	mdy@insa-toulouse.fr

Tutors:

Eric	Alata	eric.alata@laas.fr
Daniela	Dragomirescu	daniela.dragomirescu@laas.fr

Keywords: RISC-V, Hardware Security, Buffer Overflow, RIPE, FPGA

Abstract:

Buffer overflow attacks have been plaguing computer security for decades. Most traditional processor architectures such as x86 and ARM have had time to build protection mechanisms into their systems. Due to its development for educational purposes, the RISC-V (Reduced Instruction Set Computer) currently lacks proper defence systems against these attacks. This research was done as part of a national contest hosted by Thales, the GDR SOC^2 and the CNFM in which we countered attacks generated from the RIPE (Runtime Intrusion Protection Evaluator) testbench. In this paper, we introduce a buffer overflow protection unit for the CV32A6 application core based on the identification of attack patterns. We designed mechanisms from these patterns to prevent buffer overflow attacks in the stack and heap, as well as variable leaks. These mechanisms were implemented in a new functional unit of the pipeline. Our implementation countered a total of 268 RIPE attacks, including the original 10 attacks selected for the contest. This was done without modifying the critical path of the processor thus not impacting the timing performances.

1 Introduction

This report is a submission for the CV32A6 RISC-V soft-core hackathon where teams are tasked with defeating the most of ten cyber attacks on the CV32A6 core.
The attacks are extracted from the Runtime Intrusion Prevention Evaluator (RIPE). Each team may modify the CV32A6 core, the operating system, or the compiler, if those changes do not affect the application behaviour. There are multiple criteria for a submission to be accepted:

- The design has to fit on the FPGA board used for the competition.
- The memory usage and execution time overhead must not exceed +100% of the base system for a given benchmark program.
- Test applications must still be able to run.

Numerous applications and architectures remain subject to vulnerabilities similar to the buffer overflow attacks generated by RIPE. RISC-V does not currently offer protection against such attacks. They threaten the integrity of a given program, or permit the execution of malicious code. This can potentially lead to the access to secrets by an attacker or the loss of data.

For this project, attacks were grouped up by finding similar patterns to limit the memory and performance impact of the submission while countering as many attacks as possible. These patterns were then studied from the processor's point of view. From this analysis, various solutions were implemented. Each solution was tailored for the contest's regulations but potential improvements will also be referenced for real-life applications.

2 Project strategy

2.1 Setup for development

A permanent system was set up to host the code and connect to the FPGA implementing the processor. We also added a webcam to be able to see the LEDs of the FPGA from a distance. We connected to the system using SSH and Visual Studio Code's "Remote SSH" extension which made it easy for all the team to work on the same project from any location. The SSH setup made nearly all commits to our repo appear to be from the same GitHub account, but the work was done collectively.

2.2 Attack list and methodology

At the start of the project, both the C and assembly code of each attack has been analysed to determine their behaviour. Patterns were identified to generalise the solutions. The table below compiles all the similarities we have identified between the attacks.

From this preliminary analysis, we established a list of five hypotheses, from which we designed and implemented hardware solutions. The hypotheses in this report have been chosen to enforce a set of rules we deemed to be good practice. This was created to focus on clean and safe code.

The hardware implementation resulted in a new functional unit named BOP (Buffer Overflow Protection) Unit. The unit was positioned in the branch unit to have access to the target address (for crashing when an illegal operation was detected by our system) as well as the registers used and their content.

The hypotheses are as follow:

- Data related to the control flow of a program should only be handled by the processor.
- Program data and control flow should not be mixed.
- Pointers should be processed separately from each other.
- The processing of a buffer cannot extend past the boundaries of the buffer.
- The compiler, through the symbol table, will always use the same strategy to access a local variable.

We do not consider these rules to be an objective method for programming. They are only a stance we took, but we judge these hypotheses to be sensible in

Table 1: Patterns identified in each attack

Attack Number	Altering Return Address	Abusing Stack Pointer	Overwriting Function Pointer	Double Dereference	Successive loads crossing interval bounds
1	Yes				
2		Yes	Yes		
3		Yes	Yes		
4					Yes
5	Yes				
6			Yes	Yes	
7			Yes	Yes	
8			Yes		
9	Yes				
10			Yes		

the context of critical embedded systems. Developing programs with these hypotheses will be discussed in section 9.3.

Sections 3 to 8 will discuss every hypothesis and its corresponding hardware solution for countering the attacks.

2.3 Optimisations for the contest

The solutions we implemented have all been improved to optimise results as defined within the expectations of the contest. The aim was to defeat as many attacks as possible while limiting the increase in execution time and memory used. As such, we made the following decisions for our code :

- Buffers have all been designed to be as small as possible and optimised to work within the context of RIPE. In a real-world scenario, bigger buffers would provide better security.
- We also use the hypothesis that there are few instructions between the overflow and the execution, which would not necessarily be the case in real attacks. It is true for RIPE.
- Our solutions focus on detection and do not restore the control flow of the program. When buffer overflows are detected, the Program Counter (PC) is set to address 0x00000000. This address is outside of the code portion of memory, which causes the processor to crash.

These elements should not be kept as-is for a real world application. However, they are enough to satisfy the contest's restraints as well as providing a proof of concept for our suggested solutions.

3 Altering the return address

Hypothesis 1. Data related to the control flow of a program should only be handled by the processor.

3.1 Pattern & Illustration

Upon a function call, the processor stores the address of the next instruction in the Return Address (RA) register. When it is finished, the function returns to the next instruction thanks to RA. Once the processor jumps into the function, the current context is saved. The value of the registers that are going to be used in the function are saved onto the stack, including RA.

```
ADDI sp,sp,-16
SW ra,12(sp)
```

Once the execution of a function is over, the processor needs to retrieve the return address and jump to said address via the pseudo-instruction "RET".

```
LW ra,12(sp)
ADDI sp,sp,16
RET
```

Between the beginning and the end of the function, the return address sits in the stack where it is modifiable by a malicious user.

Our first pattern is as follows:

- The return address is pushed onto the stack by a legitimate process when a function is called.
- A malicious process modifies the return address between the call and the return. It sets the address to another section of the code (in order to do ROP (Return-Oriented Programming) for example).
- The legitimate process pops the return address from the stack that has been modified and jumps there. The attack is successful and the malicious process has hijacked the control flow.

3.2 Suggested solution

An easy way to counter this pattern is to encode the return address when pushing it onto the stack. There are different ways to do so:
The most lightweight but also less secure way is to simply XOR the value of the return address with a fixed secret key when it is pushed and to decode it, using XOR again, when it is popped. To lessen the performance impact of our solution, we chose to use a fixed key. However, there is a chance for the attacker to find the secret value. This can be made more complex by generating a new secret value using a reversible LFSR (shift register), making the secret value pseudo-random and a lot harder to reverse-engineer.

4 Executing overwritten function pointer

Hypothesis 2. Program data and control flow data should not be mixed.

4.1 Pattern

Function pointers can be the target of buffer overflow attacks to run malicious code. The goal of this attack is to modify the content of the function pointer. This results in the execution of a malicious function instead of the intended one. Executing a function call requires two instructions in RISC-V assembly : the program loads the function pointer's content into a register, then jumps to the register's address. Thus, the pattern is as follows:

- During the execution of the program, buffers are detected due to consecutive STORE instructions. The interval (first address, last address) is stored in our hardware buffer.
- After a load instruction, the processor checks if the address loading from is in the buffer. Our hypothesis states that program data should be distinct from control flow, so if the processor jumps to a value stored at an address in the buffer, an attack is detected.

4.2 Illustration

C Functions such as memcpy use the SB instruction to store data byte after byte. Consecutive write operations to the memory are identified by repeated SB instructions separated by few other instructions.

```
LB t2, 0(a1)
SB t2, 0(t1)
ADDI a2,a2,-1
ADDI t1,t1,1
ADDI a1,a1,1
loop
```

The BOP unit detects a function call from a function pointer stored in a saved interval.

```
LW a5, -1968(a5)
JALR a5
```

Where the address `a5-1968` is in an interval of the the circular buffer.

4.3 Suggested solution

The proposed defence mechanism identifies consecutive SB instructions and function calls from a function pointer within the modified interval.

When detecting a SB instruction, the mechanism activates and starts monitoring the target address of the next SB instructions. The mechanism is designed to tolerate a specific number of non-SB instructions before deactivating. The starting and ending addresses of the interval are saved in a circular buffer.

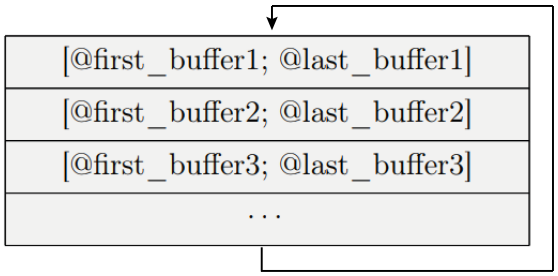

Figure 1: Illustration of the circular buffer as implemented

The mechanism also detects attempts to execute code within the intervals identified by the circular buffer. When an address in the buffer is loaded in a register, a flag signal is raised. If a jump operation is executed from the flagged address, the processor crashes.

5 Double dereference

Hypothesis 3. Pointers should be processed separately from each other.

5.1 Pattern

Function pointers may also be the target of indirect buffer overflow attacks. In this type of attack, the attacker alters the address of a pointer by dereferencing another pointer. This can be done by obtaining the address of the function pointer and using a second pointer to change its value to a different address. In this case, the BOP Unit may detect a double dereference. Our hypothesis states that pointers should be processed separately from each other, thus it is illegal to perform a double dereference of an address in a buffer. This solution reuses the circular buffer from section 4.3.

5.2 Illustration

This pattern also includes the detection of intervals as seen for Hypothesis 2.

The double dereference can be illustrated like this in RISC-V assembly language:

```
LW a5,-1744(s0)
LW a5,0(a5)
```

5.3 Suggested solution

The mechanism builds onto the previous one. It also detects double dereferences. For two consecutive LW instructions, a flag is raised when the result of the first one is used as the source address of the second one. This causes earlier crashes in RIPE since the double dereference happens when building the payload. This is why the "Executing attack..." is not logged when executing attacks 6, 7 & 8.

6 Successive loads crossing interval's bounds

Hypothesis 4. The processing of a buffer cannot extend past the boundaries of the buffer.

6.1 Pattern

There is no restriction when performing a copy of a part of a buffer in the heap to another location. This vulnerability can be exploited to copy data from unreachable buffers to a buffer controlled by a malicious user. Our hypothesis states that the processing of a given buffer must only affect the area in memory dedicated to that buffer, and cannot also affect the memory before or after it.

6.2 Illustration

The BOP Unit looks out for instructions such as:

```
LB t2, 0(a1)
```

Where the address in `a1` will be compared with previous data as per the next paragraphs.

6.3 Suggested solution

The mechanism we developed to cover this specific attack is an extension of the defence mechanism presented above. If memory is copied from the heap and eventually reaches the beginning of a buffer, a flag is raised. This solution relies on the circular buffer mentioned above.

In this case, we focus on detecting successive use of LB (Load Bytes). The defence mechanism activates if the address loading to is the first address of a buffer in a series of consecutive loads (at least eight bytes).

7 Abusing stack pointer

DISCARDED Hypothesis 5. The compiler, through the symbol table, will always use the same strategy to access a local variable.

7.1 Pattern

When manipulating a pointer function, the processor pushes the address of the pointer function onto the stack. The sp (stack pointer) or fp (frame pointer) register are always involved in this process. If its value changes upon affectation by the user, the processor still proceeds to a STORE operation, using the stack (or frame) pointer and an immediate to access the pointer function. However, functions such as memcpy do not use the stack (or frame) pointer. In the case of memcpy, the processor uses a temporary register, such as t1 in gcc for RISC-V, to hold the address to STORE to.

Our hypothesis forbids using a different method for accessing a variable once it has been initialised. The attack pattern is as follows:

- A pointer function is defined and a value is affected to it. The processor stores this value to a particular address in memory, which is referred to by the address of the stack (or frame) pointer and an immediate.
- A malicious user attempts to alter this value with a buffer overflow. The new value is stored at the same address in memory without using the stack pointer.

7.2 Illustration

In assembly, a legitimate way of accessing a pointer function is:

```
sw a1, 512(sp)
```

While an illegitimate method would be, following the line above:

```
sw a1, 0(t1)
```

Where `sp+512 = t1`.

7.3 Suggested solution

In order to counter this pattern, we implemented the following protection mechanism:

- Upon detection of a STORE instruction using the stack pointer, the address in memory of the store is written in our hardware buffer.
- Upon detection of a STORE instruction that does not use the stack pointer, a check is performed to verify the address in memory of the instruction is not inside the buffer. If it is inside the buffer, an attack has been detected according to our hypothesis.

This solution was ultimately disregarded as solution in section 4 covers a larger panel of attacks. We still decided to include it in this report because the logic is sound.

8 GCC plugin for function pointers

DISCARDED Hypothesis 6. Function pointers must be processed individually.

Many buffer overflow attacks aim at function pointers. A pointer is a variable in memory which stores the address of another element. It is possible to use buffer overflows to modify the address held by a pointer. Function pointers are particularly affected, because they will eventually be called and alter the control flow of the program.
One prevention method is to duplicate the pointers upon compilation. When the pointer is modified by direct affectation, both are modified. However, if the value is modified via a buffer overflow, only one of the pointers is corrupted. We can then compare both pointer addresses when calling said pointer. If the values are different, a flag is raised.
This idea was implemented in a GCC plugin which you can find on our GitHub repository. However, this idea was discarded as the rules of the contest were later clarified to forbid such modifications on the compiler.

9 Results

9.1 Result table

The impact of each solution can be found on Table 2. The table refers to numbered solutions according to the corresponding hypothesis as we numbered them above.

Table 2: Impact of each solution per attack

Attack Number	Solution 1	Solution 2	Solution 3	Solution 4
1	Counters			
2		Counters		
3		Counters		
4				Counters
5	Counters			
6			Counters	
7			Counters	
8	Counters	Counters		
9	Counters	Counters		
10		Counters		

9.2 Performance analysis

Both the base design and our modified design were tested using the provided `perf_baseline` program. On the base design, an average execution time of `25.3171894s` (632929729 cycles) was found. After implementing the buffer overflow protection unit, all ten given attacks were successfully countered. On average, the execution time of the test program was `25.2989816s` (632474527.2 cycles). The relative deviation of both cycle counts is 0.07%. The buffer overflow protection unit probably is not on the critical path of the CV32A6 processor since it doesn't have any impact on timing performances.

The timing reports after implementation on Vivado showed a slack of `1.615ns` for a required period of `20ns`, giving a `18.385ns` clock period. Thus, the frequency of the FPGA after implementation is roughly `54MHz` (54,392167 MHz).

The hardware modifications on the FPGA are listed in Table 3.

Table 3: Hardware impact on the FPGA

	Before	After implementation
LUT	26639	27034
Flip-Flops (included in LUT)	22415	22893
BRAM	71	71
DSP	4	4

Table 4 shows the size of the `perf_baseline` program once compiled by west after the implementation of our protection unit.

There is a 16 byte difference between the original program and our modified version. The modification of the Zephyr RTOS to take into account our new instruction is a plausible explanation.

Table 4: Size of perf_baseline after implementation

Memory Region	Used Size	Region Size	%age Used
RAM	61952 B	1 GB	0.01%
IDT_LIST	0 GB	2 KB	0.00%

We additionally have written a script to test our mechanisms on all of the RIPE testbench. On the base design, we found that out of the 587 attacks that produced a result (no errors), 236 were already countered. After implementation of our buffer overflow protection unit, this number went up to 504 attacks. Thereby, 268 attacks were stopped by the protection unit leaving only 83 attacks still working. The effectiveness of our solutions goes beyond the original ten attacks of the contest.

9.3 Implementation beyond the scope of the contest

In this paper, we identified machine behaviours which could allow an attacker to hijack the control flow of a program and we blocked them through the processor's design. While this approach does block an attacker from executing malicious code or reading protected data, it could also lead to unexpected crashes for a legitimate developer. It is possible to create a GCC plugin which detects these patterns upon compilation and creates a warning or an error that is easy to correct. We made one such plugin which identifies the pattern 5 and creates a warning. A proof of concept is available at in our team's GitHub under the `plugin` and `warning` folders.

Additionally, the solutions in this report focus on detection. They could ideally be implemented together in the form of an IDS (Intrusion Detection System). A possible continuation would be to create a language to program the IDS so that its parameters can easily be altered depending on the applications. With the plugin mentioned above, they provide a straightforward coding experience for the developer.

In the annexe available in the `report` folder of our GitHub, every solution has been formalised. We think this formalisation could help develop the tools mentioned above.

10 Conclusion

The hypotheses introduced in this report were used to implement a new functional unit in execute stage of the pipeline. These solutions countered all ten of the contest's attacks but also 236 RIPE attacks of the 587 we tested on the processor. These results show the relevancy of the implementation in targeting broader attacks. They also have a very marginal impact on the overall performance, with practically no impact on execution time (relative difference of 0.07%).
In order to defeat the remaining non-countered attacks from the RIPE testbench, we could use a similar methodology and suggest a more complete solution. In the case of real world applications, the development of a complete IDS is required as our current research crashes the processor when a flag is raised. Additional tools like a complete GCC plugin are also needed so that a legitimate developer knows if their code could be recognised as illegitimate according to our hypotheses.