CV32A6 RISC-V soft-core hackathon *Breizh Hat* Student Team Report

Alexandre Riviere
Master 2 CSSE
Université Bretagne Sud
Lorient, Bretagne
riviere.e2105264@etud.univ-ubs.fr

Valentin Geffroy
Master 2 CSSE
Université Bretagne Sud
Lorient, Bretagne
geffroy.e2003481@etud.univ-ubs.fr

Jean-Charles Gerard
Master 2 CSSE
Université Bretagne Sud
Lorient, Bretagne
gerard.e2102612@etud.univ-ubs.fr

Vianney Lapôtre
Research Professor
Université Bretagne Sud
Lorient, Bretagne
vianney.lapotre@univ-ubs.fr

Kévin Quénéhervé
Master 2 CSSE
Université Bretagne Sud
Lorient, Bretagne
queneherve.e2005011@etud.univ-ubs.fr

Philippe Tanguy
Research Professor
Université Bretagne Sud
Lorient, Bretagne
philippe.tanguy@univ-ubs.fr

***Abstract*—This document deals with our submission for the 3rd National Student Contest organized by Thales France. The goal is to defeat several software and hardware attacks on the CV32A6 core. Based on the RISC-V ISA, this compact 32-bit core developed by the OpenHW Group has been equipped with the Zephyr RTOS which runs some test applications. They are subject to some attacks mainly on different memory features like the heap, the stack and some buffers. We have worked on some countermeasures like adding security flags to the GCC compile or enabling the canaries mechanism in the ZephyrRTOS. We thought about some RISC-V security features to protect the system but we didn't have finish setting them up for now.**

***Index Terms*—hardware security, risc-v, cva6, buffer-overflow, stack, heap, ripe, fpga, zephyr rtos, memory protection**

I. Introduction

More and more embedded systems are created and used regularly in many everyday cases. This brings with it more security issues. We are currently investigating some vulnerabilities on a specific hardware platform - well known in the growing open-source industry, market and manufacturing - that is becoming more and more popular in embedded systems research: the RISC-V Instruction Set Architecture (ISA).

One of a CPU class core developped for the RISC-V ISA is named the CVA6. We'll see in this report how it works, how we ran a material implementation on the Zedboard Zynq-Z7 FPGA and interacted with it thanks to a Zephyr Real Time Operating System (RTOS) designed for embedded use.

II. CV32A6 main study

A. Some history about the CVA6

The CVA6 project was created by the OpenHW Group. It is an industrial evolution of the ARIANE project developped by the ETH Zürich school and the University of Bologna. Its goal is to provide a new family of open source processors with the special feature to support different application cores like the RISC-V class. Indeed, the Application-Specific Integrated Circuit (ASIC) and the Field Programmable Gate Array (FPGA) implementations are the main target of the CVA6. The CVA6 is written in SystemVerilog. This hardware description language is used to design, implement and test electronic systems thats why it is considered as heavily parameterizable. For example some parameters can set the ILEN[1] through the IP in the FPGA's design conception phase for example. The CV prefix refers to the technical aspect of the CVA6 as a member of the CORE-V family and the A6 indicates the application class processor composed by a six stage execution pipeline.

B. Dive into the CVA6 architecture

As the OpenHW Group thought, the CVA6 pipeline has been designed to reduce the critical path way on all RISC-V processor instructions. It fully implements the RISC-V architecture as specified in *the RISC-V Volume 1, Unprivileged Specification* and *Volume 2, Privileged Specification* to support Unix-like operating systems. Indeed, the RISC-V privileged instruction set architecture (ISA) divides its execution model into the *Machine-Mode* (M-mode) which is the most privileged level, the *Supervisor Mode* (S-Mode) which runs Unix operating systems that require virtual memory management and the *User-Mode* (U-Mode) which executes userland applications.

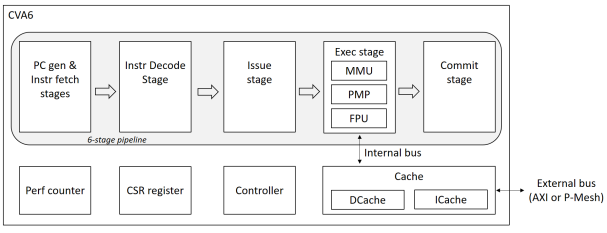

Fig. 1: The CVA6 design is a six stage execution pipeline

[1]*Nota bene*: the ILEN term refers to the maximum instruction length supported by an implementation so in this case it concerns 32 or 64 bits.

As briefly detailed in *Fig. 1*, the CVA6 core features a 6-stage pipelined to run a full OS at reasonable speed. To achieve this goal, the CPU has separate Translation Lookup Buffers (TLB), a hardware Page Table Walker (PTW) and branch-prediction (branch target buffer and branch history table). This can be explained by a scoreboard which hide latency to the data RAM (cache) by issuing data-independent instructions.

C. Some details about the CVA6 stages

We'll now have a look at each part of the six stage execution pipeline but you can find more detailed information on the CVA6 **[2]** specification. First of all, the PC gen stage is responsible for generating the next program counter. Indeed, we recall that it is a register in a computer processor that contains the address of the instruction being executed at the moment but it is necessary to note that program counters are logically addressed. If the logical to physical mapping changes, a `fence.vm` instruction should flush the pipeline and TLBs. This stage contains speculation on the branch target address as well as the information if the branch is taken or not. In addition, the Instruction Fetch (IF) gets its information from the PC Gen stage, which deals with the current PC, the branch prediction and if this request is valid. The IF stage asks the Memory Management Unit (MMU) to do address translation on the requested PC. This step is very time-critical, so the OpenHW group designed it to avoid a more elaborate handshake protocol (as the execution times would be too long).

The third part deals with the Instruction Decode (ID) stage which is the first pipeline stage of the processor's back-end as illustrated in *Fig. 2*. We don't go into as much detail as the Ariane documentation **[3]** does but its main purpose is to decompose instructions from the data stream it gets from the previous stage, decode them and send them to the Issue Stage.

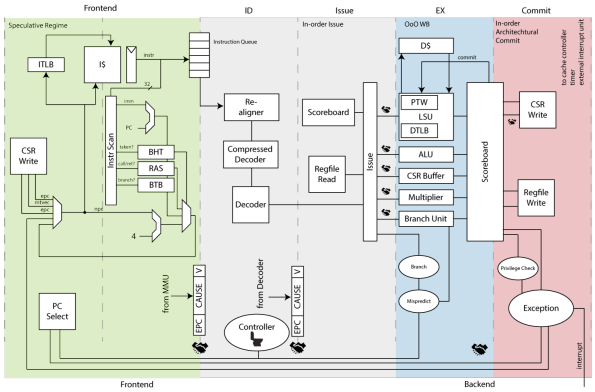

Fig. 2: A view of the CVA6 architecture [OpenHW Group]

With the introduction of variable length instructions like the compressed instructions, the ID stage gets a little bit more complicated. Indeed, it has to search the incoming data stream for potential instructions, re-align them and decompress them.

As the fourth step, the Issue Stage is designed to receive the decoded instructions and transmit them to the different functional units. In addition, the transmit stage keeps track of all transmitted instructions, the status of the functional unit and receives the rewrite data from the execute stage. It also contains the CPU register file. Using a data structure called a *scoreboard* like in *Fig. 3*, it knows exactly which instructions are issued, in which functional unit they are located and in which register they will be written back.

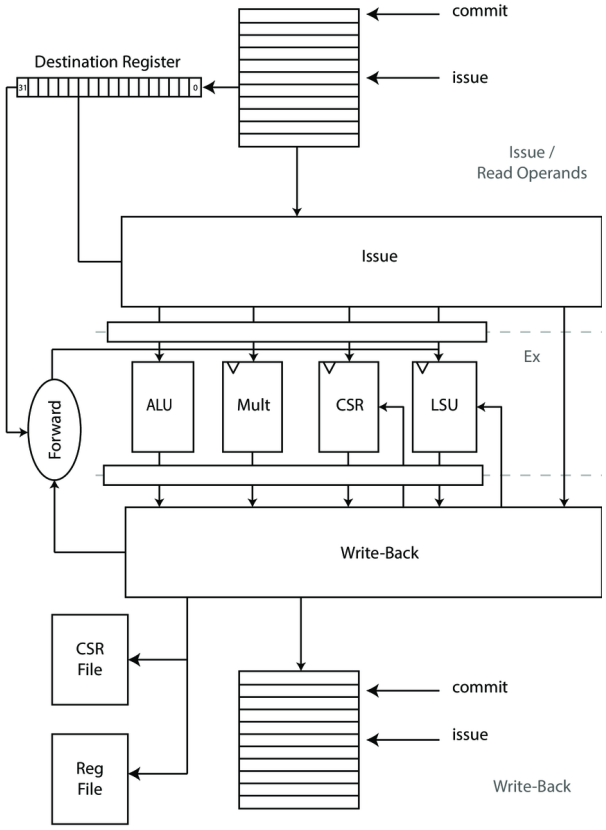

Fig. 3: The scoreboard of the Issue Stage [OpenHW Group]

The fifth step is a logical step that encompasses all functional units (FUs). The functional units are not supposed to have any dependencies on each other. Moreover, each functional unit must be able to operate independently of all other units and maintain a valid signal with which it signals valid output data and a ready signal. It tells the issue logic whether it is able to accept a new request or not then returns the transaction ID along with the valid signal and the result.

Finally, the Commit Stage is the last stage of the processor pipeline. Its purpose is to take incoming instructions and update the state of the architecture. This includes writing the CSR registers, validating the memories and writing the data back to the register file. The golden rule is that no other stage in the pipeline is allowed to update the architectural state under any circumstances. If it maintains an internal state, it must be resettable. The Commit stage controls the general shutdown of the processor. If the stop signal is set, no new instructions will be committed, which will generate back-pressure and eventually stall the pipeline. The Commit stage

also communicates extensively with the controller to execute close instructions (cache flush) and other pipeline resets.

If later we wish to compare any material changes to the CVA6 like to the memory, it is interesting to notice that the instruction RAM (or L1 instruction cache) has an access latency of 1 cycle on a hit, while accesses to the data RAM (or L1 data cache) have a longer latency of 3 cycles on a hit.

D. The 32 bit version of the CVA6

While we have clarified what the CVA6 is, we need to explain the specificity of its 32 bit version called the CV32A6 which we studied during the contest period. The processor is still a Harvard-based modern architecture which implements a 6-stage pipeline composed of PC Generation, Instruction Fetch, Instruction Decode, Issue stage, Execute stage and Commit stage. At least 6 cycles are needed to execute one instruction. Instructions are issued in-order through the DECODE stage and executed out-of-order but committed in-order. The processor is still "single issue" which means that a maximum of one instruction per cycle can be issued to the EXECUTE stage. So the main difference may be the length of the instruction bus (32 bits instead of 64 bits) but this is more about the lack of MMU. We will see later what the consequences will be and which secure aspects it concerns.

III. Zephyr RTOS

Highly known for its small footprint, high performance, and flexibility the Real Time Operating System (RTOS) Zephyr is an open-source project developed by the Linux Foundation. Ported by Thales to run our pentesting test on the CV32A6, it provides a wide range of features commonly found in all RTOSs such as task scheduling, interrupt handling, memory management (highly useful for our case), device drivers, etc.

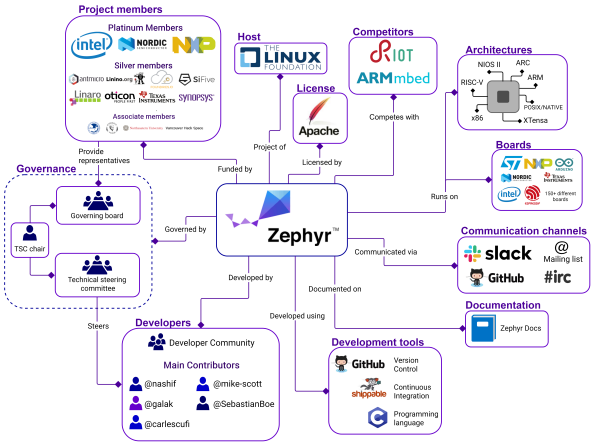

Fig. 4: Context of the Zephyr RTOS workflow [Zephyr Project]

One of the key features of Zephyr is its modular architecture. It supports ARM, x86 and RISC-V implementation. The RISC-V implementation in Zephyr is based on the RISC-V ISA like both RISC-V 32-bit and 64-bit architectures.

By default, Zephyr has a single privilege mode called kernel mode. This is the privilege level of the OS kernel itself. For hardware platforms without an unprivileged mode, threads will also run in privileged mode along with the kernel. On these platforms, a buggy or malicious thread could corrupt other threads or the kernel itself. This is obviously undesirable.

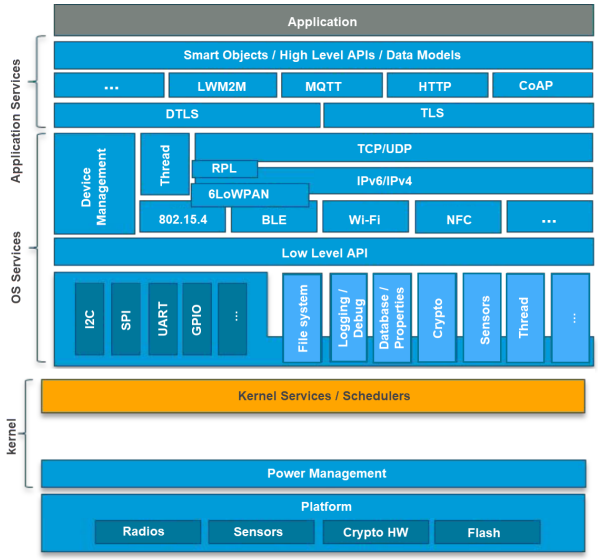

Fig. 5: The Zephyr RTOS architecture [Zephyr Project]

Zephyr supports hardware stack protection so we must consider it for our case application. Hardware stack protection is an optional feature which detects buffer overflows when the system is running in supervisor mode. This issues when the entire kernel stack buffer has overflowed, but not for individual stack frames. However Zephyr supports optional compiler features which enable stack canaries for the individual frames we could use to solve the CVA6 memory vulnerabilities.

IV. Define possible attacks vectors

As we previously said, the goal of the contest is to protect the Zephyr OS running on the CV32A6-based processor from executing a corrupted payload. With the type of CVA6 RISC-V architecture we need to explain what the possible attack issues. The corrupted payload is simply printing the string "Executing attack... success". Ten scenarios are implemented, the *Table-I* shows their parameters which come from the RIPE program we'll use, the purpose of which will now be clarified.

nb	method	target	code_ptr	location	function
1	direct	no nop	ret	stack	memcpy
2	direct	no nop	funcptrstackvar	stack	memcpy
3	indirect	no nop	funcptrstackvar	stack	memcpy
4	direct	data	var_leak	heap	sprintf
5	direct	ret to libc	ret	stack	memcpy
6	indirect	ret to libc	funcptrheapvar	heap	memcpy
7	indirect	ret to libc	structfuncptrheap	heap	homebrew
8	indirect	ret to libc	longjumpbufheap	heap	memcpy
9	direct	rop	ret	stack	memcpy
10	direct	rop	structfuncptrheap	heap	memcpy

TABLE I: RIPE parameters

For each attack we get one point if the print function is not executed during a RIPE execution in the Zephyr environment on the CV32A6 on the ZYBO Z7-20 board. The following code is extract from `ripe/src/ripe_attack_generator.c` . We changed the `ATTACK_NR` variable to try each of the ten attacks.

```
#define ATTACK_NR 1
#if ATTACK_NR == 1
    attack.technique = DIRECT;
    attack.inject_param = INJECTED_CODE_NO_NOP;
    attack.code_ptr= RET_ADDR;
    attack.location = STACK;
    attack.function = MEMCPY
#elif ATTACK_NR == 2
    attack.technique = DIRECT;
    ...
```

Listing 1: Choosing the attack parameter from 1 to 10

We are not allowed to use dishonest techniques to block RIPE feedback, for instance by blocking the print function or even show a false result for our submission. Changing the `/workspace/ripe` or `/workspace/perf_baseline` to modify something is against the rules.

V. Using RIPE to attack the target

A. What is the Runtime Intrusion Prevention Evaluator?

RIPE is an extension of Wilander's and Kamkar's testbench which covers 850 attack forms. The main goal of RIPE is to provide a standard way of testing the coverage of a defense mechanism against buffer overflows. There are various types of shellcode that an attacker can use to exploit vulnerabilities.

One option is to use **shellcode without a NOP sled**. This can be useful for testing the accuracy of attacks and challenging countermeasures that rely on detecting specific code patterns, such as the presence of NOP instructions in the process's address space. Another common form of shellcode is one that includes a **NOP sled**. This is when the attacker includes a set of NO-Operation instructions at the beginning of the code to improve the chances of redirecting the program's execution flow into the injected code. In some cases, attackers may use a polymorphic NOP sled instead of the standard set of 0x90 bytes. This makes it more difficult for countermeasures that rely heavily on detecting standardized NOP sleds to detect such attacks. The **return-into-libc** attack is where the attacker does not inject new code but rather uses existing functions to perform their attack. This is often used when countermeasures like Data Execution Prevention and WxorX **[10]** are in place. Finally, there is **Return-Oriented Programming (ROP)**, which is the most recent method of exploitation. ROP allows attackers to use chunks of functionality from existing code to create new functionality by using the return address of a function. While ROP attacks are powerful, they require control over the existing stack.

B. Testing the program

We have built the program to the target. The goal is to see if the vulnerabilities exist using the previous pentesting RIPE script as seen on the previous *Table-I*. We succeeded to do it by following the Thales tutorial step by step. Our results have been logged into our Github `reports` branch.

```
*** Booting Zephyr OS build
zephyr-v3.2.0-327-g869365ab012b  ***
RIPE is alive! cv32a6_zybo
RIPE parameters:
technique       direct
inject param    shellcode
code pointer    ret
location        stack
function        memcpy
----------------
Shellcode instructions:
lui t1,  0x80001                        80001337
addi t1, t1, 0xb1c                      b1c30313
jalr t1                                 000300e7
----------------
target_addr == 0x8000afec
buffer == 0x8000aa70
payload size == 1409
bytes to pad: 1392

overflow_ptr: 0x8000aa70
payload: 7

Executing attack... success.
Code injection function reached.
exit
```

Listing 2: The RIPE native output with the 1st attack

VI. Working Environnment

Here is the complete setup we did for the contest:

- A virtual machine (Ubuntu 2021.04 according the rules) for an easier share between members of the team
- Xilinx Vivado 2020.1
- Xilinx Zedboard Zynq-Z7
- Zephyr RTOS v3.8 (from Thales Group)
- CV32A6 v4.2 (from Thales Group)

VII. Contermeasures and results

Before explaining our work we've done, we must present each feature we found to mitigate the known vulnerabilities.

A. Modifying the compiler

1) The theory: In general, the C programs routinely suffer from memory management problems. For several years, a `_FORTIFY_SOURCE` preprocessor macro inserted error detection to address these problems at compile time and run time. To add an extra level of security, `_FORTIFY_SOURCE=3` has been in the GNU C Library (glibc) since version 2.34. Moreover, we could use the `-fstack-protector` in the GCC compiler to detect when the stack has been overwritten.

2) What we did: To implement the function, we firstly added the compiler flags in the RIPE program workspace with the `set(CMAKE_C_FLAGS "$CMAKE_C_FLAGS" -fstack-protector-strong)` command line. We didn't know it was forbidden but here the results we had:

```
*** Booting Zephyr OS build
zephyr-v3.2.0-327-g869365ab012b ***
RIPE is alive! cv32a6_zybo
RIPE parameters:
technique       direct
```

```
inject param     shellcode
code pointer     ret
location         stack
function         memcpy
* buffer overflow detected *
```

Listing 3: Attack 1st on the stack - buffer overflow detection

We known this is against the rules but it was interesting to know why. Indeed, the compiler security context has been set thanks to this option at runtime. But next, we tried to reproduce the same directly in the Zephyr configuration. We added the `zephyr_compile_options(-fstack-protector)` in the CMakeLists.txt used for the Zephyr build. We tried `-fstack-protector-strong` which enables stack protection for vulnerable functions while `-fstack-protector-all` enables it for all functions.

Furthermore, the GCC compiler provides others security flags such as `-Wformat-security` for warning about format string vulnerabilities, `-D_FORTIFY_SOURCE=2` for additional security checks on library functions, `-pie` or `-fpie` which enables the Address Space Layout Randomization (ASLR). We noticed they are enabled by default in Zephyr RTOS 3.8 even it doesn't protect from the RIPE program.

B. Canaries in Zephyr RTOS

1) The theory: As a reminder, a "canary" is a security mechanism used to detect and prevent buffer overflow attacks. Buffer overflow attacks occur when an attacker sends more data to a program than it can handle, causing the excess data to overflow into adjacent memory locations. This can allow an attacker to execute arbitrary code or gain access to sensitive information. A canary works by placing a small piece of data (often called a "canary value") in the program's memory before a buffer that is vulnerable to overflow. This canary value is designed to be checked before and after the vulnerable buffer is used. If the canary value has been modified between these two checks, it indicates that a buffer overflow has occurred and the program can take appropriate action (such as terminating the program or alerting the user) like *Fig. 6* illustrates.

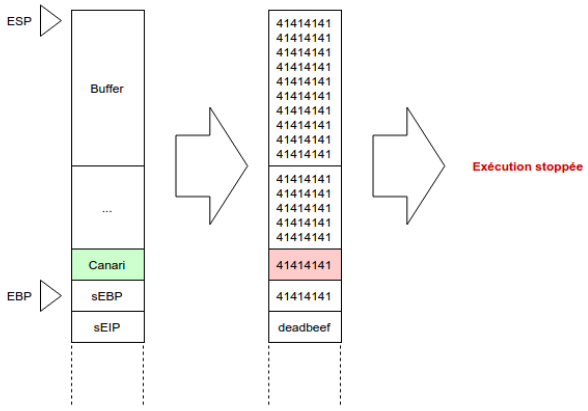

Fig. 6: A schematic of the canaries mechanism [hackndo]

In the context of GCC on RISC-V, canaries are used in the same way as on other platforms. The specific implementation may differ depending on the version of GCC and the operating system being used, but the basic idea is the same. When compiling software with canaries enabled on RISC-V, the GCC compiler will generate code that inserts canary values into vulnerable buffers to help prevent buffer overflow attacks.

2) What we did: To try to implement the function on Zephyr, we modify the `zephyr/CMakeLists.txt` as you can see in our branch `canaries-zephyr` on our Github page. The CMake is used as a build file generation tool for compiling because it generates build files and defines compilation targets that's why we needed to build again Zephyr at each time we change something in the `zephyr/CMakeLists.txt` or `zephyr/cmake/compiler/`. This gives this content:

```
if(CONFIG_STACK_CANARIES)
        zephyr_compile_options(
                $<TARGET_PROPERTY:compiler,
                security_canaries>)
endif()
```

Listing 4: Setting canaries mechanism in Zephyr

Even we haven't finished to try the feature we had some results to show. As expected, the detection worked in application:

```
*** Booting Zephyr OS build
zephyr-v3.2.0-327-g869365ab012b  ***
RIPE is alive! cv32a6_zybo
RIPE parameters:
technique        direct
inject param     returntolibc
code pointer     ret
location         stack
function         memcpy
----------------
Shellcode instructions:
lui t1, 0x80003                  80001337
addi t1, t1, 0x064               06430313
jalr t1                          000300e7
----------------
target_addr == 0x800011adc
buffer == 0x800116bc
payload size == 1061
bytes to pad: 1056

overflow_ptr: 0x80003080
payload: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Executing attack...
*** stack smashing detected ***: terminated
exit
```

Listing 5: The Canaries detection on the 5th RIPE attack

C. PMP : the Physical Memory Protection in Zephyr RTOS

1) The theory: The Zephyr Project offers the possibility to add a Physical Memory Protection (PMP) in a RISC-V architecture. When the platform has Physical Memory Protection (PMP) support, enabling it on Zephyr allows user space support and stack protection to be selected. In RISC-V, the Physical Memory Protection is implemented using

a combination of hardware and software mechanisms. The hardware provides a set of memory protection features that allow the operating system to allocate memory and set access permissions for each region. This includes setting read, write, and execute permissions as well as specifying which modes can access the memory like *Fig. 7* shows. Each PMP entry is defined by a config register and an address register.

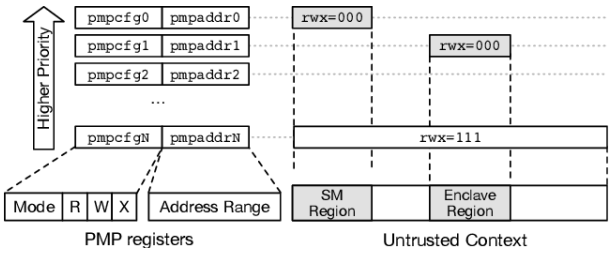

Fig. 7: The Physical Memory Protection mechanism

Additionally, the software can also configure the memory protection hardware by setting up the memory mapping tables and configuring the page tables. The memory mapping tables specify the physical memory regions and the page tables map virtual addresses to physical addresses. Overall, physical memory protection in RISC-V helps to ensure that the system's memory is protected from unauthorized access, preventing malicious code from accessing sensitive data or modifying critical system code. The Physical Memory Protection (PMP) feature was first integrated into Zephyr RTOS in version 1.14, which was released in May 2019. However, it is important to note that the level of PMP support may vary depending on the specific architecture and platform being used with Zephyr. In our case, it depends on its integration in the CVA6.

2) What we did: We noticed the PMP has been added in the version 4.3 of the CVA6. The CVA6 includes a Physical Memory Protection (PMP) unit. The PMP is both statically and dynamically configurable. The static configuration is performed through the top level parameters `ArianeCfg.NrPMPEntries`. The dynamic configuration is performed through the CSRs as Control and Status Registers. A maximum of 16 PMP entries are supported. However our contest version is based on the version 4.2 of the CVA6. We needed to cherry-pick latest branches into our project to enable the PMP. But we didn't have time to achieve this.

D. Shadow Stack ideas for a RISC-V implementation

In RISC-V architecture, the concept of a shadow stack is similar to other architectures. It provides a separate memory area that stores the return address of the function calls. This separate memory area is called the "RISC-V Shadow Stack" based on the Control-flow Integrity (CFI) feature. It provides CPU instruction set architecture (ISA) capabilities to defend against Return-Oriented Programming (ROP) and Call/Jump-Oriented Programming (COP/JOP) attacks.

When a function is called, the return address of the calling function is pushed onto the Shadow Stack, along with any additional arguments or variables that need to be saved. This separate stack is maintained by hardware, and it is separate from the normal stack that is used for storing function arguments and local variables. When the function returns, the Shadow Stack is checked to ensure that the return address on the normal stack matches the return address on the Shadow Stack. If they do not match, an error is raised, and the program is terminated. The Shadow Stack provides an additional layer of security against certain types of attacks, such as Return-Oriented Programming (ROP) attacks, which we talked about earlier. The Shadow Stack can prevent these type of attacks, which are used in the RIPE program, by ensuring that the return addresses on the normal stack are valid and match the expected return addresses on the Shadow Stack.

The use of a Shadow Stack is an effective mechanism but not officially supported in the RISC-V architecture. Indeed, we didn't find any completed implementation. Today, there are discussions **[11]** and research from the RISC-V community but this could be a great feature to defeat attacks on the stack.

VIII. Conclusion

To conclude this report, we learned a lot about the CVA6. With a complex architecture, this type of RISC-V processor implementation is interesting to study. We noticed some hardware and software vulnerabilities on the memory like the stack, the heap... To mitigate them, we tried to integrate some software solutions on the compiler and the Zephyr RTOS even we didn't finished what we would like to do. We know we haven't mitigate each of the ten attack like on the stack, the heap with the return-to-libc, the ROP attacks... However the training was very instructive. We talked on the Physical Memory Protection and the Shadow Stack. It would have been interesting to add this newest improvements to be more efficient against the vulnerabilities we studied.

Acknowledgment

We want to thank our teachers who supported us as our supervisors both Mr. Lapôtre and Mr. Tanguy. Moreover, a thanks to the organizers of the contest. It was very informative!

References

[1] **Thales Group** - Official CVA6 hackathon - Last view: 11-05-23
[2] **OpenHW Group** - The CVA6 RISC-V - Last view: 16-04-23
[3] **OpenHW Group** - A CV32A6 presentation - Last view: 29-04-23
[4] **The KeyStone Enclave** - Some RISC-V ISA explanation
[5] **RISC-V News** - Zephyr RTOS Virtualization and Memory Isolation
[6] **RIPE Github** - Runtime Intrusion Prevention Evaluator
[7] **RedHat** - GCC compiler's fortification levels - Last view: 29-04-23
[8] **Zephyr Project** - Integrating RISC-V PMP Support in Zephyr
[9] **OpenHW Group** - The PMP in the CVA6: how it works?
[10] **Wikipedia** - How works the W xor X attack? Last view: 14-03-23
[11] **Github RISC-V** - The RISC-V CFI specification - Last view: 29-04-23