CV32A6 Codesign Reporting

Fatma Ben Amor **Oussama Akennaf** **Antoine Faurie**

Abstract: *The CV32A6 RISC-V soft-core Hackathon is sponsored by Thales, the GDR-SOC² conference and the CNFM. It is a contest that invites Master students to work on a RISC-V instruction set implemented on a CVA6 architecture. For this, a Xilinx Zybo Z7 board is provided, as well as the Zephyr RTOS. This paper aims at presenting the knowledge we have acquired during the project, as well as describing the steps we have taken to counter RIPE attacks, subject of this edition of the contest.*

I. Introduction

In 2019 the OpenHW group was created, with the goal of designing industrial scale applications for the RISC-V instruction set, developed by researchers at the University of California, Berkeley. For this, a 64 bit version of the CVA6 processor was implemented. The Thales engineers then made this 64-bit processor more compact by proposing the 32-bit CV32A6 processor, the architecture that will be used during this competition. The objective of this contest is, under this environment, to thwart HOPE-RIPE attacks. All the scripts and testbench necessary for the implementation and the attacks on the OS and the architecture are provided. To thwart these attacks, several approaches are possible, especially by modifying the CV32A6 architecture, the Zephyr OS, or the compiler.

II. RISC-V

The RISC-V instruction set originated at the University of California, Berkeley, where Prof. Krste Asanović and graduated students Yunsup Lee and Andrew Waterman began developing the RISC-V architecture set. The name was chosen as a follow-up to UC
Berkeley's previous four RISC-like designs: RISC-I, RISC-II, SOAR, and SPUR. The goal of RISC-V is to provide an open-source ISA, as well as documentation, compilation tools, and support for FPGA and ASIC implementations. Today, the RISC-V community is worldwide and the interest for this instruction set and its applications is growing.

Indeed, the RISC-V ISA has several features that set it apart from other solutions. First, RISC-V is an open source project, which collects the details of commercial ISA designs and is suitable for hardware implementations. Also, RISC-V is intended to be a reduced instruction set, which allows the architecture to adapt to the desired project. This avoids having architectures that are too ambitious for their applications, whether for microarchitectures or FPGA/ASIC implementations. In both cases, RISC-V guarantees an efficient implementation. In addition, this instruction set supports 32- or 64-bit addressing systems. Secondly, massively parallel many core or multicore architectures are also supported, including heterogeneous multicore architectures.

Furthermore, the dense instruction set coding allows for variable length instructions, for a better optimization of the energy consumed and of the code size. Finally, it is also worth mentioning that this ISA is fully virtualizable and supports the 2008 IEEE 754 standard on floating point number representation [2].

III. CVA6

The CV32A6 architecture used in this competition is a six-stage pipelined, in-order processor architecture that implements the 32bit RISC-V ISA. It contains eight distinct modules, which are described in the following diagram:

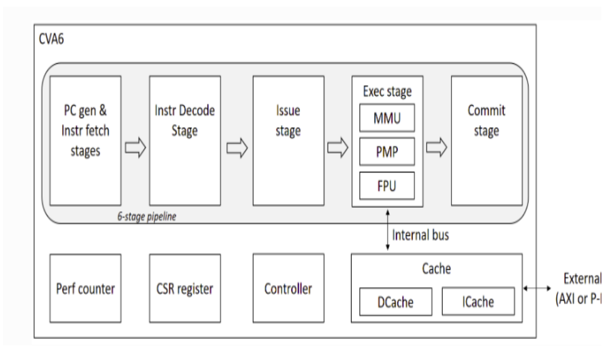


Figure 1 : *CVA6 architecture and pipelines*

The data cache is composed of 256 sets. Each set is composed of eight cache lines called “ways”. Each way contains 16 bytes according to the standard configuration of the data cache. Based on this information, we can conclude that the size of the L1 data cache is 256 (sets) × 8 (ways) × 16 (bytes in one way) = 32KB. The addresses in the CVA6 CPU are 32-bits. The cache addresses are physical (and not virtual). Therefore, the addresses’ sizes in the data cache vary with the cache’s size. The data and instruction caches are both linked to an AXI adapter module in order to connect them to a 64-bit AXI bus.

For the CVA6’s data cache, the physical addresses are 64-bit long and are composed of the following fields:
- 0 to 3: Offset => it specifies the location of a byte inside the way and thus enables to choose the appropriate data bank
- 4 to 11: Index => these bits allow to distinguish between the 256 available Sets
- 12 to 63: Tag => these 52 bits are used to represent the requested memory location

IV. Zephyr docker

The Zephyr Operating System is a popular security-oriented RTOS with a small-footprint kernel designed for use on resource-constrained and embedded systems: from simple embedded environmental sensors and LED wearables to sophisticated embedded controllers, smart watches, and IoT wireless applications. The Zephyr kernel supports multiple architectures, including:

• ARCv2 (EM and HS) and ARCv3 (HS6X)
• ARMv6-M, ARMv7-M, and ARMv8-M (Cortex-M)
• ARMv7-A and ARMv8-A (Cortex-A, 32- and 64-bit)
• ARMv7-R, ARMv8-R (Cortex-R, 32- and 64-bit)
• Intel x86 (32- and 64-bit)
• MIPS (MIPS32 Release 1 specification)
• NIOS II Gen 2
• RISC-V (32- and 64-bit)
• SPARC V8
• Tensilica Xtensa

Zephyr offers a large and ever growing number of features including:

- **Compile-time resource definition**

Allows system resources to be defined at compile-time, which reduces code size and increases performance for resource-limited systems.

- **Cross Architecture**

Supports a wide variety of supported boards with different CPU architectures and developer tools. Contributions have added support for an increasing number of SoCs, platforms, and drivers.

- **Memory Protection**

implements configurable architecture-specific stack-overflow protection,

V. Hope-RIPE attacks

Runtime Intrusion Prevention Emulator (RIPE) software was originally developed by John Wilander and Nick Nikiforakis. The RISC-V porting was developed by John Merrill.
RIPE is a free software under MIT License that provides a wide range of buffer overflow attacks in order to quantify the protection coverage of a given countermeasure. RIPE testbed is coded in C (backend) and Python (frontend).
It relies on five dimensions to evaluate intrusion prevention at runtime. The first dimension is the memory location of the buffer to be overflowed. The four supported buffer locations are Stack, Heap, BSS, and Data segment. The second dimension is the target code pointer, which determines the type of attack to perform. The two types of target code pointers are direct and indirect. The direct target code pointer points directly to the vulnerable function to be attacked, while the indirect target code pointer points to a memory location containing a pointer to the vulnerable function. The third dimension is the overflow technique. Overflow techniques include buffer overflow, format string overflow, integer overflow, lifetime overflow, and pointer type overflow.
A buffer overflow occurs when a program writes data to a buffer in memory beyond its allocated boundaries, overwriting adjacent memory locations. This can cause unpredictable behavior, such as crashes or the execution of malicious code injected into the buffer.

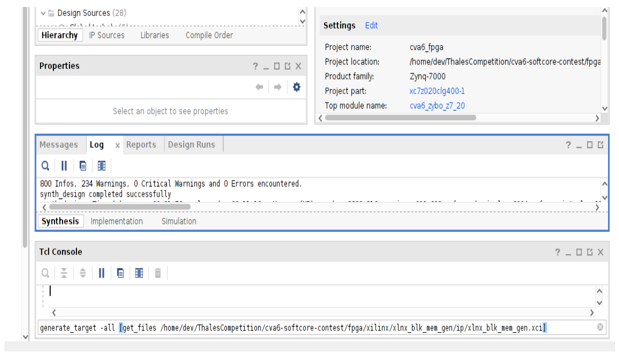
The fourth dimension of the RIPE testbed is the attack code, which allows users to test the effectiveness of attacks and countermeasures. There are several variations of shellcode available, including those with and without NOP sleds, as well as polymorphic NOP sleds. Return-into-libc and Return-Oriented Programming (ROP) attacks are also supported. Return-into-libc attacks use existing functions to carry out an attack, while ROP attacks use existing code snippets to create new functionality. However, the RIPE testbed has not yet implemented stack-pivoting techniques for ROP attacks. The purpose of this dimension is to provide users with a realistic environment to test their attack and defense skills.

The fifth and final testbed dimension in RIPE allows a user to choose from several functions that are commonly abused in buffer overflow attacks. These functions include memcpy(), strcpy(), strncpy(), sprintf(), snprintf(), strcat(), strncat(), sscanf(), fscanf(), and a custom "homebrew" function that is a loop-based equivalent of memcpy.

Although some of these functions are designed to prevent buffer overflows by taking the target buffer size into account, there are known caveats that can cancel out this protection. For example, the parameter "n" in strncpy() means total buffer size but for strncat() it means remaining buffer space [4]. Additionally, if the value of "n" is undefined due to a NULL value in the length calculation, strncpy() can allow for buffer overflow, as shown in CVE-2009-4035 [3].

VI. Development tracks:

With the Tcl command, we have generated the different IPs of the CV32A6 architecture.



Project name: cva6_fpga
Project location: /home/dev/ThalesCompetition/cva6-softcore-contest/fpga
Product family: Zynq-7000
Project part: xc7z020clg400-1
Top module name: cva6_zybo_z7_20

```
800 Infos, 234 Warnings, 0 Critical Warnings and 0 Errors encountered.
synth_design completed successfully
```

```
generate_target -all [get_files /home/dev/ThalesCompetition/cva6-softcore-contest/fpga/xilinx/xlnx_blk_mem_gen/ip/xlnx_blk_mem_gen.xci]
```

Figure 2 : Generating CV32A6 IPs

After generating the IPs, we have synthesized and implemented the CV32A6 architecture.

As a new development track, we have checked if Address space layout randomization is enabled for the Zephyr OS

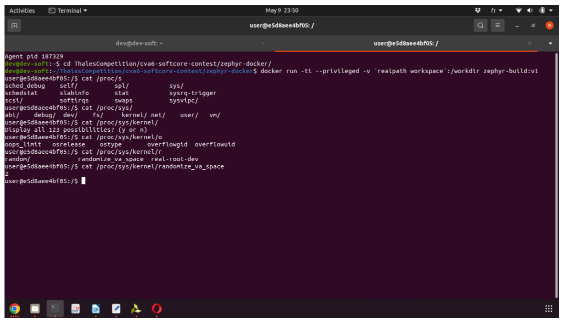
- **Address space layout randomization** (ASLR) :

is a technique that is used to increase the difficulty of performing a buffer overflow attack that requires the attacker to know the location of an executable in memory.

The possible values for "randomize_va_space" are:

- 0 : ASLR is disabled.
- 1 : ASLR is enabled for libraries, but not for the main executable.
- 2 : ASLR is fully enabled for both the main executable and libraries.

In our case the value of randomize_va_space being shown as "2" in the output of the cat command indicates that Address Space Layout Randomization (ASLR) is enabled on your system.



```
Agent pid 187329
dev@dev-soft:~$ cd ThalesCompetition/cva6-softcore-contest/zephyr-docker/
dev@dev-soft:~/ThalesCompetition/cva6-softcore-contest/zephyr-docker$ docker run -ti --privileged -v `realpath workspace`:/workdir zephyr-build:v1
user@e5d8aee4bf05:/$ cat /proc/s
sched_debug    self/          spl/           sys/
schedstat      slabinfo       stat           sysrq-trigger
scsi/          softirqs       swaps          sysvipc/
user@e5d8aee4bf05:/$ cat /proc/sys/
abi/    debug/  dev/    fs/     kernel/ net/    user/   vm/
user@e5d8aee4bf05:/$ cat /proc/sys/kernel/
Display all 123 possibilities? (y or n)
user@e5d8aee4bf05:/$ cat /proc/sys/kernel/o
oops_limit   osrelease    ostype       overflowgid  overflowuid
user@e5d8aee4bf05:/$ cat /proc/sys/kernel/r
random/            randomize_va_space  real-root-dev
user@e5d8aee4bf05:/$ cat /proc/sys/kernel/randomize_va_space
2
user@e5d8aee4bf05:/$ 
```

Figure 3 : Address Space layout randomisation

Below are some development tracks that could be implemented to increase the security of our processor against buffer overflow attacks.

- **Non-executable memory** (NX):

is a security feature that prevents the execution of code from areas of memory that are intended to hold only data. This means that memory regions marked as non-executable cannot be used to store nor execute code, which helps protect against certain types of attacks such as buffer overflow exploits.

In traditional memory architectures, code and data are both stored in memory and can be executed as long as they are in a memory region that is marked as executable. However, this can be exploited by attackers who can trick a program into executing code stored in a data region of memory. This type of attack is known as a code injection attack.

NX memory helps prevent such attacks by marking certain regions of memory as non-executable, preventing any code from being executed from those regions. By default, most modern operating systems enable NX memory on systems that support it.

To test this countermeasure on board, we could configure the memory management unit (MMU): The CV32A6 architecture includes an MMU that can be used to set up memory protection. This can be done by setting the appropriate bits in the MMU configuration registers.

Then, we use a linker script to specify memory layout. In the linker script, we can mark appropriate memory regions as non-executable to enable NX protection. We should also make sure to enable NX protection in the Zephyr operating system.

- **Stack protector**, also known as stack canary, is a security mechanism used to prevent buffer overflow attacks on a computer's stack.

A stack protector works by placing a random value, called a canary, between the stack buffer and the return address. Before the function returns, the canary value is checked to ensure that it has not been overwritten. If the canary has been modified, the program can terminate before any malicious code can be executed.
The canary value is typically a random number that is generated at runtime and placed on the stack before the function is executed. The value is then checked at the end of the function to ensure that it has not been modified. If the value has been modified, this indicates that a buffer overflow has occurred and the program can terminate to prevent further damage.
Stack protection is a widely used technique to improve the security of software, and it is enabled by default in many modern compilers and operating systems.

In order to try this countermeasure on board, we could enable the stack protector flag in the compiler. We can enable stack protection by passing the "-fstack-protector" flag to the compiler. Then, we should link it with the runtime library. In CV32A6 architecture, we can use the newlib-nano library, which includes support for stack protection.
The CV32A6 architecture includes a hardware feature that can be used to automatically generate and check the stack canary value. To enable this feature, we need to set the appropriate bits in the "Status" register (CSRs). We can do this by calling the "csrsi" instruction with the "MSTATUS" CSR and the appropriate bit mask.[6]
Finally, we would need to launch another testbed of RIPE attacks to figure out the efficiency of this approach.

- **Data execution prevention**

DEP, is a software-based security feature that works in conjunction with hardware NX support. DEP is implemented in the operating system and provides additional protection against buffer overflow attacks by preventing code from being executed from memory locations that are not designated as executable.

To implement this countermeasure on board, we should first enable memory protection in Zephyr OS in the project config file. Then, we should declare and configure MPUs (Memory Protection Units) in the device tree and assign executable and non-executable memory regions.

VII. Sources

[1] « CV32A6 Subsystem — CVA6 documentation ». https://docs.openhwgroup.org/projects/cva6-user-manual/04_cv32a6_design/source/cv32a6_subsystem.html (consulté le 8 mai 2023).

[2] A. Waterman, Y. Lee, D. Patterson, K. Asanovic, et C. Division, « Volume I: Base User-Level ISA ».

[3]Gajdos, P., and Kornacker, C. Cve-2009-4035 xpdf: buffer overflow in fofitype1.https://bugzilla.redhat.com/show_bug.cgi?id=541614, December 2009.

[4]Howard, M. Evils of strncat and strncpy -answers.http://blogs.msdn.com/b/michael_howard/archive/2004/12/10/279639.aspx, December 2004.

[5] Data Execution Prevention, Microsoft, consulté le 08/04/2023, https://learn.microsoft.com/en-us/windows/win32/memory/data-execution-prevention

[6]CV32A6, Control Status Registers, OpenHWGroup, consulté le 07/04/2023 https://docs.openhwgroup.org/projects/cva6-user-manual/01_cva6_user/CV32A6_Control_Status_Registers.html

[7] Zéphyr project, consulté le 07/04/2023 https://docs.zephyrproject.org/3.1.0/introduction/index.html