ChuckNoRisc Results Paper

Gabriel Blanchet[1*] **Hippolyte Boyer**[1*] **Anne-Charlotte Vignon**[1]

[1] CentraleSupélec Rennes
gabriel.blanchet@student-cs.fr, hippolyte.boyer@student-cs.fr, annecharlotte.vignon@student-cs.fr

Abstract

This research paper presents the results of the ChuckNoRisc team at the 3^{rd} national RISC-V student contest aimed at protecting the Zephyr OS on a RISC-V core through the implementation of hardware and software protection mechanisms. The paper highlights the importance of cybersecurity in the context of Internet of Things (IoT) devices and the potential risks and consequences of a security breach.

The Zephyr OS is a real-time operating system designed specifically for resource-constrained IoT devices, making security a critical concern. The paper provides an overview of the Zephyr OS and its architecture, and describes the various security features built into the OS. The paper then details the hardware and software protection mechanisms implemented by the ChuckNoRisc team to enhance the security of the Zephyr OS on the CV326 core.

The results of the contest are presented and analyzed, highlighting the strengths and weaknesses of each technical approach and identifying areas for further research.

The paper also emphasizes the importance of hardware design in ensuring the security of operating systems for IoT devices. The contest provides an opportunity for student teams to explore novel hardware design approaches that can enhance the security of the Zephyr OS on a RISC-V core. The paper concludes by highlighting the potential for student-led innovation in this area and the need for continued research and development in cybersecurity to protect the growing number of IoT devices.

Introduction

We took part in the 3^{rd} national RISC-V student contest as the ChuckNoRisc team. The goal of the contest is to improve the capabilities of the processor core CV32A6. Each year a new problematic and topic is created. For the 2022-2023 edition, we had to think about the security of this processor core on the board : Zybo Z7-20 but also with real time operating system : Zephyr. The scope of the attacks was based on the RIPE bank for cyber attacks (Runtime Intrusion Prevention Evaluator). The RIPE bank attack is an open source intrusion prevention evaluator provide by Wilander and Nikiforakis. They distinguish every cyber attack by 5 axes : the technique (direct or indirect), the target, the function (memcpy, sprintf...), the location (stack or heap) and the attack code (ROP, ReturnIntoLibc...). With this open source tool, it is possible to create around 850 different attacks scenarios. For the contest, only 10 attacks have been selected. We focused on this sub-sample to set up adapted protections. And this is their description :

Attack n°	Technique	Attack Code	Target Code Pointer	Location	Function
1	Direct	Shellcode without NOP sled	Return Address	Stack	Memcpy
2	Direct	Shellcode without NOP sled	Stack pointer	Stack	Memcpy
3	Indirect	Shellcode without NOP sled	Stack pointer	Stack	Memcpy
4	Direct	Data only	Variable Leak	Heap	Sprintf
5	Direct	ReturnIntoLibC	Return address	Stack	Memcpy
6	Indirect	ReturnIntoLibC	Heap pointer	Heap	Memcpy
7	Indirect	ReturnIntoLibC	Vulnerable Struct	Heap	HomeBrew
8	Indirect	ReturnIntoLibC	Longjmp buffers	Heap	Memcpy
9	Direct	Return-Oriented Programming	Return Address	Stack	Memcpy
10	Direct	Return-Oriented Programming	Vulnerable Struct	Heap	sprintf

Table 1: Selected RIPE attacks for the contest

With the continuous evolution of computing systems, the need for versatile and efficient instruction set architectures (ISAs) has become paramount. An emerging actor in this field is RISC-V, an open-source RISC (Reduced Instruction Set Computer) ISA that has garnered significant attention from both industry and academia. RISC-V distinguishes itself by its modular and extensible design, providing the flexibility necessary to cater to a wide range of application domains.

The Ariane core processor CV32A6 is a 6-stage in-order processor implementing the ISA RISC-V 32bits (RV32). The CVA6 core is Following the interests of the IoT, with a strong modularity it is perfectly adapted to various applications. This processor is interfaceable with SoC,ASIC and FPGA. For the contest we have worked on the ZYBO-Z20 FPGA board from Digilent
This CV32A6 is fully compliant with RISC-V specifications.
This core being open-source, the whole source code is avail-

[*]These authors contributed equally.

able on the Github of openhwgroup. Written in systemverilog it is possible to modify it later.

The Zephyr RTOS (Real Time Operating System) is ported to the CV32A6 processor core. This OS (Operating System) is an open-source, lightweight, and modular real-time operating system designed for resource-constrained embedded systems. It offers a compact footprint, cross-platform support, and integrated security features, making it well-suited for Internet of Things (IoT) applications. For all those reasons it was pertinent to use it in this hardware contest. This paper aims to explore the modular options and applications of different protections in order to defend the entire project from the 10 attacks.

The contest have many challenges for students. It offers the opportunity to work on very actual and popular technologies. Nevertheless, the hardware security is relevant today and challenge us to not only think on the conception but also in the security and protection of our system. Our team is composed of hardware and cybersecurity students, it creates a lot of discussion and exchange of knowledge and opinions.

Preliminary study

Before any work we spent a long time studying the different possible and existing solutions on the different CPU and OS architectures.
Many solutions to protect the stack, the heap and different other types of attacks exist on x86, ARM and others architectures processors. So we had to work to understand their mechanism and if these protections were compatible with our architecture and OS. We first think about how we can enforce the protection of the memory, and the PMP (Physical Memory Protection) seem to be a really good choice to protect the access of the memory. But as we have a special version of the CV32A6 without MMU (Memory Management Unit) this solution was impossible to implement. This example shows the purpose of this long first work on the existing solutions. We finally retained different options which seemed to us exploitable and conceivable to set up.

Stack canary is a security mechanism that is used to protect against buffer overflow attacks in software. A buffer overflow attack occurs when an attacker overwrites the memory beyond the end of a buffer, which can allow the attacker to execute arbitrary code.
In a CPU with an operating system, the stack is a region of memory that is used to store local variables, function parameters, and return addresses. When a function is called, its local variables and parameters are pushed onto the stack. The return address is also pushed onto the stack so that the function can return to the correct location.
A Stack Canary works by inserting a small value, called the Canary, between the function's local variables and the return address on the stack. The Canary is initialized with a random value when the system is booting. The Canary is added into the stack when the function is called. When the function returns, the Canary is checked to ensure that it has not been modified. If the Canary has been modified, it indicates that a buffer overflow attack may have occurred and the program can be terminated or other appropriate actions can be taken.
Stack Canary is a simple and effective security mechanism that can be implemented at the software level by compilers or operating systems.
Control Flow Integrity (CFI) is a security mechanism that aims to prevent control-flow hijacking attacks in software. Control-flow hijacking attacks are a type of exploit where an attacker manipulates the program's control flow to execute arbitrary code or to bypass security checks.
In a CPU with an operating system, the control flow of a program is managed by the operating system's kernel. The kernel controls the execution of each process and manages the program's memory and other system resources.
Control Flow Integrity works by adding checks (called landing pad) to the program's control-flow graph to ensure that the program only executes valid control-flow paths.This is typically done by inserting checks before each indirect branch instruction to verify that the target address of the branch is valid.
For example, if an attacker attempts to modify the return address of a function to execute malicious code, the CFI mechanism will detect this modification and terminate the program or take other appropriate actions to prevent the attack.
CFI is typically implemented as a compiler-based defense, which means that it is integrated into the software development process. This allows CFI to provide strong security guarantees without requiring any modifications to the underlying hardware or operating system.
Overall, CFI is an important security mechanism that can help protect against control-flow hijacking attacks in CPU and OS environments. This solution combine both hardware and software implementation
Shadow Stack is a security mechanism that helps protect against control-flow hijacking attacks in software.
Shadow Stack works by maintaining a separate copy of the return addresses on the stack, called the shadow stack. When a function call is made, the return address is pushed onto both the main stack and the shadow stack. When the function returns, the return address is popped from both stacks and compared to ensure that they match. If the return addresses do not match, it indicates that the program's control flow has been hijacked and the program can be terminated or other appropriate actions can be taken.
Shadow Stack is implemented at the hardware level using processor extensions or modifications to the operating system's kernel.
With this different protection mechanism we want to stop a maximum of RIPE attack by protecting the stack integrity and the flow execution integrity.
We have chosen to implement both hardware and software solution, the first one with the stack canary to provide a strong protection against buffer overflow attacks on the stack. As this protection is not designed to protect against control flow attacks, we have also chosen to implement

the CFI protection to prevent the ROP and ReturnIntoLibC attacks.
With this choice, we want to break the attacks n°1,2,3,5,6,7,8,9,10 (refer to the table 1 on page 1), the 4 will not be prevented because it is not based on ControlFlow or buffer overflow attacks.

Methodology

As it is the third contest organized, we had to manage an existing code and implementations. Therefore, we decided to not modify any SystemVerilog module of Thales (Except for practicality issues that we will see later in hardware part). Our strategy was to create as many modular options as we can. In fact, with modular implementations it was helpful to know that the initial code could work at any time if we remove our new modules. Moreover, the other part of our strategy was to test very often our drafts and new implementations. Also, to figure it out that the software configurations were working and don't break any thing on the hardware part.

Software implementation

Stack Canaries

First of all, we began to implement solution to counter stack based attacks. So, we installed stack canaries.

Zephyr configuration Zephyr is a modular and open source RTOS. Therefore, we managed to learn which options could help us in the project configuration. Many options are already available and thanks to its open-source capabilities, we find an option very interesting on the Stack Canaries. The implementation of this protection in Zephyr involves inserting a random value, known as a canary, between the local variables and the return address on the stack. During runtime, Zephyr constantly monitors the integrity of the canary. If it detects any modification to the canary's value, it immediately terminates the program execution, preventing the exploitation of a potential buffer overflow vulnerability. This approach helps the security posture of Zephyr-based systems, safeguarding them against malicious attempts to manipulate the stack and ensuring the reliability and integrity of the operating system. In the build option of our Zephyr project we enabled the option **CONFIG_STACK_CANARIES** and the option **CONFIG_TEST_RANDOM_GENERATOR**. This second option is a prerequisite for the stack canary, because the canary value has to be create by a random generator.

Control Flow Integrity

The Control Flow Integrity interested us a lot. This defence mechanism could help us to protect all the branchs and paths of the execution. We imagined 2 different scenarios. The first one is to add a specific instruction (personalized landing pad) at the begin of every function, that we will check at every *call*, which is **JAL** instruction. The second one is to add an other specific instruction (personalized landing pad) just after a *return*, which is **JALR**, and to check it in the same way. If the instructions in these 2 scenarios are not the good one, we protect the CPU by stopping the current execution (Have look on the hardware implementation part). We created another functionality to enable and disable the CFI, we check the detection of a particular landing pad at the beginning of the main. The CFI is disabled after leaving the main function. Our objective is to defend the attacks based on the ROP (Return Oriented Programming) and the Return-IntoLibc techniques. The landing pad instructions are personalized NOP instructions (cf Hardware implementation).

Plugin GCC In order the to add the landing pads, we decided to create a GCC plugin which add instruction at compile time. A GCC plugin is an extension that we can add in the execution flow of GCC. It gives the possibility to manipulate the data in different parts of the GCC compilation. A plugin extension in GCC is typically introduced during compilation process. This compilation phase is responsible for translating the preprocessed code into assembly code. At this stage, GCC provides an option to load and utilize plugins, which are external modules that extend the compiler's functionality. When a plugin is enabled, it is integrated with the compilation process and it can modify the behavior or perform additional tasks during the compilation phase. This could include custom optimizations, static analysis, code transformations, or other specialized operations. The plugin interacts with the compiler by hooking into specific events or compiler-generated data structures, allowing it to influence the compilation process or access and modify code representations. We decided to add our landing pads as RTL (Register Transfer Language) instructions. Our plugin detects the beginning of a function and add the correct instruction : **addi x0,x1,3**, and it detects every return (Jalr) and add with the same method the instruction **addi x0,x1,5**. Finally, the last option we add to our plugin was the capacity to enable and disable the CFI with a landing pad at the beginning of the main : **addi x0,x1,2**.

Recompiling libraries We decided to implement this functionality in order to disable the CFI during the Boot of our project. If we want our CFI to work, all the functions from different libraries, have to begin with the correct landing pad. We didn't want to recompile all the libraries, only the libc, libc_nano and lib_a has been recompiled with our plugin. So, the CFI is protecting the core from the beginning of the execution of the main until the end.

Adding the plugin in the compiling chain Finally, we had to add our plugin in the compiling chain of our project. The given plugin is inserting our personalized landing pads at the 3 different places we expose :

- at the beginning of the main
- at the beginning of every function
- after every return (JALR instruction)

We didn't manipulate the configuration files of Zephyr (build.ninja) to add the option. Our strategy was to intercept the call to the **riscv_zephyr_elf_gcc** and just add our plugin thanks to the option **"-fplugin="**. To do this, we changed the name of **riscv_zephyr_elf_gcc** for **chuck_riscv_zephyr_elf_gcc** (it didn't change the initial

compiler at all). Then, we created a script file with the name of **riscv_zephyr_elf_gcc** that receive the chain of option for the compilation. We only add the option **"-fplugin="** with our entire path plugin, and finally we are calling the "true" compiler which is **chuck_riscv_zephyr_elf_gcc**. The provided compilation chain has not been recompiled, we only add our plugin without modifying anything.

During our study, we tried all of these protections and scenarios. Nevertheless, we had to remove the **JAL** detection from our final work and the main beginning detection, that was not on purpose anymore. It didn't work correctly and we didn't manage to fix the issues to make it usable and effective for the deadline. In the final version of the project, our CFI detects only one scenario : the detection of the **JALR** and it checks that we are not returning to the beginning of another function, because it is forbidden. More precisely, when we detect a **JALR** we are checking that the next instruction is not **addi x0,x1,3**, corresponding to the beginning of a function.

To conclude with the software implementations, we have enabled the stack canaries to counter the attacks on the stack integrity (attacks n°1,2,3,5,9 from the tab 1). We also developed the CFI software part, in order to defend the attacks on the flow integrity (attacks n°6,7,8,10 from the tab 1). The hardware implementation is giving more explanation on the RISC-V instructions and also about the concrete implementation in the core processor CV32A6.

Hardware implementation

RISC-V Registers

To make the CFI software part work, it is necessary to implement a hardware module that checks the integrity of the CPU execution flow.
For that we have to understand the basics of the RV32 ISA. In RISC-V we have a convention on the use of call registers that describe the use of each registers, some of them are essential to consider for the CFI.

- **x0** is a hard-wired register, this means that its value will never differs from 0, any value assigned to this register will become 0.
- **x1** and **x5** : x1 is the basic register to store the return address after a call, x5 is also used for special runtime library functions to store a return address.

RISC-V Instructions

Our CFI have to check where we go when we do a "call" and where we go when we do a "return". So we need to be able on a hardware point of view to detect this "call" and "return" instructions.

The call instruction in RISC-V is a pseudo instruction build with the **JAL** (Jump And Link) instruction and the destination register (rd) equal to x1 or x5.
JAL instruction describes the conditionnal and unconditionnal jump (the one on which we focus), it allows to jump in range of $\pm 1MiB$, this instruction is build as following :

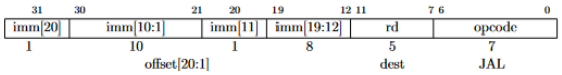

Figure 1: JAL instruction

One part of our CFI is dedicated to detect instruction following the format : JAL x1/x5
This instruction was not decoded as a "JAL" in the Ariane core, so the first modification of the core that we made is to add this opcode in the *Ariane.pkg* and also to modify the decoder to decode the JAL instruction. (This is the only major modification made in the source code of the core).

The return instruction in RISC-V is a pseudo instruction build with the **JALR** (Jump And Link Register) instruction and with the return address register (ra) equal to x1 and the destination register equal to x0.
The JALR instruction is used for indirect jump with a $\pm 2KiB$ range, and for return pseudo instruction.

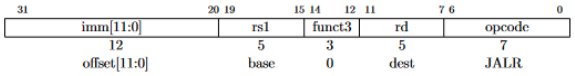

Figure 2: JALR instruction

For the CFI we have to check this instruction in 2 different scenarios, when the rd is x0 (for the return) and when rd is different of x0 (for a call).

The NOP instruction is an instruction used to modify the program counter by adding itself 4, without modify any other state of the CPU.
In RISC-V the *NOP* instruction is a pseudo-instruction build from the **ADDI** instruction with all its registers set to 0. So rd is equal to x0.
The ADDI instruction is used to make add and store operations immediately between 2 registers.

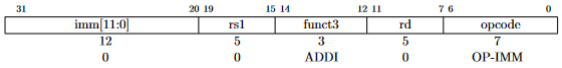

Figure 3: ADDI instruction

This instruction is usefull for us to create our landing pad, as GCC generates NOP by default, we have changed the rs1 register by loading it with the x1 register. If we translate it, we have an instruction which add immediately n to the return address and store the result in the x0 register, so this instruction is doing nothing as expected.
We have made this little trick (register x1) to avoid the GCC optimisation on NOP instruction (it translates it by li zero, n with n the value to add).
So we have 3 landing pads (1 after a call, 1 after a ret and 1 at the beginning of the main, as describe in the software section).

CFI module pipeline position

In order to control the flow integrity, we must check the instruction played by the CPU in order.
We spent some time looking for the right position for the module, initially we wanted to place it at the beginning of the pipeline at the decoder, this strategic position will have allowed us to stop the attack even before the instruction is played. However this CPU has branch prediction, this prediction can be wrong until the last moment and therefore have instructions in the instruction flow that will never be executed. With the pipeline architecture schema (4) we have seen that the impact on branch prediction is ended on the execute stage so we can check the flow of instructions from the execution stage and thus set up our CFI.

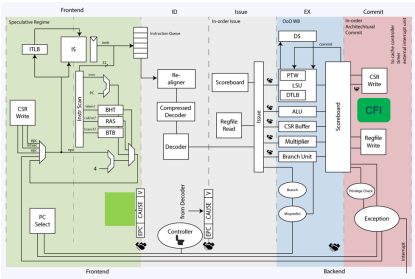

Figure 4: Ariane CPU Arch

We have connected our module to the commit module for the input, with $commit\_ack[1:0]$ and $commit\_instr[1:0]$ ports, and the output is connected to the csr_regfile module.
To check the instructions flow we have developped a simple 2 state FSM. The strategy is to track an anormal execution flow, and if it occurs it puts all the next instructions to the illegal state in the csr_reg_file module, with those conditions the CPU stops the execution of the program, and the attack is blocked.
We have not managed to make works the CFI in order to check the call instructions, so we only have a CFI to check the returns. We had troubles on the call detection with some functions of the libc that are blocking.
On the final version available on git we are checking only one landing pad.

Results

To have comparisons points, we will compare the results between different implementations to understand the impact of the different protections mechanisms.

Performance Results

	Original	Canaries	CFI	Final Version
Time Margin (ns)	28,678	28,678	26,846	26,846
Delta from ref (%)	0%	0%	6,388%	6,388%

Table 2: Time Margin Clock Path Group :clk_out_1_xlnx_clk_gen results

The critical path is one of the most important factor in FPGA, it describes the maximum frequency of our design, each modification by the hardware designer can impact this path. It is important to have a look in, after each implementation to make sure that we meet the deadline.

	Original	Canaries	CFI	Final Version
Cycle number	634021575	633811373	608033130	608033130
Delta from ref (%)	0%	-0,033%	-4,099%	-4,099%
Timing (s)	25,36086	25,352455	24,321325	24,321325
Delta Timing (%)	0%	-0,033%	-4,099%	-4,099%

Table 3: Performance BaseLine results

This tab shows us the performances of the CV32A6 core with a benchmark provided by Thales, this program aims to test the performance of the OS on the core.

	Original	Canaries	CFI	Final Version
LUT (%)	49,71	49,71	49,88	49,88
Delta (%)	0%	0%	0,342%	0,342%
Register (%)	21,47	21,47	21,52	21,52
Delta (%)	0%	0%	0,233%	0,233%
BRAM (%)	27,14	27,14	27,14	27,14
Delta (%)	0%	0%	0%	0%
DSP(%)	1,82	1,82	1,82	1,82
Delta (%)	0%	0%	0%	0%

Table 4: Size of the design on the FPGA results

Another key information in hardware design, is the size of the design, as we have seen before this CPU has a embedded aim, so it needs to be compact to fulfill all the criteria of the embedded world. With the following factor we can check the impact of our implementation on the size performance of the CPU.

Protection results

All this modifications have of course and fortunately an impact on the attacks, and here there are the results on the RIPE Attacks sub set :

Attack n°	Original	Canaries	CFI	Final Version
N°1	NO	**YES**	NO	**YES**
N°2	NO	NO	NO	NO
N°3	NO	NO	NO	NO
N°4	NO	NO	NO	NO
N°5	NO	**YES**	**YES**	**YES**
N°6	NO	NO	NO	NO
N°7	NO	NO	NO	NO
N°8	NO	NO	**YES**	**YES**
N°9	NO	**YES**	**YES**	**YES**
N°10	NO	NO	NO	NO
Total defended	0	3	3	**4**

Table 5: Prevented attacks results

With the different strategies put in place to protect the CV32A6 core, we managed to counter a total of 4 attacks out of the 10 provided by Thales.

Discussion

Interpretation of FPGA performances results

Critical path has been affected by our modifications as the tab 2 shows us. With the simple stack canaries implementation we haven't any time modification and this is logic as it is a pure software implementation. It can't modify the critical path. However, the CFI of course has an impact on the critical path by adding delay in the logic. We have a **6%** delay increase, this delta is not that important because as the design must operate at a minimum frequency of 25 MHz, with our modification the design can still run at 76,02 MHz against 88,32MHz with the initial design.

Performance BaseLine indicator in the tab 3, translate the impact of the modifications on the Zephyr OS performances. Stack canary should add a small amount of overhead to the program, in our case they have a negligible impact on performance. The reason for this is that the cost of checking the canary value is small compared to the cost of the function call and return operations that are already occurring as part of the program's normal execution.
Additionally, the use of stack canaries can help to prevent security vulnerabilities that could have much larger performance impacts if exploited by an attacker. By detecting buffer overflows upstream, stack canaries can prevent a program from crashing or behaving unpredictably.
In summary, while stack canaries may add a small amount of overhead to a program, the benefits they provide in terms of improved security generally outweigh any performance costs.
With the CFI plugin, we observe a performance boost in execution cycles and timing, this is due to the compiler used to build the GCC plugin. It includes optimisation as it is one of the last version of GCC. We haven't change the initial GCC so the *ripe_attack_generator.c* is still build with the Thales provided GCC. Moreover we observe that the plugin comes with some software optimisation.

Size of the design on the FPGA has been of course impacted by the CFI as it modify the design. The CFI only needs a little 3 states FSM, which provides really good results in term of LUT and register used. According to the tab 4 we have respectively an increase of **0,34%** and **0,23%** of resources. As we are not using BRAM or DSP of course this statistic haven't mooved. If we had implemented the shadow stack as a solution to protect the stack, we would have needed to use either a lot of BRAM or memory from the ZYBO-Z20. The space ratios would have been much more impacted.

Interpretation of protection results

During our work, we understood that we made a mistake at the beginning of the project. When we have studied the RIPE attacks, we only focused on their title without searching how they are really working in our case. In fact, the ROP and ReturnIntoLibC attacks are not generated as we were imagined them. They are a little bit customized. By definition, ROP attacks should jump into the middle of a function and only execute a part of them (gadgets). But we discovered that in the given implementation, there was no offset to allow the jump into the middle of the function. Concretely, the ROP attacks used in the contest do not jump into the middle of a function but it jumps at the beginning of them. Finally, we decided to finish the work we were working on, before trying to defend these new kind of attacks.

The 2^{nd} and 3^{rd} attacks are located on the stack, so this attacks should be prevented by the canaries. Nevertheless they are not, they are more or less broken, this is due to the way they are made. They are constructed by modifying the addresses offsets, but as the canaries are already modifying the addresses offset with a random value, the attack construction failed with the error *Error calculating size of payload*. Therefore, we did not consider foiling these 2 attacks. However, in a common context this is not a problem, the software architecture should not be based on modifying calling offsets. As the attacks are not played by the CPU, we more or less counter them.

Conclusion

Over this paper we were able to develop our researches and present our results. We succeeded in implementing two protection techniques, combining both hardware and software implementation. Our 2 protection methods allow us to counter 4 out of 10 attacks and to render 2 of them non-operable. We enjoyed this project, the team having various profiles and not all turned towards cybersecurity or hardware, we all learned a lot and increased our skills. We would have liked to develop the shadow stack, one of the main components that held us back was the depth of the stack and the fear of not having enough memory space to manage it.
This project showed us the importance and the difficulty that there is in the implementation of such low level protection. We were delighted to participate in an open-source project.

References

2017. The RISC-V Instruction Set Manual : Volume I: User-Level ISA.

2020a. Compilation avancée.

2020b. An introduction to gcc and gccs plugins.