IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2025, No. 1, pp. 227–250. DOI:10.46586/tches.v2025.i1.227-250

Call Rewinding: Efficient Backward Edge Protection

Téo Biton[1,2], Olivier Gilles[1], Daniel Gracia Pérez[1], Nikolai Kosmatov[1] and Sébastien Pillement[2]

[1] Thales Research & Technology, F-91767 Palaiseau, France
{firstname.lastname}@thalesgroup.com
[2] Nantes Université, CNRS, IETR UMR 6164, F-44000 Nantes, France
{firstname.lastname}@univ-nantes.fr

Abstract. The prevalence of memory-unsafe software prompts significant efforts by the research community to mitigate memory corruption bugs. This endeavor is crucial for safeguarding critical systems against security threats. Specifically, there is a focus to protect against code-reuse attacks through enforcing control-flow integrity (CFI). This paper introduces *call rewinding*, a novel microarchitecture-level mechanism for protection of return addresses. It is based on a property of the calling convention that is common to major architectures such as x86, ARM and RISC-V, which states that all return instructions transfer control to a valid call site. Call rewinding consists of jumping to the instruction preceding the return target for each return instruction and checking if the instruction at this address is a call or not. On systems equipped with return address prediction, a commonly employed optimization, the security check is performed only on mispredicted return addresses. The proposed protection mechanism demonstrates negligible impact on both area and performance. We implement call rewinding on the CV64A6, a RISC-V CPU with consequent branch prediction support. Our evaluation validates the effectiveness of call rewinding, both in bare-metal and in a Linux operating system (OS) environment. It triggers no false positives in bare-metal and is functional with the OS extended with a custom exception handler. Furthermore, our findings indicate that call rewinding successfully detects unauthorized return addresses, highlighting its potential as a reliable and efficient security mechanism.

Keywords: control-flow integrity · return-oriented programming · RISC-V · shadow stack · branch prediction

1 Introduction

While memory safe languages, like Rust [MK14], are getting increasingly popular nowadays, a lot of legacy software is written in memory unsafe languages. Over the years memory corruption exploits have received a lot of attention from both industry and academic communities. Initially, hackers injected easily detectable standalone payloads [One96], and efficient countermeasures were proposed, like data execution prevention (DEP) [Mic12]. However, complex exploits and especially code-reuse attacks (CRAs) like return-oriented programming (ROP) [RBSS12] and return-to-libc [Des97] have been used to bypass these defenses. CRAs reuse parts of the binary application for malicious purposes, making them hard to protect against.

Certain existing mitigation methods, based on compiler modifications or binary rewriting, aim to detect and eliminate code sections vulnerable to reuse by attackers [OBL$^+$10]. Techniques like address space layout randomization (ASLR) [Tea03] introduce code address randomness, making it harder for attackers to find the location of the code to be reused.

Received: 2024-07-15 Accepted: 2024-09-15 Published: 2024-12-09

Yet it has been shown that information leakage can nullify the shuffling, and exploits still remain achievable [SMD+13]. Additionally, control-flow integrity (CFI) [ABEL09] techniques prevent arbitrary control-flow transfers by insuring that an application follows its predetermined control-flow graph (CFG). The CFG is computed based on the source code and supposedly contains valid edges, i.e. valid flow of control between statements, that are checked for during runtime. While originally CFI—through the computation of the CFG—was exclusively software-based, many hardware solutions arose [dCV17]. This is especially true for shadow stacks [BZP19, DMW15, DHP+15, DBGJ19].

However, these approaches often come at the cost of performance or complexity, reducing their overall usability [BCN+17]. Although the increased density of modern embedded chips allows more complex systems on chip to be designed, gate count and area are still important concerns in the industry. Increasing awareness about energy consumption makes this issue even more critical. Optimally protective countermeasures are often not implemented due to the required memory space or hardware resources, or the resulting slow-down. Hence, there is a need for efficient measures that bring a sufficient degree of security while seamlessly integrating into the system. Developing defenses in the microarchitecture participates to this effort, as it can reduce the cost in memory space and performance loss.

In this paper, we propose *call rewinding*, an innovative and low-cost countermeasure to protect backward edges in central processing units (CPUs), which does not require applications to be recompiled. It is based on a property of the calling convention common to major architectures (like x86, ARM and RISC-V), according to which all return instructions transfer control to a valid call site. Call rewinding consists of fetching the instruction preceding the return target address for each return instruction in order to check if the instruction at this address is a call and raising an exception otherwise. This instruction is not executed by the CPU. On processors equipped with a return address stack (RAS), only mispredicted returns need to trigger a check, as the RAS predictions are trusted.

To validate the approach of call rewinding, we focus on RISC-V based processors, taking advantage of the openness of its instruction set architecture (ISA) and application binary interface (ABI). The RISC-V ISA is an open-source architecture perfectly suited for experimenting innovative countermeasures. It is very popular in the research community for testing both attacks [JMA+20, BGPK24] and protections [HVW+21, DBGJ19]. Moreover, several CPUs are backed by the industry and on products, like the CV64A6 [ZB19] and Ibex[1]. Despite the inherent security advantages of RISC ISAs due to their smaller attack surface compared to CISC ISAs [KOAGP12], systems using RISC architectures, especially in critical areas like industrial control or cyber-physical systems, are sensitive to potential threats with significant consequences. The open nature of the RISC-V ISA offers security benefits by learning from past experience and allows for community review, fostering trust in the architecture. While RISC-V allows for the design of a wide range of cores, from simple digital signal processors (DSPs) to high-end CPUs, its most common usage nowadays is in embedded systems, which implies limited resources and high power-efficiency context. This is the specific challenge we address in this work.

We have deployed call rewinding on the CV64A6, a RISC-V CPU equipped with consequent branch prediction capabilities. Its ability to boot Linux facilitates testing the solution within an operating system (OS) environment. Consequently, our evaluation shows the effectiveness of call rewinding in both bare-metal configurations and when operating alongside an OS. It does not generate false positives when operating directly on hardware and remains operational even when integrated with an OS supplemented with a custom exception handler. Finally, our results indicate that call rewinding effectively identifies unauthorized return addresses, making it a dependable and effective security augmentation.

[1]https://github.com/lowRISC/ibex

Contributions. We summarize our contributions as follows:

- a novel, purely hardware-based method, called *call rewinding*, to dynamically detect invalid return addresses;
- its implementation on a popular RISC-V CPU, the CV64A6;
- a security analysis of call rewinding against a wide range of exploits;
- a proof-of-concept support for call rewinding in a Linux kernel;
- an evaluation of ROP gadgets reduction and performance overhead induced by call rewinding, confirming its capacity as a reliable and efficient security enhancement.

Outline. The remainder of this paper is organized as follows. Section 2 provides necessary background on ROP and shadow stacks. A study of related works is presented in Section 3 and further relevant concepts and definitions are presented in Section 4. Section 5 describes the proposed call rewinding technique. Its implementation is presented in Section 6. Section 7 presents the evaluation results. A general discussion on call rewinding is provided in Section 8 and potential threats to validity are addressed in Section 9. Finally, conclusions are given in Section 10.

2 Background

2.1 Return Oriented Programming

Injecting payloads in memory to be executed is one of the earliest techniques of attacks. Several countermeasures were introduced to mitigate this type of abuse like DEP or ASLR. CRAs were then introduced to bypass these protections. Instead of injecting specially crafted code, control-flow of a targeted program is abused to rearrange instructions in a specific order that performs a malicious exploit.

Return-oriented programming is a popular CRA used by hackers to divert the control-flow of an application and target assets. It is a generalization of return-to-libc that allows an attacker to perform a malicious return into a function in the C standard library. It works by chaining together short sequences of instructions already present in the application, called *gadgets*. ROP allows for a chain of several gadgets, only limited by the stack's depth. The term *return-oriented* comes from the fact that the attacker constructs a chain of gadgets ending by return instructions, each pointing to the next gadget. Shasham [Sha07] defined the term *gadget* and also showed for the application used in their work that a set of gadgets can be Turing-complete.

In return-oriented programming, the attacker gains control over the program's execution flow by manipulating the stack. The attacker overwrites the return address on the stack with the address of a gadget. When the function returns, control is transferred to the attacker's chosen address. To build a gadget chain, an attack must link all return addresses on the stack in a chain. Each return instruction transfers control to the next gadget in the chain. Executed in a crafted sequence, these gadgets can perform malicious actions. The attacker's goal can be for example to spawn a shell, or, use *mprotect* function to make a memory page writable and executable and then execute injected code.

In the context of a ROP exploit, the attacker must have access to the memory space. The attacker has to parse the binary and search for gadgets, i.e. sequences of instructions that end with a return instruction. Each gadget can perform one or more operations, like a memory access, an arithmetic operation or a system call.

For an attacker to launch the execution of a ROP gadget chain, at least two events need to happen:

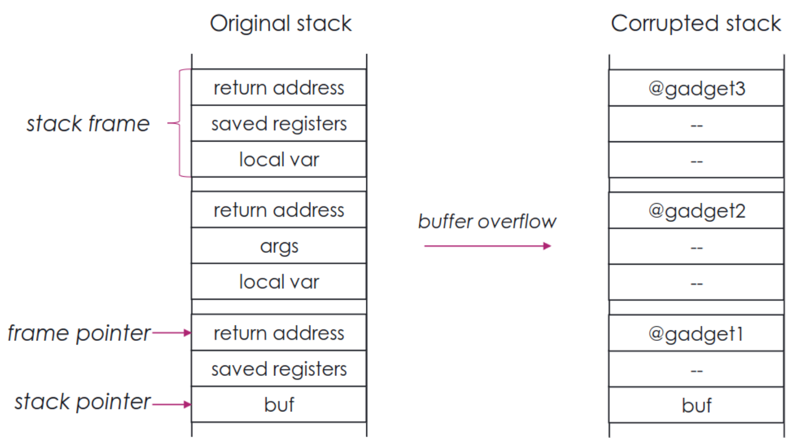

Figure 1: Stack content before and after exploitation of a software vulnerability.

1. The attacker must exploit a software vulnerability to smash the program stack and overwrite the return address to point to the first gadget of the chain;
2. The stack must be filled with the addresses of the next gadgets [RBSS12] so that they can be chained together, as shown in Figure 1.

2.2 Shadow Stacks

A *shadow stack* is a popular countermeasure to protect backward edges, i.e. return from functions, against abuses on return addresses. It serves the purpose of preserving a duplicate of the return address in a secure memory region, in addition to it being stored on the regular program stack. The primary goal of the shadow stack is to ensure the integrity of control transfers during returns, particularly in scenarios involving indirect (i.e. from a register) or direct calls, a concept commonly referred to as *backward-edge protection*. Different approaches exist to implement shadow stacks. Parallel shadow stacks are entire copies of the regular stack, while compact shadow stacks only save the return address. Example of a compact shadow stack is showed in Figure 2. Before returning from the called function, integrity of the return address is checked by comparing the two previously stored addresses. If there is a mismatch, the system reacts accordingly by raising an exception, which can eventually lead to a halt in execution. In some cases, only the return address from the shadow stack is used to avoid the performance penalty from the comparison [BZP19].

Shadow stacks are widely used in the literature [SAD$^+$16, DBGJ19, SGS19, DMW15, BZP19] and exist on several architectures to enforce a strict backward-edge policy. This protection is particularly efficient against ROP and return-to-libc attacks, that exclusively rely on backward-edge abuses. However, design constraints should be considered when implementing a shadow stack. To be fully operational, it must have a sufficiently large memory area and it must be context-aware, so that shadow stacks can be saved and restored when switching processes.

The RISC-V ISA has its own specification extension of a shadow stack [SS-23] under development. It introduces custom instructions and control and status registers (CSRs) to manage it, as well as general rules to control it in hardware. The architecture introduces a register called shadow-stack-pointer (`ssp`), which holds the address of the top of the active shadow stack. Similar to the regular stack, the shadow stack grows in a downward

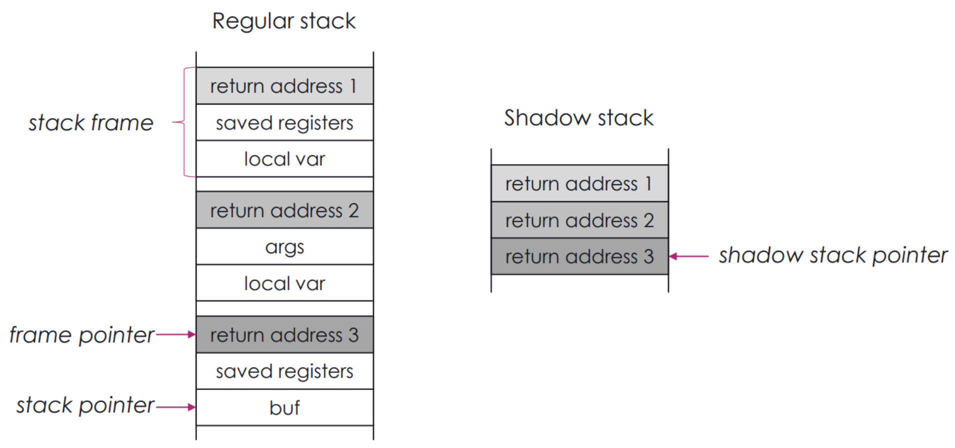

Figure 2: Regular stack and an associated compact shadow stack.

direction, progressing from higher memory addresses to lower ones. Implementing a RISC-V shadow stack in a CPU implies toolchain modifications to support the custom instructions and an OS able to manage the shadow stacks on a per process basis. This type of implementation can be impactful on the development process as it requires software, hardware and toolchain modifications.

3 Related Work

Existing countermeasures to protect against ROP are already based on the hypothesis that *all destinations of return instructions are preceded by a call*. This hypothesis is valid in all major architectures and it is specified in their ABI. Bin-CFI [ZS13] exploits this property to enforce a relaxed CFI policy by instrumenting and rewriting binaries, which is costly on the toolchain and on performance. It can be also verified when performing critical operations such as system calls. Taking advantage of the Last Branch Records (LBR) registers of the x86 architecture, kBouncer [PPK13] does exactly that by checking if all return addresses in the LBR registers are preceded by calls. However, this approach is limited by the finite size of the LBR of 16 entries and is only applicable to x86 processors.

Performance optimization mechanisms like return address prediction use the same hypothesis, by saving the return address at each function call. SIGDROP [WB16] presents a heuristic mechanism that leverages return address mispredictions and performance counters. It builds a heuristic model based on a misprediction counter and an instruction counter for a defined execution window. An alarm is raised when the number of mispredictions multiplied by a gadget size threshold is higher than the number of instructions executed in the window. While this approach is not intrusive and easy to implement, false positives or false negatives can arise depending on the defined threshold.

HAFIX [DHP+15] proposes an alternative to the shadow stack mechanism by keeping track of active functions. The active set is saved in memory, keeping track of procedure calls and their corresponding return address. It requires ISA modifications: instructions are inserted in the binary to activate and deactivate the labels. It also needs the active set to be saved between contexts, making it as complex as a shadow stack.

Branch Target Identification (BTI) [Sip23] is an ARMv8.5 extension that protects indirect branches and their targets, thus helping to limit control-flow abuses. With this extension, ARM8.5-A introduces Branch Target Instructions (BTIs), also called landing pads. The microarchitecture is adapted so that indirect branches are only allowed to target landing pad instructions. If the target of an indirect branch is not a landing pad, an exception is raised. While mostly used to protect forward edges, e.g. function calls, BTIs can also be used to protect function returns. A concrete comparison with ARM BTI on backward edges is provided in Section 8.4.

On the RISC-V architecture specifically, several propositions were made to address CRAs. Morpheus-II [HVW$^+$21] is a secure processor architecture based on code and pointers encryption. Unlike call rewinding, it tackles CRAs during the gadget search phase by making it seemingly impossible for an attacker to assess gadgets addresses. It also addresses side-channel attacks by renewing encryption keys at runtime. Despite its efficiency in terms of performance and area and its wide attacks coverage, Morpheus-II brings a significant complexity for its integration on existing platforms. FIXER [DBGJ19] is focused on the protection of backward and forward edges. It presents a coprocessor extension and custom instructions to enforce CFI on the platform. The new instructions are inserted by processing the binary, marking a strong impact on the toolchain and a significant cost in hardware resources.

An implementation close to call rewinding was mentioned in an earlier patent [PSS14]. The core of this work is to allow execution in a protected mode where the return address pushed on the stack is the address of the call instruction instead of the address of the following instruction, so that it can be decoded when returning from a function. Execution is then resumed from the expected address. Their approach differs from ours in two ways. The main one is that call rewinding only modifies the semantics of the return instruction and does not interact with program data on the stack. Legacy applications that read data from the stack, specifically the return address, will be affected. Additionally, call rewinding leverages branch prediction mechanisms to enhance performance without impacting system security, which is not mentioned nor possible with the patent's proposed implementation. Also, to the best of our knowledge, this idea was not carefully investigated, implemented and evaluated.

The present work continues earlier efforts to design and evaluate countermeasures and protection mechanisms against ROP attacks.

4 Context and Definitions

4.1 RISC-V Architecture

RISC-V is a relatively recent open-source ISA introduced in 2010 [RIS19]. It has gained a lot of popularity in the last few years due to its modularity. It provides a mandatory set of instructions, `I` for Integer instructions, and several optional extensions, each designated by a unique letter. The most common ones are integer multiplication and division (`M`), atomic operations (`A`), floating-point operations (`F` for single-precision, `D` for double-precision) and compressed instructions (`C`). The letter `G` describes the general purpose ISA that embraces `IMAFD` extensions. The most commonly used control-flow instructions are part of the base set and compressed extension set.

The RISC-V ISA defines 31 general-purpose registers. Some of them have specific use as per the ABI defined in the specification [RIS19].

The RISC-V calling convention (Table 1) defines the rules and conventions for how functions interact in a RISC-V architecture-based system. It defines the usage of registers for argument passing, and the handling of return values. In RISC-V, registers are designated for specific purposes, such as `a0` to `a7` for argument passing and `s0` to `s11` for saving values across function calls. The stack is employed for managing local variables and function call information, with the stack pointer `sp` tracking its top. Return values are typically stored in specific registers (`a0` for integers), and the convention distinguishes between callee-saved

Table 1: RISC-V first standard calling convention [RIS19].

Register	ABI Name	Description	Saver
`x0`	`zero`	Hard-wired zero	–
`x1`	`ra`	Return address	Caller
`x2`	`sp`	Stack pointer	Callee
`x3`	`gp`	Global pointer	–
`x4`	`tp`	Thread pointer	–
`x5`	`t0`	Temporary/alternate link register	Caller
`x6–7`	`t1–2`	Temporaries	Caller
`x8`	`s0/fp`	Saved register/frame pointer	Callee
`x9`	`s1`	Saved register	Callee
`x10–11`	`a0–1`	Function arguments/return values	Caller
`x12–17`	`a2–7`	Function arguments	Caller
`x18–27`	`s2–11`	Saved registers	Callee
`x28–31`	`t3–6`	Temporaries	Caller

and caller-saved registers, specifying which registers need preservation across function calls. The calling convention ensures compatibility and coherence in function interactions, facilitating interoperability between different software components on RISC-V platforms. Function calls can be performed by two instructions of the base instruction set, namely `jal` and `jalr`.

Register `x1` is commonly known as the *return address register* or `ra`. Its primary purpose is to hold the return address of a function call. When a function is called, the address of the instruction immediately following the call is stored in `ra`. This allows the program to know where to return once the function is executed. The standard way to return from a function call is to execute the pseudo-instruction `ret` that is actually a `jalr` instruction with `zero` as destination register and `ra` as source register.

The compressed counterparts of `jal` and `jalr`, i.e. encoded on 16 bits, are `c.jal` and `c.jalr` respectively. At the CPU level, these instructions are interpreted in the same way: meaning a compressed jump and link instruction will be evaluated as a regular jump and link.

The `ra` register is part of the calling convention, and it is a caller-saved register. This means that if a function uses `ra` for any purpose inside the function, it should save its original value and restore it before returning. The standard also defines `t0` as an alternate link register.

4.2 Return Address Stack

Branch prediction is a crucial unit of modern CPUs aimed at improving instruction execution efficiency [Sto07]. Conditional branches occur when the program encounters decision points, such as if-statements or loops, where the next instruction to execute depends on a condition. Traditional CPUs use a pipeline architecture. If a branch instruction is encountered, and the CPU does not know the outcome, there is a pipeline stall as it waits for the branch condition to be resolved. The primary goal of branch prediction is to minimize pipeline stalls by predicting the outcome of branch instructions before they are definitively known, allowing the CPU to speculatively execute subsequent instructions.

One of the prominent challenges in branch prediction lies in the occurrence of mispredictions, where the CPU anticipates the outcome of a branch incorrectly. This miscalculation leads to undesired consequences, primarily resulting in wasted CPU cycles spent executing the wrong speculative instructions. Flushing the pipeline of those erroneous instructions and refilling it causes a performance penalty. A challenge for designers is to reach a maximum level of accuracy on correctly predicted branches in order to limit performance impact.

A *return address stack* (RAS) focuses on optimizing the execution of return instructions, which are a special type of indirect jump. Modern processors incorporate a hardware-based stack that stores the return addresses of function calls. Whenever a call instruction is executed, its corresponding return address is pushed onto the stack. Subsequently, when a return instruction enters the pipeline, the processor seamlessly retrieves the next address by popping it off the stack, allowing the fetching of instructions from the associated address to continue without interruption. Within a few cycles, depending on the pipeline depth, the actual return address is computed, enabling the processor to recover from a branch misprediction if necessary. This mechanism is popular in high performance CPUs and can be seen in all major architectures: ARM Cortex A-53 [ARM18], Intel Core i7 6700 has an 8-entry RAS [PH17] and RISC-V CV64A6 [ZB19].

A RAS is conceptually similar to a shadow stack, in the sense that it securely stores return addresses. However, its purpose is completely different as a RAS is only used to optimize performance, i.e. mispredicts are tolerable and happen in legitimate applications in the following cases:

- **RAS overflows** caused by recursive function calls or deeply nested function calls;
- **Speculative execution** of a return instruction that is fetched but on a wrongly taken path;
- **Particular software constructs** like `setjmp` and `longjmp`;
- **Interrupts** that halt a function execution and start a specific routine;
- **Context switches** that interrupt a process control-flow, rendering the RAS outdated.

As mentioned earlier, the RISC-V specification also defines a RAS and the simplicity of RISC-V calling convention and of its instruction set allows for a straightforward management of it in hardware. It is accessible only with a push/pop mechanism. Only two instructions, part of the base instruction set, can affect on the RAS:

- `jal`, with the jump target encoded in the instruction (i.e. direct jump);
- `jalr`, with the target address in its source register `rs1` (i.e. indirect jump);

Table 2: Return address stack prediction hints encoded in the register operands of a `jalr` instruction [RIS19]. Below, *link* is true when the register is either `ra` or `t0`.

`rd`	`rs1`	`rd`=`rs1`	RAS action
!*link*	!*link*	–	None
!*link*	*link*	–	Pop
link	!*link*	–	Push
link	*link*	0	Pop, then push
link	*link*	1	Push

Hints as to the instruction's usage are implicitly encoded in its source and destination registers. For `jal` instructions, the return address is pushed onto the RAS when its destination register `rd` is either `ra` or `t0` (see Table 1). For `jalr` instructions, the RISC-V specification presents the effects on RAS as in Table 2. The *link* condition is true when the register is either `ra` or `t0`. When two different link registers (`ra` or `t0`) are given as `rs1` and `rd`, then the RAS is both popped and pushed to support coroutines.

5 Call Rewinding

5.1 Threat Model

We consider an attacker who seeks to exploit a software vulnerability in an application in user mode and to hijack the control-flow of an application, in order to execute arbitrary code. The attacker has full control of data memory once the vulnerability is exploited, and a read-only access to the software binary and shared libraries. It is assumed that ASLR is enabled on the platform, but the attacker is able to bypass it to obtain the required section addresses and find ROP gadgets [JMA+20]. The platform is a chip equipped with a RISC-V CPU, either in bare-metal or with an OS. Since code-injection attacks are outside the scope of this work, we assume that data memory is non-executable at the beginning of the exploit and that the attacker cannot inject a specially crafted program to initiate the attack. Also, as the proposed mechanism only protects the return address, it is not able to detect other CRAs such as jump-oriented programming (JOP) [BJFL11] or sigreturn-oriented programming (SROP) [BB14] that target other registers.

5.2 Strategy

Return-oriented programming attacks exploit the return address to link gadgets together. They exploit return instructions to jump from one gadget to the next. Gadgets are carefully chosen when crafting the chain to achieve a malicious goal and in most ROP exploits, gadgets entry points are at arbitrary locations in the binary. Hence the return instructions jump to invalid addresses. From the CFG point of view, this is equivalent to invalid backward edges.

In applications following the standard convention, all procedures or functions are executed in a similar way using *call* and *return* instructions. As seen in Section 4.1, call instructions `jal` and `jalr` store the return address in register `x1` (or the alternate link register `x5`). Since it is a caller-saver register, its value is also stored on the stack if needed to keep the linking information of nested functions. Call rewinding is based on the fact that all return addresses popped from the stack and loaded into `x1` must, when a `ret` instruction is executed, point to an instruction's address directly preceded by a call instruction.

Concretely, the proposed countermeasure against ROP attacks is to fetch the instruction at the address preceding the computed return address. The consequence is that one additional cycle is required before resuming execution at `ra`. The following actions are performed at the microarchitectural level when a call is rewinded:

t_0: The return instruction is executed as a jump register instruction with `ra` as source register, but instead of just fetching the instruction at `ra`, the CPU first fetches the instruction at address `ra` minus 4, called the rewinded address `rwa`;

t_1: The fetched instruction is scanned to determine its nature and then discarded;

t_2: Based on the scan result, execution resumes normally at `ra` or an exception is raised if the instruction at `rwa` is not a call.

Instruction scanning is a useful mechanism for CPUs that support branch prediction: instructions are predecoded early in the pipeline. When a control-flow instruction is detected by scanning, branch prediction modules are updated accordingly. Then, based on the encoded hints of the `jalr` and `jal` instructions, instruction scan is able to determine whether the fetched instruction is a call or not. At the fetch stage, a small module called *rewind* is added to handle the rewinded call. It registers the rewinded address `rwa` to know when to expect a call and receives information from instruction scanning to know

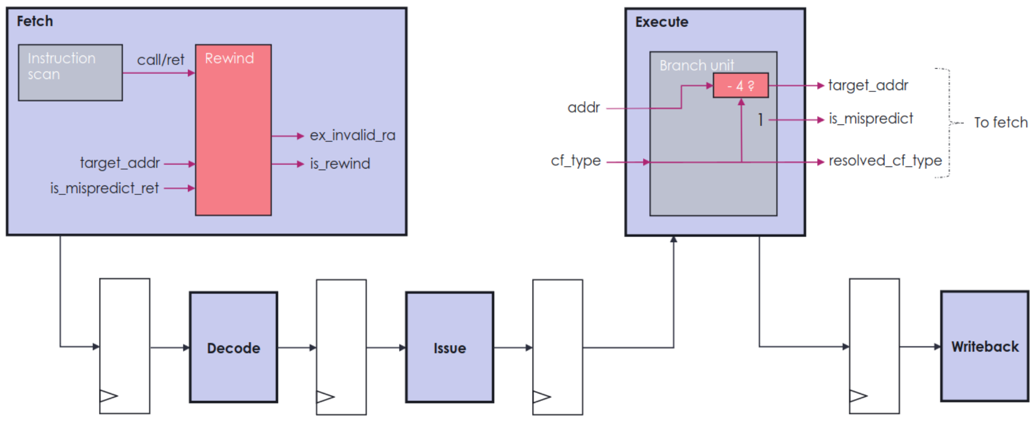

Figure 3: A standard 5-stage CPU architecture with call rewinding modifications shown in red. Only modifications in the branch unit of the execution stage to compute `rwa` and a special rewind module to handle it in the instruction fetch stage are required. There is no return address prediction and `rwa` is always jumped on.

the nature of the instruction. If it is indeed a call, execution resumes normally at address `ra`. If not, a custom exception `INVALID_RETURN_ADDRESS` with code 25 is raised by the *rewind* module. The default behavior is to stop the execution in bare-metal. In an OS environment, further actions towards the faulty application could be taken.

The crucial aspect of call rewinding is to identify the instruction fetched at `rwa` so that it does not go further in the pipeline. If not done correctly, the rewinded instruction will be decoded, executed and committed to the pipeline. This would have irreversible effects on a legitimate application as it would call and return from a function endlessly. Rewinded instructions are flushed from the fetch stage when identified.

Call rewinding is a microarchitectural modification only, i.e. it requires no binary processing. The pipeline modifications shown on Figure 3 are the only required CPU adjustments to protect a system against most ROP exploits. In the case of a platform with an OS, it must support the custom exception with a specific handler. This is addressed in Section 6.3.

While the presented approach is lightweight and does not impact performance drastically, it can be enhanced. ROP attacks have another low-level property that is true for all exploits. Indeed, linking gadgets with return instructions will trigger branch prediction mechanisms and pop addresses from the RAS. Yet these addresses were not pushed by a past procedure call and all gadgets will have mispredicted addresses. Correctly predicted return addresses on the other hand are deemed secure since the RAS is accessible by hardware only. Hence, call rewinding can benefit from branch prediction mechanisms aimed at optimizing return instructions. In the case where the CPU correctly predicted `ra`, then there is no need to fetch the instruction at `rwa`; integrity of the return address is insured by the fact it was pushed in the RAS. This approach has a positive impact on performance without degrading the security brought by call rewinding. The pipeline optimized with a RAS is shown in Figure 4. Mispredictions can still arise due to causes mentioned earlier, and then need to be checked.

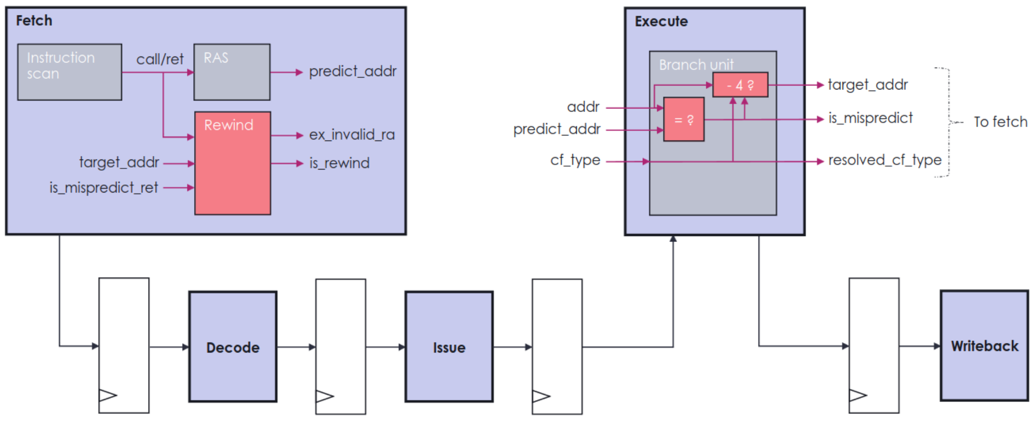

Figure 4: A standard 5-stage CPU architecture enhanced with call rewinding supporting return address prediction. Call rewinding modifications are shown in red. Here, `rwa` is computed only in the case of a mispredicted return address.

6 Implementation Details

6.1 CV64A6 Integration

The target we have chosen to experiment with call rewinding is the CV64A6 [ZB19]. The CV64A6 is an open-source RISC-V application core. It has a 6-stage pipeline and is a single issue in order 64-bit CPU. It has a wide support for branch prediction: both static and dynamic prediction are used as well as a return address stack. It supports the `IMAFC` extensions. It was chosen for its support of Linux, so that tests could be done in bare-metal and in an OS environment.

As defined in Section 5, the study delved into modifications within the fetch pipeline and branch unit of the CV64A6. These units can be seen in Figure 5 in the frontend and the execute stage. The modifications are as shown in Figure 4 since there is a RAS in the processor core. The CV64A6 has a lot of configurations based on hardware parameters. The RAS depth depends on the value of *RASDepth*. It corresponds to the number of available stack entries and must be different from 0 for the RAS to be instantiated. The predicted return address is hardwired to 0 in case it is not part of the design, meaning all return instructions will be mispredicted. We added a *CallRwEn* boolean parameter to enable or disable the call rewinding feature.

6.2 Compressed Instructions Support

The CV64A6, in its default configuration, supports the `C` extension. When executing a compressed instruction, the program counter (PC) is incremented by 2. In case of a compressed call instruction, the saved return address is then the current PC value incremented by 2 (instead of 4). Since the information regarding the size (16 or 32 bits) of the call instruction is unknown when executing a return, the choice is made to always compute `rwa` as being `ra` minus 4. This means that the instruction at address `rwa + 2` must also be checked by the rewind module. The alignment of the instruction's address does not impact the mechanism. In the base instruction set, the `jal` and `jalr` instructions must target an address aligned to a four-byte boundary. However they can target addresses aligned to a two-byte boundary if the platform supports the compressed instruction set extension.

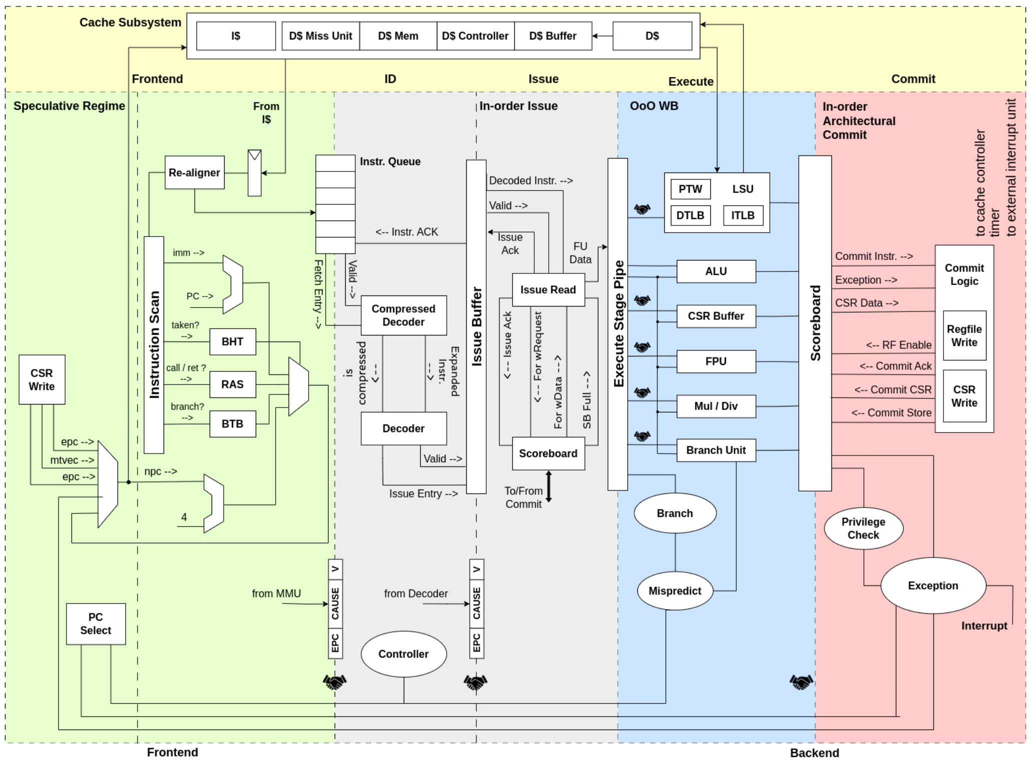

Figure 5: Overview of the CV64A6 architecture (block diagram from https://github.com/openhwgroup/cva6).

To comply with the `C` extension, the CPU must scan the instruction it receives from memory in 16-bit chunks. It can then identify compressed instructions. This means that there are four possibilities when scanning data received from the instruction bus, depicted in Figure 3. A compressed call instruction at address `rwa + 2` is valid (Figure 3b), however one at address `rwa` is considered invalid (Figure 3c). In legitimate applications, the return address cannot be four bytes away from its original compressed call site. In this case, an exception is reported. Additional logic is required to make sure the instruction at `rwa + 2` is correctly identified.

It has been shown that using compressed instructions can also increase the attack surface [JMA+20]. Overlapping instructions can form “unintended calls” or other instructions depending on their operands. Call rewinding is not affected by this as gadgets starting with an unintended call are considered invalid: the compressed call instruction will be seen on address `rwa` as on Figure 3c. When only using 32-bit instructions, there is no corner case, the instruction at `rwa` is either a call (Figure 3a) or not (Figure 3d).

6.3 Linux Support

Call rewinding is based on the hypothesis that applications respect the ABI. Running an OS with a strict security policy enforced in hardware can unveil false positives. To validate the approach, we run a minimal image of Linux 5.10-7 built using the Yocto layers for the CV64A6[2]. In order to run an OS the CPU must have support for supervisor-mode (S-mode), as is the case of the CV64A6. There is no guarantee that the OS respects the

[2] https://github.com/openhwgroup/meta-cva6-yocto

Table 3: Example of valid (a and b) and invalid (c and d) scenarios of instructions fetched at `rwa`.

rwa		rwa + 2	
01110010	11000000	00010000	11101111
72	c0	10	ef
`jal 25608`			

(a) Valid: 32-bit call instruction at `rwa`.

rwa		rwa + 2	
10010001	00010010	10011011	10000010
91	12	9b	82
`c.add sp, tp`		`c.jalr s7`	

(b) Valid: 16-bit call instruction at `rwa + 2`.

rwa		rwa + 2	
10011010	00000010	01100111	00001101
9a	02	67	0d
`c.jalr s4`		`c.lui a4, 0x3`	

(c) Invalid: 16-bit call instruction at `rwa`.

rwa		rwa + 2	
00000000	00100110	10000111	10010011
00	26	87	93
`addi a5,a3,2`			

(d) Invalid: non-call 32-bit instruction at `rwa`.

ABI when executing in-kernel operations. It might use `ra` in a way that is not specified by the ABI and trigger a call rewinding exception. Indeed, we found false positives within the boot sequence of Linux in S-mode. Since compromising of binaries used during the boot is an issue which must be addressed by secure boot and not by CFI methods, we deactivated call rewinding in S-mode. It is activated in user-mode (U-mode), for user applications, where lies the actual attack surface, as well as in machine-mode (M-mode), for bare-metal systems.

In U-mode, we discovered one false positive when returning from a signal handler, the `sigreturn` syscall. Its location would be written in `ra` and jumped on at the end of each handled signal, raising an exception as this construct does not respect the ABI. To solve this issue we wrote a handler for our custom exception `INVALID_RETURN_ADDRESS` that prevents the running application from crashing in case `ra` points to the `sigreturn` syscall. This solution is palliative to a programming issue. This does not weaken the protection against ROP exploits as shown in Section 7.1. SROP [BB14] exploits are still possible with call rewinding enforced, however vulnerability to SROP attacks is not increased.

7 Experimental Results

Our evaluations cover the following Research Questions (RQ):

RQ1 How effective is call rewinding in detecting illegal return addresses, and to what extent does it mitigate ROP exploits?

RQ2 What is the impact of implementing call rewinding on CPU performance, specifically in terms of computational overhead?

RQ3 How does call rewinding affect the utilization of hardware resources?

7.1 Security Evaluation

Experimental setting. RQ1 evaluates the actual security provided by call rewinding to a system, focusing on its ability to protect against ROP and return-to-libc attacks, which rely on return address overwrites. To validate our implementation of call rewinding, we ran 25 randomly generated ROP attacks using a RISC-V ROP generator[3] in a bare-metal

[3] https://gepgitlab.laas.fr/matana/bench/rop-generator

environment and in a Linux environment. The exploited vulnerability was an unchecked *memcopy* function. The attacks set up argument registers and perform a system call to execute a specific command. Additionally, a ROP chain was crafted to validate call rewinding against a realistic payload.

We also tested call rewinding in a more robust environment using the RIPE [MT18] benchmark, with Linux running on the CV64A6. This benchmark performs various types of buffer overflow attacks, but we focused on those that exploit the return address on the stack. The test was conducted on the Digilent Genesys 2 FPGA board, with the RIPE benchmark compiled with an executable stack and no stack protection.

Finally, a relevant metric is to evaluate the number of remaining gadgets available for attackers to build ROP attacks. To find gadgets, we use *RaccoonV*, a static binary analysis tool presented in [BGPK24]. This tool is able to extract both ROP and JOP gadgets from RISC-V binaries. We extended it to indicate *call-preceded gadgets* as well, as they are the only allowed gadgets with call rewinding. To observe the effectiveness of call rewinding against ROP exploits, we compare the number of ROP gadgets and call-preceded gadgets found in binaries.

Results. In the bare-metal environment, the CPU detected the invalid return address on the first gadget of the chain for each of the 25 tested ROP attacks. The crafted ROP chain also triggered an error on the first return instruction with an invalid address, demonstrating call rewinding's effectiveness in detecting and halting these attacks.

The RIPE benchmark provided a comprehensive evaluation, compiling 180 attacks ported to the RISC-V architecture, with 11 using the return address (`ra`) as the abused pointer. All these attacks targeting the return address raised an exception at runtime, successfully halting the corrupted application. This confirmed the robustness of call rewinding in a more complex environment.

Although call rewinding effectively mitigates common ROP attacks and basic buffer overflow attempts, it has limitations. Notably, in scenarios where a smart attacker overwrites `ra` through a buffer overflow, they may redirect it to an address following a call instruction within a code injection context. Consequently, while call rewinding provides valuable protection, it does not comprehensively address all potential attack vectors. Side-channel attacks that target the RAS [KKSAG18] are still possible: call rewinding is not suitable to detect such attacks. However, trusting the RAS predictions even in such cases does not actually decrease the security of call rewinding, since it can only contain addresses of valid call sites. Even if the prediction is a leftover of a speculatively executed call, call rewinding would not detect it as malicious even if the check was performed. Hence it does not diminish the base security brought by call rewinding to leverage the RAS for performance.

In a CRA context, it successfully detects illegal return addresses but it does not detect *call-preceded gadgets* [CW14]. The number of available gadgets is directly correlated to the ability for an attacker to build a gadget chain. Table 4 summarizes the effective reduction of exploitable gadgets on popular libraries, for gadgets of up to ten instructions. Call rewinding still offers a very valuable gadget reduction, only leaving a small set of exploitable gadgets in studied libraries. On average, gadget reduction is superior to 99%.

Observing the number of remaining gadget is a common approach to evaluate countermeasures that enforce CFI properties [ZS13, DZL16, HDZL17]. Moreover, the remaining gadgets have an average size of 5 instructions, meaning that they do several operations. They then have a higher number of side effects [KSN$^+$13], making them harder to use. Also, our study of the remaining gadgets in each libraries showed that they have limited capacities: they do not allow attackers to have system calls as part of their gadget chains, thus limiting the possible attacks to memory corruption, which are harder to set up with call rewinding.

Table 4: Number of available gadgets in relevant libraries compiled for RISC-V 64-bit.

Binary	ROP gadgets	Call-preceded gadgets	Effective reduction
libc	7,525	101	98.6%
libbluetooth	7,170	6	99.9%
libcrypto	17,733	138	99.2%
libgnutls	4,810	134	97.2%
libsqlite3	3,501	41	98.8%
libssl	3,132	20	99.4%

Table 5: CoreMarks/MHz score for the CV64A6 depending on relevant hardware parameters.

Configuration	RASDepth	CallRwEn	CoreMarks/MHz	% performance
Default	2	0	2.40	-
	2	1	2.39	-0.41%
No RAS	0	0	2.36	-
	0	1	2.35	-0.42%

Conclusion. Call rewinding effectively detects illegal overwrites of the return address in basic buffer overflow exploits and in the context of a ROP attack. While some gadgets are still available to attackers in our experiments, they provide a reduced set of operations and do not allow calls to kernel operations. Overall, call rewinding offers substantial security benefits by significantly reducing the number of exploitable ROP gadgets, although it does not address all potential attack vectors.

7.2 Performance Analysis

Experimental setting. The core concept of call rewinding is to execute return instructions by jumping to the preceding instruction to perform a security check. Although this instruction is not executed, fetching it costs at least one additional cycle, impacting CPU performance. To answer RQ2, we modified the CV64A6 processor with call rewinding in its default configuration and without return address prediction. Performance was measured by running CoreMark [GOL12]. Additionally, UCB's RISC-V benchmark[4] suite was used to provide insights into common applications. The main metric studied is the total number of CPU cycles used to execute applications.

Results. The CV64A6 showed reduced performance with call rewinding enabled, as expected. However, the performance gap is minimal, with only a 0.4% decrease in CoreMark score compared to the nominal score. The CoreMark scores are presented in Table 5.

To gain a broader perspective on the impact of call rewinding on common applications, we ran the UCB's RISC-V benchmark suite on the default CV64A6 with call rewinding both enabled and disabled. The metric studied was the total number of CPU cycles used to execute the applications. The overhead percentage, representing performance degradation, was calculated as the absolute difference between the two sets of clock cycle counts.

The overall execution time overhead induced by call rewinding was always less than 0.9%, with an average overhead of 0.1% based on the benchmarks run. These results are detailed in Table 6.

Conclusion. The integration of call rewinding into the CV64A6 processor introduces a minimal performance overhead. The slight performance degradation observed in both

[4]https://github.com/ucb-bar/riscv-benchmarks

Table 6: Evaluation of number of CPU clock cycles used by benchmark applications.

Benchmark	Default CV64A6	Default CV64A6 + CallRwEn	Performance Overhead
coremark	874,527	874,837	0.03%
dhrystone	223,550	224,072	0.23%
median	12,778	12,788	0.08%
mm	317,657	317,690	0.01%
mt-matmul	40,159	40,159	0.00%
mt-vvadd	55,036	55,048	0.02%
multiply	36,983	36,995	0.03%
qsort	224,724	224,733	0.00%
rsort	192,095	192,103	0.00%
spmv	59,829	59,840	0.02%
towers	13,341	13,457	0.86%
Average	-	-	0.12%

CoreMark and the RISC-V benchmark suite underscores that call rewinding's impact is negligible in practical scenarios. While the performance loss due to call rewinding is minor, it can be further optimized using a return address stack (RAS) without compromising the security mechanism. Profiling the system and adjusting the number of RAS entries can further mitigate any performance costs associated with call rewinding, ensuring efficient and secure processor operation. Increasing the number of RAS entries could improve performance, but the extent of improvement is highly dependent on the application.

7.3 Hardware Resources Utilization

Experimental setting. To answer RQ3, call rewinding is integrated in the CV64A6 and we measured its impact on hardware resources in a FPGA context. Synthesis for the Digilent Genesys 2 FPGA board was done using Vivado 2020.1. Since the implementation is adapted to other parameters of the CV64A6, we present results for different configurations.

Results. With the CV64A6 in its default configuration, lookup tables (LUTs) count increases by **0.25%** and flip-flops (FFs) count increases by **0.18%**. The synthesis results showing absolute values are presented in Table 7. Other configurations have been studied to show the raw complexity introduced by call rewinding. In case of no support for return address prediction (RASDepth = 0), the LUTs and FFs overhead is respectively of **0.06%** and **0.17%**. As described in Section 6.2, support for the `C` extension is responsible for most combinational resources required by call rewinding. With the CV64A6 synthesized with no support for compressed instructions, the LUTs count increases by **0.18%** and FFs count increases by **0.18%**.

Table 7: CV64A6 hardware resources utilization on Digilent Genesys 2 FPGA board in its default configuration and with call rewinding enabled.

Configuration	LUTs count		FFs count	
Default CV64A6	72,491	-	46,589	-
Default CV64A6 + CallRwEn	72,674	**+0.25%**	46,673	**+0.18%**

Conclusion. In the three configurations studied, the impact of call rewinding on hardware resources is negligible. The integration of call rewinding in the pipeline does not cause timing issues (i.e. reduce processor frequency) as the added logic is rather simple and not on the critical path of the CV64A6 design.

8 Discussion

8.1 Software Constructs

One strength of call rewinding is that it is built in respect to the calling convention, that dictates how software should use the general-purpose registers. The circumstances triggering return instructions to jump on the rewinded address `rwa` exist in legitimate applications. However, errors on invalid return addresses should only be raised in the context of a malicious exploit, specifically in a ROP context. Certain software constructs or events that can be seen in an OS environment could disrupt the expected behavior. We propose a discussion on the main ones we have identified.

Context switches. A context switch is a process by which the OS temporarily interrupts the execution of one thread, saving its current state and allowing another thread to resume execution. This operation is vital for multithreading environments, where multiple threads share a single CPU, and it ensures that each thread can continue without interference from the others. Before switching to a new thread, the OS saves the current context of the executing thread. This includes the values of processor registers (`x1` to `x31`), the PC, and other relevant information that defines the state of the thread. When a new thread is selected, its stored context is loaded into the processor. The processor resumes execution from the point where the selected thread was previously interrupted. The thread continues to execute as if there had been no interruption.

This is completely transparent to the CPU: it is just a sequence of store instructions followed by a sequence of load instructions. The RAS is completely unaware of these changes and will contain return addresses from the previous thread, i.e. it is not part of the information saved during a context switch. This will cause return mispredicts and jumps on `rwa` to perform the security check. However, the return address stored during the context switch should be valid because it comes from previous function calls, so there is no security issue during context switching.

Interrupts and exceptions. Interrupts are asynchronous events that can occur at any point during execution. They are mainly raised by system peripherals or by the debug module. Exceptions, on the other hand, are synchronous events raised when an attempt is made to execute an undefined instruction. For example, an *illegal instruction* exception is raised when the CPU is unable to decode an instruction. In the RISC-V architecture, when an exception or interrupt occurs, the virtual address of the current instruction is stored so that it can be executed on return from the trap routine. In a CPU with a pipeline stage such as the CV64A6, the stored PC corresponds to the instruction in the last stage of the processor (commit or writeback stage). The other stages of the pipeline are flushed to start the trap routine.

If an interrupt or exception occurs while fetching from the rewinded address `rwa`, the check may be interrupted. In this case, the *rewind* module must also be flushed: since execution restarts from the return instruction, the check will still be performed. An attacker cannot bypass the exception by triggering a software interrupt, because the `INVALID_RETURN_ADDRESS` exception has the highest priority. By flushing the rewinded address and ensuring that execution resumes from the return instruction, this type of abuse is mitigated.

`setjmp` and `longjmp`. These functions are specific to the C programming language and provide a mechanism for non-local jumps, allowing a program to jump back to a saved execution state from another point in the code. They are typically used to implement exception handling or to deal with complex control-flow scenarios.

The `setjmp` function is used to save the current execution state, including register values and the PC, in a `jmp_buf` data structure. In RISC-V, the return address is part of the saved data.

The `longjmp` function is used to jump back to a previously set checkpoint. It restores the saved execution state (stored in the `jmp_buf`) and continues execution from that point. The return address `ra` is then overwritten when the execution state is restored: it becomes the return address of the `setjmp` function. Since `ra` is overwritten with a return address preceded by a call, call rewinding will not raise an exception due to this construct. Experimentation with a small application has confirmed that `ra` is saved by the `setjmp` function and that no false positives are triggered by this software construct.

8.2 Supporting other conventions

Call rewinding enhances the detection of invalid return addresses, but it is only effective for code that adheres to the calling convention. Software that deviates from this convention will not function correctly on CPUs that implement call rewinding, as any alternative use of the register `ra` would trigger exceptions. The false positive discussed in Section 6.3 is an example of this issue. To support code that does not strictly follow the calling convention, we propose two solutions:

- **Extend the exception handler**: Adapt the exception handler to manage false positives specific to an application. This involves identifying potential false positive addresses in the binary and providing this information to the exception handler. However, if these addresses are frequently encountered during execution, this approach may degrade performance due to the interruptions caused by each false positive.
- **Control call rewinding via a Control and Status Register (CSR)**: Allow the operating system to enable or disable call rewinding before starting an application. This method requires ensuring that code running in user mode (U-mode) cannot alter this setting, thus maintaining system integrity.

These considerations apply to hardware shadow stacks as well, especially the RISC-V extension that only supports `ra` and the alternate link register `t0` as valid registers for the return address [SS-23].

8.3 Comparison with Shadow Stacks

Shadow stacks are the most common countermeasure to protect backward edges, however their implementation in systems is complex. Software approaches have an overhead of around 10% for traditional designs, and of 3.5% for a parallel design [DMW15]. Hardware approaches, on the other hand, have a significantly lower performance overhead: HAFIX [DHP+15] indicates an average overhead of around 2% while FIXER [DBGJ19] indicates one of approximately 1.5%. Shadow stack designs studied in [BZP19] show performance overheads varying from 1% to 50% on certain applications. Table 8 summarizes the overheads based on performance and code size for a sample of shadow stack schemes. However, other issues are raised by the utilization of shadow stacks. A shadow stack is specific to a particular thread, which means that in multithreading contexts, multiple shadow stacks must coexist. This has an performance implications when switching threads: information about the shadow stack associated with the current thread must be saved somewhere. This also has an increasing impact on memory as multiple pages must be allocated to store the shadow stacks. The `setjmp/longjmp` constructs often needs a special handling as information on the shadow stack must be part of the saved context. Finally, hardware implementations of shadow stacks often add custom instructions that have an impact on the ISA, hardware resources and on code size.

Table 8: Comparison of performance overhead between call rewinding and different implementations of shadow stacks.

Implementation	Average performance overhead
Call rewinding	0.12%
FIXER [DBGJ19]	1.5%
HAFIX [DHP+15]	2%
Parallel software design [DMW15]	3.5%
Traditional software design [DMW15]	10%

We argue that call rewinding is easier to integrate into CPUs and in the overall platform than a shadow stack with only a slight decrease of security (quantitatively, approximately 2% of remaining gadgets as shown in Section 7.1). Code size is not impacted at all by call rewinding, since the detection of illegal return addresses is done solely using hardware mechanisms. In terms of hardware resources, call rewinding requires very few LUTs (183 in absolute value) and very low additional registers as only the current return address is saved during the check sequence. Smashguard [OVB+06] requires an additional memory bank of 1 KB to store the shadow stack, as expected to match the requirement of storing many addresses. Relative overheads are given by HAFIX (1.0% LUTs, 2.49% FFs) and FIXER (2.9% area), but it should be noted that the hardware platforms are different. Call rewinding also causes a strictly lower average performance overhead than HAFIX, FIXER and software shadow stacks implementations presented in [DMW15]. It should be taken into consideration that hardware resources usage and performance results presented in other papers were obtained on different architectures, using specific tools and benchmarks. An undeniable comparison would be with a hardware shadow stack implemented on the same architecture and synthesized with the same tools or with a software shadow stack tested on the same platform. We are confident about the really low impact on area and performance of our solution and that it is competitive with the state of the art in these regards.

8.4 Comparison with Landing Pads

Landing pads are special instructions inserted to validate the target of an indirect jump. While their most common use is to protect forward edges, as is the case of the RISC-V landing pad extension, these instructions can also be used for backward edges. ARM BTI [Sip23], mentioned in Section 3, provides this functionality. Call rewinding is as secure as a landing pad implementation on backward edges: remaining gadgets would have to be preceded by a landing pad instruction. Introducing labels in the landing pads allows for a more fine-grained protection, as done by RISC-V landing pad extension for forward edges [SS-23]. We implemented a landing pad scheme similar to ARM BTI on the CV64A6 and compared it to call rewinding based on three metrics: performance (CPU cycles), hardware resources and code size. The performance and code size comparison is based on UCB's RISC-V benchmark, as was done in experimentations presented in Section 7.2. The overheads are evaluated compared to the baseline of CV64A6. The results are presented in Table 9.

Table 9: Comparison of performance overhead, hardware resources utilization and code size overhead between call rewinding and an implementation of landing pads on the CV64A6.

Metric	Call rewinding	Landing pads
LUTs	0.3%	1%
FFs	0.18%	0.4%
Performance	0.12%	0.65%
Code size	–	0.85%

The main advantage of call rewinding is its seamless integration into the execution flow, i.e. it does not need additional instructions to perform security checks. It uses the hints directly encoded in call instructions. Hence the code size shows no increase compared to landing pads, and call rewinding can be used with legacy applications. The landing pad implementation roughly has 1% LUTs and 0.4% FFs overhead, mainly due to the logic required to save the information that a landing pad is expected, to support execution of exception routines. Also, as call rewinding uses branch prediction on return instructions, performance is less impacted. It proves to be a strong alternative to landing pads on backward edges.

9 Threats to Validity

Threats to external validity relate to the sample of applications used to test call rewinding. The benchmarks run on the platform in a bare-metal environment did not cause any false positives, but there is no guarantee that all legitimate applications can run properly, as discussed in Section 8.2. Because call rewinding closes the gap between ISA and standard call conventions, it makes platforms more restrictive regarding authorized manipulation of the return address. We mitigated this risk by stress testing our solution in a Linux operating system environment, where we encountered only one false positive in user mode. Also, as mentioned in Section 7.1, there is a need for more appropriate metrics to measure the effectiveness of reducing the number of available gadgets.

10 Conclusion

Since the introduction of ROP attacks, numerous countermeasures have been proposed to safeguard the integrity of backward edges. Approaches such as shadow stacks, have been explored, with the major pain points being performance overhead, interrupt and context switch management, and a broader impact on development workflows.

Call rewinding emerges as a microarchitectural defense seamlessly integrated into the processor's pipeline. Its distinctive feature lies in its transparency to software developers, requiring no modifications to the toolchain. It is a purely hardware solution requiring no binary analysis or rewriting. Since it restricts the return instruction, the ISA is slightly modified: most standard applications are not impacted, however applications that do not respect the calling convention must be able to handle false positives. The only discernible effects are on area and performance, with the implementation in CV64A6 revealing hardware resources utilization overhead of less than 0.3% and a mean performance overhead of less than 0.1%. The core concept of call rewinding is moreover applicable to other architectures, e.g. ARM and x86.

While it is acknowledged that call rewinding does not address all ROP exploits, it provides a strong level of security, particularly beneficial for embedded systems that cannot afford the expense of fine-grained CFI. Our results demonstrate a significant reduction in the number of available ROP gadgets within a binary, leaving only call-preceded gadgets susceptible to exploitation. A method to evaluate the effective exploitability of this remaining set of gadgets could further support this reasoning. Furthermore, we argue that integrating it into CPUs is straightforward and easily configurable. It proves its utility by maintaining effectiveness across a wide range of scenarios, demonstrating equal security in both bare-metal environments and on top of an OS. The OS should be extended to handle the custom exception and potential false positives. The appropriate action taken by this custom handler is totally dependent on the operational context. We are convinced that it presents a promising alternative to shadow stacks in system architectures, depending on their available resources and security requirements. Other CRAs that do not corrupt the return address, like jump-oriented programming (JOP) or call-oriented programming (COP) still need to be addressed.

Acknowledgements

We thank the ANR which supports this work under the convention ANR-21-CE-39-0017.

References

[ABEL09] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information and System Security*, 13(1), nov 2009.

[ARM18] ARM. *Arm Cortex-A53 MPCore Processor Technical Reference Manual, revision r0p4*, 2018.

[BB14] Erik Bosman and Herbert Bos. Framing Signals - A Return to Portable Shellcode. In *2014 IEEE Symposium on Security and Privacy*, pages 243–258, 2014.

[BCN$^+$17] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys*, 50(1), apr 2017.

[BGPK24] Loïc Buckwell, Olivier Gilles, Daniel Gracia Pérez, and Nikolai Kosmatov. Execution at RISC: Stealth JOP Attacks on RISC-V Applications. In *Computer Security. ESORICS 2023 International Workshops*, pages 377–391, Cham, 2024. Springer Nature Switzerland.

[BJFL11] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, page 30–40, New York, NY, USA, 2011. Association for Computing Machinery.

[BZP19] Nathan Burow, Xinping Zhang, and Mathias Payer. SoK: Shining Light on Shadow Stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 985–999, 2019.

[CW14] Nicholas Carlini and David Wagner. ROP is still dangerous: breaking modern defenses. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, page 385–399, USA, 2014.

[DBGJ19] Asmit De, Aditya Basu, Swaroop Ghosh, and Trent Jaeger. FIXER: Flow Integrity Extensions for Embedded RISC-V. In *2019 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 348–353, 2019.

[dCV17] Ruan de Clercq and Ingrid Verbauwhede. A survey of Hardware-based Control Flow Integrity (CFI). *CoRR*, abs/1706.07257, 2017.

[Des97] Solar Designer. "return-to-libc" attack. *Bugtraq, Aug*, 1997.

[DHP$^+$15] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. HAFIX: hardware-assisted flow integrity extension. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, New York, NY, USA, 2015. Association for Computing Machinery.

[DMW15] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, page 555–566, New York, NY, USA, 2015. Association for Computing Machinery.

[DZL16] Sanjeev Das, Wei Zhang, and Yang Liu. A Fine-Grained Control Flow Integrity Approach Against Runtime Memory Attacks for Embedded Systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(11):3193–3207, 2016.

[GOL12] Shay Gal-On and Markus Levy. Exploring coremark a benchmark maximizing simplicity and efficacy. *The Embedded Microprocessor Benchmark Consortium*, 2012.

[HDZL17] Wenjian He, Sanjeev Das, Wei Zhang, and Yang Liu. No-jump-into-basic-block: Enforce basic block CFI on the fly for real-world binaries. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2017.

[HVW$^+$21] Austin Harris, Tarunesh Verma, Shijia Wei, Lauren Biernacki, Alex Kisil, Misiker Tadesse Aga, Valeria Bertacco, Baris Kasikci, Mohit Tiwari, and Todd Austin. Morpheus II: A RISC-V Security Extension for Protecting Vulnerable Software and Hardware. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 226–238, 2021.

[JMA$^+$20] Georges-Axel Jaloyan, Konstantinos Markantonakis, Raja Naeem Akram, David Robin, Keith Mayes, and David Naccache. Return-oriented programming on RISC-V. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 471–480, 2020.

[KKSAG18] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *Proceedings of the 12th USENIX Conference on Offensive Technologies*, WOOT'18, page 3, USA, 2018. USENIX Association.

[KOAGP12] Mehmet Kayaalp, Meltem Ozsoy, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Branch regulation: Low-overhead protection from code reuse attacks. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 94–105, 2012.

[KSN$^+$13] Mehmet Kayaalp, Timothy Schmitt, Junaid Nomani, Dmitry Ponomarev, and Nael Abu-Ghazaleh. SCRAP: Architecture for signature-based protection from Code Reuse Attacks. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 258–269, 2013.

[Mic12] Microsoft. Microsoft security toolkit delivers new bluehat prize defensive technology. https://news.microsoft.com/2012/07/25/microsoft-security-toolkit-delivers-new-bluehat-prize-defensive-technology/ accessed April, 2024, 2012.

[MK14] Nicholas D Matsakis and Felix S Klock. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014.

[MT18] John Merril and Arun Thomas. Hope-RIPE. https://github.com/draperlaboratory/hope-RIPE, 2018.

[OBL+10] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-Free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, page 49–58, New York, NY, USA, 2010. Association for Computing Machinery.

[One96] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[OVB+06] Hilmi Ozdoganoglu, T.N. Vijaykumar, Carla E. Brodley, Benjamin A. Kuperman, and Ankit Jalote. SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address. *IEEE Transactions on Computers*, 55(10):1271–1285, 2006.

[PH17] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. ISSN. Elsevier Science, 2017.

[PPK13] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, page 447–462, USA, 2013.

[PSS14] Baiju Patel, Vedvyas Shanbhogue, and Ravi Sahita. Returning to a control transfer instruction, 2014. US14/484,75.

[RBSS12] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.*, 15(1), mar 2012.

[RIS19] RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*, Dec 2019.

[SAD+16] Dean Sullivan, Orlando Arias, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, and Yier Jin. Strategy without tactics: policy-agnostic hardware-enhanced control-flow integrity. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, New York, NY, USA, 2016. Association for Computing Machinery.

[SGS19] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '19, New York, NY, USA, 2019. Association for Computing Machinery.

[Sha07] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, page 552–561, New York, NY, USA, 2007. Association for Computing Machinery.

[Sip23] Sipearl. Anti-Malware active protection on Arm64 Systems. 2023.

[SMD+13] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy*, pages 574–588, 2013.

[SS-23] SS-LP-CFI Task Group. *RISC-V Shadow Stacks and Landing Pads, Document Version 0.4.0*, nov 2023.

[Sto07] Jon Stokes. *Inside the Machine: An Illustrated Introduction to Microprocessors and Computer Architecture.* Ars technica library. No Starch Press, 2007.

[Tea03] PaX Team. PaX address space layout randomization (ASLR). *http://pax. grsecurity. net/docs/aslr. txt*, 2003.

[WB16] Xueyang Wang and Jerry Backer. SIGDROP: signature-based ROP detection using hardware performance counters. *CoRR*, abs/1609.02667, 2016.

[ZB19] Florian Zaruba and Luca Benini. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, Nov 2019.

[ZS13] Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, page 337–352, USA, 2013.