RetTag: Hardware-assisted Return Address Integrity on RISC-V

Yu Wang[1], Jinting Wu[1], Tai Yue[2,1], Zhenyu Ning[1,*], Fengwei Zhang[1]
{12032879,wujt,yuet2021}@mail.sustech.edu.cn,
{ningzy,zhangfw}@sustech.edu.cn
[1]*Research Institute of Trustworthy Autonomous Systems and COMPASS Lab, Southern University of Science and Technology*
[2]*College of Computer Science and Technology, National University of Defense Technology*

Abstract

Memory-corruption-based return address hijacking, such as Return-oriented Programming (ROP), is a prevalent attack technique that compromises the program's control flow integrity. So far, software-based defenses against these attacks either introduce heavy performance overhead or trade-off security for performance. Meanwhile, some hardware-assisted defense mechanisms are not practical for large-scale deployment due to additional requirements of hardware features and flaws caused by complicated design.

In this paper, we present RetTag, a hardware-assisted and crypto-based defense scheme on RISC-V architecture that leverages Pointer Authentication Code (PAC) embedded into the unused bits of function return address to ensure return address integrity. We extend RISC-V ISA with Return Address Authentication (RAA) instructions to generate the PAC efficiently. We integrate RetTag into the mainstream compilers GCC and LLVM to help developers transparently employ the defense and implement a prototype of RetTag on the Rocket emulator and FPGA development board to demonstrate its effectiveness by detecting various ROP attacks. Moreover, the performance evaluation shows that RetTag only introduces 0.11% performance overhead on NBench and 7.69% on Coremark.

CCS Concepts

• **Security and privacy → Software and application security**;
• **Software and its engineering**;

Keywords

Return Address Integrity, Pointer Authentication Code, RISC-V

ACM Reference Format:
Yu Wang[1], Jinting Wu[1], Tai Yue[2,1], Zhenyu Ning[1,*], Fengwei Zhang[1]. 2022. RetTag: Hardware-assisted Return Address Integrity on RISC-V. In *15th European Workshop on Systems Security (EUROSEC'22), April 5–8, 2022, RENNES, France*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3517208.3523758

EUROSEC'22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9255-6/22/04…$15.00
https://doi.org/10.1145/3517208.3523758

1 Introduction

Memory-corruption-based return address hijacking attacks, such as Return-oriented Programming (ROP) [10, 38], exploit memory vulnerabilities to hijack a program's control flow by overwriting function return addresses in memory. Attackers construct gadgets from existing code and chain them via return instructions by a carefully crafted stack. Since the gadgets are made by existing code, these attacks can bypass the traditional W⊕X defense policy, which prevents writable memory execution.

Stack Canary [16] and Shadow Stack [14, 18, 19, 43] are widely deployed methods to mitigate buffer-overflow attacks. Stack Canaries protect against sequential overwrites of return addresses by storing special values in stack frames. The program checks canary values before returns. However, these values are stored in insecure memory, where an attacker can read or overwrite them. Thus, it can be reliably bypassed [9, 23]. Shadow stacks store the return addresses in a separate, isolated region of memory that is not accessible by the attacker. Upon returning, the integrity of the return address is checked against the copy on the shadow stack. However, it is difficult to guarantee the security of the memory isolation in actual deployment. It has been proven that various technologies can bypass them in real-world scenarios [25, 30, 39].

Control-flow Integrity (CFI) [6] is another prominent defense technique. It guarantees a trusted process by limiting control flow to the original Control Flow Graph (CFG). Software-based CFI [12, 15, 17, 22, 31, 34, 44, 46] leverages compile-time instrumentation and runtime monitoring to achieve integrity. Unfortunately, due to the high performance overhead introduced by instrumentation and monitoring, researchers have to use coarse-grained approaches (e.g., adopting a more permissive policy for return instructions, rather than tracking the return addresses precisely) to trade off security for performance [12, 15, 22, 44, 46]. Moreover, even the fine-grained approaches are also proved to be vulnerable [34]. In contrast, hardware-assisted CFI [27, 35, 36, 45] offers a strong security guarantee at a negligible overhead. However, the hardware-assisted approaches have to face some limitations. First, the requirement of additional hardware components introduces implementation difficulty on Commercial Off-The-Shelf (COTS) devices [27, 36]. Second, the design of some hardware-assisted approaches [35, 45] is complicated, which makes them not practical to be deployed on COTS devices and increases the risk of potential vulnerabilities.

To overcome these limitations, we design a simplified and efficient return address authentication mechanism on RISC-V architecture [40]. RISC-V is an open and free Instruction Set Architecture (ISA). The open nature of RISC-V allows the developers to design and implement customized hardware features, which further helps

*Zhenyu Ning is the corresponding author.

to deploy our protective mechanism. There are two paged virtual-memory schemes in 64-bit RISC-V architecture: Sv39 and Sv48 [41] with supporting a 39-bit and 48-bit virtual address space, respectively. In other words, not all bits are used as a valid virtual address in a 64-bit architecture. Consequently, we utilize the unused bits of the pointer to store authentication code rather than memory, which improves the performance and reduces the attack surface. Moreover, it requires no additional hardware components, making it capable of large-scale deployment.

In this paper, we present RetTag, a hardware-assisted and crypto-based defense scheme that uses Pointer Authentication Code (PAC) embedded into the unused bits of function return address to protect return address integrity on RISC-V. Specifically, we extend RISC-V ISA with additional Return Address Authentication (RAA) instructions and efficiently authenticate function return addresses before being used as the returning targets. Furthermore, to transparently embed our defense scheme in the applications, we integrate RetTag into mainstream compilers including GCC and LLVM, and the defense is enabled automatically at the compile time. RetTag is deployed on the Rocket emulator and FPGA development board and tested with benchmarks to evaluate its effectiveness and efficiency, and the details will be discussed in Section 6.3.

In summary, we make the following contributions:

- We introduce a hardware-assisted defense scheme that uses PAC embedded into the unused bits of return address to protect return address integrity on RISC-V.
- We further integrate RetTag into mainstream compilers including GCC and LLVM, and the application developer can enable the defense scheme at the compile time.
- We test RetTag with 3 popular benchmarks, and the evaluation result shows that RetTag is capable of detecting various buffer-overflow attacks with a reasonable overhead.
- We open-source the code of RetTag on `github.com/Compass-All/RetTag`.

2 Related work

Shadow stack is widely deployed to enforce return address integrity. There are two shadow stack designs. Parallel shadow stacks [18] place shadow stack entries at a constant offset from the program stack. It's efficient but suffers from high memory overhead, compatibility problems, and low security. Compact shadow stacks [11] dedicate a general-purpose register to store the shadow stack pointer. Although it reduces memory overhead and shows reasonable performance, it still can be inferred the location of the shadow stack by exploiting memory vulnerabilities [28]. Hardware support for shadow stacks Intel Control-flow Enforcement Technology (CET) [1] uses a new extension to page permissions to protect a copy of the return address pointer. It offers greater security and performance. Nevertheless, the new page attribute might be modified like the similar approach in Data Execution Prevention (DEP) [26]. Besides, such a custom hardware mechanism incurs development and deployment costs. The main challenges of shadow stack are the memory overhead and requiring memory isolation. Compared to them, RetTag makes use of the unused bits of pointers to store return addresses' authentication codes without requiring additional memory or hardware resources. It effectively decreases the overhead and reduces the attack interfaces while facing the challenge of reuse attacks.

Cryptographic Control Flow Integrity (CCFI) [31] uses Message Authentication Codes (MACs) to protect control flow elements such as return addresses. However, it faces the problem of reuse attacks, where the adversary can reuse previously observed valid MACs. Compared to CCFI, RetTag provides better performance and is able to resist reuse attacks.

ARMv8.3 architecture introduces a dedicated Pointer Authentication (PA) [37] mechanism to protect the integrity of pointers. However, the COTS devices armed with the architecture are limited. To the best of our knowledge, the only COTS processors with ARMv8.3 are the most recent Apple processors, and the ARM Cortex series are not updated to ARMv8.3. Furthermore, current PA schemes are vulnerable to reuse attacks.

PARTS [27] implements compiler instrumentation that integrates PA-based defenses to protect all code and data-pointers at runtime. PARTS claims that a unique function id is used as part of the PA Modifier for each code pointer. Nevertheless, it is not completely unique in the implementation of PARTS, which exposes an attack surface for pointer reuse attacks. Compared to PARTS, RetTag improves the implementation of PAC generation (see Section 5.1.2). Moreover, RetTag is implemented with real hardware on an FPGA development board, while PARTS is only developed on a simulator.

PACStack [28] re-purposes the ARM PA instructions to create a MAC chain of function return addresses. They use a Chain Register (CR) to store PAC values generated from the previous CR values and the return address. Similar to PACStack, Zipper Stack [26] presents a lightweight mechanism to protect return addresses against reused attacks, which authenticates all return addresses by a chain structure using cryptographic MACs. However, it depends on the FILO sequence of the stack. When this sequence is destroyed, it needs to backup and restore the delicate register frequently. Compared to Zipper Stack, RetTag requires no additional operations.

3 Threat Model and Design Goal

Similar to other RISC architectures, RISC-V has a dedicated `ra` register that stores current return address. The `ra` register is generally set during regular and indirect function calls. Because the `ra` register is overwritten on call, non-leaf functions must store the return address onto the stack. While the return address is saved on the stack in a nested function, an attacker can exploit a stack overflow vulnerability to overwrite return address and subsequently redirect the control flow to attacker-chosen locations. Such control-flow attacks remain a prominent threat against computer systems. Some works [20, 24] have proved that they can implement ROP attacks on the new RISC-V architecture. To prevent these attacks, return addresses must be protected when stored on the stack.

In this work, we consider an adversary that performs attacks consistent with what we discussed before. The attacker corrupts return addresses on the stack by exploiting stack overflow vulnerabilities and subsequently redirects the program control flow to attacker-chosen locations. Since it is technically possible to control PAC generation from a high privilege level (i.e., kernel), we consider this is out of the scope for RetTag and limit the attacker to user

space. Furthermore, cryptanalysis attacks based on known PAC values and side-channel attacks are out of the scope as well.

Our goal is to thwart attackers who corrupt function return addresses on the stack to control the program flow. We specify the following design goals for our defense scheme.

- *G1:* Return address integrity: Ensure function return addresses on the stack remain unchanged.
- *G2:* Resistance against reuse attacks: Resist PAC reuse attacks.
- *G3:* High performance: Minimize performance and memory overhead.

4 Design

4.1 Overview

Figure 1 shows the overall architecture of RetTag. RetTag is a hardware-assisted and crypto-based defense scheme that consists of two parts, RetTag-enabled RISC-V platform (showing on the bottom of the Figure 1) and RetTag-enhanced compiler (showing on the top of the Figure 1). The gray area in the Figure 1 shows the modified component in our architecture design.

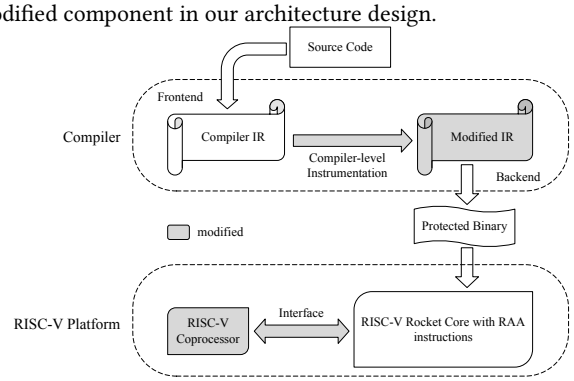

Figure 1: Architecture Overview of RetTag.

Among them, RetTag-enabled RISC-V platform is used to support RAA instructions. Specifically, we extend RISC-V ISA with dedicated RAA instructions that authenticate function return addresses. RetTag-enhanced compiler is responsible for generating the protected binary during compiling. Detailedly, it analyzes the Intermediate Representation (IR) and uses compiler-level instrumentation to transparently add RAA instructions when return address is saved into or restored from the stack.

4.2 RAA instructions

We use PACs as the ciphertext of the return addresses to protect them. For creating and verifying PACs efficiently, we extend RISC-V ISA with RAA instructions, which are a set of `pac` and `aut` instructions. Among them, `pac` is an instruction that creates a PAC of the return address when the protected program calls a function. `aut` is an instruction to calculate a new PAC and verify it at the end of the function call. If the new PAC matches the original one (i.e., verification is successful), it will delete the PAC and subsequently function returns as expected. Otherwise, it will trigger an interrupt and terminate the process execution.

Figure 2 shows the generation and storage of PACs in RetTag. For the return address, `va_size` is the valid virtual address size, which depends on the paged virtual-memory schemes. RetTag calculates the PAC over the virtual address, a 128-bit key, and a 64-bit modifier that is used as a tweak. After generating the PAC, RetTag stores it on the top `unused bits` of the return address.

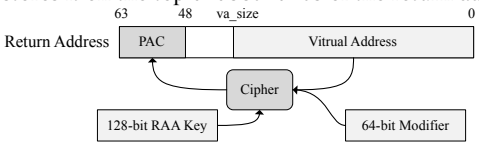

Figure 2: Generation and Storage of PAC.

Generation. As mentioned in [27, 28], the attacker can perform PAC reuse attacks by substituting an authenticated pointer with another one using the same modifier. Thereby, a unique PA modifier can effectively prevent authenticated pointers from being arbitrarily interchangeable with a malicious one. According to these, RetTag generates PAC by using 1) the value of return address, 2) a 128-bit RAA key and 3) a 64-bit modifier, which are shown in Figure 2. The RAA key is fixed in read-only memory by vendors and protected by hardware. Thus, the attackers cannot obtain the key. Furthermore, the key is unique among different devices. The 64-bits modifier contains two parts: the 32-bit SP value and the 32 most significant bits of the function-specific id. Specifically, the function id is a compile-time nonce, which is unique for each function. Note that only function id as the tweak is not enough. The same modifier will be generated when a function is called by different callers. Thus, the SP value is used as part of the tweak to distinguish callers with different stack layouts effectively. The unique modifier further defends against the PAC reuse attacks.

Storage. For storing PAC securely and efficiently, we use the `unused bits` of the return address to store PAC. There are two advantages of using unused bits rather than some other technologies (e.g., Shadow Stack). First, compared to shadow stack, storing PAC in the unused bits will not introduce any additional memory requirements, reducing the attack surface. Moreover, doing operations on unused bits is reached by accessing the registers. That means that reading and writing PAC to unused bits introduces less performance overhead than some memory-based technologies. As PAC is stored in the unused bits of the return address, the security provided by PAC is partly affected by the size of the unused bits. A longer PAC provides a stronger security guarantee since it increases the difficulty of brute-forcing the PAC. For supporting both two paged virtual-memory schemes, we use the upper 16 bits of the return address to store PAC. Theoretically, the attacker with probability p needs $\frac{\log(1-p)}{\log(1-2^{-16})}$ guesses, which provides adequate protection for Sv39 or Sv48. Additionally, the size of the PAC can be extended to 25 bits at most on Sv39 to guarantee more security.

4.3 Compiler-level Instrumentation

In RetTag, when function return addresses are pushed into or restored from the stack, the program should be instrumented with the aforementioned RAA instructions to create or authenticate PAC correspondingly. A `prologue` is a set of instructions that prepare the stack and store the return address on the stack at the beginning of a function. In contrast, an `epilogue` is a set of instructions to restore the stack and registers at the end of a function. Therefore, RetTag inserts the `pac` instruction to create PAC before the return address is pushed into stack in function prologue and `aut` instruction to verify PAC after the return address is restored from the stack in

function epilogue. In this case, we ensure the return address is authenticated when being used as the target of a function return.

```
Function:
Prologue:
  lui    tmp, func-id[0:19]
  addi   tmp, tmp, func-id[20:31]  ① get 32-bit function-id
  slli   tmp, tmp, 0x20
  add    tmp, tmp, sp              ② get modifier
  pac    ra,  ra,  tmp             ③ PAC generation
  ...
Function body
Epilogue:
  ...
  lui    tmp, func-id[0:19]
  addi   tmp, tmp, func-id[20:31]  ④ get 32-bit function-id
  slli   tmp, tmp, 0x20
  add    tmp, tmp, sp              ⑤ get modifier
  aut    ra,  ra,  tmp             ⑥ PAC authentication
  ret
```

Listing 1: Instrumented Assembly Instruction by RetTag.

Listing 1 shows the instrumented assembly instruction by RetTag. When a function is called, PAC generation instructions are executed instantly in the function prologue. The 32 most significant bits of function-id and current stack pointer are used to compose a 64-bit modifier (①, ②). Subsequently, the program executes a `pac` instruction to generate PAC and store it in the unused bits of the return address (③). Thus, the return address tagged with PAC is pushed into the stack.

Before the function return, in the epilogue, the program regenerates the modifier (④, ⑤) and executes an `aut` instruction to authenticate return address (⑥). If the return address is modified by an attacker, the authentication will fail. Subsequently, a PAC comparison failure leads to program termination. It is noteworthy that not all returns need to be authenticated. For instance, in a leaf function, the value of `ra` register is not saved into the stack. Therefore, the program only executes RAA instructions when the `ra` register is pushed into or restored from the stack rather than emitting instructions in all returns.

4.4 Customized Coprocessor

In order to support the abovementioned RAA instructions, we design a coprocessor to handle our custom instructions. The coprocessor can assist a general-purpose processor in dealing with the processing work that can be offloaded. Our coprocessor consists of two modules: the encryption module and the control module. The control module controls the processing of the coprocessor, and the encryption module is responsible for encrypting and generating PACs. Figure 3 shows the Finite-State Machine (FSM) of the coprocessor, which includes 5 states: `idle`, `data-in`, `pac`, `aut` and `last`.

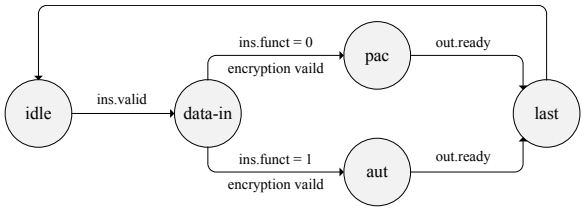

Figure 3: Finite-State Machine to Handle `pac` and `aut`.

(1) **idle**. The state is initiated to `idle` in the beginning. It stays `idle` until the main processor sends the custom instructions to the coprocessor. Then the coprocessor changes the state to `data-in`, which means preparing data for encryption.
(2) **data-in**. In this state, the control module accepts a 64-bit return address, a 128-bit key, and a 64-bit modifier in the registers and sends them to the encryption module. The encryption module sends ciphertext back to the control module and generates a valid signal when finished. In RISC-V instructions, the funct7 and funct3 fields combined with opcode describe what operation to perform. The instruction format of `pac` and `aut` are designed with the same opcode and different funct7 filed. When the encryption signal is valid, the state changes to `pac` or `aut` according to the `funct7` field of the instruction, which selects the type of operation.
(3) **pac**. In this state, the control module uses the 16 most significant bits of the ciphertext as PAC and inserts it into the unused bits of the return address. Subsequently, it writes the return address with PAC back to the `ra` register.
(4) **aut**. In this state, the control module regenerates PAC and verifies it. If the verification is successful, it removes the PAC and returns normally. Otherwise, it triggers an interrupt.
(5) **last**. After that, the state changes to `last`, which waits until the instruction is written back.

5 Implementation

5.1 Complier Integration

We integrate RetTag into the latest version of the mainstream compiler GCC [3] and LLVM [29] to build the RetTag-enhanced compiler. We add new passes to the optimizer and RISC-V backend to recognize and emit RAA instructions.

5.1.1 GCC Compiler Backend We add a new RTL pass and modify the pass that expands the prologue and epilogue patterns to emit RAA instructions automatically. Moreover, we add the corresponding machine description to guide RAA instructions generation. Moreover, we use a pseudo-random, non-repetitive sequence to generate unique function-specific ids during compile time.

5.1.2 LLVM Compiler Backend We insert RAA instructions in the function prologue and epilogue similar to the GCC compiler backend by using software APIs (i.e., `IRBUILDER` and `BUILDMI`) in frame lowering. The frame lowering is an interface that describes the layout of a stack frame on the target machine. We modify it to emit RAA instructions for authenticating return addresses. Additionally, some technologies (e.g., [27]) use the id of function prototypes as the modifier in the implementation (by calling `getType` [1]), which leads to duplicated modifiers in different function instances with the same function prototype. We improve the modifier generation process by calling `getFunctionNumber` that returns a unique id for the current function.

5.2 Hardware Extension

We develop a full-system FPGA prototype based on the Rocket-chip [7], which generates RISC-V Rocket core [5], a 64-bit 5-stage in-order synthesizable core. We use Chisel [8] Hardware Description Language (HDL) to configure the Rocket core [5] and customize a coprocessor consisting of control and encryption modules. Similar

[1] https://github.com/pointer-authentication/parts-llvm/blob/master

Table 1: The RIPE Test Suite [42] on RISC-V. In our evaluation, we perform various tests, including stack overflow, ROP gadget ,and ret2lib gadget. The results show that attacks are detected by RetTag.

Attack Dimensions	Attack Variant	Simple Description
Attack Location	stack	The attack location describes the memory section in which the buffer to be overflowed.
	heap	
	bss	
	data segment	
Target Code Pointer	return address	The target code pointer describes the code pointer to redirect towards the attack code.
Attack Code	return-into-libc	Redirecting the target pointer to the entry point of an otherwise inaccessible function.
	ROP	Redirecting the PC to an illegal jump target.
	shellcode	Performing a similar transfer of control flow.
Overflow Technique	indirect	Overwriting a target pointer with an integer value.
	direct	Overwriting a pointer to the target with a pointer elsewhere in memory.
Function Abused	memcpy	The function abused describes the vulnerable functions.

to [21], Rocket Custom Coprocessor (RoCC) communicates with Rocket core by RoCC interface. In addition,we choose AES128 [13] for the PAC encryption. As a widely implemented block-cipher encryption algorithm, AES performs well on a wide variety of hardware. After receiving a 128-bit RAA key and a 128-bit plaintext composed of a 64-bit return address and a 64-bit PA modifier, it generates corresponding 128-bit ciphertext. We truncate the upper 16 bits of the ciphertext to be the PAC.

5.3 Bootloader

We modify the Berkeley Bootloader (BBL) to support the custom instructions and RoCC interrupt. Specifically, we configure Machine Status Register (`mstatus`) and Machine Interrupt Enable Register (`mie`) to enable custom ISA and RoCC interrupt. Moreover, we add an interrupt handler in the Linux kernel (for the FPGA) and Proxy Kernel (PK) [4] (for the emulator) to terminate the process when the PAC authentication fails.

6 Evaluation

In this section, we focus on evaluating the security provided by RetTag and its performance overhead and resource cost. We address three questions in this section:

- RQ1. Can RetTag protect the return address integrity against tested buffer-overflow attacks?
- RQ2. Is it low-cost and lightweight on hardware?
- RQ3. How about the performance overhead?

6.1 Security Analysis

Benchmark We perform the security analysis on the RIPE test suite [42]. RIPE is a benchmark that comprises various buffer-overflow attacks, including code injection, return-to-libc ,and ROP attacks. By performing a wide range of buffer-overflow attacks and recording their success or failure, this benchmark can be used to quantify the protection coverage of the defense technology, which is suitable to evaluate the security provided by RetTag.

Configuration We run the test suite on the Rocket emulator running on Ubuntu 18. We use the PK to host statically-linked RISC-V ELF binaries, using BBL to boot up. All programs are built using LLVM 11.0 for the RV64GX (G for the general combination of standard instructions and X for customized instructions) architecture.

Results We implement the combinatorial set of buffer overflow attack forms built on four locations of buffers in memory, one target code pointers (i.e., return address), two overflow techniques, three variants of attack code being executed, and one function being abused. Table 1 shows the results of the experiments. We perform various attacks that corrupt return addresses in RIPE. The results show that all attacks, which can be recurred on the RISC-V platform, are detected and resisted successfully by RetTag. We can conclude that RetTag is capable of defending ROP attacks by enforcing return address integrity. Some attacks targeting other code pointers (e.g., function pointers) in the RIPE test suite can bypass the proposed protection mechanism. Moreover, we do not consider non-control data attacks, such as Data-oriented Programming (DOP) attacks, which can influence program behavior without modifying code pointers. Although RetTag can't defend against these attacks currently, we can extend protection all code and data pointers with more instrumentation in future work.

Table 2: Hardware Resource Cost of RetTag.

	Whole Systems		Rocket Cores	
	LUT	Slice Registers	LUT	Slice Registers
Without RetTag	58,483	29,351	31,993	14,028
With RetTag	59,213	30,015	32,744	14,691
%	+0.8%	+0.16%	+0.37%	+0.16%

6.2 Hardware Resource Cost

Configuration To evaluate the hardware resource cost of RetTag, we instantiate the original and the modified RISC-V Rocket Cores and synthesize them on the Xilinx Kintex-7 FPGA KC705 evaluation board using Vivado 2018.3. RetTag is configured with a 16KiB 4-way L1 instruction cache, a 16KiB 4-way L1 data cache, a 32-entry instruction TLB, and a 32-entry data TLB. Moreover, we integrate the above Rocket Cores with a 1GiB DDR3 and a 128KiB boot ROM.

Results Table 2 shows the hardware resource cost of systems without and with RetTag. Whole Systems refer to the hardware resource cost of whole systems with a Rocket Core and peripherals. Rocket Cores refer to the hardware resource cost of the out-of-context

synthesis of Rocket Cores. Table 2 shows the utilization percentage for slice look up tables (LUT) and slice registers. The number of extra slice LUTs and slice registers are 0.8% and 0.16% in the Whole Systems, 0.37% and 0.16% in the RISC-V Rocket Cores. The result shows the hardware resource cost of RetTag is low (less than 0.8%), which indicates RetTag is adaptable for real-world systems.

6.3 Performance Overhead

Benchmark We perform performance analysis on NBench [32] and CoreMark [2] benchmark. Specifically, NBench is a synthetic computing benchmark used to measure a computer's CPU, FPU and memory system speed. This benchmark consists of ten different tasks such as Huffman compression and LU decomposition. CoreMark is a sophisticated benchmark designed to test a processor core's functionality. Since these benchmarks are designed to evaluate the performance of a processor, we consider evaluating the performance overhead introduced by RetTag on them.

Configuration We tested benchmarks on the Xilinx Kintex-7 FPGA KC705 evaluation board with Linux 4.20.0. The system has 1GiB memory provided by a Micro SD Card installed on our FPGA board. All the benchmarks are compiled and instrumented by LLVM 11.0 for RV64GX architecture, and all programs are statically linked. We ran each benchmark 10 times.

Results Figure 4 shows the performance overhead of RetTag on NBench. The baseline is the result of the benchmark without RetTag (i.e., the program is not be instrumented). In Figure 4, the baseline is standardized as 1, and we can find that the performance overhead introduced byRetTag is around −0.66% to 1.10%. Since there are fluctuations in the running environment, there may be some imprecision of the performance overhead. The average performance overhead of RetTag on NBench is 0.11%. Thus, it is a negligible performance overhead for RetTag on NBench. The average performance overhead of RetTag on Coremark is 7.69%.

To further understand the performance overhead introduced by RetTag, we discuss it in detail. The performance of RetTag mainly depends on two factors: 1) the CPU cycles of `pac/aut` instructions, 2) the ratio of instrumented custom instructions.

CPU Cycles Since the performance is impacted by the encryption module , we calculate additional CPU cycles that RetTag introduced to a protected function. In RetTag, we insert a sequence of `lui,addi,slli,add,pac/aut` instructions to get the current function-id and stack pointer value and generate or authenticate PAC, etc. By observing the waveform, we calculate that the overall process needs 57 cycles. The CPU cycles of `pac/aut` are heavily affected by the encryption module. Currently, we choose AES128 to generate PACs in the implementation, which consumes more than 20 cycles for encryption. If we use a faster hardware encryption module, the number of CPU cycles will be significantly decreased. Consequently, we assume that the PAC is computable with an approximate overhead of 4 cycles like [27] and reevaluate the performance based on it. We rerun the Coremark benchmark and find that the CPU cycles of instrumented custom instructions for each function are decreased from 57 cycles to 12 cycles, and the performance overhead introduced by RetTag is 7.69% to 3.42%.

Ratio Furthermore, the number of function calls and the number of instructions in the function also affect the performance. Table 3 shows the number of instructions executed in the benchmark and

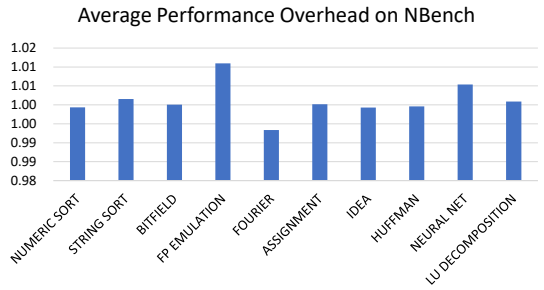

Figure 4: Performance Overhead of RetTag on NBench.

Table 3: Number of Executed Instructions in Benchmarks with RetTag.

Benchmarks	All Instructions	Instrumented Ins	Ratio
NBench	256,101,997,912	13,916,804	0.005%
Coremark	19,225,103,401	870,957,759	4.530%

the number of instructions added by RetTag The table indicates that the ratio of instrumented instructions on NBench is less than 0.01%, while around 4.530% on Coremark. That is the reason why RetTag introduces 0.11% performance overhead on the NBench while incurring 7.69% on the Coremark. Since we only insert a fixed amount of custom instructions (i.e., 57 cycles) at the beginning and the end of a function in RetTag, the performance also depends on the number of instructions in that function. For example, in a function with 1,000 instructions, the slowdown caused by the inserted instructions would be negligible. Note that the loops and code reuse make the actual number of executed instructions much more than that in the binary. Additionally, not every function needs to add and verify PAC; some functions (e.g., leaf nodes) do not contain epilogue such as `exit(0)`. Thus, the ratio of instrumented custom instructions will be lower.

Conclusion. According to the results and analysis, we can conclude that RetTag introduces around 0.11% to 7.69% performance overhead depending on the ratio of instrumented custom instructions. The runtime overhead of RetTag is reasonable. It incurs a negligible overhead when using the hardware encryption module.

7 Conclusion and Future Work

In this paper, we present RetTag, a hardware-assisted defense scheme against return address hijacking on RISC-V. Specifically, we use PAC embedded into the unused bits of a pointer to achieve return address integrity, and we demonstrate that RetTag can efficiently protect function return addresses on the stack. We integrate RetTag into the mainstream compilers and implement a prototype on the Rocket emulator and FPGA development board. The evaluation result shows that RetTag prevents an array of memory attacks on RISC-V platform with a reasonable performance overhead.

Utilizing the unused bits of a pointer to enhance security or improve performance is promising. There are still unused bits in Sv39 after deploying the proposed protection mechanism, and the size of the PAC can be dynamically adjusted depending on the memory addressing scheme and security features. We will use the remaining unused bits to improve the performance of Dynamic Taint Analysis (DTA) [33] technique. We intend to utilize the unused bits to store the taint tag and accelerate taint propagation.

8 Acknowledgments

We sincerely thank our shepherd Matthias Neugschwandtner and the anonymous reviewers for their insightful suggestions. This work was supported by the National Natural Science Foundation of China under Grant 62102175 and 62002151, and Science, Technology and Innovation Commission of Shenzhen Municipality under Grant SGDX20201103095408029.

References

[1] 2017. Intel Control-flow Enforcement Technology Preview document. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[2] 2021. Riscv-coremark. https://github.com/riscv-boom/riscv-coremark.

[3] 2021. Riscv-gnu-toolchain. https://github.com/riscv-collab/riscv-gnu-toolchain.

[4] 2021. Riscv-pk. https://github.com/riscv/riscv-pk.

[5] 2021. Rocket-chip. https://github.com/chipsalliance/rocket-chip.

[6] Martín Abadi, Mihai Budiu, and et. al. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)* (2009).

[7] Krste Asanovic, Rimas Avizienis, and et. al. 2016. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* (2016).

[8] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*. IEEE, 1212–1221.

[9] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. 2014. Hacking blind. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 227–242.

[10] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. 2008. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM conference on Computer and communications security*. 27–38.

[11] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 985–999.

[12] Nicholas Carlini, Antonio Barresi, and et. al. 2015. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of USENIX Security'15*.

[13] Cristian Chitu. [n.d.]. AES128 Reference Manual. ([n. d.]).

[14] Tzi-cker Chiueh and Fu-Hau Hsu. 2001. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings 21st International Conference on Distributed Computing Systems*. IEEE, 409–417.

[15] Mauro Conti, Stephen Crane, and et. al. 2015. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of ACM CCS'15*.

[16] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks.. In *USENIX security symposium*, Vol. 98. San Antonio, TX, 63–78.

[17] John Criswell, Nathan Dautenhahn, and et. al. 2014. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of IEEE S&P'14*.

[18] Thurston HY Dang, Petros Maniatis, and David Wagner. 2015. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. 555–566.

[19] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. 2011. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. 40–51.

[20] Bogdan Pavel Deac and Adrian Colesa. 2020. Following the Pebble Trail: Extending Return-Oriented Programming to RISC-V. In *Proceedings of ACM CCSW'20*.

[21] Leila Delshadtehrani, Sadullah Canakci, Boyou Zhou, Schuyler Eldridge, Ajay Joshi, and Manuel Egele. 2020. {PHMon}: A Programmable Hardware Monitor and Its Security Use Cases. In *29th USENIX Security Symposium (USENIX Security 20)*. 807–824.

[22] Enes Göktas, Elias Athanasopoulos, and et. al. 2014. Out of control: Overcoming control-flow integrity. In *Proceedings of the IEEE S&P'14*.

[23] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 191–205.

[24] Georges-Axel Jaloyan, Konstantinos Markantonakis, Raja Naeem Akram, David Robin, Keith Mayes, and David Naccache. 2020. Return-Oriented Programming on RISC-V. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 471–480.

[25] Vinay Katoch. 2011. Whitepaper on bypassing aslr/dep. *S. Technologies, Ed., ed* (2011).

[26] Jinfeng Li, Liwei Chen, and et. al. 2020. Zipper Stack: Shadow Stacks Without Shadow. arXiv:1902.00888 [cs.CR]

[27] Hans Liljestrand, Thomas Nyman, and et. al. 2019. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *Proceedings of USENIX Security'19)*.

[28] Hans Liljestrand, Thomas Nyman, Lachlan J Gunn, Jan-Erik Ekberg, and N Asokan. 2021. PACStack: an Authenticated Call Stack. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*.

[29] LLVM. 2020. *LLVM 10 User Guides*. https://releases.llvm.org/10.0.0/docs/UserGuides.html.

[30] Hector Marco-Gisbert and Ismael Ripoll-Ripoll. 2016. Exploiting Linux and PaX ASLR's weaknesses on 32-and 64-bit systems. *BlackHat Asia* (2016).

[31] Ali Jose Mashtizadeh, Andrea Bittau, and et. al. 2015. CCFI: Cryptographically enforced control flow integrity. In *Proceedings of ACM CCS'15*.

[32] Uwe F Mayer. 2003. Linux/unix nbench.

[33] James Newsome and Dawn Xiaodong Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software.. In *NDSS*, Vol. 5. Citeseer, 3–4.

[34] Ulziibayar Otgonbaatar. 2015. *Evaluating modern defenses against control flow hijacking*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[35] Vasilis Pappas, Michalis Polychronakis, and et. al. 2013. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proceedings of USENIX Security'13*.

[36] Pengfei Qiu, Yongqiang Lyu, and et. al. 2017. Control flow integrity based on lightweight encryption architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2017).

[37] Qualcomm. 2017. *Pointer authentication on ARMv8.3*. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf.

[38] Hovav Shacham. 2007. 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*. 552–561.

[39] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*. 298–307.

[40] Editors Andrew Waterman and Krste Asanović. 2019. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. RISC-V Foundation.

[41] Editors Andrew Waterman and Krste Asanović. June 2019. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified*. RISC-V Foundation.

[42] John Wilander, Nick Nikiforakis, and et. al. 2011. RIPE: runtime intrusion prevention evaluator. In *Computer Security Applications Conference*.

[43] Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel, and Ravishankar K Iyer. 2002. Architecture support for defending against buffer overflow attacks. *Coordinated Science Laboratory Report no. UILU-ENG-02-2205, CRHC-02-05* (2002).

[44] Chao Zhang, Tao Wei, and et. al. 2013. Practical control flow integrity and randomization for binary executables. In *Proceedings of IEEE S&P'13*.

[45] Jiliang Zhang, Binhang Qi, and et. al. 2018. HCIC: Hardware-assisted control-flow integrity checking. *IEEE Internet of Things Journal* (2018).

[46] Mingwei Zhang and R Sekar. 2013. Control flow integrity for COTS binaries. In *Proceedings of USENIX Security'13*. 337–352.