RAGuard: A Hardware Based Mechanism for Backward-Edge Control-Flow Integrity

Jun Zhang
State Key Laboratory of Computer Architecture, ICT, CAS
Beijing, China
University of Chinese Academy of Sciences
zhangjun02@ict.ac.cn

Rui Hou
Institute of Information Engineering, CAS
Beijing, China
hourui@iie.ac.cn

Junfeng Fan
Open Security Research
Shenzhen, China
fan@opsefy.com

Ke Liu
State Key Laboratory of Computer Architecture, ICT, CAS
Beijing, China
liuke@ict.ac.cn

Lixin Zhang
State Key Laboratory of Computer Architecture, ICT, CAS
Beijing, China
zhanglixin@ict.ac.cn

Sally A. McKee
Computer Science and Engineering, Chalmers University of Technology
Gothenburg, Sweden
sallyamckee@gmail.com

ABSTRACT

Control-flow integrity (CFI) is considered as a general and promising method to prevent code-reuse attacks, which utilize benign code sequences to realize arbitrary computation. Current approaches can efficiently protect control-flow transfers caused by indirect jumps and function calls (forward-edge CFI). However, they cannot effectively protect control-flow caused by the function return (backward-edge CFI). The reason is that the set of return addresses of the functions that are frequently called can be very large, which might *bend* the backward-edge CFI. We address this backward-edge CFI problem by proposing a novel hardware-assisted mechanism (RAGuard) that binds a message authentication code to each return address and enhances security via a physical unclonable function and a hardware hash function. The message authentication codes can be stored on the program stack with return address. RAGuard hardware automatically verifies the integrity of return addresses. Our experiments show that for a subset of the SPEC CPU2006 benchmarks, RAGuard incurs 1.86% runtime overheads on average with no need for OS support.

CCS CONCEPTS

•Security and privacy →Hash functions and message authentication codes; *Key management; Embedded systems security;*

KEYWORDS

backward-edge control-flow integrity, return oriented programming, return address, message authentication code, physical unclonable function, hardware hash function

CF'17, Siena, Italy

DOI: http://dx.doi.org/10.1145/3075564.3075570

1 INTRODUCTION

Code-reuse attacks, which repurposes existing code to a malicious end, have been developed to exploit memory corruption vulnerabilities in modern software. Such attacks are widespread: they have been reported on x86, ARM, SPARC, PowerPC, and Atmel AVR architectures [11, 23, 26]. The simplest code-reuse attack employs a *return-into-libc* technique that allows the attacker to execute an arbitrary sequence of libc functions via a buffer overflow that overwrites the stack with their parameters and return addresses. More complex approaches effect arbitrary computations by redirecting program control flow and chaining small fragments of benign code (called *gadgets*). Return Oriented Programming (ROP) [23] is a typical code-reuse attack, which utilizes gadgets ended with a *ret* instruction to hijack the backward-edge control-flow[1].

Control-flow integrity (CFI) is considered as a general method to prevent code-reuse attacks. It restricts the control transfers along the edges of the program's predefined Control-Flow Graph (CFG) [1]. CFG is constructed by statically analyzing the source code or binary of a given program. But it is possible for static program analysis to over-approximate the valid target addresses for indirect branches[2] at runtime, especially for the return from frequently called function that might have a large set of valid target addresses [4]. Hence, the return transfer checking might be prone to *bend* [4] the backward-edge CFI due to limited context information.

[1]Forward-edge control-flow represents transfers caused by indirect jumps and function calls. Backward-edge control-flow represents transfers caused by *ret* instructions.
[2]including indirect *call/jmp* instructions and *ret* instructions.

Shadow stack has been used to keep track of the called functions and to store the return addresses in a protected dedicated memory region. It is considered as an essential mechanism to guarantee the backward-edge CFI [1, 4, 13]. However, the shadow stack still suffers from the following problems: (1) the software shadow stack needs extra memory protection mechanism [10], and is vulnerable to memory disclosure [8]; (2) the hardware shadow stack relies on the operation system (OS) to save and restore its content during process context switches and deeply nested function calls [22, 25]; (3) the implementation of shadow stack needs to take care of some corner cases, such as *setjmp/longjmp* [4, 22], which might lead to false positives.

In this paper, we propose Return Address Guard (RAGuard) that guarantees backward-edge CFI based on hardware. Unlike shadow stack, which guarantees the backward-edge CFI by tracking the functions called, RAGuard guarantees the integrity of the stored return addresses instead. RAGuard binds each stored return address with a message authentication code (RAMAC). Compared to the state-of-art methods [20, 21] that need support from the OS, RAGuard utilizes a Physical Unclonable Function (PUF) to generate authoritative information for newly-forked or rescheduled process. Thus, RAGuard can verify the integrity of return addresses by hardware automatically without the support from OS. The Trusted Computing Base (TCB) is limited only to the system hardware. Additionally, RAGuard also can be used to provide protection for the corner cases in shadow stack.

Our contributions are summarized below.

- We propose RAGuard, which enforces the backward-edge CFI by binding every return address with a RAMAC. RAGuard utilizes the RAMACs to verify the integrity of return addresses.
- RAGuard leverages a PUF and a hardware hash function to compute and verify the RAMAC. It completely isolates the computation and verification of RAMAC from the software, thus its TCB is limited only to the system hardware.
- The runtime performance overhead of RAGaurd is evaluated using the SPEC CPU2006 benchmarks on a Intel Xeon CPU. The evaluation shows that it only incurs 1.86% runtime overhead.

2 BACKGROUND

RAGuard is a hardware based mechanism for backward-edge CFI. Before diving into the details of RAGuard, this section introduces background on CFI enforcement in the following sections.

2.1 ROP Attacks

Backward-edge CFI is considered as a general method against return-oriented programming based code reuse attacks (ROP attackes). ROP attacks utilize gadgets ended with a *ret* instruction to hijack control-flow [23]. An example of ROP attacks is shown in Figure 1. Initially, the adversary locates

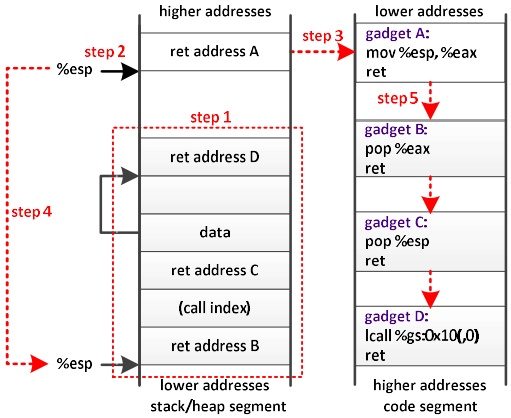

Figure 1: An example ROP attack.

the payload (a number of return addresses and necessary data in the red dotted box) into the application's stack or heap (Step 1). The adversary launches ROP attacks by leveraging a buffer overflow vulnerability to overwrite the return address A on the stack (Step 2). Then, the control-flow is hijacked and illegally transferred to gadget A (Step 3). Gadget A changes the stack pointer (*%esp* for X86 architecture) to the beginning of the payload (Step 4), and redirects the control-flow to the next gadget by executing a *ret* instruction (Step 5). Then, the gadgets are executed one by one until the adversary achieve his malicious goal.

2.2 CFI

The first CFI work, which is proposed by Abadi *et al.* [1], instruments software with runtime label checks. Generally, CFI is coupled with a protected shadow stack to ensure that each *ret* instruction only returns to its *call-site* [1, 4]. As the software-based instrumentation and shadow stack induce high performance overheads [1, 10], coarse-grained CFI approaches have been proposed [27, 28] and shadow stack is omitted [4] for better performance. However, coarse-grained CFI approaches are not secure enough for some recent attacks [4, 5, 8].

Several hardware-based CFI approaches [3, 12, 25] have been developed to reduce the performance overhead of software CFI approaches. Budiu *et al.* [3] introduces new CFI instructions and a CFI label register to enforce label checks on each indirect branch. However, a subroutine could be frequently called by different routines, the compiler inserts the same label at each possible call-side. This allows the attacker to *bend* the backward-edge control-flow [4]. Lucas *et al.* [12] address this problem by decoupling source from destination labels, and enforcing CFI based on label status. Specifically, they allocate a different label to every function. When a function is called, its label is activated. They enforce backward-edge CFI by restricting each *ret* instruction to return to a active function. These approaches could be considered as variants of shadow stack, as they keep track of

the called functions by label's status instead of the first-in, last-out principle. Similar to shadow stack, these approaches also require dedicated memory, and require OS to handle process context switches, then the OS has to be in the TCB. Therefore, if the OS is subverted, the protection can not be ensured. In contrast, our proposed RAGuard utilizes the RAMACs to verify the integrity of return addresses, which does not need the support from OS, and reduces TCB only to the system hardware. On the other hand, RAGuard is an orthogonal approach, their methods provide wonderful forward-edge CFI protection for us, our method improves the flexibility and enhances the security for their backward-edge CFI protection.

2.3 MAC of Return Address

The first work that inserts a MAC for return address is StackGuard [9]. StackGuard places a pre-defined secure value on the program stack next to a return address. The secure value could be string terminators or randomly generated numbers. When a function returns, the secure value is verified by the function epilogue to make sure that it remains intact. Then, Frantzen *et al.* [13] and Tuck *et al.* [21] protect the return address by storing the encrypted return address on the program stack. However, the MACs of these techniques only contain the information of return address, and can be used to implement replay attacks. So these techniques can not guarantee backward-edge CFI.

Cryptographic CFI (CCFI) [20], which encrypts the signature of the return address instead of the the return address, can provide CFI protection effectively. Because CCFI randomly generates the key of AES at program start, it needs the OS to save and restore the MAC key during the process context switches. Moreover, CCFI incurs non-ignorable performance overhead, even though it takes advantage of cryptographic CPU instructions (AES.NI). In contrast, RAGuard does not need the support from the OS by proposing a novel key management method based on PUF to support the process context switches. It also improves the performance of backward-edge CFI by using a hardware hash function.

3 SYSTEM MODEL

3.1 Threat Model and Assumptions

This paper focuses on backward-edge CFI. We assume that the forward-edge CFI has been efficiently enforced by the previous solutions, but adversary can exploit memory corruption vulnerabilities to launch ROP attacks. Adversaries have arbitrary read access to application's code, full control over the program's stack and heap, and circumvent ASLR (Address Space Layout Randomization) schemes to get the layout of applications memory. We assume the starting stack location is randomly chosen at program startup to prevent alignment of return address and RAMAC, which makes replay attacks possible.

Another assumption is that the Operating System (OS) may have vulnerabilities, which could be exploited to undermine security mechanisms, such as saving and restoring the consent of shadow stack. We assume that the TCB of our system is limited to the system hardware, and the hash function of RAGuard resists preimage attacks. We also assume that hardware attacks are not part of the thread model.

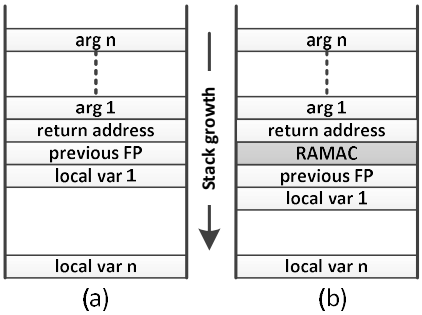

Figure 2: Comparison between (a)traditional stack layout, and (b)RAGuard stack layout.

3.2 Desired Properties

In order to overcome the shortage of pervious mechanisms, RAGuard should contain the following desired properties.

- ***P1* Secure:** The RAMAC computation and verification should be completely isolated from software, and the secret information should never leave the hardware chip.
- ***P2* Transparent:** Developers should not need to design programs any differently in order to make use of our mechanism.
- ***P3* Low-cost:** No substantial increase of area overhead and manufacturing complexity.

4 RAGUARD MECHANISM

RAGuard guarantees the backward-edge CFI by verifying the integrity of return addresses. Unlike shadow stack that stores return addresses in a dedicated memory, RAGuard binds a RAMAC to each stored return address as shown in Figure 2. To guarantee the security of RAMAC, PUF is used to generate authoritative information for program process, and a hardware hash function is used to compute the RAMAC. The authoritative information is generated by hardware, and the integrity verification of return addresses is completely isolated from software, thus our TCB is limited only to the system hardware (*P1*). Additionally, RAGuard also can be used to provide protection for *setjmp/longjmp*.

4.1 RAMAC Computation

Similar to CCFI [20], the signature of our method contains the following items:

- **Return Address (*RA*):** It is the key information that MAC verifies.
- **Stack Pointer (*SP*):** The position of the return address in the program stack. Including the SP in the signature ensures that an attackers cannot swap

a return address stored in one memory address with a pointer stored in a different memory address.

- **Authoritative Information (*P_AI*):** The signature must contain some secret information (called *P_AI*), which cannot be obtained by attackers. It is challenging that *P_AI* should never leave the hardware and also be retrievable when the process is scheduled to run. Thanks to the fundamental characteristics of PUF [14, 16, 17, 19, 29], RAGuard can securely generate and retrieve exclusive *P_AI* for each process. We use the process characteristic information as the PUF challenge. In our implementation, we use process ID (PID) as the process characteristic information. The response of that PUF is used as *P_AI*.

First, we derive the signature of the return address, denoted by *SIG*, thus we have,

$$SIG = RA||SP||P\_AI \quad (1)$$

Then, RAMAC is computed as the hash value of *SIG*, thus we have,

$$RAMAC = HASH(SIG) \quad (2)$$

where *HASH(.)* denotes the hardware hash function.

4.2 Architecture Support

In the design of RAGuard, two dedicated registers are introduced: *ramac_c* and *ramac_v*, and the semantics of *call* and *ret* instructions are extended to perform the RAMAC computation and verification. Specifically,

***Call Instruction*:** Besides pushing its return address on stack and transferring to the target function, RAGuard computes the RAMAC of the return address, and stores it in *ramac_c* register.

***Ret Instruction*:** Besides popping the return address out of stack and decreasing the *esp* as usual, the RAMAC on the stack is loaded into *ramac_v* register. RAGuard recomputes the RAMAC for the popped return address and matches it with the one in *ramac_v* register. A mismatch indicates that the popped return address or the RAMAC was modified by attackers, thus an exception is raised by RAGuard.

We propose two methods to place RAMAC next to the return address on the stack, i.e., RAGuard-I and RAGuard-II. Specifically,

RAGuard-I: RAGuard pushes the *ramac_c* register on the stack after the *call* instruction and pop the RAMAC out before the *ret* instruction. This method can be implemented by inserting a *push* instruction and a *pop* instruction in the function's prologue and epilogue, respectively, as shown in Figure 3. In order to guarantee the correctness of the program executions, the architecture register *ramac_c* must be saved if there is a context switch after the *call* instruction.

RAGuard-II: As shown in Figure 4, RAGuard-II extends the semantics of original *call* and *ret* instructions, and these two instructions access *ramac_c* and *ramac_v* register internally. Thus these two new registers are considered as internal registers and they are architectural invisible. The atomicity

Function prologue code:		Function epilogue code:	
1: push	ramac_c	1: mov	rbp, rsp
2: push	rbp	2: pop	rbp
3: mov	rsp, rbp	3: pop	ramac_v
4: sub	rsp, N	4: ret	

Figure 3: Function prologue and epilogue modification to support RAGuard-I.

```
Extended call semantics:
1: IF relative near call
2:    THEN IF OperandSize = 32
3:       THEN
4:             tempEIP ← EIP + DEST
5:             push(EIP);
6:             ramac_c ← fun(EIP);
7:             push(ramac_c);
8:             EIP ← tempEIP;
9:       FI
10: FI
Extended ret semantics:
1:  IF near return
2:    THEN IF OperandSize = 32
3:       THEN
4:             ramac_v ← pop();
5:             EIP' ← pop();
6:             ramac_c ← fun(EIP');
7:             IF match(ramac_v, ramac_c)
8:                EIP ← EIP';
9:             ELSE
10:               exception();
11:            FI
12:      FI
13: FI
```

Figure 4: Extension of instruction semantics to support RAGuard-II.

does not require any OS modifications with acceptable hardware cost. However, pushing the RAMAC on the program stack by hardware directly will break the relative position between the frame pointer and parameters. Thus RAGuard-II also needs to patch the compiler to allocate address space for RAMAC.

To protect *setjmp/longjmp*, the RAMAC is saved into the environment buffer by the *setjmp* functions, and restored by the *longjmp* functions.

As stated above, these two methods are both practicable and transparent to program developers (*P2*).

Traditionally the return address will be used as Program Counter (PC) in next cycle after being loaded. It is difficult to verify the RAMAC in one cycle. According the previous study [18], verification result can be reported at a later time. Thus these both methods only add extra memory operations to each function.

5 IMPLEMENTATION

As instroduced in Section 4, our RAGuard mechanism can be applied to the traditional processor architectures. In this section, we implement our RAGuard mechanism on the open-source LEON3 processor [6], which is released in open-source, for example.

The structure of our RAGuard mechanism is shown in Figure 5. We add a hardware RAMAC generator module (*ramac_gen*) to the iu3 module (inside the dash line)[3]. The *ramc_gen* module contains the following components: *raguard_ctrl*, *PUF*, *hw_hash*, and *lsuq* (for RAGuard-I, RAMAC is stored and loaded by instructions, *lsuq* is not needed). The added data pathes are denoted as blue lines. In order to facilitate the RAMAC compution, we also add four registers: *p_inf*, *p_ai*, *ramac_c* and *ramac_v*.

In 64-bit architectures, the maximum PID number could be up to 4,194,303 [2]. Thus, PIDs can be expressed by a 22-bit binary. We set the input and output of the PUF module as 32-bit. The width of that input is larger than that of PIDs, which allows us to include more process characteristic information in the future. *RA* and *SP* are both set to 32-bit[4], thus the input of the hardware hash function is 96-bit. The time complexity of attack, which can be expressed as 2^n, depends on hash function's output size n. The larger the output size, the higher the bar for attacks in practice. In order to simplify the performance evaluation, we set the output size as 32-bit and 64-bit for 32-bit and 64-bit architectures respectively, and use a hardware hash function with only 1 cycle latency [15]. The area overhead of the hash function is about 4K GE(*P3*).

As shown in Figure 5, *raguard_ctrl* module monitors the instructions at the decode stage (DE), and maintains a state machine to control the RAMAC computation and verification. The state machine is shown in Figure 6.

When a process is scheduled to run, the PID register of the processor[5] is reset. Meanwhile, the value of *p_inf* register is updated automatically. *raguard_ctrl* module sends a process authoritative information update command (*pai.update*) to the PUF module, and its status becomes *s_puf*. The PUF module uses the new value of *p_inf* register as its challenge. Its response is stored in the *p_ai* register. When the process authoritative information update is done, the PUF module sends a response (*pai.update_done*) back to *raguard_ctrl* module.

When there is a *call* instruction at the DE stage, a RAMAC computing command (*call.mac_generate*) is sent to *hw_hash* module, and the status of *raguard_ctrl* module becomes *s_hw_hash*. After receiving this command, *hw_hash* module uses the signature of the return address as the input of the hardware hash function to compute RAMAC (i.e., Equation (2)). When the RAMAC is prepared, *raguard_ctrl* module sends a RAMAC storing command (*call.mac_store*) to *lsuq* module. Then the *call* instruction is submitted (*call.mac_submit*). As the RAMAC is computed in parallel with the processor pipeline, and it can be sent to *lsuq* later, the RAMAC computation will not incur any pipeline stall.

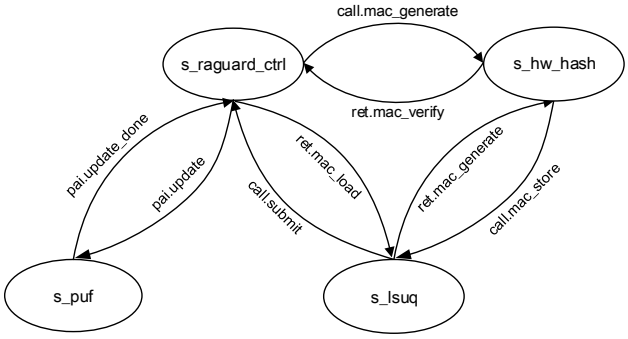

Figure 6: State machine of the *raguard_ctrl* module.

When there is a *ret* instruction at the DE stage, firstly, *raguard_ctrl* module sends a RAMAC loading command (*ret.mac_load*) to *lsuq* module. RAMAC is loaded from the program stack and stored in *ramac_v* register. Meanwhile, the *ret* instruction pops the return address out of the program stack at the XC stage. Then, *raguard_ctrl* module sends a RAMAC calculation command (*ret.mac_generate*) to *hw_hash* module. After receiving the RAMAC calculation command, *hw_hash* module recomputes the RAMAC of that return address, and sends it back to *raguard_ctrl* module. Finally, *raguard_ctrl* module compares the re-computed RAMAC with the one in the *ramac_v* register. If a mismatch occurs, an exception will be raised, and the execution is transferred to the exception handler routine.

5.1 PUF Module

Silicon PUFs exploit uncontrollable process variations in fabrication technology to generate secret key information [14, 16, 17, 19, 29]. Their novel properties, such as unclonability, robustness, unpredictability, and tamper-evidence, make them very appealing for the process's authoritative information management of our RAGuard mechanism. However, the response latency of PUFs is critical to our RAGurad mechanism. PUF is used to generate the authoritative information when a process is scheduled to run. Long response latency will incur extra overhead during process context switches[6].

Several candidates are listed in Table 1. All these PUFs will not incur notable context switch and area overhead. However, err-puf [29], Bitline PUF [14] and MECCA PUF [19] are required to modify the control logic of memory. In contrast, the LR-PUF [17] and VIA PUF [16] can be integrated into system as intellectual property (IP) cores. We prefer to the

[3]The code modification and the system evaluation on a Digilent Nexys4DDR board will be finished about two month later.

[4]In 64-bit architecture, the upper 32 bits of virtual address are 0xFFFFFFFF, so we only use the lower 32 bits.

[5]For LEON3, the context number is the corresponding PID register.

[6]Usually, the context switch time is about 100-200 microseconds [30]. Context switch time is the time spend in the OS to switch from one process to another.

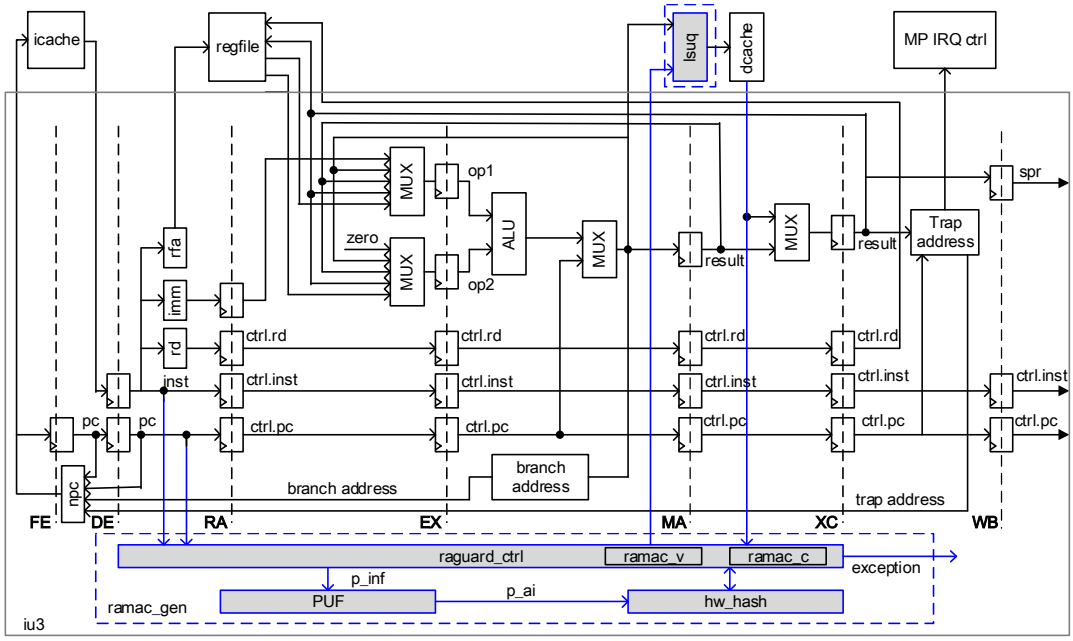

Figure 5: Our RAGuard mechanism is implemented on LEON3, which uses a 7 stages pipeline (Instruction Fetch (FE), Decode (DE), Register Access (RA), Execute (EX), Memory (ME), Exception (XC), and Write (WR)). A RAMAC generator module (ramac_gen, inside the dash line) is added to the LEON3 pipeline. The added data pathes are denoted as blue lines.

Table 1: Candidates of PUF module

	Structure	Response latency	Area overhead
err-PUF[1][29]	cell error rate distribution of STT-RAM	2.32 µs	2.9×10^2 µs[2]
Bitline PUF [14]	SRAM with modified wordline drivers	64 memory write operations and 1 memory read operation (about 0.16 µs)[2]	2048 GE[3]
MECCA PUF [19]	SRAM with a programmable delay generator	2 memory write operations and 1 memory read operation	20 GE[4]
LR-PUF[5][17]	arbiter PUF with reconfiguration ctrol logic	1069 clock cycles (about 1 µs at 1 GHz)	6974 GE
VIA PUF [16]	via holes between two metal layers	1 memory read operation	nearly zero[6]

[1] The response latency and area overhead is evaluated with the 45nm technology.
[2] For a 256-column by 256-row SRAM with a 5ns cycle time, the response latency is 512 write memory operations and 1 memory read operation (2.6 µs) [14].
[3] Bitline PUF's area overhead is only a single flip-flop and two logic gates (8 GE) per row of SRAM. The area overhead is estimated based on a 256-column by 256-row SRAM.
[4] MECCA PUF's area overhead is a programmable delay generator. The area overhead is estimated based on programmable delay generator with 4 duty cycles.
[5] LR-PUF is run with 80-bit challenge line and 64-bit response line.
[6] VIA PUF is not built on via holes between two metal layers, but needs dedicated reading circuit.

Table 2: Experiment setup

CPU Type	Intel Xeon E5645
#Cores	6 cores@2.4GHz
#Sockets	2
Memory	16G, DDR3
Operating system	Centos 6.6 with Linux kernel 2.6.32
Workload	SPEC CPU2006
Compiler	LLVM 3.8.1
Baseline	-O3
RAGuard	-O3 and optimized with a LLVM pass

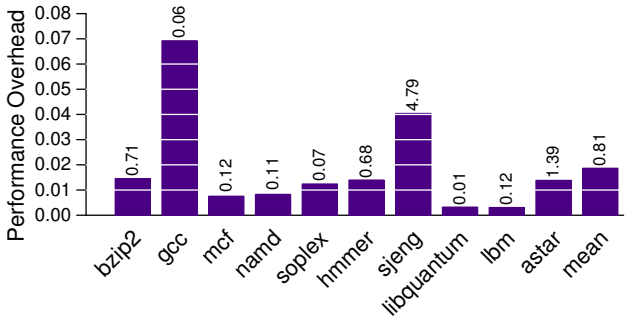

Figure 7: Performance overhead of RAGuard.

latter one, as it can be implemented without bit-error, and it is avialable [16](*P3*).

6 PERFORMANCE EVALUATION

To evaluate the performance overhead of our RAGuard mechanism, we re-compile the SPEC CPU2006 benchmark applications [24] with Low Level Virtual Machine (LLVM) [7], and insert two extra memory operation instructions to each function by a LLVM pass. The details of experiment environment is listed in Table 2.

We evaluate the runtime performance overhead of RAGuard. As shown in Figure 7, the y-asis shows that the runtime overhead normalized to the baseline, i.e., the runtime of the processor architecture without RAGuard. The value on each bar is the absolute runtime overhead in seconds resulted from RAGuard. In average, RAGuard incurs 1.86% runtime degradation. The worst-case is 6.92% for `gcc`. Except for `gcc` and `sjeng`, RAGuard incurs less than 1.5% degradation for the rest of the cases.

Figure 8 shows the code size of RAGuard-I normalized to the baseline, i.e., the code size of the processor architecture without RAGuard. The value on each bar is the absolute code size overhead. The average code size overhead resulted from RAGuard-I is 2.72%, and the worst-case is 4.22% for `soplex`. For RAGuard-II, RAMAC is pushed on the program stack by the *call* instruction and popped out of the program stack by the *ret* instruction, thus this method does not incur any code size overhead.

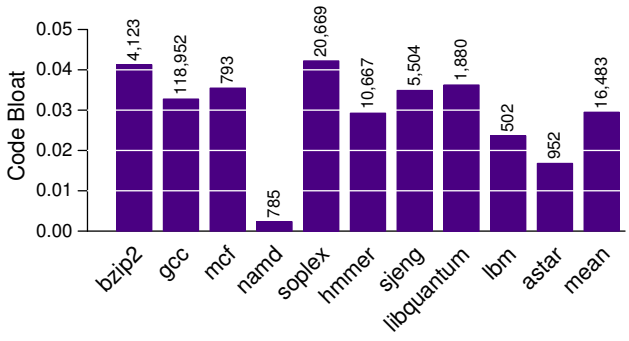

Figure 8: Size of added code size for RAGuard-I.

7 SECURITY ANALYSIS

In this section, we analyze the security of our RAGuard mechanism. We only discuss the attacks that leverage the backward edges, such as ROP attacks. Additionally, the security of RAGuard also relies on the complexity of computed RAMAC. Hence, we also analyze the effect of the complexity of computed RAMAC in the security of RAGuard.

7.1 Return-Oriented Programming

In ROP attacks, attackers have to overwrite return addresses on the stack and locate their payload into the application's heap/stack (as shown in Figure 1). The malicious return addresses in the payload may redirect control flow to arbitrary code locations or unintentional call-sites. In the former case, attackers have to compute the RAMACs for their dummy return addresses. In the latter case, attackers might carry out the splicing attack by constructing payload with the existing return address and corresponding RAMAC. This way violates our RAMAC verification. When a return address and its corresponding RAMAC are inserted on another location, the stack pointer of the new location mismatches with the RAMAC. Hence, attackers also have to recompute the RAMACs for their malicious return addresses.

Recently, a special ROP attack, Control-Flow Bending (CFB) [4], has been proposed, which implements code-reuse attacks within the valid CFG to achieve Turing-complete computation. In the CFB attack, attackers could utilize the dispatcher function to overwrite the return address to return to any location where the dispatcher function is called. Our mechanism also can prevent CFB attacks, because attackers also have to overwrite the RAMAC correctly besides the return address stored on the program stack.

7.2 RAMAC Security

As shown in Section 5, RAMAC is computed with a one-way hardware hash function, which takes the signature of the return address as the input. A PUF module is used to dynamically update the *p_ai* register during context switching. As the hardware hash function is not visible to attackers, and the *p_ai* register is only accessed by hardware, they cannot infer the the algorithm of the hardware hash function.

Moreover, the *p_ai* register is updated at runtime, which makes RAMAC more difficult to attack.

8 CONCLUSION

This work presents RAGuard, an efficient hardware-based mechanism for backward-edge CFI. RAGuard binds every return address with RAMAC to guarantee the integrity of return addresses on the program stack. The security of RAMAC is provided by a PUF module and a hardware hash function, thus it can be stored on the program stack. By completely isolating the computation and verification of RAMAC from the system software, RAGurad reduces the TCB to the system hardware. Compared to the baseline processor architecture,RAGuard only incurs 1.86% runtime overhead on average.

9 ACKNOWLEDGMENTS

The authors would like to thank to Chen Zheng, Zhen Jia, Haiyang Pan, Lili Sun, Yizhong Hu, and Yang Cao for their kind help on the work. We also thank the anonymous reviewers for their valuable comments. This work was supported by the China National Science Fund for Outstanding Young Scholars under grant No. 61522212, Frontier Science Research Projects, Chinese Academy of Science, under grant No. QYZDB-SSW-JSC010, National Key R&D Plan under grant No. 2016YFB1000400, and National Natural Science Foundation of China (NSFC) under grant No. 61521092.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *proc. 12th ACM Conference on Computer and Communications Security (CCS)*. 340–353.

[2] Daniel Bovet and Marco Cesati. 2005. *Understanding the Linux kernel*. O'Reilly Media, Inc.

[3] Mihai Budiu, Úlfar Erlingsson, and Martín Abadi. 2006. Architectural support for software-based protection. In *1st Workshop on Architectural and System Support for Improving Software Dependability (ASID)*. 42–51.

[4] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-flow bending: on the effectiveness of control-flow integrity. In *Proc. 24th USENIX Security Symposium (SEC)*. 161–176.

[5] Nicholas Carlini and David Wagner. 2014. ROP is still dangerous: breaking modern defenses. In *Proc. 23rd USENIX Security Symposium (SEC)*. 385–399.

[6] Cobham Gaisler. 2017. LEON3 processor. http://www.gaisler.com/index.php/products/processors/leon3. (2017).

[7] The LLVM compiler infrastructure. 2017. LLVM Overview. http://llvm.org/. (2017).

[8] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. 2015. Losing control: on the effectiveness of control-flow integrity under stack attacks. In *Proc. 22nd ACM Conference on Computer and Communications Security (CCS)*. 952–963.

[9] 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 *Proc. 7th Usenix Security Symposium (SEC)*. 63–78.

[10] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2015. The performance cost of shadow stacks and stack canaries. In *Proc. 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 555–566.

[11] Lucas Davi. 2015. *Code-reuse attacks and defenses*. Ph.D. Dissertation. Technische Universität, Darmstadt. http://tuprints.ulb.tu-darmstadt.de/4622/

[12] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. 2015. HAFIX: hardware-assisted flow integrity extension. In *Proc. 52nd ACM Annual Design Automation Conference (DAC)*. 74:1–74:6.

[13] Michael Frantzen and Michael Shuey. 2001. StackGhost: hardware facilitated stack protection. In *Proc. 10th USENIX Security Symposium (SEC)*.

[14] Daniel E. Holcomb and Kevin Fu. 2014. Bitline PUF: building native challenge-response PUF capability into any SRAM. In *16th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. 510–526.

[15] Nan Hua, Eric Norige, Sailesh Kumar, and Bill Lynch. 2011. Non-crypto hardware hash functions for high performance networking ASICs. In *Proc. 7th IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 156–166.

[16] ICTK. 2017. VIA PUF. http://www.ictk.com/servicenproduct/puf. (2017).

[17] Stefan Katzenbeisser, Ünal Koçabas, Vincent van der Leest, Ahmad-Reza Sadeghi, Geert-Jan Schrijen, Heike Schröder, and Christian Wachsmann. 2011. Recyclable PUFs: logically reconfigurable PUFs. In *13th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. 374–389.

[18] Mehmet Kayaalp, Timothy Schmitt, Junaid Nomani, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2013. SCRAP: architecture for signature-based protection from code reuse attacks. In *Proc. 19th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 258–269.

[19] Aswin Raghav Krishna, Seetharam Narasimhan, Xinmu Wang, and Xinmu Wang. 2011. MECCA: a robust low-overhead PUF using embedded memory array. In *13th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. 407–420.

[20] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. 2015. CCFI: cryptographically enforced control flow integrity. In *Proc. 22nd ACM Conference on Computer and Communications Security (CCS)*. 941–951.

[21] Tuck Nathan, Calder Brad, and Varghese George. 2004. Hardware and binary modification support for code pointer protection from buffer overflow. In *Proc. 37th IEEE International Symposium on Microarchitecture (Micro)*. 209–220.

[22] H. Ozdoganoglu, T. N. Vijaykumar, C. E. Brodley, B. A. Kuperman, and A. Jalote. 2006. SmashGuard: a hardware solution to prevent security attacks on the function return address. *IEEE Trans. Comput.* (2006), 1271–1285.

[23] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proc. 14th ACM Conference on Computer and Communications Security (CCS)*. 552–561.

[24] Standard Performance Evaluation Corporation (SPEC). 2011. SPEC CPU2006 Benchmark. (2011). http://www.spec.org/cpu2006/

[25] Dean Sullivan, Orlando Arias, Lucas Davi, Per Larsen, Ahmadreza Sadeghi, and Yier Jin. 2016. Strategy without tactics: policy-agnostic hardware-enhanced control-flow integrity. In *proc. 53rd ACM Design Automation Conference (DAC)*. 163:1–163:6.

[26] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *Proc. 23rd USENIX Security Symposium (SEC)*. 941–955.

[27] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *Proc. 34th IEEE Symposium on Security and Privacy (S&P)*. 559–573.

[28] Mingwei Zhang and R. Sekar. 2013. Control flow integrity for COTS binaries. In *Proc. 22nd USENIX Security Symposium (SEC)*. 337–352.

[29] Xian Zhang, Guangyu Sun, Yaojun Zhang, Yiran Chen, Hai Li, Wujie Wen, and Jia Di. 2016. A novel PUF based on cell error rate distribution of STT-RAM. In *Proc. 21st IEEE Asia and South Pacific Design Automation Conference (ASP-DAC)*. 342–347.

[30] Cheng Zheng, Jianfeng Zhan, Zhen Jia, and Lixin Zhang. 2013. Characterizing OS behavior of scale-out data center workloads. In *7th Workshop on the Interaction amongst Virtualization, Operating Systems and Computer Architecture (WIVOSCA)*.