RAGuard: An Efficient and User-Transparent Hardware Mechanism against ROP Attacks

JUN ZHANG, State Key Laboratory of Computer Architecture, ICT, CAS and Hubei University of Arts and Science
RUI HOU and WEI SONG, State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences
SALLY A. MCKEE, Electrical and Computer Engineering, Clemson University
ZHEN JIA, CHEN ZHENG, MINGYU CHEN, and LIXIN ZHANG, Institute of Computing Technology, Chinese Academy of Sciences
DAN MENG, State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences

Control-flow integrity (CFI) is a general method for preventing code-reuse attacks, which utilize benign code sequences to achieve arbitrary code execution. CFI ensures that the execution of a program follows the edges of its predefined static Control-Flow Graph: any deviation that constitutes a CFI violation terminates the application. Despite decades of research effort, there are still several implementation challenges in efficiently protecting the control flow of function returns (Return-Oriented Programming attacks). The set of valid return addresses of frequently called functions can be large and thus an attacker could bend the backward-edge CFI by modifying an indirect branch target to another within the valid return set. This article proposes RAGuard, an efficient and user-transparent hardware-based approach to prevent Return-Oreiented Programming attacks. RAGuard binds a message authentication code (MAC) to each return address to protect its integrity. To guarantee the security of the MAC and reduce runtime overhead: RAGuard (1) computes the

This article is an extension of the conference paper "RAGuard: A hardware based mechanism for backward-edge control-flow integrity," which appeared in the Proceedings of the International Conference on Computing Frontiers 2017 (CF'17) [61]. This submission improves the original mechanism by: (1) **Raising the bar for brute-force attacks**: We compute the RAMAC by encrypting the signature of the return address with AES-128 to raise the complexity of brute force attacks to 2^{128}; (2) **Evaluating on a real implementation**: We implement RAGuard on the open-source LEON3 processor to evaluate the runtime overhead due to AES-128 latency. Then we evaluate the performance and area overhead of RAGuard based on the Modelsim SE 10.2c simulator and Xilinx Vivado Design Suite, respectively; (3) **Optimizing based on leaf functions**: The RAGuard hardware dynamically identifies the leaf functions to skip unnecessary RAMAC load and store operations, thereby reducing RAGuard's performance overhead.

This work was supported by the China National Science Fund for Excellent Young Scholars under Grant No. 61522212, National Key R&D Plan under Grant No. 2017YFB1001602, Frontier Science Research Projects, Chinese Academy of Science, under Grant No. QYZDB-SSW-JSC010, the CAS Pioneer Hundred Talents Program and National Natural Science Foundation of China (NSFC) under Grants No. 61521092 and No. 61502459.

Authors' addresses: J. Zhang, Hubei University of Arts and Science, Xiangyang, 441053, China; email: zhangjun@hbuas.edu.cn; R. Hou (corresponding authors), W. Song, and D. Meng (corresponding authors), Institute of Information Engineering, Chinese Academy of Sciences, Beijing, 100190, China; emails: {hourui, songwei, mengdan@}iie.ac.cn; S. A. McKee, Electrical and Computer Engineering, Clemson University, Clemson, SC, 29630, USA; email: samckee@clemson.edu; Z. Jia, C. Zheng, M. Chen, and L. Zhang, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, 100190, China; emails: {jiazhen, zhengchen, cym, zhanglixin@}ict.ac.cn.

1544-3566/2018/11-ART50
https://doi.org/10.1145/3280852

MAC by encrypting the signature of a return address with AES-128, (2) develops a key management module based on a Physical Unclonable Function (PUF) and a True Random Number Generator (TRNG), and (3) uses a dedicated register to reduce MACs' load and store operations of leaf functions. We have evaluated our mechanism based on the open-source LEON3 processor and the results show that RAGuard incurs acceptable performance overhead and occupies reasonable area.

CCS Concepts: • **Security and privacy** → **Key management**; **Artificial immune systems**; **Embedded systems security**; **Hardware-based security protocols**;

Additional Key Words and Phrases: Code-reuse attacks, return-oriented programming attacks, message authentication code, AES-128, key management, PUF

ACM Reference format:
Jun Zhang, Rui Hou, Wei Song, Sally A. Mckee, Zhen Jia, Chen Zheng, Mingyu Chen, Lixin Zhang, and Dan Meng. 2018. RAGuard: An Efficient and User-Transparent Hardware Mechanism against ROP Attacks. *ACM Trans. Archit. Code Optim.* 15, 4, Article 50 (November 2018), 21 pages.
https://doi.org/10.1145/3280852

1 INTRODUCTION

Code-reuse attacks exploit memory corruption vulnerabilities in modern software by re-purposing existing codes to a malicious end [57]. Such attacks are widespread: They have been reported on x86, ARM, SPARC, PowerPC, and Atmel AVR architectures [18, 45, 54], and they certainly take place on countless others. One typical example of a code-reuse attack employs a *return-into-libc* technique that allows the attacker to execute `libc` functions via buffer overflow attacks. More complex approaches effect arbitrary computations by redirecting program control flow and chaining small fragments of benign code (called *gadgets*). Return Oriented Programming (ROP) [45] is another typical code-reuse attack that utilizes gadgets ending with a *ret* instruction to hijack the backward-edge control flow.[1]

Control-flow integrity (CFI) is a general approach for preventing such code-reuse attacks. It restricts control transfers along the edges of a program's predefined Control-Flow Graph (CFG) [1]. The CFG is constructed by statically analyzing a program's source code or binary, but this statically constructed CFG is normally an over-approximation of the valid runtime target addresses for indirect branches, including indirect *call*, *jump*, and *ret* instructions. This is a particular problem for returns from frequently called functions that might have many valid target addresses [25]. Limited context information makes this backward-edge CFI vulnerable to *bending* in which attackers make a program return to a different address within the set of valid target addresses [6].

Shadow stacks are essential mechanisms for guaranteeing backward-edge CFI [1, 6, 8, 10, 19, 27, 44]. A shadow stack keeps track of called functions by storing the return addresses in a dedicated and protected memory region [22]. However, this approach still suffers from several challenges: (1) a software shadow stack needs extra memory protection [34, 38] and is vulnerable to memory disclosure attacks [9, 21, 26, 42]; (2) a hardware shadow stack relies on the OS to save and restore its contents during context switches and stack overflows [10, 12, 19, 35, 59]; and (3) the implementation of the shadow stack must handle corner cases such as *setjmp/longjmp*, which might lead to false positives [6, 16, 19, 43, 44, 59].

This article describes RAGaurd, an efficient and user-transparent hardware mechanism for backward-edge CFI. RAGuard guarantees the integrity of the stored return addresses. The major contributions of this work are:

[1]Forward-edge control flow represents transfers caused by indirect jumps and function calls. Backward-edge control flow represents transfers caused by return instructions.

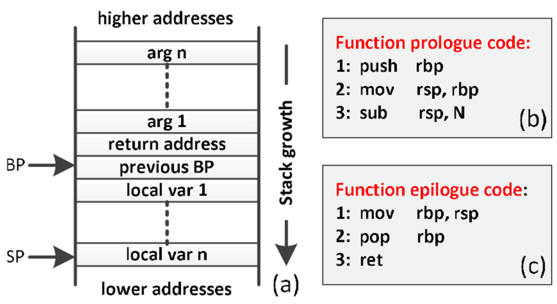


Fig. 1. Stack frame and related operations.

- We design a backward-edge control-flow integrity mechanism RAGuard that raises attack complexity to 2^{128}. RAGuard binds every return address with a message authentication code (called an RAMAC and pronounced R-A-mac) and verifies the integrity of the return address automatically. It guarantees the security of its mechanism by computing the RAMAC with AES-128 and implementing a key management module based on a Physical Unclonable Function (PUF) and a True Random Number Generator (TRNG).
- We demonstrate an implementation of RAGuard on the open-source LEON3 processor, which we use to evaluate runtime overheads due to AES-128 latency. We evaluate RAGuard's performance and area overhead via the Modelsim SE 10.2c simulator and Xilinx Vivado Design Suite, respectively.
- We develop an optimization method based on leaf functions. By monitoring the *call-ret* instruction sequences, RAGuard dynamically identifies each leaf function and compares the return address of the leaf with the stored one. Our evaluation shows that our optimized RAGuard mechanism incurs negligible performance overhead for application with high percentage (above 70%) of leaf functions.

The rest of this article is organized as follows. Section 2 provides a background introduction. Section 3 explains the threat model and the design goal. Section 4 describes our proposed RAGuard mechanism. Section 5 presents our proposed hardware solution in details and analyzes the evaluation results. Section 6 analyzes and compares related work in terms of both performance and effectiveness. Finally, Section 7 concludes this work.

2 BACKGROUND

Before describing RAGuard in detail, we introduce necessary background on CFI enforcement.

2.1 Stack Frames

Program stacks store information about active subroutines. As Figure 1(a) shows, arguments are pushed onto the stack before calling a function. The actual *call* instruction then pushes the return address, i.e., the address of the instruction immediately following the *call*. The function prologue (shown in Figure 1(b)) pushes the previous stack frame base pointer (BP) onto the stack and allocates space for the called function's local variables by adjusting the stack pointer (SP). The function arguments, return address, previous BP, and local variables comprise its stack frame. When a function returns, the function epilogue (shown in Figure 1(c)) restores the SP and pops the return address from the stack. Finally, control flow is redirected to the return address.

The return address is stored in the stack frame during the call, which introduces a vulnerability: Attackers can redirect control flow by overwriting this original return address. For an example,

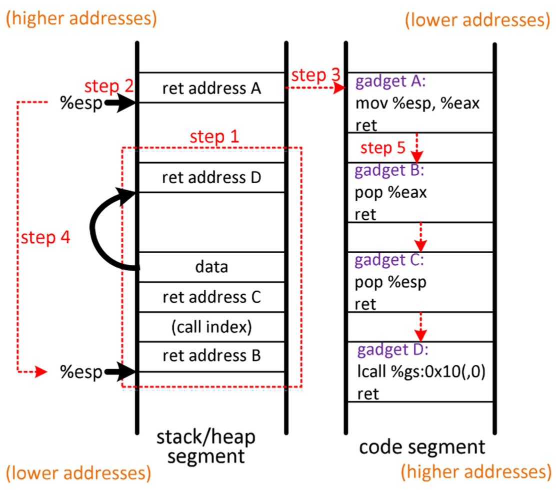


Fig. 2. An ROP attack example.

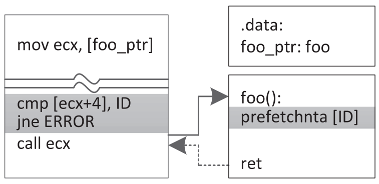


Fig. 3. A CFI [1] example.

attackers can exploit a buffer overflow vulnerability to overwrite the return address and thus hijack control flow.

2.2 ROP Attacks

Backward-edge CFI is a general defense against return-oriented programming (ROP) code reuse attacks [16]. ROP attacks hijack control flow by using gadgets (code segments ending with a *ret* instruction [45]). Figure 2 shows an example in which the adversary initially locates the payload (a number of return addresses and necessary data in the red dotted box) in the application's stack or heap (Step 1). The adversary launches an ROP attack by leveraging buffer overflow or use-after-free vulnerabilities to overwrite the return address *A* (Step 2) [53]. The control flow is then hijacked, i.w., it is illegally transferred to gadget *1* (Step 3). This gadget *1* changes the stack pointer (*%esp* in x86 architectures) to the beginning of the payload (Step 4) and redirects the control flow to the next gadget by executing a *ret* instruction (Step 5). These gadgets are then executed one by one until the system is compromised.

2.3 Control-Flow Integrity

Abadi et al. [1] introduced CFI by instrumenting software with runtime label checks. Figure 3 shows an example of CFI. There are two kinds of control-flow transitions in this example. The first is the function pointer `foo_ptr`, which holds the address function `foo`. The second is the function return of `foo`. The former is an example of forward-edge control flow and the latter is backward-edge control flow. To protect the forward-edge control flow, CFI inserts a label (e.g., the `prefetchnta` instruction in Figure 3) before the function entry and a check before the indirect function `call` instruction. Backward-edge control flow integrity is generally implemented via a protected shadow stack that ensures that each *ret* instruction returns only to its call site [1, 6].

Such software-based instrumentation with shadow stacks introduces high-performance overheads [1, 13]. Researchers proposed coarse-grained CFI approaches to alleviate this problem [60, 62]. These coarse-grained CFI approaches, such as avoiding shadow stacks, have better performance [6], but they are not secure enough to thwart some recent attacks [6, 7, 9]. Thus, recent research focuses on implementing fine-grained CFI [25, 52, 54]. Tice et al. [54] enforce fine-grained forward-edge control-flow integrity in GCC and LLVM. The compiler inserts a check before every call to verify that the destination is within a jump table recording the valid targets for indirect calls [20]. For the state-of-the-art CFI mechanisms, the average runtime overhead is 4.0% for forward-edge control-flow integrity [54] and 9.7% for backward-edge control-flow integrity [13].

Hardware-based CFI approaches [5, 8, 16, 17, 52] incur lower performance overheads than software approaches. Intel now provides an interface to its hardware solutions via an augmented Instruction Set Architecture (ISA) [10]. Budiu et al. [5] introduce new CFI instructions and a CFI label register to enforce label checks on indirect branches. Since a subroutine could be called by different routines, the compiler inserts the same label at each possible call site. This allows the attacker to *bend* the backward-edge control-flow [6]. Davi et al. [16, 17] address this problem by decoupling source and 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 be to an active function.

3 THREAT MODEL AND DESIGN GOALS

Here, we focus primarily on ROP attacks by remote adversaries against user-space applications We assume that forward-edge CFI has been efficiently enforced by previous solutions and that an adversary can exploit memory corruption vulnerabilities to launch ROP attacks. We assume that adversaries have no control over the operating system kernel, which ensures that its services, such as Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR), cannot be subverted. The DEP mechanism prevents adversaries from modifying code regions or mapping data regions as executable. The ASLR mechanism ensures that stack space is randomly allocated. We assume the loaded program is benign but may contain memory safety errors. Adversaries could conduct side-channel attacks to get the application memory layout, and they could leverage the memory errors to read arbitrary application code. This would give them full control over the program's stack and heap. For example, adversaries may keep forking malicious processes to collect the return addresses and their corresponding RAMACs.

To overcome the shortcomings of previous mechanisms, RAGuard needs the following properties.

- ***P1* Security:** The RAMAC computation and verification should be completely isolated from software, and the encryption key should never leave the chip.
- ***P2* User-transparency:** Developers should not need to design programs differently to make use of RAGuard.
- ***P3* Low cost:** RAGuard should cause no substantial increase in chip area or manufacturing complexity.

4 RAGUARD MECHANISM

Our RAGuard mechanism guarantees backward-edge CFI by verifying the integrity of return addresses. Figure 4 shows the traditional versus RAGuard stack layouts. RAGuard binds an RAMAC to each stored return address. The RAMAC is computed automatically and solely by hardware,

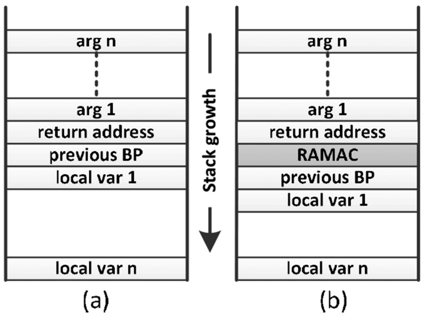


Fig. 4. Comparison between (a) the traditional stack layout and (b) the RAGuard stack layout.

without any support from the OS. When a function is called, an RAMAC is stored directly on the program stack with the corresponding return address. When the function returns, the RAMAC of the return address is loaded from the program stack and verified (property *P1*).

RAGuard has the following advantages over shadow stacks. Shadow stacks track called functions as raw data, and thus they must be stored in protected or dedicated memory regions to prevent information leakage attacks. The OS must save and restore each process's shadow stack contents during context switches and must handle stack overflow. In contrast, RAGuard guarantees the integrity of a return address by checking its RAMAC. Since the (encrypted) RAMACs are treated as normal local variables, RAGuard is much more flexible and secure than a shadow stack.

Informally, a MAC consists of three algorithms: a key generation algorithm, a signing algorithm, and a verifying algorithm. Realizing the benefits of RAGuard requires answering the following questions. (i) How do we generate a key such that the key management is completely isolated from software (Section 4.1)? (ii) How do we compute an RAMAC such that it can be used to verify the integrity of return addresses (Section 4.2)? (iii) How do we modify the processor architecture to support the signing and verifying algorithms (Section 4.3)? (iv) How do we implement RAGuard with acceptable performance overheads (Section 4.4)?

4.1 Key Management Based on a PUF and a TRNG

An RAGuard key should be generated when the process is forked. To completely isolate key management from software, a key should never leave the hardware to be read by software. However, it is challenging to generate the key when the process is scheduled to run again. The properties of Physical Unclonable Functions (PUFs)—reliability, uniqueness, and unclonability [4, 24]—make them appealing for RAGuard key management. Reliability comes from the fact that a PUF consistently generates the same response to a given challenge.[2] This property guarantees that the same key is recovered when the process is scheduled to run again. Uniqueness means that the key from different PUFs (i.e., chips) are never the same. As PUFs exploit the uncontrollable variations in the fabrication process, they are practically impossible to duplicate.

The PUF is used to generate the encryption key when a process is newly forked or rescheduled to run (property *P1*). Long response latency increases the cost of context switching.[3] Thus, the response time of the PUF is critical to our RAGuard mechanism. The cost of a process switch varies significantly among processors and operating systems [14], but it usually takes about 1–200ms [36, 63]. Table 1 lists several candidate PUFs. All incur little context switch time and area

[2]For PUF, an input and its corresponding output are called as a challenge-response pair.
[3]In this work, a context switch means process switching only. The cost of context switching is the time spent in the OS to put one process to sleep and to wake another to run.

Table 1. Candidates for the PUF Module

	Structure	Response latency	Area overhead
err-PUF[1] [58]	cell error rate distribution of STT-RAM	2.32µs	$2.9 \times 10^2 \mu m^2$
Bitline PUF [28]	SRAM with modified wordline drivers	64 memory writes and one memory read[2]	2,048GE[3]
MECCA PUF [33]	SRAM with a programmable delay generator	two memory writes and one memory read	20GE[4]
LR-PUF[5] [32]	arbiter PUF with reconfiguration control logic	1,069 clock cycles (about 1 µs at 1GHz)	6,974GE
VIA PUF [30]	via holes between two metal layers	one memory read	nearly zero[6]

[1]The response latency and area overhead are evaluated within a 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 (2.6µs) [28].
[3]Bitline PUF's area overhead amounts to a single flip-flop and two logic gates (eight Gate Equivalents (GE)) per row of SRAM. The area overhead is estimated based on a 256×256 SRAM array.
[4]MECCA PUF's area overhead is a programmable delay generator whose area is estimated based on four duty cycles.
[5]LR-PUF is run with an 80-bit challenge line and 64-bit response line.
[6]VIA PUF requires dedicated reading circuits.

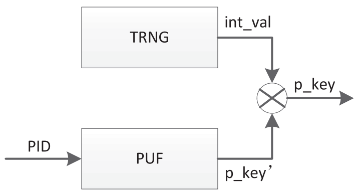


Fig. 5. Key management based on PUF and TRNG.

overhead. However, err-PUF [58], Bitline PUF [28], and MECCA PUF [33] must modify the memory control logic. In contrast, LR-PUF [32] and VIA-PUF [30] can be integrated into the system as IP (intellectual property) cores. We prefer the latter option, as it adds almost no area overhead and can be implemented without bit error [30] (property *P3*).

RAGuard's current key-management implementation uses the process ID (PID) as the process characteristic for the PUF challenge. If the PUF response is used directly as the key for the current process, then attackers may fork a large number of processes to collect return addresses and their corresponding RAMACs for later use. An attacker may perform replay attacks on a process with the anticipated PID by replacing the return address at location x with a collected return address at the same location. We deploy two mechanisms to prevent replay attacks. First, stack space is randomly allocated at runtime. This makes it more difficult for the attacker to replace the return address with a collected return address. The relationship between PID and the encryption key is configured during the power-up process. As shown in Figure 5, we use a True Random Number Generator (TRNG) [50, 56] to generate the initial value (*int_val*) during power-up. *int_val* is valid throughout the runtime. When a process is launched or scheduled to run, its PID is used for the PUF challenge. The key for the current process (*p_key*) is generated by XORing the PUF response (*p_key'*) with the initial value. van der Leest et al. implement a TRNG based on an SRAM [56], which is available in most Integrated Circuits (ICs). Thus, a TRNG could be implemented with almost no area overhead (property *P3*). In 64-bit architectures, the maximum PID number is $0 \times 400{,}000$ (4,194,303) [14], which can be represented in 22 bits. The inputs and outputs of the PUF

Table 2. Open Projects of Pipelined AES Implementations

Project	# LUTs	# BRAMs	Latency
Muehlberghuber [40]	8,319	—	12 cycles
Hsing [29]	1,686	59	21 cycles
Das [15]	3,314	38	32 cycles
Strömbergson [51]	3,155	—	46 cycles

module are 32 and 128 bits, respectively. Since the input width is larger than that of PIDs, we can include more process-specific information in the future.

4.2 RAMAC Computation

To verify the integrity of a return address, the RAMAC should define the signature (or context) of the return address. As claimed in CCFI [39], this signature contains the the return address (RA) itself and the stack pointer (SP). The former is the key information that the MAC verifies. The latter gives the position of the return address in the program stack. Including the SP in the signature ensures that an attacker cannot swap return addresses by simply copying a return address along with its MAC (property *P1*).

First, we derive the signature (*SIG*) of the return address,

$$SIG = RA||SP, \tag{1}$$

where || denotes concatenation. The RAMAC is computed as the hash value of *SIG*,

$$RAMAC = HASH_K(SIG), \tag{2}$$

where $HASH_K$ is a hardware cryptographic hash function (property *P1*). The AES-CMAC [47] authentication algorithm is based on a Cipher-based Message Authentication Code (CMAC) with AES-128 [41] and is used to implement the cryptographic hash function. As the key generated by PUF corresponds to the PID, attackers cannot swap a return address of one process with one of another. In 32-bit architectures, the signatures are 64 bits. The length of the signatures is less than the block length (128 bits) of AES-128. We pad the signatures with a bit string to bring length up to the block length [47]. Here, the bit string is a single “1” followed by 63 “0”s. In 64-bit architectures, the signature length equals the block length, and there is no need for adjustment.

Since the RAMAC is stored on the program stack along with the return address, its width should be the same as the return address (64 bits in 32-bit systems), which gives us

$$RAMAC_{32} = RAMAC[31:0]. \tag{3}$$

Likewise, 128-bit RAMACs are represented as

$$RAMAC_{64} = RAMAC[63:0]. \tag{4}$$

AES can be implemented in software or hardware, with the latter providing more physical security and higher speed. AES architectural designs are driven by system requirements in terms of latency, resources, and frequency. Table 2 lists four open-source pipelined AES implementations. The results shown are from implementations on a Xilinx Artix-7 XC7A100T-csg324-1 FPGA device. We choose these open-source projects to study the impact of AES latency on our hardware mechanism.[4] We discuss their impact on performance in Section 5.2.

[4]RAGuard works in parallel with the processor pipeline, and so the frequency of the AES design is also critical. AES can be implemented at high frequencies. For example, the design of Muehlberghuber [40], which is synthesized with 65nm CMOS technology can run above 600MHz. Sub-pipelining techniques can also be employed to increase operational frequency [46].

```
Extended call semantics:
IF relative near call
    THEN IF OperandSize = 32
        THEN
            tempEIP ← EIP + DEST
            push(EIP);
            ramac_c ← fun(EIP);
            push(ramac_c);
            EIP ← tempEIP;
        FI
FI
```

```
Extended ret semantics:
IF near return
    THEN IF OperandSize = 32
        THEN
            ramac_v ← pop();
            EIP' ← pop();
            ramac_c ← fun(EIP');
            IF match(ramac_v, ramac_c)
                EIP ← EIP';
            ELSE
                exception();
            FI
        FI
FI
```

Fig. 6. Extended instruction semantics to support RAMAC.

4.3 Baseline Design

RAGuard introduces two dedicated registers—*ramac_c* and *ramac_v*[5]—and extends the semantics of the *call* and the *ret* instructions to perform the RAMAC computation and verification, respectively. Figure 6 shows how RAGuard extends the semantics of the original *call* and *ret* instructions such that these two instructions access *ramac_c* and *ramac_v* implicitly.

call: In addition to pushing the return address on the stack and transferring control to the target function, RAGuard computes the RAMAC of the return address and stores it in *ramac_c*. Then the computed value is pushed on the stack next to the return address.

ret: In addition to popping the return address off the stack and adjusting *%esp*, RAGuard loads the RAMAC from the stack into *ramac_v*. RAGuard recomputes the RAMAC for the popped return address and compares it with the oldest one in *ramac_v*. A mismatch indicates that either the popped return address or the RAMAC was modified by attackers, and RAGuard raises an exception.

These new registers are invisible to software. As parts of the atomic operation of the *call* and the *ret* instructions, they introduce no OS modification and only marginal hardware cost. Thus, our RAGuard mechanism is transparent to program developers (property *P2*). However, pushing the RAMAC on the program stack breaks the relative positions of the frame pointer and parameters. RAGuard therefore needs a compiler patch to allocate address space appropriately.

4.4 Optimization Based on Leaf Functions

We next analyze the impact of our RAGuard mechanism on different execution sequences of function calls. We identify several opportunities to avoid unnecessary RAMAC *store* and *load* operations.

As shown in Figure 7(a), function `sort2()` calls function `sort()`, then function `sort()` calls function `lt()`. Functions like `lt()`, which does not make further calls, are called *leaf functions* [2]. The number “1” represents the return address of function `sort()`, the number “2” represents the return address of function `lt()`. For RAGuard, the RAMAC operations corresponding to the *call* and *ret* instructions are shown in Figure 7(b). When a *call* instruction is detected, RAGuard computes the RAMAC of the return address (C-1 and C-2) and stores the RAMAC on the stack (S-1 and S-2). When a *ret* instruction is detected, RAGuard loads the RAMAC of return address from the stack (L-1 and L-2) and verifies the integrity of the return address (V-1 and V-2). The RAMAC of return address “2” is stored on the stack (S-2) and then loaded from the stack (L-2) sequentially. In

[5]The *ramac_v* register is implemented as a First-In-First-Out (FIFO) buffer that decouples the pipeline and RAMAC verification.

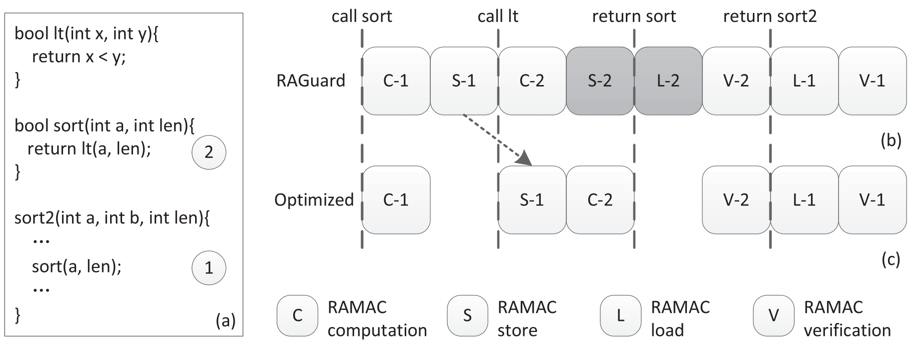


Fig. 7. Example program fragment and an outline of the execution sequences of the *call* and the *ret* instructions.

this situation, we could save the return address of the leaf-function *call* on the chip and verify its integrity by comparing it with the saved return address directly. Consequently, the RAMAC store (S-2) and load (L-2) for can be avoided. We thus introduce a dedicated register that is invisible to software to save the latest return address.

A similar optimization based on leaf functions has been reported in CCFI [39], which relies on the compiler to statically identify leaf functions. However, the hardware pipeline cannot identify a leaf function when it is called, and the RAMAC may not be fully generated when the `call` instruction completes. To deal with these challenges, we defer storing the RAMACs until functions are verified to be non-leaf functions. As shown in Figure 7(c), a function is not a leaf function if there is no *ret* executed in between its corresponding *call* and the immediate following *call*. The RAMAC of such a non-leaf function is stored on the stack when this following *call* is executed.

To determine the likely impact of these optimizations, we extract the leaf function calls of the C and C++ benchmarks from SPEC CPU2006 [49]. We instrument the function entry and exit points [64] and ran the benchmarks on an Intel machine to collect runtime profiling information. Table 3 shows the percentage of leaf function calls. Most of the benchmarks (12 of 19) contain up to 70% leaf calls. Leaf calls account for half of the calls in the remaining benchmarks (7 of 19). Five benchmarks have over 95% leaf calls. These results indicate that optimizing for leaf functions could save most of the RAMAC load/store operations.

4.5 Handling `setjmp()` and `longjmp()`

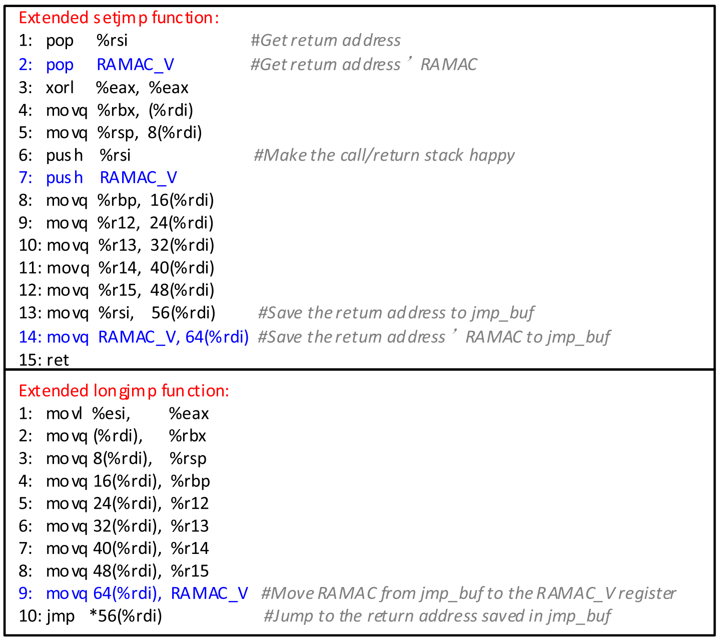
RAGuard correctly handles `setjmp()`/`longjmp()` functions. `setjmp()` stores the context information for a predefined location into a jump buffer (`jmp_buf`). Then `longjmp()` uses the saved context information to quickly return execution to the predefined location. To correctly handle these calls, we rewrite the `setjmp()` and `longjmp()` functions as shown in Figure 8. `setjmp()` gets the RAMAC from the stack and saves it to the `jmp_buf`. When `longjmp()` is about to return, the RAMAC is moved from `jmp_buf` to the RAMAC_v register. The hardware detects the update of the RAMAC_v register and verifies the target of the *jmp* instruction as it does the *ret* instruction.

5 IMPLEMENTATION AND EVALUATION

We implement the RAGuard mechanism on the open-source LEON3 processor [23]. We evaluate its performance and area overhead based on the Modelsim SE 10.2c simulator and the Xilinx Vivado Design Suite, respectively. Finally, we analyze the security of our RAGuard mechanism.

Table 3. Percentage of Leaf Function Calls

Applications	# Total Function Calls	# Leaf Functions Calls	Fraction
400.perlbench	181,974	101,991	56.0%
401.bzip2	9,536,770	9,165,288	96.1%
403.gcc	96,520,855	53,349,059	55.2%
429.mcf	87,399,945	84,274,221	96.4%
445.gobmk	1,608,123	960,508	59.7%
456.hmmer	63,255,863	31,703,700	50.1%
458.sjeng	66,572,836	50,622,855	76.0%
462.libquantum	639,881	485,511	75.9%
464.h264ref	78,762,299	70,579,741	89.6%
471.omnetpp	165,078,087	93,292,028	56.5%
473.astar	681,047,522	566,481,863	83.2%
483.xalancbmk	125,204,873	96,046,228	76.7%
433.milc	196,131,289	195,858,156	99.8%
444.namd	219,233,757	218,436,801	99.6%
447.dealII	514,340,741	261,184,875	50.8%
450.soplex	2,532,788	2,048,022	50.9%
453.povray	166,991,701	122,737,149	73.4%
470.lbm	81	77	95.1%
481.sphinx	21,973,395	15,876,815	72.2%



```
Extended setjmp function:
pop    %rsi                  #Get return address
pop    RAMAC_V               #Get return address’ RAMAC
xorl   %eax, %eax
movq   %rbx, (%rdi)
movq   %rsp, 8(%rdi)
push   %rsi                  #Make the call/return stack happy
push   RAMAC_V
movq   %rbp, 16(%rdi)
movq   %r12, 24(%rdi)
movq   %r13, 32(%rdi)
movq   %r14, 40(%rdi)
movq   %r15, 48(%rdi)
movq   %rsi,   56(%rdi)      #Save the return address to jmp_buf
movq   RAMAC_V, 64(%rdi)     #Save the return address’ RAMAC to jmp_buf
ret
```

```
Extended longjmp function:
movl  %esi,       %eax
movq  (%rdi),     %rbx
movq  8(%rdi),    %rsp
movq  16(%rdi),   %rbp
movq  24(%rdi),   %r12
movq  32(%rdi),   %r13
movq  40(%rdi),   %r14
movq  48(%rdi),   %r15
movq  64(%rdi),   RAMAC_V    #Move RAMAC from jmp_buf to the RAMAC_V register
jmp   *56(%rdi)              #Jump to the return address saved in jmp_buf
```

Fig. 8. Rewritten `setjmp()` and `longjmp()`. functions.

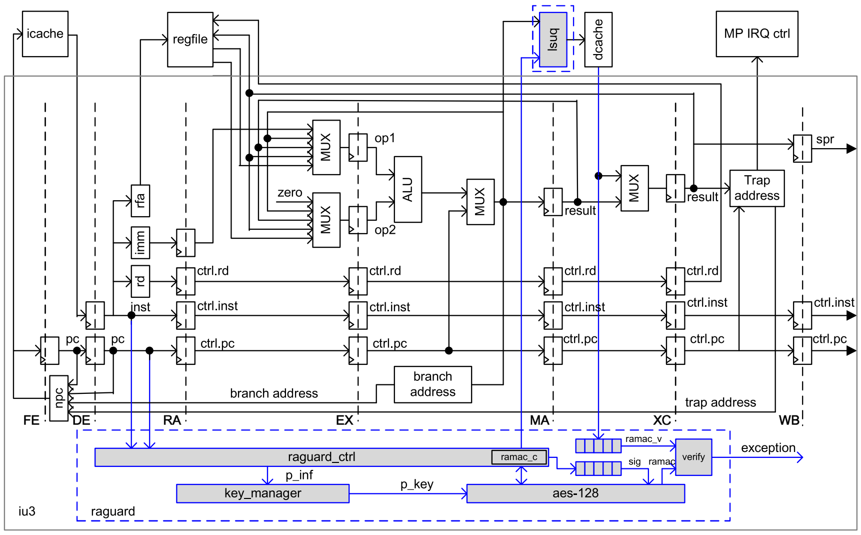


Fig. 9. Our RAGuard mechanism is implemented in a LEON3 (SPARC) processor, which uses a seven-stage pipeline (Instruction Fetch (FE), Decode (DE), Register Access (RA), Execute (EX), Memory (ME), Exception (XC), and Write (WR)). We add an RAMAC generator module (*raguard*, inside the dashed blue box) and associated data paths (denoted as dotted blue lines) to the LEON3 pipeline.

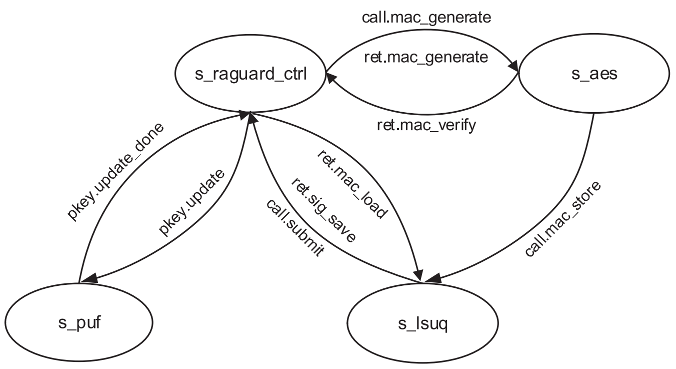


Fig. 10. The state machine of *raguard_ctrl* module includes: RAGuard control stage (s_raguard_ctrl), encryption key update stage (s_puf), RAMAC generation stage (s_aes), and RAMAC load/store stage (s_lsuq).

5.1 Details of RAGuard

LEON3, which is distributed by the European Space Research and Technology Centre, is a synthesizable 32-bit processor compliant with the SPARC V8 architecture. It implements a single-issue instruction pipeline with seven stages, as shown in Figure 9. We add a hardware RAMAC computation and verification module *raguard* to the *iu3* module. The structure of *raguard* is illustrated inside the dashed box of Figure 9. The *raguard* module contains the following components: *raguard_ctrl*, *key_manager*, *AES-128*, and *lsuq* (the load-store update queue). The added data paths are denoted by dotted blue lines. To facilitate the RAMAC computation, we also add three registers (*p_inf*, *p_key*, and *ramac_c*) and two FIFOs (*ramac_v* and *sig*). The *raguard_ctrl* module monitors instructions at the decode stage (DE), and maintains a state machine to control RAMAC computation and verification. The state machine is shown in Figure 10.

When a process is scheduled to run, the OS process ID (PID)[6] is reset. At the same time, the value of register *p_inf* is automatically updated. The *raguard_ctrl* module sends a process encryption key update command (*pkey.update*) to the *key_manager*, and its status becomes *s_puf*. The *key_manager* module uses the new value of the *p_inf* register as its input, and its output is stored in the *p_key* register. When the process encryption key is finished updating, the *key_manager* sends a response (*pkey.update_done*) back to the *raguard_ctrl* module.

When a *call* instruction enters the decode stage, a command is sent to the *AES-128* module to compute the RAMAC (*call.mac_generate*), and the status of the *raguard_ctrl* module becomes *s_aes*. After receiving this command, the *AES-128* module uses the signature of the return address as its input to compute the RAMAC (Equations (1–4)). When the RAMAC is ready, the *raguard_ctrl* module sends the RAMAC (*call.mac_store*) to the *lsuq* (load store update queue) module. Then the *call* instruction is executed (*call.mac_submit*).

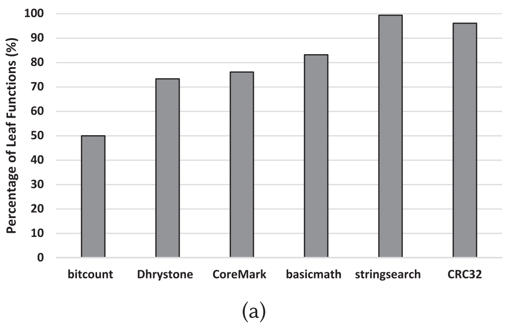
When a *ret* instruction enters the decode stage, the *raguard_ctrl* module sends a command to load the RAMAC (*ret.mac_load*) to the *lsuq* module. The RAMAC is loaded from the program stack and stored in the *ramac_v* FIFO. Meanwhile, the *ret* instruction pops the return address from the program stack in the exception stage. The *raguard_ctrl* module then stores the signature of the loaded return address into the *sig* FIFO (*ret.sig_store*). When the *AES-128* module is free and the *sig* FIFO is not empty, the *raguard_ctrl* module sends an RAMAC calculation command (*ret.mac_generate*) to the *AES-128* module (*ret.mac_generate*). After receiving the RAMAC calculation command, *AES-128* recomputes the RAMAC of that return address and compares it to the oldest one in the *ramac_v* FIFO (*ret.mac_verify*). If there is a mismatch, then an exception is raised, and execution is transferred to the exception handler.

Although our proof of concept RAGuard implementation uses a single-issue processor, RAGuard can be naturally applied to superscalar and simultaneous multithreading (SMT) processors. For superscalar processors, only one process is active at a time. The *mac_generate* commands (*call.mac_generate* and *ret.mac_generate*) from different pipelines use the same encryption key. We can implement the RAGuard mechanism in a similar way to the LEON3. The *raguard* module can be shared between the pipelines directly. For SMT processors, more than one process run on the processor core at a time. The *mac_generate* commands from different hardware threads use different encryption keys. The AES-128 cannot process *mac_generate* commands from different processes at the same time. The *raguard_ctrl* module may employ a round-robin scheduler to assign time slices of AES-128 to the hardware threads. More than one AES-128 module may be used to accelerate RAMAC computation.

5.2 Runtime Overhead

The percentages of leaf functions and the call frequencies per 1K instructions in our benchmarks are illustrated in Figure 11. Table 4 gives the details of the experimental setup. To evaluate RAGuard's performance overhead, workloads are loaded to the DRAM. The entry point is 0x4000_0000 and the stack pointer is set to 0x47FF_FFFF. Since we did not modify the compiler, the RAMACs are stored at shadow locations to guarantee the correct execution. In other words, the RAMACs are stored in a separate location in DRAM, instead of on the stack as Figure 4 shows. For example, if the return address is 0x47FF_FE04, the RAMAC is stored at 0x45FF_FE04 during

[6]To reduce the software overhead of TLB maintenance, the Address Space Identifier (ASID) is used to identify pages associated with a specific process. The hardware can monitor changes to the ASID and decide whether there is a context switch. For Intel 64 and IA-32 architectures, the current ASID (also called the Process-Context Identifier (PCID)) is the value of bits 11:0 of CR3 [31]. For the LEON3, the context number is the corresponding ASID [48].



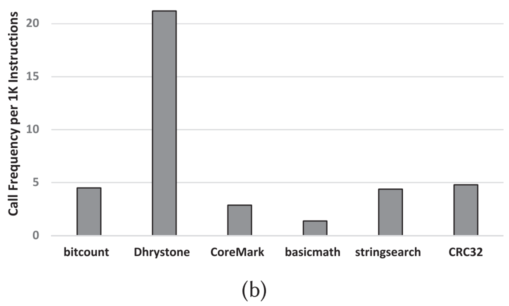


Fig. 11. (a) Percentage of leaf functions and (b) call frequency per 1K instructions for the six benchmarks.

Table 4. Experimental Setup

LEON3 CPU	7 stage, single issue
I-cache	16KB 4-set, 4KB/way, 32B/line
D-cache	16KB 4-set, 4KB/way, 32B/line
DRAM	128MB DDR2 (0x4000_0000 0x47FF_FFFF)
Compiler	sparc-elf-4.4.2
Workload	bitcount, Dhrystone, CoreMark, stringsearch, CRC32
Simulator	Modelsim SE 10.2c
Synthesis Tool	Xilinx Vivado Design Suite

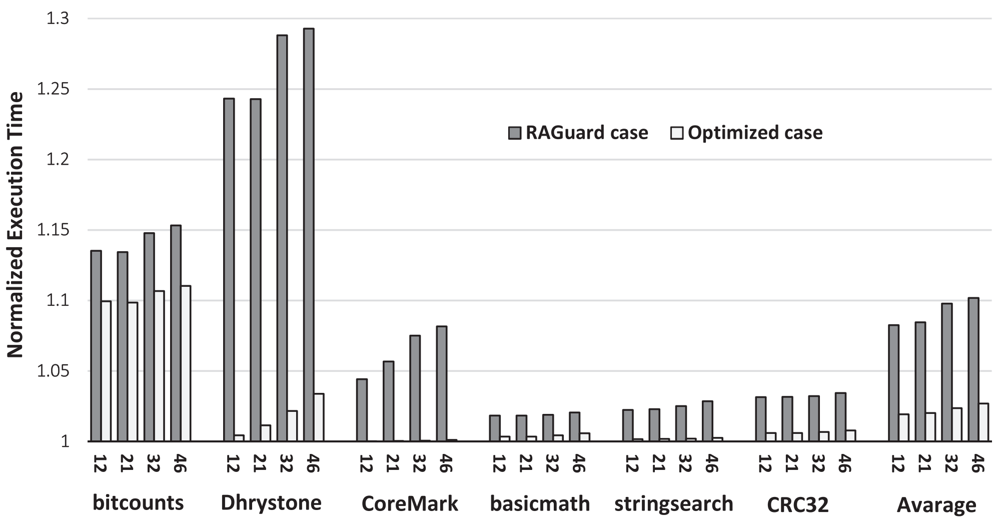


Fig. 12. Runtime overhead of different implementations.

evaluation. Meanwhile, to analyze the effect of AES latency on performance, we integrate the four AES implementations (shown in Table 2) into our projects.

Figure 12 shows the runtime overhead normalized to the baseline, i.e., the original LEON3 processor from Cobham Gaisler [23]. The RAGuard case represents the design described in Section 4.3, and the optimized case represents the RAGuard design with leaf-function optimization described

Table 5. Increased Hold Time in Cycles for D-Cache, I-Cache, and RAMAC

		Increased H_{Dcache} (cycles)		Increased H_{Icache} (cycles)		Increased H_{RAMAC} (cycles)	
		RAGuard case	Optimized case	RAGuard case	Optimized case	RAGuard case	Optimized case
bitcount	12	12,565	9,592	120	67	835	171
	21	12,501	9,603	−13	−196	958	402
	32	12,685	9,770	198	192	2,146	829
	46	12,423	9,658	118	167	2,976	1,406
Dhrystone	12	27,431	593	53	152	2,188	71
	21	23,560	589	46	36	6,027	1,048
	32	20,566	587	94	42	14,432	2,287
	46	19,417	455	−56	−41	16,025	3,868
CoreMark	12	8,693	312	−4	97	350	36
	21	8,608	312	−133	92	2,931	67
	32	8,533	310	115	−11	6,568	140
	46	8,593	305	−56	82	7,993	251
basicmath	12	1,332	233	92	26	142	32
	21	1,323	233	91	20	166	54
	32	1,314	261	50	44	293	148
	46	1,315	261	3	41	457	275
string	12	2,955	218	58	4	179	30
	21	2,924	218	94	8	172	40
	32	2,930	218	75	−1	184	95
	46	2,978	211	0	−10	197	217
CRC32	12	1,209	223	120	125	139	34
	21	1,203	223	108	119	178	45
	32	1,198	220	79	105	262	74
	46	1,307	216	204	80	411	130

in Section 4.4. The numbers on the x axis denote the latencies of the AES implementations in Table 2. As expected, our optimization mechanism efficiently reduces RAGuard's runtime overhead. For the AES with a latency of 12 cycles, RAGuard incurs 8.3% performance overhead, on average. Its worst-case degradation is 24.3% for `Dhrystone`. The optimized RAGuard incurs almost no performance degradation except for `bitcount`, which has a much lower percentage (49.9%) of leaf function calls than the other benchmarks.

The relatively large degradations for the basic (unoptimized) RAGuard are not surprising. Since we extend the semantics of the *call* and *ret* instructions to insert the RAMAC operations, there is more contention on the bus, causing extra pipeline stalls. For LEON3, the delay consists of three parts: D-cache hold time (H_{Dcache}), I-cache hold time (H_{Icache}), and RAMAC hold time (H_{RAMAC}). When D-cache, I-cache, and *raguard_ctrl* cannot submit their command immediately, they stall the pipeline. Table 5 shows the increases in the number of cycles spent in each type. The I-cache hold time is increased little by our hardware mechanism, but it substantially increases the D-cache hold time, especially for `bitcount` and `Dhrystone`. It is notable that the the RAMAC hold time is much smaller than the extra D-cache hold time. That means the RAMAC computation and verification degrade performance very little when the AES latency is small. Thus, the large drop in performance for basic RAGuard is due to the sharply increased D-cache hold time. In contrast, the optimized RAGuard sharply decreases the extra D-cache hold time, except for `bitcount`. This shows that

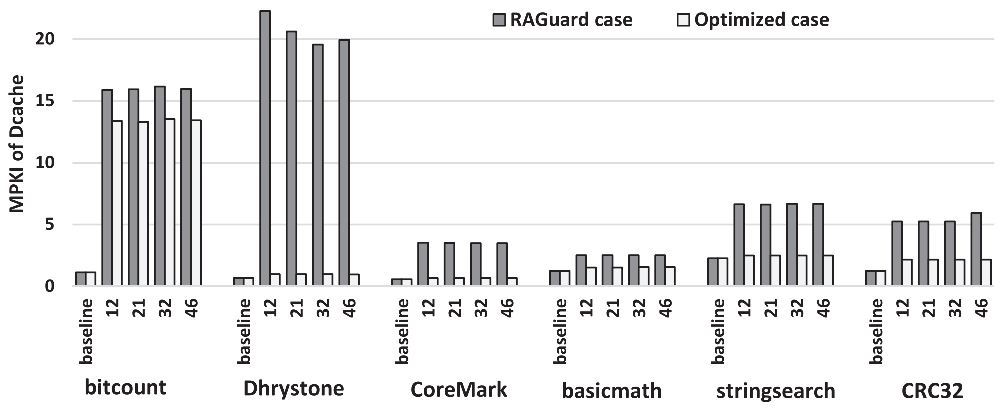


Fig. 13. D-cache miss per thousand instructions.

Table 6. RAGuard Area

	# LUTs as Logic	# LUTs as Memory	# LUT FF Pairs	# Total
Original LEON3 processor	10,150	76	2,335	12,561
raguard_ctrl	473	0	150	623
AES-128 (Muehlberghuber [40])	6,866	0	1,453	8,319

our optimization can effectively reduce the performance overheads for applications with many leaf function calls.

The sharply increased D-cache hold time for the unoptimized RAGuard is due to the high D-cache miss rate. As shown in Figure 13, the D-cache misses per thousand instructions (MPKI) increases substantially, especially for `Dhrystone`. This is corroborated by Figure 11(b), where `Dhrystone` has the highest call frequency. The increased misses may be due to our not storing the RAMAC with the corresponding return address on the program stack for these experiments If we were to modify the compiler to store the RAMAC directly with the corresponding return address, then performance will be further improved. As our optimization mechanism saves most of the load/store operations, the optimized RAGuard shows similar MPKI values to those of the original LEON3 processor. The only exception is `bitcount`, which has 49.9% leaf-function calls.

The long latency of the AES used in RAGuard slightly increases the performance overhead. As shown in Figure 12, the average performance degradation of RAGuard increases from 8.2% to 10.1% when the AES latency increases from 12 to 46 cycles. The average performance degradation of the optimized RAGuard increases from 1.9% to 2.6% when the AES latency increases from 12 cycles to 46 cycles. The optimized RAGuard incurs negligible performance overhead for application with higher percentage (above 70%) of leaf functions.

5.3 Resource Overhead

We synthesize and implement our RAGuard prototype with Xilinx Vivado HLx 2016.2 [57]. The required FPGA resources are shown in Table 6. The second row shows the resources used by the unmodified LEON3 processor. The extra resources needed by our RAGuard mechanism comprise three parts: *raguard_ctrl*, *AES-128*,[7] and *key_manager* (or PUF). The resource overheads

[7]We choose the biggest AES implementation from Table 2.

of the former two are shown in the third and fourth rows, respectively, and the resource overhead of the PUF is estimated in Table 1. The *raguard_ctrl* module requires 623 extra LUTs/FFs, which is only 4.96% of the baseline. *AES-128* consumes more FPGA resources (8,319 LUTs/FFs). The total FPGA resources required by RAGuard (8,942 LUTs/FFs) amount to a little more than half (58.9%) of those required by the original Xilinx MIGv7 memory controller (15,178 LUT/FF) (property *P3*).

5.4 Security Analysis

We restrict our discussion to attacks that leverage backward control-flow edges, such as ROP attacks. Note that RAGuard security also relies on the complexity of the RAMAC computation. In ROP attacks, the adversary overwrites return addresses on the stack and locates payloads in the application's heap/stack (as shown in Figure 2). Malicious return addresses in the payload may redirect control flow to arbitrary locations, unintended call sites, or even valid potential call sites within the CFG [6]. In the first case, attackers must compute the RAMACs for their dummy return addresses. In the latter two cases, attackers might carry out splicing attacks by constructing payloads with the existing return address and the corresponding RAMAC. This approach violates our RAMAC verification, as well: if a return address and its corresponding RAMAC are inserted into another location, the stack pointer of the new location disagrees with the RAMAC. Hence, attackers must also recompute the RAMACs for their malicious return addresses.

As noted in Section 4.2, the RAMAC is computed with AES-128, which encrypts the signature of the return address. A PUF module dynamically updates the encryption key on context switches. Since the encryption key never leaves the hardware and the *p_key* register is only accessed by hardware, adversaries cannot obtain the encryption key to produce a valid RAMAC. Moreover, the *p_key* register is updated at runtime, which makes the RAMAC less vulnerable to brute-force attacks.

Attackers may fork a large number of processes to collect return address and RAMAC pairs for a specific PID. When the PID is reused, the attackers could conduct replay attacks by replacing the return address at location x with a collected return addresses at the same location.

6 RELATED WORK

6.1 Using Shadow Stacks

Shadow stacks are considered to be an essential mechanism for the security of CFI [1, 6, 10]. They usually work by keeping a copy of the return address when a function is called. When the function returns, it uses the return address stored on the shadow stack to ensure integrity. Although shadow stacks enforce strong backward-edge CFI policy, they present several implementation challenges. First, the effectiveness of the shadow stack relies on its integrity. The shadow stack must be isolated against the software it protects. Information hiding is often used to secure software shadow stacks [13, 34, 38] with acceptable performance overhead. Unfortunately, these shadow stacks are vulnerable to information leakage attacks [9, 21, 42]. Hardware-assisted shadow stacks [22, 35, 43] copy the return addresses into a dedicated memory region. Intel recently introduced Control-Flow Enforcement Technology (CET) [10], in which hardware-based shadow stacks verify return addresses. Furthermore, CET provides indirect branch tracking to validate forward CFI edges. Such indirect branch tracking complements our RAGuard mechanism. Both mechanisms rely on the OS to save and restore the contents of shadow stack on context switches and stack overflows [22, 43, 59].

The use of `setjmp()/longjmp()` violates *call* and *ret* matching. `setjmp()` saves the current execution environment into a platform-specific data structure (`jmp_buf`). Then `longjmp()` restores that saved program state to continue execution after the `setjmp()`, allowing functions to unwind multiple stack frames. To deal with this corner case, StackGhost [22] walks back through the program's stack to find the corresponding `setjmp()` and manipulates the shadow stack accordingly. Smashguard [43] pops return addresses off the shadow stack until a match is found or the shadow stack is empty. Davi et al. [16, 17] design a hardware shadow stack variant that allocates 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 an active function. Their approach tracks called functions by their label status instead of the first-in, last-out principle. In doing this, they avoid having to do anything special to correctly handle mismatches between `calls` and `rets`.

In contrast to these prior efforts, RAGuard uses RAMACs to verify the integrity of return addresses, rather than tracking the return addresses precisely using a shadow stack. While these prior efforts require dedicated memory and require that the OS handle process context switches and stack overflows, RAGuard relies on cryptography to verify the integrity of return addresses. RAGuard can also correctly handle mismatches between `call` and `ret` instructions by rewriting the `setjmp()` and `longjmp()` functions (as shown in Figure 8).

6.2 Using a Reversible Transform of the Return Address

A reversible transform can be used to prevent attackers from overwriting the stored return address with a legal value. When the return address needs to be stored, a reversible transform is applied to it and the result is stored on the stack instead. When the return address is popped from the stack, the reverse transform is applied to it before it is used. If attackers do not know the transform or the key to the transform, then they cannot overwrite the return address with their intended value.

StackGhost [22] applies XOR encryption to the return address value. As this approach is proposed based on the SPARC architecture, the transform is used whenever a register window overflow or underflow occurs. Similarly, PointGuard [11] XORs a key with every pointer that is loaded from or stored to memory. Unfortunately, both of approaches are susceptible to information leakage attacks. The XOR key can be inferred if attackers can read the transformed return address off the stack.

To prevent information leakage attacks, Tuck et al. [55] protect the return address with cryptography implemented in hardware. When the return address is loaded back, it is decrypted with the inverse function before being used. However, this approach is susceptible to ROP attacks. Attackers may swap a return address stored in one memory address with a pointer stored in a different memory address.

6.3 Using MACs with Return Addresses

To prevent attacks from swapping one return address with another, a message authentication code (MAC) can be bound to the return address. StackGuard [12] is the first work to employ this approach. It places a predefined secure value on the program stack next to a return address. The secure value could be a string terminator or a randomly generated number. When a function returns, the secure value is verified by the function epilogue to make sure it has remained intact. However, applying StackGuard directly to enforce backward-edge CFI does not work for two reasons. First, attackers could overwrite the secure value and the address of StackGuard's (software) security handler [37]. We therefore suggest generating and verifying the secure value automatically in hardware. Second, the secure value contains nothing about the return address, and attackers can obtain it via brute force attacks [3].

Cryptographic CFI (CCFI) [39] uses MACs instead of a predefined secure value to efficiently provide CFI protection. CCFI's MACs are implemented as a single block of AES applied to the signature of the pointer. Because CCFI randomly generates the AES key at program initiation, it also needs the OS to save and restore the MAC key during context switches. CCFI incurs non-negligible performance overheads, even though it takes advantage of cryptographic CPU instructions (AES.NI). In contrast, RAGuard uses a novel key management method based on PUFs to isolate key management from software, and it improves the performance of backward-edge CFI with a hardware AES-128 function. Furthermore, CCFI relies on the compiler to statically identify leaf functions to reduce the cost of protecting return addresses. Instead, RAGuard proposes a dynamic hardware-based approach to avoid unnecessary *store* and *load* operations.

7 CONCLUSIONS

This work presents RAGuard, an efficient, user-transparent hardware mechanism for backward-edge Control Flow Integrity. RAGuard binds each return address with an RAMAC to guarantee its integrity on the program stack. Since the security of the RAMAC is guaranteed by a PUF module and a hardware cryptographic hash function, the RAMAC can be stored directly on the program stack. By dynamically avoiding storing the RAMAC for leaf functions, our optimized RAGuard incurs negligible performance overheads for application with higher percentages (above 70%) of leaf function calls. If the RAMAC is stored directly with the corresponding return address, then performance will be further improved, including for applications with few leaf function calls.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*. 340–353.

[2] ARM Limited. 2011. What Is a Leaf Function? Retrieved from http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13785.html.

[3] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking blind. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'14)*. 227–242.

[4] Christoph Böhm and Maximilian Hofer. 2012. *Physical Unclonable Functions in Theory and Practice*. Springer Publishing Company, Inc.

[5] Mihai Budiu, Úlfar Erlingsson, and Martín Abadi. 2006. Architectural support for software-based protection. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID'06)*. 42–51.

[6] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15)*. 161–176.

[7] Nicholas Carlini and David Wagner. 2014. ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium (SEC'14)*. 385–399.

[8] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. 2016. HCFI: Hardware-enforced control-flow integrity. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY'16)*. 38–49.

[9] 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 *Proceedings of the 22rd ACM Conference on Computer and Communications Security (CCS'15)*. 952–963.

[10] Intel Corporation. 2016. Control-flow Enforcement Technology Preview. Retrieved from https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[11] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. 2003. Pointguard TM: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th Conference on USENIX Security Symposium (SEC'03)*. 7–7.

[12] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. 1998. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium (SEC'98)*. 5–5.

[13] Thurston H. Y. 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 (ASIA CCS'15)*. 555–566.

[14] Bovet Daniel and Cesati Marco. 2006. *Understanding the Linux Kernel* (3rd ed.). O'Reilly Media, Inc., Sebastopol.

[15] Subhasis Das. 2010. Fully Pipelined AES Core. Retrieved from http://opencores.org/project,aes_pipe.

[16] 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 *Proceedings of the 52nd Design Automation Conference (DAC'15)*. 74:1–74:6.

[17] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. 2014. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Proceedings of the 51st Design Automation Conference (DAC'14)*. 133:1–133:6.

[18] Lucas Vincenzo Davi. 2015. *Code-Reuse Attacks and Defenses*. Ph.D. Dissertation. Technische Universität, Darmstadt.

[19] Ruan de Clercq and Ingrid Verbauwhede. 2017. A survey of hardware-based control flow integrity (CFI). *CoRR* abs/1706.07257 (2017).

[20] Clang 6 documentation. 2017. Control Flow Integrity. Retrieved from http://clang.llvm.org/docs/ControlFlowIntegrity.html.

[21] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. 2015. Missing the point(Er): On the effectiveness of code pointer integrity. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'15)*. 781–796.

[22] Mike Frantzen and Mike Shuey. 2001. StackGhost: Hardware facilitated stack protection. In *Proceedings of the 10th Conference on USENIX Security Symposium (SEC'01)*. 5–5.

[23] Cobham Gaisler. 2017. LEON3 Processor. Retrieved from http://www.gaisler.com/index.php/products/processors/leon3.

[24] Yangsong Gao, Damith C. Ranasinghe, Said F. Al-Sarawi, Omid Kavehei, and Derek Abbott. 2015. Memristive crypto primitive for building highly secure physical unclonable functions. *Sci. Rep.* 5, 12785 (2015), 1–14.

[25] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. 2016. Fine-grained control-flow integrity for kernel software. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P'16)*. 179–194.

[26] Enes Göktaş, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. 2016. Undermining information hiding (and what to do about it). In *Proceedings of the 25th USENIX Security Symposium (USENIX Security'16)*. 105–119.

[27] Wenjian He, Sanjeev Das, Wei Zhang, and Yang Liu. 2017. No-jump-into-basic-block: Enforce basic block CFI on the fly for real-world binaries. In *Proceedings of the 54th Design Automation Conference (DAC'17)*. 23:1–23:6.

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

[29] Homer Hsing. 2015. Tiny Encryption Algorithm. Retrieved from http://opencores.org/project,tiny_encryption_algorithm,Overview.

[30] ICTK. 2017. VIA PUF. Retrieved from http://www.ictk.com/servicenproduct/puf.

[31] Intel Corp. 2018. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide*.

[32] 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 *Proceedings of the 13th International Conference on Cryptographic Hardware and Embedded Systems (CHES'11)*. 374–389.

[33] Aswin Raghav Krishna, Seetharam Narasimhan, Xinmu Wang, and Xinmu Wang. 2011. MECCA: A robust low-overhead PUF using embedded memory array. In *Proceedings of the 13th International Conference on Cryptographic Hardware and Embedded Systems (CHES'11)*. 407–420.

[34] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. 147–163.

[35] Ruby B. Lee, David K. Karig, John P. McGregor, and Zhijie Shi. 2003. Enlisting hardware architecture to thwart malicious code injection. In *Proceedings of the 1st International Conference on Security in Pervasive Computing*. 237–252.

[36] Chuanpeng Li, Chen Ding, and Kai Shen. 2007. Quantifying the cost of context switch. In *Proceedings of the Workshop on Experimental Computer Science (ExpCS'07)*.

[37] David Litchfield. 2003. Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server. Retrieved from https://pdfs.semanticscholar.org/cf20/2abd9d5e866659be3cfbb1cb094d60d5484c.pdf.

[38] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. 2015. ASLR-guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*. 280–291.

[39] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. 2015. CCFI: Cryptographically enforced control flow integrity. In *Proceedings of the 22rd ACM Conference on Computer and Communications Security (CCS'15)*. 941–951.

[40] Michael Muehlberghuber. 2014. A High-throughput VHDL and SystemVerilog Implementation of AES-128 Including Scripts for a Full Front-end Design Process. Retrieved from https://github.com/mbgh/aes128-hdl.

[41] National Institute of Standards and Technology (NIST). 2006. FIPS 197, Advanced Encryption Standard (AES). Retrieved from http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[42] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. 2016. Poking holes in information hiding. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security'16)*. 121–138.

[43] Hilmi Ozdoganoglu, T. N. Vijaykumar, Carla E. Brodley, Benjamin A. Kuperman, and Ankit Jalote. 2006. SmashGuard: A hardware solution to prevent security attacks on the function return address. *IEEE Trans. Comput.* 55, 10 (2006), 1271–1285.

[44] Das Sanjeev, Zhang Wei, and Liu Yang. 2016. A fine-grained control flow integrity approach against runtime memory attacks for embedded systems. *IEEE Trans. Very Large Scale Integr. Syst.* 24, 11 (2016), 3193–3207.

[45] 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 (CCS'07)*. 552–561.

[46] Abolfazl Soltani and Saeed Sharifian. 2015. An ultra-high throughput and fully pipelined implementation of AES algorithm on FPGA. *Microprocess. Microsyst.* 39, 7 (2015), 480–493.

[47] JunHyuk Song, Radha Poovendran, Jicheol Lee, and Tetsu Iwata. 2006. The AES-CMAC Algorithm. Retrieved from https://tools.ietf.org/html/rfc4493.

[48] SPARC International Inc. 1998. *The SPARC Architecutre Manual Version 8*.

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

[50] Mario Stipčević and Çetin Kaya Koç. 2014. *True Random Number Generators. Open Problems in Mathematics and Computational Science*. Springer, 275–315.

[51] Joachim Strömbergson. 2017. Verilog implementation of the symmetric block cipher AES. Retrieved from https://github.com/secworks/aes.

[52] Dean Sullivan, Orlando Arias, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, and Yier Jin. 2016. Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity. In *Proceedings of the 53rd Design Automation Conference (DAC'16)*. 163:1–163:6.

[53] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal war in memory. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'13)*. 48–62.

[54] 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 *Proceedings of the 23rd USENIX Security Symposium (SEC'14)*. 941–955.

[55] Nathan Tuck, Brad Calder, and George Varghese. 2004. Hardware and binary modification support for code pointer protection from buffer overflow. In *Proceedings of the 37th IEEE/ACM International Symposium on Microarchitecture (MICRO'04)*. 209–220.

[56] Vincent van der Leest, Erik van der Sluis, Geert-Jan Schrijen, Pim Tuyls, and Helena Handschuh. 2012. Efficient implementation of true random number generator based on SRAM PUFs. *Cryptography and Security: From Theory to Applications*. Lecture Notes in Computer Science, Vol. 6805. Springer, 300–318.

[57] Xilinx inc.2017. Vivado Design Suite - HLx Editions. Retrieved from https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2016-2.html.

[58] Chaofei Yang, Beiye Liu, Yandan Wang, Yiran Chen, Hai Li, Xian Zhang, and Guangyu Sun. 2016. The applications of NVM technology in hardware security. In *Proceedings of the 26th Great Lakes Symposium on VLSI (GLSVLSI'16)*. 311–316.

[59] Shalabi Yasser, Yan Mengjia, Honarmand Nima, Ruby B. Lee, and Torrellas Josep. 2018. Record-replay architecture as a general security framework. In *Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. 180–193.

[60] 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 *Proceedings of the 2013 IEEE Symposium on Security and Privacy (S&P'13)*. 559–573.

[61] Jun Zhang, Rui Hou, Junfeng Fan, Ke Liu, Lixin Zhang, and Sally A. McKee. 2017. RAGuard: A hardware based mechanism for backward-edge control-flow integrity. In *Proceedings of the ACM International Conference on Computing Frontiers*. 27–34.

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

[63] Chen Zheng, Jianfeng Zhan, Zhen Jia, and Lixin Zhang. 2013. Characterizing OS behavior of scale-out data center workloads. In *Proceedings of the ACM 7th Workshop on the Interaction amongst Virtualization, Operating Systems and Computer Architecture*. 311–316.

[64] Intel Developer Zone. 2016. Finstrument-functions, Qinstrument-functions. Retrieved from https://software.intel.com/en-us/node/693332.

Received November 2017; revised June 2018; accepted September 2018