Control Flow Integrity Based on Lightweight Encryption Architecture

Pengfei Qiu, Yongqiang Lyu, *Member, IEEE*, Jiliang Zhang, *Member, IEEE*, Dongsheng Wang, *Member, IEEE*, and Gang Qu, *Senior Member, IEEE*

***Abstract*—Control-flow integrity (CFI) plays a very important role in defending against code reuse attacks by protecting the control flows of programs from being hijacked. However, previous CFI methods suffer from performance overheads, cost, or security issues. In this paper, we propose a new CFI based on a lightweight encryption architecture with advanced encryption standard (LEA-AES) to address the challenges above. The LEA exploits AES to encrypt and decrypt return addresses and instructions at indirect jump destinations, which protects function calls and indirect jumps from being reused by return-oriented programming (ROP) and jump-oriented programming (JOP) attacks. For ROP, the encryption and decryption of return addresses are performed when the *call* and *ret* instructions are executing; for JOP, the encryption of instructions are performed when programs are loading into memory and the decryption of instructions are performed right before they are executing. The LEA-AES does not need to revise instruction sets of CPU and its security is also guaranteed by the encryption mechanism in addition to its high performance. Experimental results showed that the run-time and loading time overheads of LEA-AES are both less than 4% and the memory overhead is 0.62%.**

***Index Terms*—Advanced encryption standard (AES), code reuse attacks (CRAs), control-flow integrity (CFI), jump-oriented programming (JOP), lightweight encryption architecture (LEA), return-oriented programming (ROP).**

I. Introduction

BEFORE code reuse attacks (CRAs) [1]–[6], code injection attacks [7], [8] were widely used by adversaries to achieve attacks. In code injection attacks, attackers often inject malicious codes into programs by exploiting the software vulnerabilities, for example, stack or heap buffer overflow vulnerabilities and string formatting bugs. Then attackers overwrite the stack with the addresses of viciously injected codes. At run-time, the malicious codes will be executed and finally the system is controlled. Data execution prevention (DEP) [9] can defend against code injection attacks efficiently, and it thus has been widely employed on most operating systems (e.g., Windows and Linux). In DEP, memory units are marked as *executable* or *nonexecutable* and only the original programs can be executed. The maliciously injected codes cannot be executed because the memories available for code injections (data area) are dedicatedly marked *nonexecutable*. However, adversaries may bypass DEP by reusing small fragments of existing codes to modify the control flow of programs, which is known as CRAs. The small fragments ending with the *ret* or *indirect jmp* instructions are called *gadgets* in CRAs.

Return-oriented programming (ROP) [1], [3]–[6] and jump-oriented programming (JOP) [2], [6] are the most important instances of CRAs. In ROP, attackers use gadgets that end with the *ret* instructions to change the execution flow of programs, so that the programs will be executed as the wishes of intruders and finally the intruders may control the systems. ROP has been theoretically verified to be turing-complete [1]. Compared with ROP, JOP attacks use gadgets which end with the *indirect jmp* instructions to assault programs. CRAs have been successfully realized on numerous architectures, such as x86 [10], ARM [11], SPARC [12], Atmel AVR [13], Z80 [14], and PowerPC [1], [3], [5], [15]. Recently, CRAs are increasingly used to hijack the programs. Many well-known software products have been attacked by CRAs successfully, such as Adobe Reader [16], [17], Adobe Flashplayer [18], or Quicktime Player [5], [19]. In addition, Hund *et al.* [20] presented an ROP-based rootkit for the Windows operating systems which can steer by kernel integrity protection mechanisms [5]. This is a very fearful menace to computers. Moreover, several tools have been developed to search the useful gadgets automatically [21], which make CRAs easier.

Many methods have been proposed to defend against CRAs recently. These defenses can be categorized into four types.

1) *Randomization-based defenses*, such as address space layout randomization (ASLR) [3]. However, it is still probable for attackers to construct enough gadgets to perform CRAs with the knowledge of some randomized codes [5].

Manuscript received September 15, 2016; revised December 27, 2016, March 11, 2017, and May 22, 2017; accepted July 16, 2017. Date of publication August 31, 2017; date of current version June 18, 2018. This work was supported in part by the National Natural Science Foundation of China under Grant 61602107 and Grant 61373025, in part by the State Key Development Program for Basic Research of China under Grant 2012CB315801, in part by the Tsinghua University Initiative Scientific Research Program and the AFOSR MURI under Grant FA9550-14-1-0351, in part by the Research Fund from the Beijing Innovation Center for Future Chip under Grant KYJJ2016005, in part by the Central Universities Fundamental Research Funding of China, and in part by the National Natural Science Foundation of Hunan Province, China, under Grant 618JJ3072. This paper was recommended by Associate Editor F. Firouzi. *(Corresponding author: Jiliang Zhang.)*

P. Qiu, Y. Lyu, and D. Wang are with the Research Institute of Information Technology and TNList, Tsinghua University, Beijing 100084, China (e-mail: qpf15@mails.tsinghua.edu.cn; luyq@tsinghua.edu.cn; wds@tsinghua.edu.cn).

J. Zhang is with the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China (e-mail: zhangjiliang@hnu.edu.cn).

G. Qu is with the Department of Electrical and Computer Engineering, University of Maryland at College Park, College Park, MD 20742 USA (e-mail: gangqu@umd.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2017.2748000

2) *Detection-based defenses*, for instance, ROPdefender [5], signature-based CRAs protection (SCRAP), and detecting ROP malicious code (DROP) [22]. But some exceptions are not be considered in these methods [5].
3) *Compiler-based defenses*, for example, G-Free [13]. However, these methods need to modify the compiler and may result in large false positives.
4) *Instrumentation-based defenses*, such as branch regulation (BR), transparent run-time shadow stack (TRSS) [23] and control-flow integrity (CFI) [6], [24]–[27]. But these methods may introduce large performance overheads or require to modify the ISAs.

We will describe these defenses minutely in Section III.

Among those defenses, CFI approach is the most general solution. When achieving CFI, first, the control flow graph (CFG) [6] of a program will be constructed, and then the CFG checking mechanisms will check whether the program is executed as the CFG at run-time [24]. There are two kinds of implementations of CFI, i.e., the software-based CFIs [24] and the hardware-based CFIs [25]–[27]. Software-based CFI [24] methods insert IDs and ID-checking instructions before control flow transitions according to the preconstructed CFG when the programs are compiling. When *call*, *ret*, and *indirect jmp* instructions are executing, the ID-checking instructions will be executed to judge whether the control flow transitions are legal. The illegal jumps of control flow will be checked and rejected at run-time. However, the implementations of software-based CFIs often result in high performance overheads due to the IDs creating, querying, comparing, and storing [28]. Therefore, a range of hardware-based CFIs are proposed to reduce the performance overheads of CFI, such as hardware-assisted fine-grained CFI [26], hardware-assisted flow integrity extension (HAFIX) [27], and physical unclonable functions (PUFs)-based CFI [28]. However, these methods still suffer from high cost [need to extend the instruction set architectures (ISAs) of CPU] or security issues (key leakage issues).

In this paper, we propose a new hardware control flow integrity based on a lightweight encryption architecture with advanced encryption standard (LEA-AES) to encrypt and decrypt return addresses and instructions at indirect jump destinations to defend against CRAs, which shows a good balance among performance, security and cost in comparison with previous methods. This method is similar to the PUF-based CFI [28], but it gains remarkable advantages in security. Cryptographic CFI (CCFI) [29] also protects control flow elements with AES. However, CCFI is built on source codes and has limitations in performance overhead and defending against JOP attacks. Although the AES is not a linear encryption method, many studies [30]–[33] have greatly reduced the latency of AES by using parallel techniques and adding hardware support. Besides, Intel has embedded the AES instructions [33] (*AESENC*, *AESENCLAST*, *AESDEC*, and *AESDECLAST*) into the x86 instruction set, which improves the efficiency of AES further. There are several modes of AES, including electronic codebook, cipher block chaining, cipher feedback, output feedback, and counter (CTR). Different ISAs and different architectures may have different return address lengths and different instruction lengths. The counter mode (CTR) of AES encrypts a self-increase counter with encryption key and exclusive OR (XOR) the result with plaintext to get the ciphertext. It is very suitable for protecting alterable length data since its encryption block is not necessary to be 128-bit long. Besides, it can be implemented to encrypt or decrypt data in parallel. The experiments on the CTR mode of AES using the library provided by Gueron [33] show that the latency of encrypting or decrypting a byte is only about three cycles on average with the plaintext block is 64-bit long. So, using AES to encrypt and decrypt data will take very little influences on programs. Besides, it is nearly impossible to get the key of AES even the length of the key used is 128, so that LEA-AES has no key leakage issues. In our design, LEA-AES will encrypt return addresses of function-call and certain length of instructions at indirect jump destinations before they are stored in the relevant memory units. When executing the correlative instructions, they will be decrypted with the same key and the same length. Those operations can be performed with current ISAs that support AES, which enhance its usability a lot. The key used in AES and the length of instructions to be encrypted or decrypted are produced by a random number generator before the programs are loaded into the memory. Our contributions are as follows.

1) We employ AES to encrypt or decrypt the return addresses of function-calls and certain length of instructions at indirect jump destinations to prevent ROP and JOP attacks. This paper has no need for adding new CPU instructions.
2) Compared with [28], this paper does not have the key leakage issues since that it is very hard to break AES. Besides, this paper employs random number generator to generate the encryption–decryption key instead of PUF [52], which makes it easier to be applied as most computers may not have PUF.
3) The LEA-AES has very low performance overheads. Performance evaluations indicated that the average run-time and loading time overheads of LEA-AES are both less than 4%; the average memory overhead is 0.62%, which greatly outperform the software-based CFIs and is in the same level as other hardware-based CFIs.

The remainder of this paper is organized as follows. Section II introduces the basic mechanism of CRAs. Section III indicates some existing defenses against CRAs, especially describes CFI. The proposed method and its working mechanisms are elaborated in Section IV. The procedures of protecting *call* and *ret* instructions to defeat ROP are given in Section V. And then, Section VI shows the processes of protecting *indirect jmp* instructions to defeat JOP. Performance and potential security threats are analyzed in Section VII. The evaluation results and analysis are reported in Section VIII. Finally, we conclude in Section IX.

II. Code Reuse Attacks

CRAs, including ROP and JOP approaches, make use of existing codes rather than injected codes to attack a

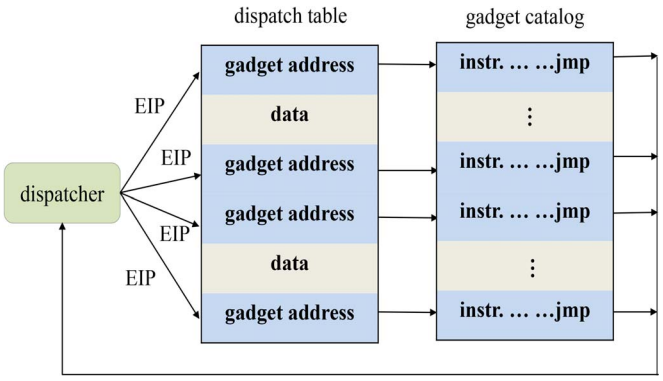


Fig. 1. Basic mechanism of JOP [2].

program [25], [28]. Therefore, CRAs may bypass the security measures that prevent code injection attacks, e.g., the DEP [25].

A. Return-Oriented Programming

After a *call* instruction is executed, the return address will be pushed into the stack, which will be popped from the stack again while the corresponding *ret* instruction is executed. However, adversaries may exploit some software vulnerabilities, for example, the stack or heap overflow vulnerabilities and the string formatting bugs, to modify the return addresses in the stack. If the return addresses are modified successfully, CPU will then execute the instructions at the modified return addresses, which will change the original control flow and may help fulfill the attack. Following the idea, ROP attacks can make use of the code gadgets that end with the *ret* instructions to compose a new execution flow to form or partially form an attack. First, the attackers exploit the useful gadgets in target programs, and then organize the gadget chain carefully to form a malicious program behavior. The transitions between code gadgets of a chain are usually activated by stack overwriting with the target gadgets. As a normal result of such an attack, the attackers may control the system.

B. Jump-Oriented Programming

The *direct jmp* instructions have fixed jump destinations in codes and cannot be employed to form a JOP attack. However, the destinations of *indirect jmp* instructions are determined by registers, which may be modified through some techniques (e.g., memory leakages and stack or heap buffer overflows) and software bugs (e.g., string formatting bugs) to form JOP attacks. JOP attacks make use of the code gadgets that end with the *indirect jmp* instructions [2] to form gadget chains. In practice, near indirect jumps are enough for JOP attacks [34]. Using far indirect jumps to construct the JOP gadgets must select the correct segments [34]. Fig. 1 shows the basic idea of JOP [2]. There are two types of gadgets frequently used in JOP attacks, i.e., the control gadgets and the function gadgets. Function gadgets are employed to achieve malicious actions (e.g., calling a specific system instruction). Control gadgets are employed to modify the registers to transit the control flow to function gadgets. Attacks can be fulfilled successfully by executing dedicated sequences of function gadgets.

III. Existing Defenses Techniques

A range of defense methods against CRAs were proposed, which can be classified into four types.

A. Randomization-Based Defenses

ASLR [1], [3], [5], [35], [36] is a randomization-based solution to defend against CRAs. It is supposed to be hard for attackers to find correct entry addresses of gadgets because the code and data regions will be randomly located in memory by ASLR. The randomized addresses of code and data regions are managed by an middle layer between the operating systems and hardware. Nevertheless, the code and data regions are actually not fully randomized when ASLR is employed in modern operating systems [5], [28]; it is still probable for attackers to acquire enough gadgets with the knowledge of partial randomized codes. For example, Snow *et al.* [3] proposed the *just-in-time code reuse* that achieved CRAs on programs with ASLR techniques applied.

B. Detection-Based Defenses

ROPdefender [5], [37] defends against ROPs with detecting techniques. ROPdefender dynamically detects ROP behaviors based on *ret* instruction evaluation [5]. It duplicates return addresses onto a shadow stack and employ the idea of just-in-time-based binary instrumentation to evaluate each *ret* instruction during program's execution [5]. However, it cannot defend against JOP attacks in addition to performance and memory overheads.

Kayaalp *et al.* [38] developed the *SCRAP*, a signature-based detection which observes the behavior of programs and detects the gadget execution patterns to prevent CRAs. First, SCRAP builds a single state machine to record behaviors of programs [38]. Then, SCRAP integrates sate counters of state machine into secure call stack to track the information about the state of the attack [38]. At run-time, SCRAP detects the execution patterns of gadgets, thus judging whether programs are attacked by CRAs. SCRAP can be implemented entirely in hardware at the commit stage of the pipeline using simple logic circuits [38]. Besides, SCRAP is designed to be configurable using a privileged system call that sets the detection machine state, so that it can be applied on different architectures easily [38]. However, SCRAP does not consider *just-in-time code reuse* [3] which constructs gadgets and attacks programs at run-time.

DROP [22], [39]–[41] focuses on tracing malicious behaviors of ROP through tainted data [5]. First, DROP marks untrusted data as tainted. The tainted data may be user input or any input from an untrusted device [5]. After marking data as tainted, taint analysis tracks the propagation of tainted data. If tainted data is misused, the system will alert or kill the program. Misuse of the tainted data is, for instance, using the tainted data as the target of *jmp*/*call* or *ret* instructions. Similar with ROPdefender, DROP cannot protect programs from JOP attacks and introduces high performance overhead

(30× to 50× for TaintCheck [41] and DYTAN [22]) when it is applied [5].

C. Compiler-Based Defenses

G-Free [13] is a compiler-based approach by using gadgetless binaries [5]. G-Free eliminates all unaligned free-branch instructions inside an executable binary, and protect the aligned free-branch instructions from being abused by an attacker [13]. First, G-Free modifies the compiler to guarantee that the resulted binary does not contain unintended instruction sequences. Next, the G-Free will protect the intended *ret* instructions at runtime by inserting a short header at the entry of a function and a relevant footer before its *ret* instruction [13]. Additionally, G-Free has been applied on GNU libc [5], [13]. However, each linked library and the original program codes have to be compiled with G-Free if we want to provide a full protection against ROP attacks, which might result in large false positives if a library is not compatible to G-Free [5]. In addition, G-Free cannot protect programs from JOP attacks either.

D. Instrumentation-Based Defenses

BR [25] enforces simple control flow rules present in function-based programming languages [25]. Those rules disallow arbitrary control flow transfers from one function into the middle of another function, thus dynamically reduces the possibility of the attackers to find exploitable gadgets needed for CRAs [25]. BR also adds a shadow stack to record legal return addresses and check them before *ret* instructions are executed [25], [28]. Besides, BR can protect programs from attacks that exploit unintended instructions by providing simple hardware support [25]. However, BR by design does not support stack unwinding and tail jumps [25], [28].

TRSS [4], [23], [42] is built on top of DynamoRIO [43], a dynamic binary rewriting framework. In TRSS, when *call* instructions are executing, the return addresses will be pushed into a shadow stack and return addresses check procedures will be enforced when *ret* instructions are executing [23]. TRSS is also able to detect unintended sequences issued in an ROP attack due to just-in-time-based instrumentation [5]. However, the mechanisms of DynamoRIO framework [43] do not allow to instrument a program from its very first instruction and it depends on the variable of LD_PRELOAD to map the DynamoRIO code into the addresses space of the applications [5]. Moreover, complex exceptions still need to be considered in TRSS.

In addition, CFI [6], [24]–[27] is another representative instrumentation-based defense technique that guarantees *call*, *ret* and *indirect jmp* instructions only jumping to valid destinations with respect to CFG. We describe CFI in detail in the following section.

E. Control-Flow Integrity

As an instrumentation-based technique, CFI is a general solution for CRAs. The main idea of CFI is to check whether the program's execution violates the CFG that is constructed during compiling. There are two fashions of CFI, i.e., the software-based CFI and the hardware-based CFI.

1) Software-Based CFI: Software-based CFI [24], [44] ensures that *call*, *ret*, and *indirect jmp* instructions can only direct to legal destinations specified in CFG by checking preinserted IDs. The ID-checking instructions are inserted before each legal *call*, *ret*, and *indirect jmp* transitions according to CFG. When CPU executes those instructions, the CFI checking mechanism will check whether the programs are executed as the preconstructed CFG. Any control flow transitions that do not pass the ID checking mechanism will be rejected.

Theoretically, we can insert different IDs to different control flow transitions to provide strong protections for programs. However, this will introduce high performance overheads and require more resources for IDs and checking IDs. The performance overhead of software-based CFI is about 21% [24]. To reduce the cost of traditional software-based CFIs, a bunch of researches propose to use less IDs or to comply with loosened CFGs. Compact CFI and randomization (CCFIR) [45] proposes to redirect the *indirect jmp* instructions to a new springboard that is constructed during compilation. By enforcing the *indirect jmp* instructions only to jump to springboard, the CCFIR can provide a protection for reusing *indirect jmp* instructions. There are three types of IDs used in CCFIR for restricting control flows to reduce the performance and memory overheads. Two IDs are used to protect sensitive or nonsensitive functions and one ID is used to protect *call* or *indirect jmp* instructions. Therefore, CCFIR still have security drawbacks that attackers may use the control flow transitions with same IDs to construct a successful CRA. In addition, there is a springboard used in CCFIR to ensure CFI, which also takes more memory resources.

In summary, current software-based CFIs need to insert IDs and IDs checking instructions or create accompanying data structures, which introduces remarkable performance and memory overheads [26]. Consequently, some recent researches have turned to adding hardware support to defend against CRAs.

2) Hardware-Based CFI: Several hardware-based CFIs focus on reducing performance overheads of CFI, such as hardware-assisted fine-grained CFI [26], [46], HAFIX [27], and PUFs-based linear encryption [28].

Davi *et al.* [26] proposed a *hardware-assisted fine-grained* CFI approach which introduces new instructions to ISAs to protect *call*, *ret*, and *indirect jmp* instructions from being reused by attackers. It guarantees *indirect call* instructions and *ret* instructions comply with the new CPU instructions through assigning different labels to different functions. The main drawback of this method is that its cost of introducing new ISA instructions, which results in corresponding changes of the compiler and related system software. Based on *hardware-assisted fine-grained* CFI, Davi *et al.* [27] proposed an HAFIX to particularly enhance ROP defense. HAFIX gives real hardware implementations of backward-edge (returns) CFI targeting bare metal code, and needs to add new assembler instructions to ISA.

CCFI [29], which is based on cryptographic message authentication codes (MACs) to protect control flow elements, such as return addresses and frame pointers, function pointers, vtable pointers, and exception handlers. The MACs are

calculated by employing a single block of AES to those elements and stored beside them. They are used to verify the control flows to check whether the programs are attacked. CCFI enables a much fine-grained classification of sensitive pointers through dynamic checks and it can prevent CRAs. This paper has the same basic mechanism with CCFI except that it protects return addresses and indirect jumps rather than code pointers. However, CCFI and this paper protect programs on different layers. CCFI is built on source codes layer and our method is designed on architecture layer. Besides, it may introduce more memory overhead to store MACs than this paper when protecting return address, theoretically, because that we do not insert additional information to function calls. Additionally, it provides no protection for *indirect jmp* instructions since they do not appear in source codes. The *indirect jmp* instructions are used in assembly codes. Therefore, CCFI may not prevent JOP attacks.

PUFs-*based linear encryption* [28] protects programs from ROP and JOP by linearly encoding and decoding the return addresses and the instructions at indirect jump destinations. It encodes some bytes of the first instruction at the target address of a *indirect jmp* instruction by XOR and decodes them after the *indirect jmp* instruction is executed [28]. Meanwhile, it similarly encodes the return address corresponding to a *call* instruction and decodes it when the *ret* instruction is executing [28]. However, the PUF-generated key for XOR encoding/decoding is vulnerable to memory leakage and debugging attacks (using both encoded and original plain text to deduce the key) [28], in which attackers may get the keys to construct encoded CRA gadget chains successfully.

In this paper, we propose an LEA using CPU-built-in AES to balance the security, performance and cost challenges previous works were faced with.

IV. Control Flow Integrity Based on Lightweight Encryption Architecture

The security drawback of the PUF-based linear encryption method [28] severely degrades its usability in practice in spite of its advantages in performance and limited impact to the CPU architecture over the traditional methods. In order to resolve the vital security issue and to reduce the changes to current CPU architecture further, we propose the LEA using CPU-built-in AES to defend against CRAs. AES is one of the proved state-of-the-art secure primitives and has been adopted by modern CPU architectures as a default feature, e.g., the Intel i7 CPUs. In addition, the CTR mode of AES can encrypt alterable length of blocks and it can be implemented in parallel. Therefore, it is very secure and low cost to use the CTR mode of AES to encrypt return addresses and instructions at indirect jump destinations in comparison with the PUF-based linear encryption method [28] due to its security against key leakage and minimal revision to current CPU architectures.

A. Assumptions

In this paper, we first make several assumptions as follows.

1) We assume that sufficient basic security measures have been applied on the target systems, such as the DEP, under which the attackers cannot modify the executable binaries at run-time. Actually, DEP has been applied on most well-known operating systems (e.g., Windows and Linux).
2) We assume that the CFGs of target programs are available. Actually, researchers have proposed a range of methods to construct CFGs based on program binaries and the coverages of CFGs is also increasing.
3) We assume that the encryption–decryption keys stored in private registers cannot be accessed and modified by any software. Actually, this is also easy to realize by restricting the data flow of those registers off the public data paths.

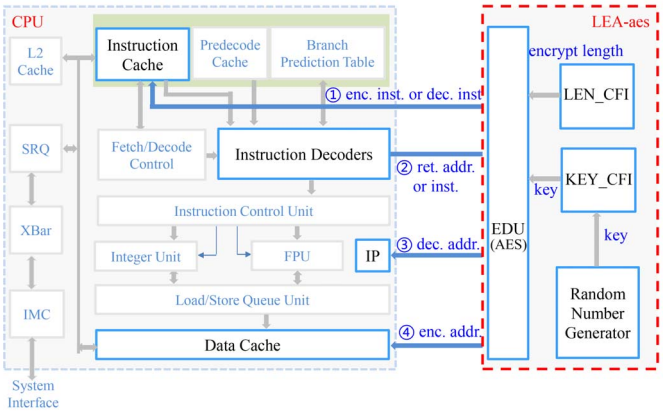


Fig. 2. CFI based on LEA and data flows. The data flows are: ① the encrypted or decrypted instructions between instruction cache and EDU, ② the return addresses or instructions to be encrypted from instruction decoders to EDU, ③ the decrypted addresses from EDU to IP, and ④ the encrypted addresses from EDU to data cache (stack).

B. Architecture Overview

Fig. 2 shows the new CFI based on an LEA-AES and its connections with CPU. We add a LEA-AES module to the current CPU architecture. There are four units in the LEA-AES, namely a encryption–decryption unit (EDU), a register KEY_CFI, a register LEN_CFI, and a random number generator unit (RGU). EDU is responsible for encrypting and decrypting return addresses and instructions of indirect jump destinations with the CPU-built-in AES. The register KEY_CFI stores the key for EDU to encrypt or decrypt data, and the register LEN_CFI stores the length of instructions to be encrypted or decrypted. Both KEY_CFI and LEN_CFI value are randomly generated by the RGU before the program is loaded into the memory. When the program is loading, EDU encrypts LEN_CFI length of instructions at the destination of each *indirect jmp* instruction with the key KEY_CFI. When an *indirect jmp* instruction is executing, CPU sends the LEN_CFI length of instructions at its destination to EDU to decrypt, after which the decrypted instructions (stored in register *xmm*) corresponding to the jump destination will be connected with the relevant memory units to be decoded and executed correctly. Similarly, EDU also encrypts the return addresses corresponding to each *call* instruction before they are stored into the stack. When CPU executes *ret* instructions, CPU sends the

return addresses popped from the stack to EDU to decrypt. The decrypted results are loaded into the register instruction point (IP) to continue.

C. Random Number Generator

Random number generators [47] are widely used for various security related applications, for instance, randomizing the addresses of data or instructions and generating keys for encryption and decryption. Although RGU may still not generate theoretically randomized data with many perfect improvements [47]–[49], its security is also accepted since the random numbers are only regarded as keys and data lengths for AES that is securely guaranteed. In case of more perfect solution, a PUF unit can also be considered to be the random number generator with certain cost invested [53].

D. Control Flow Graph

The CFG is a directed graph which includes the control flow transitions of the programs. Generally, the *call*, *ret*, and *indirect jmp* instructions are the most responsible instructions that take the control flow changes. Therefore, the CFG normally consists of every function-call and every indirect jump. The CFG can be constructed by source-codes analysis, binary analysis or execution profiling [24]. In our design, we only utilize the CFGs when the programs are loading into the memory and the EDU will encrypt certain length of instructions at indirect jump destinations according to the CFG. After that, either the decryption of jump-related instructions or the encryption and decryption of the return addresses are only performed at run-time without the CFG involved.

V. Defending ROP Attacks

The return addresses are the main elements that are modified in ROP attacks. Therefore, protecting return addresses is effective to thwart ROP attacks. In this paper, we employ the LEA to encrypt the return addresses to protect against ROP attacks.

Fig. 3 shows the processes of protecting return addresses against ROP attacks. Normally, when a *call* instruction executes, CPU will first push the return address into stack and then jump to the target function to execute instructions. When a *ret* instruction executes, CPU will first pop the return address out of the stack and then jump to it to execute instructions. In LEA-AES, CPU first sends the return address to the EDU to encrypt when *call* executes, and then pushes the encrypted return address into the stack. When *ret* executes, CPU first pops the encrypted return address out of the stack for EDU to decrypt, and then sends the decrypted address that is securely guaranteed to IP register for the program to jump to.

If attackers modify the return addresses in the stack, the default decryption operations of LEA-AES on those addresses will be forced to execute when functions are returning. The unexpected decryption cannot support a successful jump to the expected address of attackers, which may cause a system error or exception. Fig. 4 shows an example of using LEA-AES to protect against ROP.

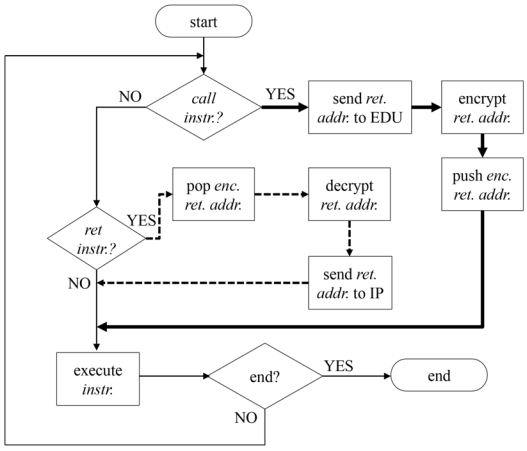


Fig. 3. Protecting return addresses (run-time). Heavy lines and dotted lines represent the procedures of protecting *call* instructions and *ret* instructions, respectively.

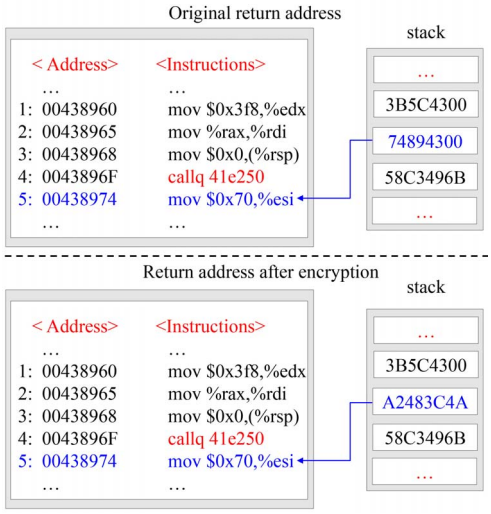


Fig. 4. Example of protecting return addresses using LEA-AES.

Normally, when the instruction of *callq* 41e250 is executing, CPU will push the return address of 00 43 89 74h into stack. When the corresponding *ret* instruction is executing, the program will return to 00 43 89 74h that is popped from stack to execute instructions. In our design, CPU will send the return address to EDU for encrypting when *callq* 41e250 is executing and EDU will push the encrypted result of 4A 3C 48 A2h into stack. Meanwhile, when the *ret* instruction is executing, CPU will pass the return address popped from stack to EDU for decryption, after which, EDU will pass the decrypted address into the register IP to continue correctly. If the return address popped from stack is not 4A 3C 48 A2h, the decrypted result will not be 00 43 89 74h and a system error may occur.

VI. Defending JOP Attacks

The *indirect jmp* instructions are the main elements that are used by attackers to achieve JOP attacks. In this paper, we protect the *indirect jmp* instructions by encrypting and decrypting certain bytes of codes with AES at the target addresses.

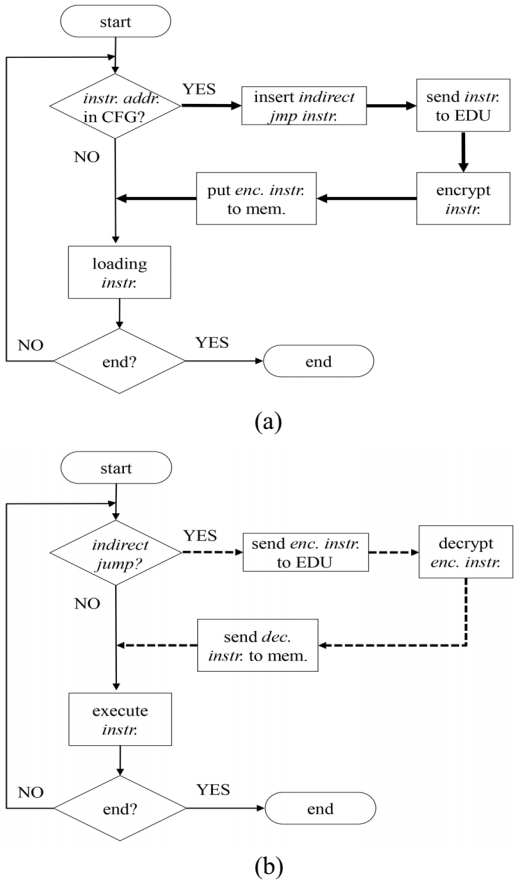


Fig. 5. JOP defense using LEA-AES. (a) Preprocess for indirect jump (loading time). (b) Protecting indirect jump while executing (run-time).

Fig. 5 shows the procedures of protecting *indirect jmp* instructions to defend against JOP attacks. Under normal conditions, when a program is loading into memory, the data and codes of the program are loaded into corresponding memory addresses. Additionally, after a *indirect jmp* instruction is executed, CPU will send the first instruction at target address to instruction register and execute instructions at target address. In LEA-AES, when programs are loading into the memory [Fig. 5(a)], EDU will encrypt some bytes of instructions at indirect jump destinations with the key stored in KEY_CFI and the encryption–decryption length stored in LEN_CFI. When CPU executes a *indirect jmp* instruction [Fig. 5(b)], some bytes of instructions at its target address will be decrypted by EDU with the same key and the same length. The decrypted results of AES instructions provided by Intel are stored in register *xmm*. However, the lengths of instructions in some ISAs are alterable, such as Intel x86 ISA, the decrypted result may not a instruction. In our design, after decrypting the instruction, CPU decodes the instructions (determined by decrypted results and instructions behind the encrypted instructions in memory) and continues the execution correctly. The instructions stored in memory are not altered, including the encrypted instructions. Hence, when the same *indirect jmp* instruction is executed, the first LEN_CFI bits of instructions at its destination are remain encrypted.

If the target address of a *indirect jmp* instruction is changed to elsewhere by a tentative JOP attack, the instruction segments

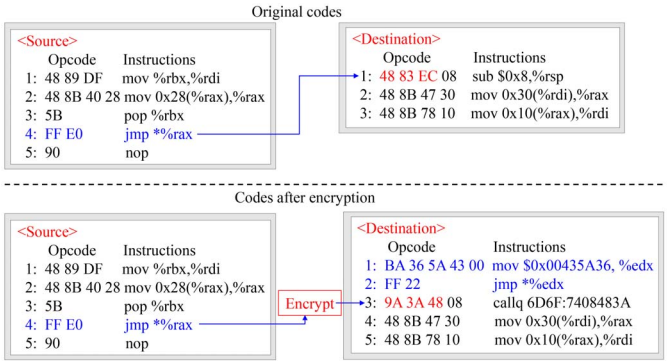


Fig. 6. Example of protecting *indirect jmp* instructions.

at the changed destinations will be forced to be decrypted after the *indirect jmp* instructions are executed. An exceptional error can occur when unencrypted codes are decrypted and run as normal. A potential attack, however, may be still valid when the target address is changed right to another candidate jump destination since all candidate destinations are encrypted at loading-time. In this case, there is a theoretical possibility of being attacked by reusing candidate jump destinations, which is also discussed in detail in the section of security analysis.

As a special case, some instructions at indirect jump destinations may also need to execute sequentially without a preceding jump, e.g., the do-while statement in which the instructions inside the loop do not have a jump to start in the first pass (the second and later passes start with jump instructions out of the while-statement). In such cases, errors will occur because the EDU does not decrypt those instructions without a preceding jump instruction triggering it. In order to make those instructions decrypted normally, we introduce a redundant *indirect jmp* instruction before each candidate indirect jump destination, which just redirects to its succeeding instruction and takes the responsibility of triggering the decryption at the destination. Fig. 6 shows an example of protecting *indirect jmp* instructions using LEA-AES.

In this example, the value of LEN_CFI is 3 and the target address of *jmp *%rax* is 00 42 5A C8h. EDU will encrypt the first 3 bytes of the instruction 48 83 EC 08h at the address of 00 42 5A C8h when the program is loading into memory. The encrypted result is 9A 3A 48 08h. After the instruction of *jmp *%rax* is executed, the EDU will decrypt the first 3 bytes of the instruction 9A 3A 48 08h, which is 48 83 EC 08h. Then CPU will execute programs with correct instructions. In this example, we insert two instructions of *mov* $0x00435A36, *%edx* and *jmp *%edx* before the indirect jump destination to guarantee the programs to be executed correctly. Normally, the addresses of indirect jump target instructions after applying the LEA-AES are not the same with them in original programs due to the inserted instructions. Therefore, the value moved to the %edx is 00 43 5A 36h instead of 00 42 5A C8h.

VII. Performance and Security Analysis

A. Performance Analysis

We employ the CTR mode of AES as the encryption method in EDU. Let p be the target text to encrypt, e.g., the return

addresses and instructions at indirect jump destinations, and let c be the corresponding ciphertexts. Let e be the encryption method (i.e., the AES in this paper), and let d be the corresponding decryption method. Let k be the encryption–decryption key. The encryption of p and decryption of c can be formulated as

$$c = e_k(p)$$
$$p = d_k(e_k(p)). \tag{1}$$

Suppose that there are n *call* instructions and m *indirect jmp* instructions are executed in programs, we assume that: 1) the average execution time of a instruction is t and 2) the average execution time of encryption or decryption in EDU is t^*. So that we can get the loading and run-time overheads of LEA-AES as

$$\text{Overhead} = 2nt^* + 2m\big(t^* + t\big). \tag{2}$$

In (2), $2nt^*$ represents the run-time of encryption and decryption operations for *call* and *ret* instructions since that every function-call has a relevant function-return, $2mt^*$ is the run-time of the encryption and decryption operations for the *indirect jmp* instructions, and $2mt$ is the run-time of the two inserted instructions before each indirect jump destination.

Additionally, the CTR mode of AES in library provided by Gueron [33] is implemented as assembly function with parallelizing four blocks, its efficiency is very high. We have tested the average cycles for a byte in CTR mode with 64-bit long blocks and 128-bit long key. The results indicated that it will take about three cycles to encrypt a byte. Hence, the run-time for encrypting or decrypting a return address or instruction in EDU is acceptable. We assume that the $t^* = kt$, (2) will be

$$\text{Overhead} = (2nk + 2m(k+1))t. \tag{3}$$

The k in (3) is a constant. Hence, the loading time and run-time overheads of LEA-AES is $O(n + 2m)$. The run-time overhead can be further reduced by store the encryption result and decryption result into cache. Besides, we insert two instructions (*mov* and *indirect jmp*) before every indirect jump destination, therefore, the memory overhead of LEA-AES is $O(m)$. Hence, the LEA-AES will introduce less run-time and memory overheads than traditional CFI methods.

B. Security Analysis

1) Key Leakage Issue: We employ the AES in LEA-AES to encrypt and decrypt return addresses and instructions at indirect jump destinations. Compared to the PUF-based method [28] that employed the XOR as the encryption method with the keys generated by PUF, using AES is much securer but takes more cycles than XOR encryption. The XOR encryption method is a linear encryption method, and it can encrypt or decrypt data in one cycle [28]. So, using XOR to encrypt and decrypt data will introduce very little performance overheads [28]. However, the XOR encryption method is not secure any more when the plaintext and the ciphertext are obtained by attackers in the same time. The attackers may compare the unencrypted data of programs with the encrypted data to get the encryption–decryption keys by memory leakage, debugging or other techniques [28]. It is very easy to calculate the key in such circumstances, which causes the key leakage issues. If attackers get the keys, they can use those keys to construct the encrypted return addresses to organize ROP attacks. Even the keys can be different for different programs and for different executions of the same program, attackers can still use the same method to get the key for every execution and they can use those keys to organize CRA attacks.

Since occasionally cracking an AES key must take a lot of time, we believe it is practically securer to use RGU to generate keys randomly for each program at each run with mentioned limited impact to current architectures. However, if we use fixed AES keys in a similar way of trusted computing architectures, it will be not secure enough any more because the attackers may get the key through many techniques like side channel attacks [50].

In our experiments which will be showed in Section VIII, the self-increase counter value of CTR mode is start at 0 for every encryption and decryption. In order to improve the security, for one return address encryption or decryption, the counter value may be set as the stack address where it will be stored, for instructions encryption or decryption, the counter may be set as the addresses of instructions.

2) False Positive/Negative Rate: The false postive rate and false negative rate are two important indicators to estimate the security and reliability of a security strategy [28]. The false positives mean that some normal executions of programs are judged as attacks and the false negatives mean that some attacks are not prevented. In this paper, the LEA-AES relies on the CFG to preprocess the jump instructions to defend against JOP. Researchers have proposed several methods to increase the coverage of the CFG. Unfortunately, no practical tools that can analyze a program to form a complete CFG [25]. Similarly with all other CFI methods, the CFG of a program used in LEA-AES is not complete. The result is that some *indirect jump* destinations do not appear in CFG and the instructions at those destinations will not be encrypted when loading the programs. When a *indirect jmp* instruction jump to those destinations, LEA-AES tries to decrypt unencrypted instructions before jumping, the decrypted instructions will be abnormal instructions, a false positive will occur. This should be avoided as far as possible.

The false positives are introduced by the incomplete CFG. Therefore, improving the coverage of indirect jumps in CFG is responsible in reducing the false positive rate. The following solutions may be useful to construct more complete CFG.

1) *Vulcan* [51] is publicly the best tool available to construct the CFG, which provides both static and dynamic code modification as well as a system-level analysis for heterogeneous binaries across instruction sets [51]. The Vulcan can be used to improve the completeness of CFG.
2) The CFG may be updated dynamically as long as a false positive happens. When programs are executing, the indirect jumps formerly not included in CFG can be added into CFG.
3) A software testing procedure may also be added to construct more complete CFG for a software, which is similar with traditional software testing.

TABLE I
CHARACTERISTICS OF BENCHMARKS

Category	Basic	Game	Language	Plotting	Picture	IO	Text	Chat	Sever	Mail	Player	Database
Program	Vul	Mines	Lua	Orage	Eog	Hardinfo	Gedit	Xchat	Nginx	Mutt	SMplayer	Sqlite3
#instructions	373	32148	33854	90442	126112	130988	133912	171733	198677	234651	400768	406057
#*call*	27	1218	2633	1331	3391	3945	4607	5402	4677	7115	19728	9398
Intend tar. of jmp	10	305	123	600	1212	896	1271	831	181	315	1630	1832
#*ret*	14	613	717	382	1862	2589	2563	3387	1333	1284	4899	2074

In addition to false positives, there are still possible false negatives existing in LEA-AES. For example, a transition between the legal jump destinations may not be a legal jump of the control flow in CFG, but get no exceptions from LEA-AES because LEA-AES uses the same key and the same length to encrypt and decrypt instructions. As another example, attackers may record the encrypted return addresses when they are pushed into stack and overflow the stack with those encrypted return addresses to attack programs with ROP. Consequently, such violations against the control flow cannot be found by LEA-AES, which is called false negatives, i.e., the abnormal control flow violations are not detected. Nevertheless, the false negative rate is actually very low due to the low proportion of the candidate indirect jump destinations and available return addresses for attacks over the entire code. Evaluation results will also be shown in the following section.

The LEA-AES uses the same key and the same length to encrypt and decrypt data, which is responsible for false negatives. Therefore, distinguishing different return addresses and different indirect jumps are useful to reduce false negative rate. We are working on this topic. The following solutions may be useful to distinguish different control flow transitions.

1) For one return address encryption or decryption, the initial self-increase counter value used in CTR mode may be set as the stack address which it is stored. The result is that the same function call may get different encrypted return addresses for different executions since its stack addresses may be not the same.
2) For one instruction encryption or decryption, the initial self-increase counter may be set as a specific value, such as a hash value of addresses of all indirect jump instructions that can jump to it. The hash maps value may be calculated before the programs are loaded into the memory according to the CFG and may be implemented with hardware to increase the efficiency. As a result, for an indirect jump destination, only the indirect jump instructions that can jump to it have the correct decryption counter value. However, more hardware and memories are required to manage the hash map.

VIII. EVALUATION RESULTS

The performance overhead of the LEA-AES were evaluated on the benchmark of 12 different types of open source programs by simulation, in which the CTR mode of full ten-round 128-bit AES instructions (provided by Gueron [33]) were inserted into the programs before every function call, function return and indirect jump instruction. For every encryption or decryption, the self-increase counter value starts at 0. Some indirect jump instructions were also inserted before indirect jump targets according to the CFG to guarantee the target instructions can be correctly executed when they are executed in order. Table I shows the detailed characteristics of the benchmark. The programs focus on different application types, such as game, plotting, picture tool, etc. The tests were performed on a desktop computer with a 64-bit Intel x86 core i5 CPU (with the 128-bit AES built-in), 4-GB memory and an Ubuntu 12.04 OS.

A. Constructing CFG

In this paper, we employed a binary-analyzing CFG construction tool [24] to extract the *indirect jmp* instructions and their intended destinations, in which the executable binaries were disassembled to acquire the addresses of indirect jump instructions and basic blocks in the same segment. The CFG is a directed graph. The nodes represent the addresses of the *indirect jmp* instructions and their destinations. Each indirect jump transition becomes an edge in the CFG.

B. Inserting Instructions

In this paper, we inserted two instructions before each candidate destination of *indirect jmp* instructions to avoid a sequential execution disorder as mentioned in JOP defense. One instruction is to move the candidate destination address to a temporary register, and the other is a indirect jump instruction whose target address is the address stored in the temporary register. The two instructions do not have meaningful functions except for guaranteeing the correct execution at those candidate destinations when the program just sequentially flow there without a jump instructions (e.g., the first loop of the do-while statement).

C. Performance of LEA-AES

We took the total run-time and memory usages as performance gauges. The run-time of a program is the time during its start execution and end execution. The loading time of a program is the time during its start-up and start execution. We inserted time functions before the first executable instruction and the last executable instruction to record the time of start execution and end execution. The start-up time can be acquired directly. We compared our method (LEA-AES) with the PUF-based architecture (XOR) [28]. Table II shows the time and memory overheads of the two methods. The LEN_CFI is set as 64 that is the word length of our experiment machine. We can see that the LEA-AES had 0.62% average memory overhead, about 3.19% average run-time overhead and 3.53% average loading time overhead. The memory overhead of LEA-AES is caused by the inserted instructions, and the loading time overhead is caused by the encryption

TABLE II
TIME AND MEMORY OVERHEADS OF LEA-AES [28]

Program	Origin			XOR				LEA-AES				Memory	
	Memory(B)	Load time(s)	Run time(s)	Load time(s)	Load over-head(%)	Run time(s)	Run over-head(%)	Load time(s)	Load over-head(%)	Run time(s)	Run over-head(%)	XOR/LEA-AES mem-ory(B)	Memory over-head(%)
Vul	10834	0.1995	628.8023	0.2017	1.1028	629.0072	0.0326	0.2075	4.0100	634.022	0.8301	10850	0.1477
Mines	720662	0.8329	702.4461	0.8384	0.6603	722.3897	2.8392	0.8639	3.7219	758.6366	7.9993	721177	0.0715
Lua	253729	0.4026	640.8576	0.4039	0.3229	645.4098	0.7103	0.4129	2.5584	658.8155	2.8022	256599	1.1311
Orage	1276085	1.6046	772.6099	1.6160	0.7105	781.2068	1.1127	1.6541	3.0849	798.4008	3.3382	1290487	1.1286
Eog	2337126	3.8212	718.4958	3.9015	2.1014	721.4262	0.4079	4.0382	5.6762	728.0666	1.3321	2359332	0.9501
Hardinfo	2555039	5.0187	696.3642	5.0526	0.6755	701.4154	0.7254	5.1588	2.7916	720.1073	3.4050	2568092	0.5109
Gedit	2751830	5.5406	713.3477	5.6128	1.3031	719.3737	0.8447	5.7217	3.2686	730.8688	2.4562	2763656	0.4298
Xchat	3656663	7.8458	702.2962	7.8735	0.3531	705.5597	0.4647	8.0507	2.6116	718.0052	2.2368	3687750	0.8501
Nginx	3659443	8.4638	645.0556	8.5144	0.5978	650.6002	0.8596	8.7956	3.9202	662.625	2.7237	3668668	0.2521
Mutt	2935781	6.1588	666.7161	6.1997	0.6625	671.4144	0.7047	6.4566	4.8354	689.9391	3.4832	2966065	1.0315
SMplayer	5951334	12.5028	716.3991	12.5636	0.4863	718.9045	0.3497	12.8616	2.8698	727.9363	1.6104	5967418	0.2703
Sqlite3	4343715	10.9271	642.9779	11.0146	0.8008	651.2733	1.2902	11.3250	3.6414	681.8645	6.0479	4379777	0.8302
Average	2537686.75	5.2759	8246.3685	5.3161	0.7620	8317.9809	0.8684	5.4622	3.5312	8509.2877	3.1883	30639871	0.6162

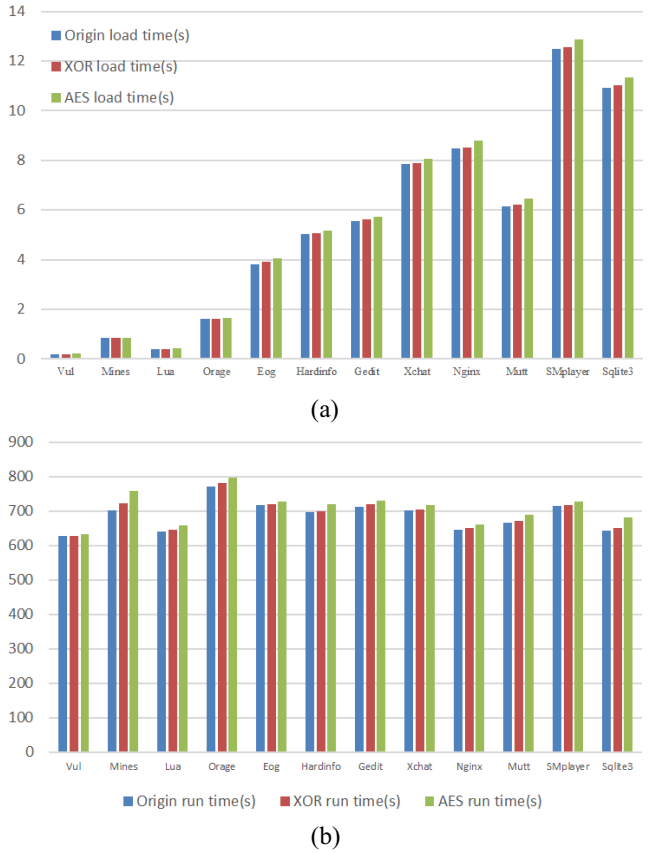


Fig. 7. (a) Loading time and (b) run-time of origin programs and modified programs.

of the instructions at indirect jump destinations. The PUF-based method (XOR) had 0.87% average run-time overhead, about 0.76% average loading time overhead [28] and the same memory overhead as LEA-AES (the same inserted instructions). From the results, we can see that LEA-AES obtained similar level of performance with the PUF-based method, which is in the best level compared to traditional CFI methods in addition to its good security and limited architecture impact.

Fig. 7 gives a schematic illustration about the loading time and run-time overheads of the LEA-AES and XOR method. The average performance overheads of our method are almost

TABLE III
GADGETS REDUCTION AND FALSE NEGATIVE [28]

Program	Origin gadgets	Result gadgets	% reduc-tion(%)	# dec. and enc.	%false negative
Vul	3	1	66.67	48	5.8824
Mines	208	8	96.15	2086	2.3408
Lua	760	128	83.16	3453	0.5759
Orage	563	57	89.88	2279	1.4700
Eog	906	62	93.16	6404	0.9435
Hardinfo	1159	85	92.67	7395	0.4711
Gedit	1036	165	84.07	8327	1.3506
Xchat	2176	287	86.81	9498	1.2682
Nginx	4252	329	92.26	6084	1.7283
Mutt	3839	551	85.65	8692	0.2525
SMplayer	5028	422	91.61	26088	0.6058
Sqlite3	5618	734	87.47	13169	1.0147
Total	25552	2829	88.93	93523	0.9133

negligible. The reason of the extremely low overheads is that the CPU-built-in AES has been implemented to encrypt and decrypt data within very little cycles and it is only used to encrypt very small proportion of the code in LEA-AES. Furthermore, LEA-AES does not need to insert IDs and check IDs, which also greatly reduces its run-time and memory overheads.

D. False Positive/Negative Rate

The false positive rate of a CRA defense technique is very hard to acquire because of the difficulty of constructing a complete CFG as the gold [25]. In fact, if we can find those absent control flow transitions to CFG, we can put them into the CFG and eliminate the false positives. The false negatives of LEA-AES are caused by the indirect jumps and available return addresses. Normally, the gadgets that can be used to achieve CRAs should not be very long, therefore, not all instructions at indirect jump destinations and return addresses can be used to construct gadgets. We evaluated the total gadgets available and the false negative rate of LEA-AES as shown in Table III. From the results, we can see that the gadgets which can be used for attackers to organize ROP attacks and JOP attacks were reduced about 88.93% on average by using LEA-AES for return addresses and *indirect jmp* instructions. This means that the probability to achieve CRAs can be reduced a lot with LEA-AES employed. The results also showed that the false negative rate is less than 1%.

IX. Conclusion

Traditional CFI methods suffer from performance, cost or security issues in protecting programs from CRAs. For example, the software-based CFIs have high performance overheads and most hardware-based CFIs need to revise current ISAs, which greatly reduces their usabilities. A most recent PUF-based architecture can have outstanding performance and limited architecture impact [28]. However, it still suffers from the security issue of key leakages. In this paper, we propose a new CFI based on an LEA-AES to resist CRAs with extremely low run-time overheads and architecture impacts. The LEA-AES can protect *call*, *ret*, and *indirect jmp* instructions by encrypting and decrypting return addresses and certain length of the instructions at indirect jump destinations without introducing new ISA instructions. For ROP, the encrypting and decrypting operations for return addresses are performed when the *call* or *ret* instructions are executing. For JOP, the encrypting operations for instructions are performed when the programs are loading into the memory, and the decrypting operations for instructions are performed when the *indirect jmp* instructions are executing. Performance evaluations showed that the proposed LEA-AES only introduced 3.19% run-time overhead and 0.62% memory overhead on average.

In future work, we may employ hash-mapping schemes in JOP protection to lower the false negative rate further. We can set multiple candidate keys for different *indirect jmp* instructions to choose through a hash-mapping scheme. In such cases, the instructions at different indirect jump destinations may be encrypted with different keys and the malicious control flow jump among those instructions may not appear any more. Meanwhile, we can also use advanced program analysis to increase the coverage of the CFG to reduce the false positive of LEA-AES. A pure-hardware LEA-AES evaluation may also be considered to validate the practical usability of this paper.

References

[1] S. Checkoway *et al.*, "Return-oriented programming without returns," in *Proc. 17th ACM Comput. Commun. Security (CCS)*, Chicago, IL, USA, 2010, pp. 559–572.

[2] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proc. 6th ACM Symp. Inf. Comput. Commun. Security (ASIACCS)*, Hong Kong, 2011, pp. 30–40.

[3] K. Z. Snow *et al.*, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proc. IEEE Symp. Security Privacy*, Berkeley, CA, USA, May 2013, pp. 574–588.

[4] A. Francillon, D. Perito, and C. Castelluccia, "Defending embedded systems against control flow attacks," in *Proc. 1st ACM Workshop Secure Execution Untrusted Code*, Chicago, IL, USA, 2009, pp. 19–26.

[5] L. Davi, A.-R. Sadeghi, and M. Winandy, "RopdeFender: A detection tool to defend against return-oriented programming attacks," in *Proc. 6th ACM Symp. Inf. Comput. Commun. Security (ASIACCS)*, Hong Kong, 2011, pp. 40–51.

[6] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proc. 12th ACM Comput. Commun. Security (CCS)*, Alexandria, VA, USA, 2005, pp. 340–353.

[7] A. Francillon and C. Castelluccia, "Code injection attacks on Harvard-architecture devices," in *Proc. 15th ACM Comput. Commun. Security (CCS)*, Alexandria, VA, USA, 2008, pp. 15–26.

[8] R. Riley, X. Jiang, and D. Xu, "An architectural approach to preventing code injection attacks," *IEEE Trans. Depend. Security Comput.*, vol. 7, no. 4, pp. 351–365, Oct./Dec. 2010.

[9] S. Andersen and V. Abella. (2004). *Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies*. [Online]. Available: https://technet.microsoft.com/en-us/library/bb457155.aspx

[10] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. 14th ACM Comput. Commun. Security (CCS)*, Alexandria, VA, USA, 2007, pp. 552–561.

[11] T. Kornau, "Return oriented programming for the arm architecture," Ph.D. dissertation, Horst Görtz Inst. IT Security, Ruhr-Universität Bochum, Bochum, Germany, 2010.

[12] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to RISC," in *Proc. 15th ACM Comput. Commun. Security (CCS)*, Alexandria, VA, USA, 2008, pp. 27–38.

[13] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-Free: Defeating return-oriented programming through gadget-less binaries," in *Proc. 26th Annu. Comput. Security Appl. Conf. (ACSAC)*, Austin, TX, USA, 2010, pp. 49–58.

[14] S. Checkoway *et al.*, "Can DREs provide long-lasting security? The case of return-oriented programming and the AVC advantage," in *Proc. Elect. Voting Technol. Workshop Trustworthy Elections (EVT/WOTE)*, Montreal, QC, Canada, 2009, p. 6.

[15] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, "Defeating return-oriented rootkits with 'return-less' kernels," in *Proc. 5th Eur. Comput. Syst. (EuroSys)*, Paris, France, 2010, pp. 195–208.

[16] Jduck. (2010). *The Latest Adobe Exploit and Session Upgrading*. [Online]. Available: http://blog.metasploit.com/2010/03/latest-adobe-exploit-and-session.html

[17] S. Ragan. (2010). *Adobe Confirms Zeroday ROP Used to Bypass Windows Defenses*. [Online]. Available: http://www.thetechherald.com/article.php/201036/6128/

[18] *Security Advisory for Flash Player, Adobe Reader and Acrobat: CVE-2010–1297*, Adobe Syst., Mountain View, CA, USA, 2010. [Online]. Available: http://www.adobe.com/support/security/advisories/apsa10-01.html

[19] D. Goodin. (2010). *Apple Quicktime Backdoor Creates Code-Execution Peril*. [Online]. Available: http://www.theregister.co.uk/2010/08/30/apple_quicktime_critical_vuln/

[20] R. Hund, T. Holz, and F. C. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms," in *Proc. USENIX Security Symp.*, 2009, pp. 383–398.

[21] T. Dullien, T. Kornau, and R.-P. Weinmann, "A framework for automated architecture-independent gadget search," in *Proc. WOOT*, Washington, DC, USA, 2010, Art. no. 1.

[22] P. Chen *et al.*, "Drop: Detecting return-oriented programming malicious code," in *Proc. Int. Conf. Inf. Syst. Security*, Kolkata, India, 2009, pp. 163–177.

[23] S. Sinnadurai, Q. Zhao, and W. F. Wong. (2008). *Transparent Runtime Shadow Stack: Protection Against Malicious Return Address Modifications*. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.120.5702

[24] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Security*, vol. 13, no. 1, pp. 1–40, Nov. 2009.

[25] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Branch regulation: Low-overhead protection from code reuse attacks," *ACM SIGARCH Comput. Archit. News*, vol. 40, no. 3, pp. 94–105, Jun. 2012.

[26] L. Davi, P. Koeberl, and A.-R. Sadeghi, "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation," in *Proc. 51st Annu. Design Autom. Conf. (DAC)*, San Francisco, CA, USA, 2014, pp. 1–6.

[27] L. Davi *et al.*, "HAFIX: Hardware-assisted flow integrity extension," in *Proc. 52nd Annu. Design Autom. Conf. (DAC)*, San Francisco, CA, USA, 2015, pp. 1–6.

[28] P. Qiu *et al.*, "Physical unclonable functions-based linear encryption against code reuse attacks," in *Proc. 53rd Annu. Design Autom. Conf. (DAC)*, Austin, TX, USA, 2016, pp. 1–6.

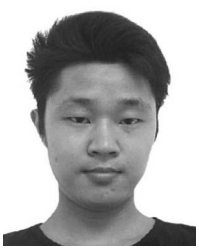
[29] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI: Cryptographically enforced control flow integrity," in *Proc. 22nd ACM SIGSAC Comput. Commun. Security (CCS)*, Denver, CO, USA, 2015, pp. 941–951.

[30] A. Kaur, P. Bhardwaj, and N. Kumar, "FPGA implementation of efficient hardware for the advanced encryption standard," *Int. J. Innov. Technol. Exploring Eng.*, vol. 2, no. 3, pp. 186–189, 2013.

[31] B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen, "Trade-offs for threshold implementations illustrated on AES," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 34, no. 7, pp. 1188–1200, Jul. 2015.

[32] J. Delvaux, R. Peeters, D. Gu, and I. Verbauwhede, "A survey on lightweight entity authentication with strong PUFs," *ACM Comput. Surveys*, vol. 48, no. 2, pp. 1–42, Oct. 2015. [Online]. Available: http://doi.acm.org/10.1145/2818186

[33] S. Gueron, *Intel Advanced Encryption Standard (AES) New Instructions Set*, Intel Corporat., Santa Clara, CA, USA, 2010.
[34] P. Chen *et al.*, "Automatic construction of jump-oriented programming shellcode (on the x86)," in *Proc. 6th ACM Symp. Inf. Comput. Commun. Security*, Hong Kong, 2011, pp. 20–29.
[35] *Address Space Layout Randomization*, Phrack Team, Ljubljana, Slovenia, Mar. 2003.
[36] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi, "Surgically returning to randomized lib(c)," in *Proc. Annu. Comput. Security Appl. Conf.*, Honolulu, HI, USA, Dec. 2009, pp. 60–69.
[37] H. Shacham *et al.*, "On the effectiveness of address-space randomization," in *Proc. 11th ACM Comput. Commun. Security (CCS)*, Washington, DC, USA, 2004, pp. 298–307.
[38] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu-Ghazaleh, "SCRAP: Architecture for signature-based protection from code reuse attacks," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Shenzhen, China, Feb. 2013, pp. 258–269.
[39] P. Chen, X. Xing, H. Han, B. Mao, and L. Xie, "Efficient detection of the return-oriented programming malicious code," in *Proc. Int. Conf. Inf. Syst. Security*, 2010, pp. 140–155.
[40] L. Davi, A.-R. Sadeghi, and M. Winandy, "Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks," in *Proc. ACM Workshop Scalable Trusted Comput. (STC)*, Chicago, IL, USA, 2009, pp. 49–54.
[41] J. Clause, W. Li, and A. Orso, "Dytan: A generic dynamic taint analysis framework," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, London, U.K., 2007, pp. 196–206.
[42] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP exploit mitigation using indirect branch tracing," in *Proc. 22nd USENIX Security Symp.*, Washington, DC, USA, 2013, pp. 447–462.
[43] D. L. Bruening, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., Massachusetts Inst. Technol., Cambridge, MA, USA, 2004.
[44] Z. Wang and X. Jiang, "HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *Proc. IEEE Symp. Security Privacy*, Berkeley, CA, USA, May 2010, pp. 380–395.
[45] C. Zhang *et al.*, "Practical control flow integrity and randomization for binary executables," in *Proc. IEEE Symp. Security Privacy*, Berkeley, CA, USA, May 2013, pp. 559–573.
[46] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Hardware-assisted run-time monitoring for secure program execution on embedded processors," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 14, no. 12, pp. 1295–1308, Dec. 2006.
[47] M. Bucci and R. Luzzi, "Random number generator," U.S. Patent 15/045 272, Feb. 17, 2016.
[48] P. Haddad and V. Fischer, "Random number generator," U.S. Patent 20 160 004 510, Jan. 7, 2016.
[49] I. Vasyltsov, B. Karpinskyy, H. Lee, and Y. Choi, "Random number generator," U.S. Patent 9 377 997, Jun. 28, 2016.
[50] M. Kayaalp, D. Ponomarev, N. Abu-Ghazaleh, and A. Jaleel, "A high-resolution side-channel attack on last-level cache," in *Proc. 53rd Annu. Design Autom. Conf. (DAC)*, Austin, TX, USA, 2016, pp. 1–6.
[51] A. Edwards, H. Vo, and A. Srivastava, "Vulcan binary transformation in a distributed environment," Microsoft Corporat., Redmond, WA, USA, Tech. Rep. MSR-TR-2001-50, 2001.
[52] J. Zhang, G. Qu, Y. Lyu, and Q. Zhou, "A survey on silicon PUFs and recent advances in ring oscillator PUFs," *J. Comput. Sci. Technol.*, vol. 29, no. 4, pp. 664–678, 2014.
[53] J. Zhang and G. Qu, "A survey on security and trust of FPGA-based systems," in *Proc. 13th Int. Conf. Field Program. Technol.*, Shanghai, China, 2014, pp. 147–152.

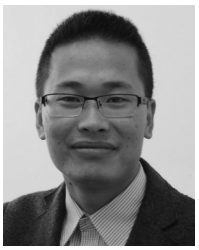


Pengfei Qiu received the B.S. degree in computer science and technology from the Harbin Institute of Technology, Harbin, China, in 2015. He is currently pursuing the Ph.D. degree in computer science and technology from Tsinghua University, Beijing, China.

His current research interests include hardware security, covering code reuse attacks, physical unclonable function, hardware trojan, and Intel sgx.

Yongqiang Lyu (M'09) received the B.S. degree in computer science from Xidian University, Xi'an, China, in 2001 and the M.S. and Ph.D. degrees in computer science from Tsinghua University, Beijing, China, in 2003 and 2006, respectively.

From 2006 to 2009, he was with Synopsys ATG and Magma, Beijing, China, where his main responsibility was the research and development on the leading techniques and products in physical synthesis. He is currently an Associate Professor with the Research Institute of Information Technology, Tsinghua University. His current research interests include cyber physical system usability and security.



Jiliang Zhang (M'15) received the Ph.D. degree in computer science and technology from Hunan University, Changsha, China, in 2015.

From 2013 to 2014, he was a Research Scholar with the Maryland Embedded Systems and Hardware Security Laboratory, University of Maryland at College Park, College Park, MD, USA. From 2015 to 2017, he was an Associate Professor with Northeastern University, Shenyang, China. Since 2017, he has been with Hunan University. He has authored over 30 papers in refereed international conferences and journals, such as the IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY, the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, *ACM Transactions on Design Automation of Electronic Systems*, and ACM/IEEE Design Automation Conference. His current research interests include machine learning security, hardware/hardware-assisted security, field programmable gate array, embedded system, and emerging technologies.

Dongsheng Wang (M'09) received the B.S., M.S., and Ph.D. degrees in computer science from the Harbin Institute of Technology, Harbin, China.

He is currently a Professor with the Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing, China. His current research interests include computer architecture, high performance computing, big data processing, and system security.

Gang Qu (SM'07) received the B.S. and M.S. degrees in mathematics from the University of Science and Technology of China, Hefei, China, and the Ph.D. degree in computer science from the University of California at Los Angeles, Los Angeles, CA, USA.

He is currently a Professor with the Department of Electrical and Computer Engineering and the Institute for Systems Research, University of Maryland at College Park, College Park, MD, USA, where he leads the Maryland Embedded Systems and Hardware Security Laboratory. He studies optimization and combinatorial problems and applies his theoretical discovery to applications in very large-scale integration (VLSI) computer-aided design (CAD), wireless sensor network, bioinformatics, and cybersecurity. He is one of the pioneers in hardware security and promoted this research field through organizing and participating numerous conferences, workshops, invited talks, tutorials, and panels. He has developed and taught an MOOC on hardware security through Coursera. His current research interests include embedded systems and VLSI CAD as well as wireless sensor network with a focus on low power system design and hardware related security and trust.