IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 21, 2018, accepted January 2, 2019, date of publication January 10, 2019, date of current version January 29, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2891762

HW-CDI: Hard-Wired Control Data Integrity

YONGSUK LEE AND GYUNGHO LEE
Department of Computer Science and Engineering, Korea University, Seoul 02841, South Korea
Corresponding author: Gyungho Lee (ghlee@korea.ac.kr)

This work was supported by the National Research Foundation of Korea funded by the Ministry of Education, Science, and Technology under Grant NRF-2015R1A2A2A01003242.

ABSTRACT Ensuring that a program follows an uncompromised control flow at the machine instruction level can provide sound protection from control flow attacks that transfer a control flow to the attacker's flow during program execution. This paper proposes an enhanced control data protection for control flow integrity called hard wired control data integrity (HW-CDI). The HW-CDI hides the control data via encoding with a key and requires proper decoding with the key for a correct control flow transfer. A unique aspect of HW-CDI is that this key changes in terms of not only the location but also the value of the control data. This paper describes the features necessary to make HW-CDI, an effective approach for securing program control flows with low-performance overhead. More specifically, this paper describes how to incorporate the HW-CDI into the processor's instruction pipeline so that it becomes an integral part of indirect branch instruction execution. It also provides information on how to generate the encoding/decoding keys without additional instrumented code. The HW-CDI is able to differentiate control flow transfer instances, providing context-based protection at negligible performance overhead.

INDEX TERMS Control data, control flow integrity, indirect branch, instruction set architecture, software security.

I. INTRODUCTION

Control flow attacks that change the ''program control flow'', i.e., the sequence of instructions to be executed for a program at run-time, represent one of the prevalent types of on-going threats to modern computer systems. Many researchers have made efforts to develop various defenses; however, control flow attacks have become more sophisticated, outpacing the defenses. There have been various mitigation strategies such as stack smashing protection (SSP) [1], data execution prevention (DEP) [2], and address-space layout randomization (ASLR) [3]. However, memory exploit vulnerability and trial-and-error replay attacks, including side-channel attacks, can bypass both ASLR and SSP [4]–[8]. Additionally, code reuse attacks (CRAs) bypass the DEP [4].

CRAs [9], [10] find the address of a short sequence of instructions (called a ''gadget'') that ends with an indirect branch instruction such as *call*, *ret*, or *jmp* from the code already existing in the memory. These gadgets are like virtual machine instructions, and CRAs link the gadgets to perform an arbitrary functionality (''Turing complete''). CRAs have evolved into more sophisticated types [4], [7], [11]–[14] from the original return oriented programming (ROP) [10]. For example, the BROP attack [4] showed that a generalized ROP attack is practical without knowledge of the code binary in the memory.

Control flow integrity (CFI) [15], which ensures program control flow at runtime to follow the control flow information described in the program at the machine instruction level, can represent the basic principle for protection against CRAs. CFI performs instrumenting of the program to validate a control flow transfer per a control flow graph (CFG) generated from the program. Two issues arise in implementing the CFI: firstly, how to check every control flow, and secondly, how to generate a precise and complete CFG. For practical implementations of CFI, these two issues are handled in a less than desirable manner [4], [11]–[13], [16]. CFI uses the static analysis to determine the target address of indirect branch instructions, which leads to overly permissive checks. Additionally, the need for the inline code instrumentation to check the legitimacy of each control flow transfer at run-time incurs performance overhead, which can be significant [15]. Also, having a ''shadow'' call stack, a protected copy of the return address stack, has been found to be essential for better protection [11], [15], [17], [18]. Although it is helpful to have more comprehensive identification of each control transfer distinctively using hardware support or heuristics, it is still possible to exploit CRAs that evade the recent sophisticated CFI implementations [4], [11]–[13], [19]. Additionally, in the Intel x86 instruction set architecture (ISA) of variable instruction length, an attack can ''create'' indirect branches

by fetching the instruction byte from the middle of multi-byte instructions [10]. These unintended branch instructions are difficult to handle because it is difficult, if not impossible, to deduce the correct location to insert the inline code instrumentation to check them.

A desirable program control flow protection scheme should be based on precise and complete control flow information and should be able to handle the dynamics of program control flows with minimal performance overhead. However, the existing protection schemes are either based on incomplete and imprecise CFG or have high performance overhead. We believe that a fine-grained control flow protection in hardware is needed to handle the dynamics of program control flows at a low performance overhead. This paper introduces a hardwired-control data integrity (HW-CDI) protection scheme to ensure control flow integrity with little performance overhead that is based on precise and complete information of dynamic control flows.

HW-CDI blocks the memory exploits for control flow attacks via encoding and decoding the program control data, i.e., function pointers and target addresses for indirect branch instructions, with dynamically varying keys generated from the location and value of the control data. For each encoding/decoding process, HW-CDI generates a different key to identify each control flow transfer instance distinctively, making the key an identifier (ID) of each control flow transfer instance. This also makes the return stack obfuscated with respect to the return addresses it holds: Each stack location is differentiated per the return address it holds, making the shadow stack not essential for control flow integrity. HW-CDI provides the integrity of the control data to ensure fine-grained control flow integrity at the machine instruction level at runtime. This paper describes the features utilized to incorporate the HW-CDI as an integral part of an indirect branch instruction execution.

While existing schemes for control flow integrity and control data protection utilize additional code, sometimes with added hardware features to reduce the overhead, instrumented for checking the integrity on top of existing machine instructions, HW-CDI incorporates the integrity checking into an inherent part of indirect branch instruction execution in the processor's instruction pipeline. Making the integrity checking an inherent part of a machine instruction that changes program control flow is the unique aspect of HW-CDI, which allows a tighter protection and a less overhead than existing protection schemes.

The contributions of this paper are as follows:

- This paper introduces the concept of obfuscating control data based not only on location but also on value.
- This paper describes a hardware efficient dynamic key generator that can be integrated into the processor's instruction pipeline so that the HW-CDI can be an integral part of machine instruction execution.
- This paper shows that HW-CDI prevents control flow attacks such as return-to-libc and CRAs with little performance overhead and without the need for a shadow stack.

The remainder of this paper is organized as follows. Section II provides background information on control flow integrity along with several existing schemes for control flow protection. Section III describes our control flow protection scheme, HW-CDI, and introduces the concept of hardwired control data integrity along with details for encoding/decoding processes for the program control data and how to incorporate the processes in the processor's instruction execution pipeline. Section IV demonstrates the effectiveness of the proposed HW-CDI scheme in terms of its efficacy on attack prevention, while Section V shows the impact of HW-CDI on a pipelined processor's instruction execution performance. Section VI discusses desirable aspects of a program control flow protection scheme and compares HW-CDI with other well-known existing control flow protection schemes. Finally, Section VII offers concluding remarks.

II. CONTROL FLOW INTEGRITY AND CONTROL DATA INTEGRITY

To ensure that software behaves as intended, one may want to ensure control flow integrity, i.e., that each control flow transfer follows the legitimate control flows described in the software. For control flow integrity, one may collect all legitimate control flow transfers and check if a specific control flow transfer instance is one of them. CFI [15] is a well-known software approach to ensure control flow integrity. CFI determines where an indirect instruction can branch to per a static analysis. In practice, static analysis of the program control flow has its limits; there are cases in which it cannot determine all possible target address values of an indirect branch. The indirect branch target is often determined at runtime.

CFI implementations [20], [21] seek to minimize the performance overhead of the validation. This is done by classifying the return addresses and the function pointers into a few different groups; each group has its unique identifier (ID), and the CFG is represented with the IDs (coarse grain) instead of the specific locations of indirect branches and its target addresses (fine grain). Most CFI implementations classify possible branch target locations into only two or three groups: a function pointer can branch to any function in the same group, and the return instructions can return to any return site. The coarse-grained CFI allows attackers to generate "new" control flows by swapping the source addresses or target addresses from the same group.

To ameliorate the coarse-grained CFI, heuristics about the legitimacy of the target addresses have been introduced [22], e.g., a return target should be the location below the corresponding call site. Additionally, hardware assistance including efforts to utilize branch prediction and monitoring/debugging features have been proposed for less performance overhead and better distinction of the control flow transfers [22], [23]. Coarse-grained CFI implementations with the hardware support cause generally less performance

overhead; however, the vulnerability of the CFI implementations remains more or less intact [17], [24].

CET from Intel is a well-known CFI scheme [25] and is a hardware supported variation of Microsoft's control flow guard [26]. CET is a coarse-grained CFI in that it coalesces the indirect branch target addresses together without the information on which the indirect branch is associated with which target addresses. It is essentially a single ID CFI with the shadow call stack: Its protection efficacy is limited, though the shadow call stack in CET provides good protection for the return targets. However, providing a safe shadow call stack in the memory is not trivial [22]. Hardware-enforced CFI (HCFI) [24] is similar to the CET but attempts to be closer to the original software-based CFI [15] with a somewhat limited implementation of the shadow call stack. Note that HCFI is also a coarse-grained CFI, which leads to the same vulnerability of software-based CFI implementations.

CFI has recently shifted away from coarse-grained CFI to fine-grained CFI [18], [27]–[30] by identifying each control flow transfer distinctively. Fine-grained CFI may be less vulnerable to those attacks targeting the coarse-grained CFI implementations, but they are prone to high performance overhead, and it is still possible to exploit CRAs that evade the fine-grained CFI implementations [4], [11], [12], [14]. CFI schemes including fine-grained ones are for user level programs and cannot protect the control data at the kernel level. The ret2usr attack [31] compromises the control data only in the kernel (e.g., a kernel function pointer or return address).

Relying on the static CFG as the reference for CFI, whether it is coarse-grained or fine-grained, is based on the assumption that the software is immutable. This implies that it cannot be used for self-modifying code or for the code generated just-in-time. Software-based CFI policies typically verify each indirect branch target by executing the instrumented code before the execution of an indirect branch instruction. This may introduce a risk of TOCTOU (time-of-check and time-of-use) vulnerability.

Control flow integrity can be assumed if the integrity of the control data is maintained. Maintaining the integrity of all program data has been studied from early on, e.g., the trusted computing proposal by the US Air Force in 1973 [32]. It is difficult to maintain the data integrity for general program data despite many proposals for tracking data flows and sophisticated memory protection [33]–[37]. Ensuring the integrity of program data in general is difficult to do, especially without incurring a significant overhead [33]. For example, Intel's MPX for memory safety is reported to incur 15% to 400% performance overhead [38]. Control data integrity (CDI) schemes seek to ensure uncompromised control data at runtime. CDI has a limited scope in ensuring data integrity; it ensures the integrity of function pointers and target addresses for indirect branch instructions [18], [39], which allows the run-time protection to be more precise but with less overhead. This paper proposes CDI as an intrinsic part of indirect branch instruction execution in a way similar to the virtual address to physical address translation via TLB (Translation Lookaside Buffer) for every memory access in modern processors.

III. HARD WIRED CONTROL FLOW INTEGRITY (HW-CDI)

HW-CDI is designed to enforce each control transfer to be legitimate without adding the instrumentation code for the control flow validation. It does not utilize a static CFG for the reference control flow. Instead, it obfuscates the control data via encoding with a key and mandates decoding with the key. In describing the HW-CDI, we use the Intel x86 instructions [40] for examples, but the general principle of HW-CDI can apply to any instruction set architecture (ISA).

A. BASIC APPROACH

The static CFG used in most CFI schemes may be imprecise and incomplete because of its conservative nature and lack of dynamic information. In addition, checking the control data against the CFG requires an added instrumented code, which incurs not only performance overhead but also the potential risk of TOCTOU. The HW-CDI approach involves encoding control data when it is stored to the memory and decoding it when it is read from the memory. This hides the control data value until the last minute before loading it to the program counter. However, obfuscating the data incurs a difficulty in managing the key for encoding and decoding and leads to the overhead associated with encoding and decoding. Additionally, relying on a single key for the whole program execution may make it an easy target for a replay attack [4].

Consider an example of CRA [11] in Fig. 1 The attack creates a loop by compromising the return address for the second call in the return stack: Calling Function_A for the second call returns to RA_1 for the first call instead of RA_2. Most CFI schemes check the legitimacy of the source-target address pair, i.e., whether it is a valid branch site and its proper target address. Because RA_1 and RA_2 are both legitimate for the *ret* instruction of Function_A, the attack bypasses

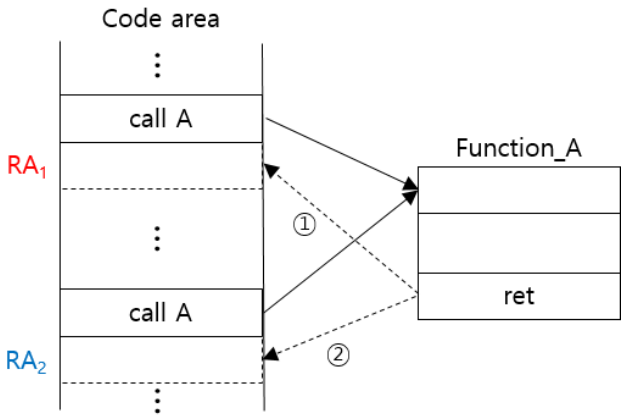

FIGURE 1. A CRA example [11]: A loop can be formed by returning from the second call to the return site, RA1 of the first call (①), instead of its proper return site RA2 (②).

the CFI schemes. To provide proper protection, one needs context information to distinguish the return addresses for the corresponding calls. CFI schemes suggest having the shadow stack, i.e., a protected copy of the return stack. The shadow stack provides a clean uncompromised return address matching the current call-return sequence. Having a shadow stack is considered essential for proper implementation of the CFI [11], [13], [15], [17], [18]. The industry has recently provided provisions to facilitate the shadow stack, e.g., Control-flow Enforcement Technology [25]. However, the shadow stack not only introduces a significant overhead but is also difficult to properly manage [41].

HW-CDI varies the encoding key dynamically by generating the key from the control flow information of a given indirect branch instance. Control data and their association with a particular control flow transfer are hidden via encoding. In the example in Fig. 1, each call instance encodes its return address with its own key before saving it to the return stack. In other words, the encoding key represents the context of a particular call sequence. This differentiates HW-CDI from previously proposed control data protection schemes [1], [23], [42], [43]. At the return after finishing Function_A, the *ret* instruction decodes the return address with the key before loading the return address to the program counter. With RA_1 and RA_2 encoded with different keys, the two returns for the first call and the second call are differentiated. Passing the key from the encoding to its corresponding decoding is essential and may require a complete pointer variable tracking that is often difficult to realize in practice. Additionally, managing many different keys is a difficult task to properly perform in practice. HW-CDI uses the target address for an indirect branch and the memory location holding the target address together to generate the key. Note that for a specific control flow transfer, the memory location and the value for the target address should remain the same when they are used and when they are defined prior to use.

In the example in Fig. 1, when the upper call instruction is executed, it generates a dynamic key with RA_1 and the address of the top of the stack at which RA_1 is to be saved. At the ret instruction after finishing Function_A, HW-CDI decodes the top of the return stack using not only the address of the top of the stack but also the encoded return address saved. In other words, the address of the top of the stack and the return address itself, the invariants for the call-return sequence, are utilized as the seed for generating and restoring the key. For a general indirect branch, the memory location for storing the target address before an indirect branch and the target address itself together make the seed for generating the encoding key.

HW-CDI attempts to have the indirect branch instruction itself do the decoding as a part of its execution, which obviates the need for inserting additional instrumented code. Several potential scenarios can exist for managing the key, and the next section describes the details of the scheme that we propose for the Intel x86 ISA.

B. ENCODING AND DECODING

HW-CDI obfuscates the target address for indirect branch instructions via encoding when it is stored in the memory. Indirect branch instruction, such as call or return, needs to decode the stored target address before loading it to the program counter for a control flow transfer. One may have the compiler add an instrumented code for the encoding and decoding, but HW-CDI does the encoding and decoding without the instrumented code. For example, consider a call-return sequence: The *call* instruction itself may encode the return address before saving it into the return stack, while the return instruction decodes it before loading it to the program counter. HW-CDI intends to have the encoding and decoding as an integral part of the indirect branch instruction execution with a key that is unique to a particular call-return sequence. For a call-return pair, the address for the top of the stack and the return address itself, the invariants for the call-return sequence, can be utilized as the seed for generating and restoring the key.

However, unlike the return address, it is not trivial to have the encoding key be the same for the decoding of a generic target address. For general indirect branches, the relationship between who defines the target and which indirect branch instruction uses it needs to be established. The encoding may have the key explicitly passed to the decoding, which will create another level of attack vulnerability and additional overhead. Another option would be to have all the writes to the memory encode data and the corresponding reads of the memory decode the data first prior to actual use as in the schemes for general data integrity [44]–[47], which will cause another level of complexity and overhead. HW-CDI introduces variants of the instruction for accessing the memory, e.g., a variant of *mov* in Intel x86 ISA, *emov* for storing control data to the memory with encoding and *dmov* for loading control data from the memory with decoding. To make a proper control flow transfer, the encoded control data needs to be decoded before being loaded into the processor's program counter. For this, HW-CDI enhances the indirect branch instructions (*jmp*, *call*, and *ret*) with decoding functionality. Note that for the *ret*, the return address saved in the runtime stack is encoded at the *call* without using a separate *emov* instruction.

Table 1 shows typical control flow transfer cases of setjmp/longjmp and call/return. The control data stored by the *emov* instruction are encoded first (with a dynamically changing key value, which is described later in the following subsection) before being moved to the memory, and the indirect branch instruction decodes the control data prior to loading it into the program counter as an intrinsic part of its execution.

The new *emov* instruction is reserved for storing the control data, i.e., the address pointing to the target address of an indirect branch. This instruction is introduced to avoid instrumented code inserted for the encoding. Without the *emov*, the compiler can identify the instruction that defines the target

TABLE 1. Control flow transfer examples: (a) setjmp/longjmp, (b) function pointer. With HW-CDI, *emov* instruction stores a function pointer or a branch target address after encoding it. HW-CDI enhances indirect branch instructions, e.g., *jmp*, *call*, with decoding of the control data encoded by the *emov* instruction.

without HW-CDI	with HW-CDI
setjmp	
mov %ecx, 20(%eax);	***emov*** *%ecx, 20(%eax);* //store %ecx with **encode** to 20(%eax)
…	
longjmp	
…	
mov 20(%edx),%ecx;	//copy pointer of 20(%edx) to %ecx
*jmp *%ecx;*	//**decode** pointing %ecx data and jump

(a)

without HW-CDI	with HW-CDI
mov $fun1, 12(%esp);	***emov*** *$func1, 12(%esp);* //store address of $func1 with **encode** to 12(%esp)
…	
mov 12(%esp), %eax;	//copy pointer of 12(%esp) to %eax
*call *%eax;*	//call pointing of %eax data with **decode** and store return address with **encode**

(b)

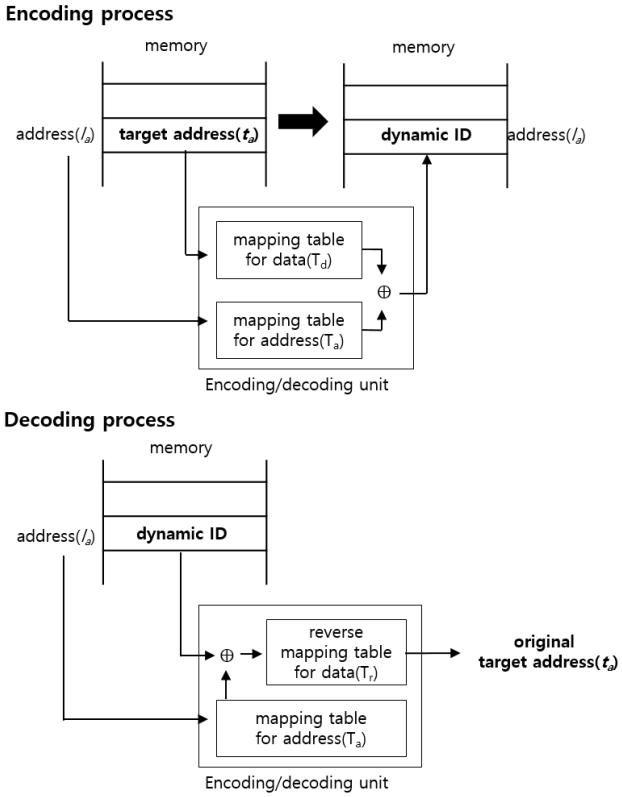

FIGURE 2. Encoding and decoding of control data for ID.

for a particular indirect branch and inserts the instrumented code. However, this introduces additional overhead and presents a potential vulnerability for the TOCTOU. Additionally, note that the control flow transfer via the indirect branch instructions is primarily "indirect" with Intel x86 ISA, meaning that it branches via the pointer as shown in Table 1 instead of to the target directly. When a compiler generates an assembly code, it is aware of the control data via the use-define chain data structure. So, the compiler can find the proper pointer as the control data for an indirect branch stored by the *emov* instruction. If an indirect branch instruction uses the target address in the register directly instead of indirectly via the pointer, the compiler may replace the memory read to the register prior to the indirect branch with *dmov* instruction.

C. KEY FOR ENCODING AND DECODING

HW-CDI maintains a pool of keys and generates a specific key dynamically per the control data value, i.e., the target address itself and the address holding the target address for the indirect branch. This specific key makes the identifier (ID) of each control flow transfer because the key may be different for different control transfer instances. Even for the same indirect branch instruction, the key may change if its target address changes.

For a fast and low-cost key generation, HW-CDI performs XOR-ing of the two values from the tables (see Fig. 2). The XOR operation possesses three desirable characteristics. First, it is a simple operation causing little delay. Second, it is invertible, which allows the extraction of the original value. This characteristic enables the HW-CDI to convert a particular ID, i.e., the encoded target, to its original value of the target. Finally, the XOR operation scatters the resulting bit pattern [48] providing a randomization effect, which motivates the encryption schemes to be generally based on XOR operations.

For the decoding, HW-CDI has to determine the original value of the control data from the encoded control data value. When the processor executes an indirect branch instruction, only the encoded control data value (ID) and its address are available. HW-CDI accesses the mapping table T_a with the address to get the mapped value for the address used in the encoding. Then, HW-CDI performs XOR-ing of the mapped value from T_a with the ID from the memory, which gives the proper index for the reverse mapping table for data (T_r) to recover the original control data.

Fig. 3 illustrates the encoding and decoding processes with the mapping tables. They are a simplified scheme of a typical hardware realization of the Advanced Encryption Standard (AES) [49]. This paper assumes the size of the mapping tables of T_d and T_a to be 8bits * 2^8 = 256bytes, which can be implemented with a fast SRAM with a linear indexing. We assume that the table T_r would be implemented in content-addressable memory (CAM). The table T_r has a tag for each entry that is merely an index value of T_d, and the tag holds the contents of the corresponding T_d entry for the reverse mapping. We assume that the tables T_d and T_a are initialized at the start of a process with 8-bit random numbers. We assume that

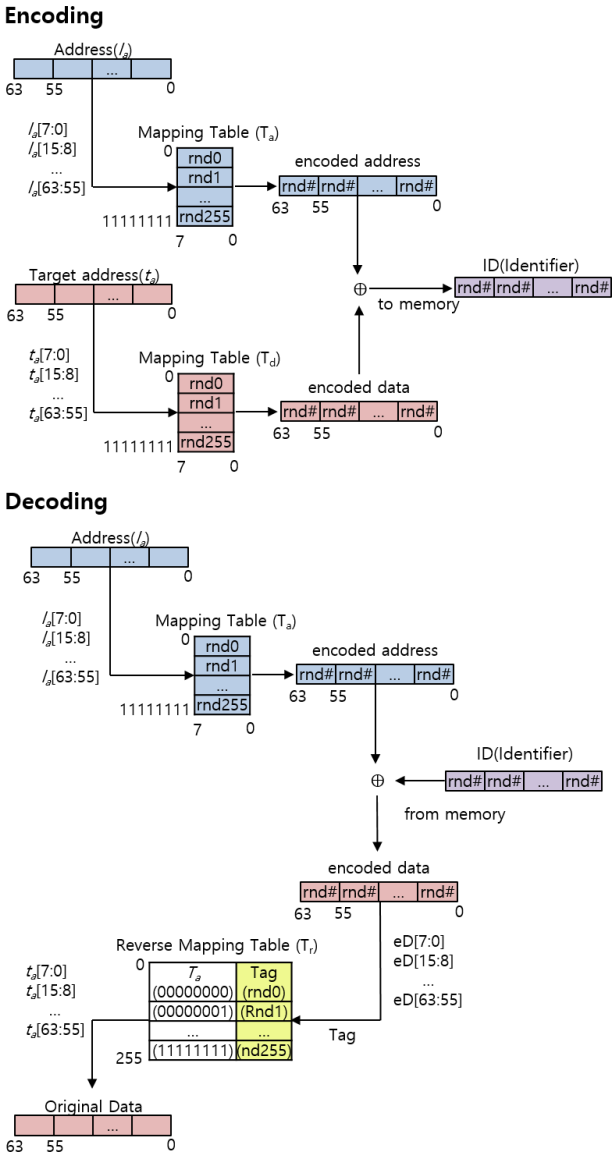

FIGURE 3. Mapping tables, T_a, $T_d$and T_r, for encoding and decoding. Note that the reverse mapping table T_r is implemented in content addressable memory (CAM).

the random numbers are unique, i.e., there are no two same values in the tables, which is necessary for a correct reverse mapping. The encoding and decoding can be described as follows (ID is the encoded value of the control data):

Encoding: $T_d(t_a) \bigoplus T_a(l_a) = ID$

Decoding: $T_r(ID \bigoplus T_a(l_a)) = t_a$ where $\bigoplus$ is exclusive-or. Note that T_r is a CAM and gives the index value of T_d via tag matching with $ID \bigoplus T_a(l_a)$.

We assume that the tables T_d and T_a are initialized with 8-bit random numbers. To access the mapping tables, the control data and its address are divided into 8 bit subunits. All eight subunits (for 64-bit address and data) access the table in parallel (CAM with an eight ported read is readily available as can be seen in the Translation Lookaside Buffer (TLB) design for the virtual-to-physical address translation in modern processors). We may have the table index in 16 or more bits with an increase in the table size for a higher entropy of randomness or use the AES instructions [38]. However, HW-CDI aims to have the decoding be an intrinsic part of an indirect branch instruction execution; a larger table index or the use of AES may introduce a delay in the processor's instruction pipeline.

D. INSTRUCTION PIPELINE

HW-CDI aims to have the encoding/decoding done in a single clock cycle and strives to have the encoding/decoding done as an intrinsic part of a machine instruction execution. HW-CDI incorporates the encoding/decoding process into the processor's instruction pipeline. For illustration purpose, consider a simple five stage instruction pipeline [50].

Fig. 4 illustrates HW-CDI implementation of the encoding process in the processor's instruction execution pipeline per the simple five stage pipeline. HW-CDI generates the key after the Execute (EX) stage, because the effective address is available after the EX stage. We add the Encoding (EC) stage after the EX stage. For the decoding, HW-CDI generates the key after the Memory (M) stage because it needs the encoded data read from the memory. We add the Decoding (DC) stage after the M stage as shown in Fig. 5.

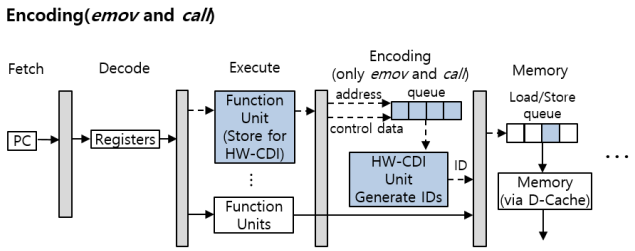

FIGURE 4. Encoding with HW-CDI; the features for the HW-CDI encoding process are in blue shade.

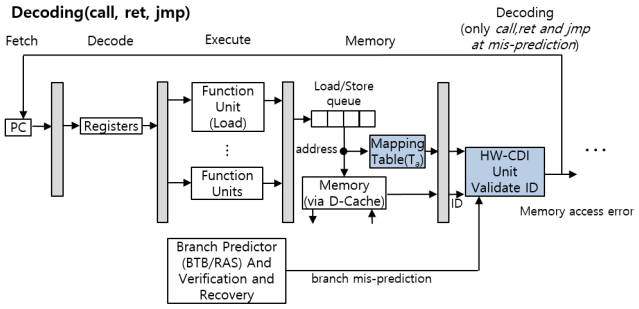

FIGURE 5. Decoding with HW-CDI; the features for the HW-CDI decoding process are in blue shade.

The actual instruction pipeline varies from processor to processor. However, the incorporation of the HW-CDI into the processor's instruction pipeline seems straightforward because HW-CDI does not introduce any conditions to impede the flow of the instruction pipeline. As long as the

table access for the key generation does not incur a significant delay, the performance impact by HW-CDI should be insignificant. Considering the TLB access or the tag access of the first level cache in modern processors, the table access for the key generation and the encoding should be feasibly performed in one cycle.

One important aspect to emphasize is that the decoding is not in the normal flow of the instruction pipeline in modern pipelined processors. Most modern pipelined processors are with branch prediction; the instruction pipeline for the next instruction moves on with the predicted target address without waiting, while the verification of the prediction goes on in the background. The delay incurred by the decoding in HW-CDI does not materialize if the branch prediction is correct. For the same reason, the delay caused by the encoding for the *call* instruction also does not materialize because the processor fetches the target instruction for the *call* with the predicted target address without waiting. Only when the branch prediction fails, the decoding delay introduced by HW-CDI may affect the performance: In terms of instruction pipeline delay, HW-CDI creates a delay effect akin to an increased mis-prediction penalty for indirect branch instructions including return instruction.

IV. EFFECTIVENESS AGAINST CONTROL FLOW ATTACKS

The threat model assumed in our study is a usual one utilized in most research works on user level software security [4], [14], [18], [24], [27], [44]: The attacker has no control over the operating system to ensure that the attacker has no ability to tamper with the mapping tables, T_a, T_d and T_r, that HW-CDI utilizes for encoding and decoding the control data. Note that the mapping tables are software transparent, which are not a part of the processor's memory space and accessible only by the processor as a part of an instruction execution. Some protection schemes [11], [17], [19] need additional assumption in the threat model: Memory pages are protected by data execution prevention (DEP) [2], which disallows execution of data as instructions. Most modern processors provide a NX (No Execute) bit so that the operating system marks the data area as non-executable. However, the protection of DEP is not essential for HW-CDI. Attackers can overwrite the program control data such as a return address or inject a malicious code into the run-time stack by exploiting memory vulnerabilities, e.g., overflowing a buffer near the control data or by dereferencing a data pointer compromised to point to the control data stored in the memory. HW-CDI disallows any control flow transfer by an indirect branch instruction without properly decoded target address.

In CRAs, the attacker compromises one of the target addresses first, usually one of the return addresses, for an indirect branch instruction. HW-CDI obfuscates the target address via encoding it with a key, and the key for the encoding and decoding changes; different keys correspond to different target address values and the memory locations holding them. Even if an attacker determines the key for a particular memory location holding the encoded target address via a replay attack or a side-channel attack, the key leaked to the attacker is an inappropriate one. As the target address or the location holding the target address changes, HW-CDI mandates a different key. Additionally, HW-CDI does XOR-ing of the results from the mapping table accesses to obfuscate the mapping tables (T_a and T_d) against the attacker's guessing attack.

Note that the attackers need to determine the key from the mapping tables to perform proper decoding and to have a control flow transfer to the attacker's flow. As described in the previous section, the two key tables are software-transparent; the tables can be read only as an integral part of indirect branch instruction execution or a memory operation to store the control data. With the mapping table size as suggested in the previous section, the attacker needs $(2^8)^8/2$ guesses for a repeating trial and error attack because each 8-bit section of the eight sections needs to be guessed correctly. As with any encryption schemes, the keys can leak, but it does not seem straightforward, and one may increase the table size to make the encryption entropy higher. Also note that the tables are reinitialized whenever the program (re)starts. To reduce the chance of a successful attack further, one may refresh the mapping tables instead of keeping the table contents during the whole period of program execution. For example, the compiler may insert a table re-generation point when there are no pending indirect branches using the control data encoded, i.e., the variables for the control data are no longer alive. One may even force re-encoding and re-storing the target addresses per a newly generated table during the program execution at the cost of higher overhead.

HW-CDI generates the encoding key to make a dynamic ID; each control flow transfer instance has a different key from the others. These IDs of control flow instances are different one from another and provide advantages over other CFI schemes. This can prevent a control flow transfer branching to a compromised address in a fine grain manner as in an ideal fine-grained CFI. Also, HW-CDI saves each return address differentiated through the encoding key, making the shadow stack non-essential for control flow integrity.

Consider again the CRA example of Fig. 1, redrawn here as Fig. 6. The *ret* instruction in 'Function_A' has targets RA_1 and RA_2. HW-CDI uses the return address and its location in the stack together to generate the dynamic key. When the upper *call* instruction in the code area is executed, it generates a dynamic ID with the return address RA_1 and its location in the return stack PA (① in Fig. 6). HW-CDI generates a different key with RA_2 and PA for ② in Fig. 6. HW-CDI differentiates dynamic instances of the same static control flow transfer and disallows the CRA to create a loop by replacing RA_2 with RA_1.

For dynamically generated target addresses, e.g., object addresses in the *vtable*, HW-CDI differentiates them with the keys generated dynamically, which disallows the attack compromising the *vtable* [14]. HW-CDI also validates unintended instructions from the middle of multi-byte instructions [10]. CFI schemes with a static CFG are not able to validate

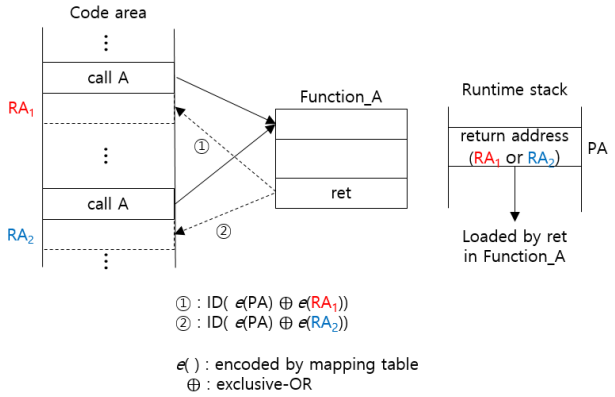

FIGURE 6. Dynamic Key (ID) for the same *ret* instruction; each instance of the two returns will have a different key from the other instance.

TABLE 2. Attack methods that HW-CDI prevents per RIPE [53] test suite exploits: for each attack method, the exploited control data types are marked with "○". The types of control data are: RA = Return Address, FP = Function Pointer, LB = Longjmp Buffer, VS = Vulnerable Structure and OBP = Old Base Pointer.

Attack methods	Control data types				
	RA	FP	LB	VS	OBP
Code inject [51]	○	○	○		○
Return-to libc [52]	○	○			○
Return-oriented programming [10]	○	○			
jump-oriented programming [9]	○	○	○		
Out of control [13]	○	○	○		
Control flow bending [11]	○	○		○	
Counterfeit Object-Oriented Programming [14]	○	○		○	

the unintended instruction. Below is a code example of an unintended instruction of *ret*.

```
1: f7 c7 07 00 00 00      test $0x00000007, %edi
2: 0f 95 45 c3            setnzb -61(%ebp)

3: f7 c7 07 00 00 00 0f   movl $0x0f000000, (%edi)
4: 95                     xchg %ebp, %eax
5: 45                     inc %ebp
6: c3                     ret
```

A hexadecimal representation of "*setnzb -61(%ebp)*"on line 2 is "*0f 95 45* ***c3***" on line 2. The "*c3*" byte is the *ret* instruction in Intel x86. An attack sets up a compromised address at the top of the stack, and point a to the byte of "*c3*" will cause the processor to follow a compromised control data that the attacker wants. However, the HW-CDI incorporates the *ret* instruction execution with the CFI validation process and does not allow the unintended *ret* to bypass the CFI validation.

To determine the efficacy of HW-CDI for various control flow attack methods, we utilized the RIPE test suite [53], which consists of 850 control flow attacks. Of 850 exploits in RIPE tested on the Fedora 12 with DEP and ASLR enabled, 127 exploits made successful attacks despite protection from DEP and ASLR. The successful exploits are on 10 return addresses, 40 function pointers, 40 vulnerable structures and 37 base pointers. Table 2 shows the control flow attack methods along with the exploit types. While HW-CDI is able to prevent all 127 exploits listed, the existing CFI schemes have been reported to not be able to do so [11], [13], [14].

V. PERFORMANCE

Because HW-CDI is implemented as an integral part of the machine instruction execution, the performance overhead is lower than that of other CFI schemes. The performance overhead comes from the one additional pipeline stage for the encoding or the decoding, which does not materialize unless the immediately following instruction(s) needs the encoded or decoded control data.

The actual control transfer can occur only after the decoded control data value is available to update the program counter. However, one should note that the branch predictor can significantly reduce the potential overhead. From a software security perspective, entries in the branch target buffer (BTB) and Return Address Stack (RAS) represent uncompromised addresses because every BTB entry from the past target address is already a validated one if the validation process is in place. The decoded value goes to the BTB or the RAS if the branch prediction turns out to be incorrect at the time of verifying the prediction. RAS and BTB are software-transparent storage and can be employed to detect potential malicious control flows and unusual return paths because a target address for such cases has never been executed, which causes a mis-prediction. RAS provides a prediction success rate of 98% or above in general [23], which will drastically reduce the needs for the control flow validation for CFI. Note that since the control data are function pointers and return addresses, it is less likely that dependencies between the instruction defining the control data and the indirect branch instruction using it within the pipeline timing window of one cycle will cause an extra delay over the case of no encoding applied. The major performance effect of HW-CDI is more or less the same as the case with the mis-prediction penalty for indirect branch instructions but a one cycle increase, which is the delay for the decoding.

The usual metric for measuring the processor's performance is *cpi* (cycles per instruction), i.e., the number of processor cycles for the execution of a machine instruction on average during a program execution [50]. We utilized the Simplescalar-3.0 simulator [54], simulating a 4-wide issue out-of-order 9 stage pipeline core with 64KB L1 data and instruction caches. To obtain a more concrete idea of the performance overhead in terms of *cpi*, we studied SPEC2000 CPUint benchmark programs. Each benchmark was simulated for one billion committed instructions after fast-forwarding for the first 100 million instructions. The evaluation environment assumed for the processor is as follows:

Pipeline – 4-issue, 9-stage
Issue queue size – 16
Reorder buffer size – 64
Branch predictor – gshare predictor with 4096 counters and 16-entry RAS
Mis-prediction penalty – 5 cycles
L1 caches – 64 KB Inst/64 KB Data, 4-way, 2 cycle hit latency
Memory – dual ported with 100-cycle latency

We assumed a one cycle performance penalty in our experiments for each of the decoding and the encoding stages as described in the previous section. Our experimental results show that the performance overhead in terms of *cpi* is 0.19% on average with the highest overhead of 0.5% for *perlbmk* (see Fig. 7). The overhead observed was 0.23% for *gcc* and 0.09% for *gzip* (see Fig. 7). Compared to HW-CDI, the average overhead of the original CFI [15] for the SPEC2000 CPUint benchmark programs is reported to be 21% (11% for *gcc* and 5% for *gzip*). Although recent CFI implementations are with less performance overhead, HW-CDI still produces less overhead than the recent CFI implementation (see the next section).

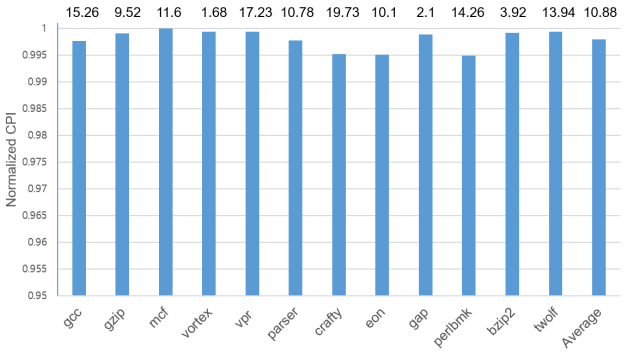

FIGURE 7. Normalized cycles per instruction (CPI) overhead of HW-CDI. Atop the bars for each program is the mis-prediction rate (%) for indirect branches including the return.

VI. DISCUSSION: COMPARISON WITH RELATED WORKS

As noted earlier, most CFI schemes rely on the instrumentation, inserting a few instructions to perform CFI checks on indirect branches with respect to a static CFG. The instrumentation can be done as part of a compiler optimization step, static binary rewriting, or through dynamic library translation. The use of a static CFG as the reference for checking each control flow limits the scope of the protection because dynamically linked routines are not included. In addition, one needs to make a conservative estimation for the details of program control flows, which may allow illegitimate control flows from attacks. The instrumented checks may rely on runtime data structures in writable memory area, which can be exploited for CRAs [55], [56] and incur significant performance overhead as noted previously. Another aspect of most CFI schemes is that they are coarse-grained, coalescing the target addresses into two or three groups, which allows sophisticated CRAs as mentioned earlier.

A desirable control flow protection scheme should have the following characteristics:

- It is based on precise and complete information on program control flows, not only static flows but also dynamic flows, and
- It checks each control flow for its legitimacy with little or no additional instrumented code and should incur little performance overhead.

Our HW-CDI is not based on a CFG generated from a program a prior to its execution and instead uses the encoding to protect the program control data that a machine instruction uses for its target address for a control flow transfer. As a result, HW-CDI is able to handle the dynamics of control flow changes at run time per the control flows exactly as described in the program without additional instrumented code inserted. HW-CDI does the checking of each control flow in hardware as an intrinsic part of an indirect branch instruction execution, which facilitates no additional instrumented code for the checking and allows little performance overhead as shown in the previous section.

HW-CDI has an order of magnitude or two less performance overhead than the CFI implementations with added instrumentation for control-flow validation. For example, an early CFI implementation with a static CFG [15] reported an average overhead of 21%, while recent coarse-grain CFI implementations have reported lower average overhead of 1% [24] and 2.14% [57] by utilizing heuristics on indirect branch behavior along with hardware support. Hardware CFI [58] is another attempt to have the CFI based on a static CFG implemented more precisely by introducing a set of new machine instructions. Hardware CFI reports an average overhead of 1.75% for several programs from the SPEC CPU2006 benchmark. Note that these schemes are coarse-grained protection. The coarse-grained CFI protections are based on incomplete control flow information and allow attackers to generate ''new'' control flows by swapping the source addresses or target addresses from different control flows.

A fine-grained CFI proposal called CCFI [18] is similar to our HW-CDI in the sense that it enforces a fine-grained CFI by hiding the control flow objects via encryption. However, CCFI is based on a static CFG and is not able to handle the dynamics of control flow at run-time. CCFI also needs additional instrumented code inserted: CCFI compute and store a message authentication code of control data that is encrypted using the AES-NI instruction [40] and is stored with the checksum. In doing so, CCFI prevents hijacking of a control flow, but involves a relatively higher performance overhead than other CFI schemes. CCFI reports a rather high average overhead of 52% on SPEC CPU2006.

Another scheme similar to HW-CDI is CPI [27], which collects and identifies all of the sensitive code pointers into a safe region to prevent attackers from overwriting the control data. CPI needs additional instrumented code for the checking

and is reported to have an average overhead of 8.4% for SPEC CPU2006. CPI is based on static data flow analysis of a program and has been demonstrated to be subverted [56], though it is claimed that the CPI implementation, using either hardware-enforced segmentation or software fault isolation, cannot be subverted [59].

ARM v8.3 architecture includes a feature that allows the attachment of a cryptographic checksum to the control data (pointer authentication) [60]. The checksum will be recalculated and verified via a separate AUTH instruction before the control data are used. An attacker without the key for the checksum is not able to create valid pointers for use in an exploit. This may allow the implementation of CPI and CCFI with less performance overhead. However, these schemes are based on static information and need additional instrumented code inserted.

As noted earlier, HW-CDI described in this paper is with code examples for the Intel x86 ISA that allows an indirect branch instruction to specify a memory operand as the pointer to its target address. Modern commercial processors other than Intel x86 ISA mostly have RISC style ISA, and RISC style ISA like ARMv8-A ISA, requires the target address of an indirect branch instruction to be loaded to the processor's register from the memory prior to the indirect branch instruction execution [50]. To have HW-CDI with RISC style ISA, *dmov* instruction is essential for reading the encoded control data with the decoding. The *dmov* instruction replaces the normal instruction for loading data from the memory to the processor's register for a specific indirect branch instruction. Note that a *dmov* instruction with decoding can be fused together with the following indirect branch instruction that uses the decoded target instruction as in the case of fusing the memory read with its dependent next instruction [50]. As a result, there is little to no change in terms of implementation complexity and performance overhead when HW-CDI is implemented for processors with RISC style ISA.

One aspect of HW-CDI that distinguishes it from other existing schemes is that HW-CDI differentiates control flow transfer instances. HW-CDI varies the key not only depending on the location but also on the value of the control data. This provides context information that differentiates control flow transfers from each other. If an attack swaps two legitimate control data values, the attack would fail under HW-CDI, while it goes through under most existing schemes including CCFI and the pointer authentication of ARM v8.8.

Although HW-CDI can protect program control flow with little performance overhead, it is not a panacea for all possible control flow attacks. Like other control data protection schemes cited in this paper, it is not able to protect the control data defined indirectly from non-control data. Also, note that HW-CDI needs recompilation of a program, which requires the program source code.

VII. CONCLUSION

This paper has introduced a HW-CDI scheme that provides protection for the control data as an intrinsic part of machine instruction execution, in a manner similar to the virtual-to-physical address translation via TLB. An instruction storing the control data encodes it with a key as a part of its execution, and an indirect branch instruction using the control data decodes it prior to loading it to the program counter for a control flow transfer. We have proposed necessary features for enhanced control data protection with HW-CDI. A unique aspect of HW-CDI is that the key for the encoding varies not only depending on the location but also on the value of the control data. This facilitates distinction of the control flow transfers from each other. While an attack attempts to compromise the control data, HW-CDI mandates different keys for different control data, making the change of the control transfer to the attacker's way infeasible with the compromised control data.

This paper has shown that HW-CDI can be incorporated into the processor's instruction pipeline with little overhead, allowing a fine-grained CFI without the need for the instrumented code to validate each control flow transfer instance. HW-CDI can make a more precise and comprehensive control flow protection than the existing CFI schemes at less overhead.

REFERENCES

[1] C. Cowan *et al.*, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proc. 7th Conf. USENIX Secur. Symp.*, San Antonio, TX, USA, vol. 7, 1998, p. 5. [Online]. Available: https://dl.acm.org/citation.cfm?id=1267554

[2] Microsoft. (2006). *Data Execution Prevention (DEP)*. [Online]. Available: https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in

[3] PaX Team. (2003). *Address Space Layout Randomization (ASLR)*. [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt

[4] A. Bittau, A. Belay, A. Mashtizadeh, D, Mazières, and D. Boneh, "Hacking blind," in *Proc. IEEE Symp. Conf. Secur. Privacy*, San Jose, CA, USA, May 2014, pp. 227–242.

[5] T. Hobson, H. Okhravi, D. Bigelow, R. Rudd, and W. Streilein, "On the challenges of effective movement," in *Proc. 1st ACM Workshop Moving Target Defense*, Scottsdale, AZ, USA, 2014, pp. 41–50.

[6] J. Seibert, H. Okhravi, and E. Söderström, "Information leaks without memory disclosures: Remote side channel attacks on diversified code," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Scottsdale, AZ, USA, 2014, pp. 54–65.

[7] K. Z. Snow, L. Davi, A. Dmitrienko, C. Liebchen, F. Monrose, and A. R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proc. IEEE Symp. Conf. Secur. Privacy*, Berkeley, CA, USA, May 2013, pp. 574–588.

[8] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *Proc. 2nd Eur. Workshop Syst. Secur.*, Nuremburg, Germany, 2009, pp. 1–8.

[9] S. Checkoway, L. Davi, A. Dmitrienko, A. R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proc. 17th ACM Conf. Comput. Commun. Secur.*, Chicago, IL, USA, 2010, pp. 559–572.

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

[11] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *Proc. 24th Conf. USENIX Secur. Symp.*, Washington, DC, USA, 2015, pp. 161–176.

[12] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *Proc. 23rd Conf. USENIX Secur. Symp.*, San Diego, CA, USA, 2014, pp. 401–416.

[13] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Proc. IEEE Symp Secur. Privacy*, San Jose, CA, USA, May 2014, pp. 575–589.

[14] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *Proc. IEEE Symp. Secur. Privacy*, San Jose, CA, USA, May 2015, pp. 745–762.

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

[16] N. Carlini and D. Wagner, "ROP is still dangerous: Breaking modern defenses," in *Proc. 23th Conf. USENIX Secur. Symp.*, San Diego, CA, USA, 2014, pp. 385–399.

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

[18] A. J. Mashtizadeh, A. Bittau, D. Mazières, and D. Boneh, "CCFI: Cryptographically enforced control flow integrity," in *Proc. 22th ACM Conf. Comput. Commun. Secur.*, Denver, CO, USA, 2015, pp. 941–951.

[19] M. Theodorides and D. Wagner, "Breaking active-set backward-edge CFI," in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust*, McLean, VA, USA, May 2017, pp. 85–89.

[20] C. Zhang *et al.*, "Practical control flow integrity and randomization for binary executables," in *Proc. IEEE Symp. Secur. Privacy*, Berkeley, CA, USA, 2013, pp. 559–573.

[21] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *Proc. 22rd Conf. USENIX Secur. Symp.*, Washington, DC, USA, 2013, pp. 337–352.

[22] Y.-J. Park, Z. Zhang, and G. Lee, "Microarchitectural protection against stack-based buffer overflow attacks," *IEEE Micro*, vol. 26, no. 4, pp. 62–71, Jul./Aug. 2006, doi: 10.1109/MM.2006.76.

[23] Y. Lee and G. Lee, "Detecting code reuse attacks with branch prediction," *IEEE Comput.*, vol. 51, no. 4, pp. 40–47, Apr. 2018, doi: 10.1109/MC.2018.2141035.

[24] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis, "HCFI: Hardware-enforced control-flow integrity," in *Proc. 6th ACM Conf. Data Appl. Secur. Privacy*, New Orleans, LA, USA, 2016, pp. 38–49, doi: 10.1145/2857705.2857722.

[25] Intel. 2017. *Intel Control-Flow Enforcement Technology Preview*. Accessed: Jun. 1, 2018. [Online]. Available: https://software.intel.com/sites/default/files/managed/4d/2a/

[26] Microsoft. MSDN *Control Flow Guard*. Accessed: Dec. 21, 2018. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx

[27] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *Proc. 11th USENIX Conf. Oper. Syst. Design Implement.*, Broomfield, CO, USA, 2014, pp. 147–163.

[28] Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen and M. Franz, "Opaque control-flow integrity," in *Proc. Netw. Distrib. System Secur. Symp.*, San Diego, CA, USA, 2015, pp. 1–15.

[29] B. Niu and G. Tan, "RockJIT: Securing just-in-time compilation using modular control-flow integrity," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Scottsdale, AZ, USA, 2014, pp. 1317–1328.

[30] C. Tice *et al.*, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *Proc. 23rd Conf. USENIX Secur. Symp.*, San Diego, CA, USA, 2014, pp. 941–955.

[31] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kGuard: Lightweight kernel protection against return-to-user attacks," in *Proc. 21st USENIX Conf. Secur. Symp.*, Bellevue, WA, USA, 2012, pp. 1–16.

[32] J. Anderson, "Computer security technology planning study," Air Force Electron. Syst. Division, Tech. Rep. ESD-TR-73-51, 1972.

[33] D. Ahn and G. Lee, "A memory-access validation scheme against payload injection attacks," *IEEE Trans. Dependable Secure Comput.*, vol. 12, no. 4, pp. 387–399, Jul./Aug. 2015, doi: 10.1109/TDSC.2014.2355844.

[34] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *Proc. 37th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Portland, OR, USA, Dec. 2004, pp. 221–232.

[35] H. Kannan, M. Dalton, and C. Kozyrakis, "Decoupling dynamic information flow tracking with a dedicated coprocessor," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Lisbon, Portugal, Jun./Jul. 2009, pp. 105–114.

[36] F. Qin, C. Wang, Z. Li, H.-S. Kim, Y. Zhou, and Y. Wu, "LIFT: A low-overhead practical information flow tracking system for detecting security attacks," in *Proc. 39th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2006, pp. 135–148.

[37] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proc. 11th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, Boston, MA, USA, 2004, pp. 85–96.

[38] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel MPX explained: A cross-layer analysis of the intel MPX system stack," in *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 2, no. 2, pp. 28-1–28-30, 2018.

[39] G. Lee and C. Pyo, "Encoding function pointers and memory arrangement checking against buffer overflow attack," in *Proc. 4th Int. Conf. Inf. Commun. Secur.*, 2002, pp. 25–36.

[40] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Accessed: Dec. 21, 2018. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf

[41] T. H. Y. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Proc. 10th ACM Symp. Inf., Comput. Commun. Secur.*, Singapore, 2015, pp. 555–566.

[42] O. Aleph, "Smashing the stack for fun and profit," *Phrack Mag.*, vol. 7, no. 49, pp. 14–16, 1996.

[43] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Pointguard TM: Protecting pointers from buffer overflow vulnerabilities," in *Proc. 12th Conf. USENIX Secur. Symp.*, Washington, DC, USA, 2003, pp. 91–104.

[44] X. Chen, H. Bos, and C. Giuffrida, "CodeArmor: Virtualizing the code space to counter disclosure attacks," in *Proc. IEEE Eur. Symp. Secur. Privacy*, Paris, France, Apr. 2017, pp. 514–529.

[45] S. Crane *et al.*, "Readactor: Practical code randomization resilient to memory disclosure," in *Proc. IEEE Symp. Secur. Privacy*, San Jose, CA, USA, May 2015, pp. 763–780.

[46] D. Grawrock, *Dynamics of a Trusted Platform: A Building Block Approach*, 1st ed. Santa Clara, CA, USA: Intel, 2009.

[47] *Trusted Platform Module Library: Part 1: Architecture, Family 2.0, Level 00, 01.16 ed*, Trusted Comput. Group, Beaverton, OR, USA, 2014.

[48] A. Gonzalez, M. Valero, N. Topham, and J. M. Parcerisa, "Eliminating cache conflict misses through XOR-based placement functions," in *Proc. 11th Int. Conf. Supercomput.*, Vienna, Austria, 1997, pp. 76–83.

[49] P. Hamalainen, T. Alho, M. Hannikainen, and T. D. Hamalainen, "Design and implementation of low-area and low-power AES encryption hardware core," in *Proc. 9th EUROMICRO Conf. Digit. Syst. Design*, Dubrovnik, Croatia, 2006, pp. 577–583.

[50] L. John Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. San Mateo, CA, USA: Morgan Kaufmann, 2011.

[51] Y. Ahn, Y. Lee, J.-Y. Choi, G. Lee, and D. Ahn, "Monitoring translation lookahead buffers to detect code injection attacks," *Computer*, vol. 47, no. 7, pp. 66–72, Jul. 2014.

[52] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *Proc. 14th Int. Conf. Recent Adv. Intrusion Detection*, Menlo Park, CA, USA, 2011, pp. 121–141.

[53] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "RIPE: Runtime intrusion prevention evaluator," in *Proc. 27th Annu. Comput. Secur. Appl. Conf.*, Orlando, FL, USA, 2011, pp. 41–50.

[54] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," *ACM SIGARCH Comput. Archit. News*, vol. 25, no. 3, pp. 13–25, Jun. 1997.

[55] M. Conti *et al.*, "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Denver, CO, USA, 2015, pp. 952–963.

[56] I. Evans *et al.*, "Missing the point(er): On the effectiveness of code pointer integrity," in *Proc. IEEE Symp. Secur. Privacy*, San Jose, CA, USA, May 2015, pp. 781–796.

[57] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP exploit mitigation using indirect branch tracing," in *Proc. 22nd USENIX Conf. Secur. Symp.*, 2013, pp. 447–462.

[58] Y. Jin, D. Sullivan, O. Arias, A.-R. Sadeghi, and L. Davi, "Hardware control flow integrity," in *Proc. Continuing Arms Race, Code-Reuse Attacks Defenses*, 2018, pp. 181–210.

[59] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, and D. Song, "Poster: Getting the point(er): On the feasibility of attacks on code-pointer integrity," in *Proc. 36th IEEE Symp. Secur. Privacy*, 2015, pp. 1–2. [Online]. Available: http://www.ieee-security.org/TC/SP2015/posters/paper_48.pdf

[60] *Pointer Authentication on ARMv8.3 Design and Analysis of the New Software Security Instruction*, Qualcomm Product Secur., San Diego, CA, USA, 2017.

YONGSUK LEE received the M.S. degree in computer science and engineering from Korea University, Seoul, South Korea, in 2011, where he is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering. His research interests include computer architecture, trusted computing, and systems security.

GYUNGHO LEE received the Ph.D. degree in computer science from the University of Illinois at Urbana–Champaign, in 1986. He is currently a Professor with the Department of Computer Science and Engineering, Korea University. His research and teaching interests include computer architecture, especifically in the areas of microprocessor architecture, systems security, and code optimization. He is a Fellow of the American Association for the Advancement of Science.