Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity

Vedvyas Shanbhogue
Intel Corporation
Austin, TX, USA
vedvyas.shanbhogue@intel.com

Deepak Gupta
Intel Corporation
Hillsboro, OR, USA
deepak1.k.gupta@intel.com

Ravi Sahita
Intel Corporation
Hillsboro, OR, USA
ravi.sahita@intel.com

ABSTRACT

Intel has developed Control-flow Enforcement Technology (CET) [27] that provides CPU instruction set architecture (ISA) capabilities to defend against Return-oriented Programming (ROP) and call/jmp-oriented programming (COP/JOP) style control-flow subversion attacks. This attack methodology uses code sequences in authorized modules with at least one instruction in the sequence being a control transfer instruction that depends on attacker-controlled data either in the return stack or in a register/memory for the target address. Attackers stitch these sequences together by diverting the control flow instruction (e.g. RET, CALL, JMP) from its original target address to a new target (via modification in the data stack or in the register or memory used by these instructions). This paper describes CET security objectives, threat model and various architectural design choices to ensure that the design meets the security objectives. We conclude the paper with performance data and related work in this domain.

CCS CONCEPTS

• Security and privacy → Security in hardware → Hardware security implementation; • Security and privacy → Intrusion/anomaly detection and malware mitigation.

KEYWORDS

Control-flow integrity, control flow subversion attacks, shadow stack, ROP, JOP, COP.

ACM Reference format:

Vedvyas Shanbhogue, Deepak Gupta and Ravi Sahita. 2019. Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *Proceedings of the 8th International workshop on Hardware and Architectural Support for Security and Privacy (HASP' 19), June 23, 2019, Phoenix, AZ, USA. ACM, New York, NY, USA, 11 pages.*
https://doi.org/10.1145/3337167.3337175

HASP'19, June 23, 2019, Phoenix, AZ, USA

978-1-4503-7226-8/19/06...$15.00
https://doi.org/10.1145/3337167.3337175

1. INTRODUCTION

Introduction of security features like No-Execute [1] to prevent data execution (DEP), supervisory mode execute prevention (SMEP) [1], supervisory mode access prevention (SMAP) [1], etc. have pushed the state of art of exploitation of software to code-reuse attacks based exploitation techniques like Return-Oriented Programming (ROP) [2], Jump-Oriented Programming (JOP) [3] and Call-oriented programming (COP) [4]. Defending against such exploits requires prevention of malicious attempts to invoke control flows that are not part of the program's control-flow graph. Control- flow Integrity (CFI) [3, 16] proposed that a program should only execute control flows that are programmed in by the programmer. Specifically, CFI requires that an indirect branch should only target instructions in the program that have been designated as targets of indirect branches. Consequently, a forward branch through an indirect call or jump should only transfer control to valid targets for calls and jumps in the program and a return instruction should only transfer control to the call site that initiated the call to the procedure being returned from.

Intel Control-flow Enforcement Technology (CET) is a CPU instruction set extension to implement CFI and defend against ROP/JOP style control-flow subversion attacks. It adds the following capabilities to the Intel instruction set architecture:

- Shadow Stack – return address protection to prevent Return Oriented Programming
- Indirect branch tracking – free branch protection to enforce security properties on Jump/Call Oriented Programming.

CET shadow stack is a second stack used exclusively during control transfer operations to store a copy of the return address pointer. The shadow stack is write protected using a new extension to page permissions to prevent software-originated unintended writes to the shadow stack. The CALL instruction pushes a copy of the return address on the shadow stack in addition to the legacy behavior of pushing the return address on the data stack. The RET instruction is modified to pop the return address from both stacks and if the two return addresses do not match, causes an exception and thus prevents and reports attempts to modify the return address maliciously (or in error).

CET Indirect branch tracking introduces a new instruction ENDBRANCH that is used to mark valid code targets for indirect

call and jumps in the program. If an indirect call or jump targets an instruction other than ENDBRANCH, the processor generates an exception and thus prevents and reports attempts to redirect control flow to unintended code targets in the program. The approach taken here is a coarse-grained forward branch enforcement that can be further restricted using software instrumentation and compiler techniques.

CET introduces a new fault-type exception – control protection (#CP) – to notify privileged software when control flow violations are detected. The #CP exception also reports an error code to notify the reason to the exception handler.

1.1. Adversary model and CET objectives

We enumerate the following threats (written as adversary capabilities) to define the scope of mitigations we aim to provide with CET:

- T1: Can find software vulnerabilities that allows the adversary to read and write anywhere in software-accessible memory.
- T2: Can discover the complete layout of the address space (e.g. where stacks, heaps, and images are mapped).
- T3: Can repeat memory reads and writes at will.
- T4: Can produce stimulus that make code take different paths and observe the state of the program including the stack state in these paths.
- T5: Can send data to a "computation server" to compute payload needed for subsequent reads/writes.
- T6: Can perform control transfers to existing code to execute (code re-use) in order to change state of processor registers.

Following assumptions were made about constraints on the adversary:

- R1: Cannot add new code without verification and all existing code is read-only and not modifiable (e.g. W $\oplus$ X policy).

While defining the technology, we enumerate following design goals and constraints:

- D1: Must provide protection mechanism for new architectural assets (i.e. new hardware register, memory) against code re-use and memory safety errors.
- D2: Must be applicable to CPU-enforced privilege levels (user/supervisor)
- D3: Must be applicable to CPU modes used by commodity software, such as 32/64-bit, hypervisor, system management mode, enclaves, etc.
- D4: Must ensure no loss of control-flow enforcement protection occurs at transitions points between modes and context switches.
- D5: Must have minimal overheads on performance, memory usage and code size growth.
- D6: Avoid embedding programming language specific constructs in the ISA
- D7: Provide only the necessary (minimal) capabilities for software to be able to impose instruction alignment and return address protection for language and usage-specific policies.
- D8: Preserve the stack and function call ABI
- D9: Must not place any restrictions on common software constructs like tail calls, co-routines, etc.

In section 2, we describe the security model of the shadow stack capability and the new instructions introduced to manage the shadow stack. In section 3, we describe the security model of the indirect branch tracking capability. In section 4, we discuss the speculation-safe hardening properties of CET. In later sections, we present the performance evaluation, security metrics, and discuss related work, with conclusions.

2. CET SHADOW STACK

A shadow stack is a second stack used exclusively for control transfer operations to store and to retrieve the return address pointers. The shadow stack is distinct from the ordinary data stack, and holds no data.

2.1. SSP register encoding and type safety

CET defines a shadow stack pointer (*SSP*) register that contains the linear address of the top of the current shadow stack. Since *SSP* is a new architectural register and must be protected (per design goal D_1) against an adversary that can use code re-use capability (per threat T_6) to change it. Thus opcode encoding does not allow *SSP* register to be directly encoded as a source, destination or memory operand in instructions other than shadow stack management instructions. This reduces potential useful code re-use gadgets in program memory and helps mitigate techniques like stack pivoting used to subvert control flow.

CET enforces type safety by enforcing that values that are generated as part of CET instructions or implicit ISA flows (e.g. privilege transitions) are consumed only by complementary CET instructions or complimentary ISA flows. Examples of such enforcement are cited in section 2.3, 2.4, 2.7 and 2.8.

2.2. Shadow stack for each privilege and exception delivery

One of the design goals of CET has been to provide support for control flow enforcement at each privilege level of processor (D_2, D_3 and D_4). The x86 segmentation protection mechanism supports four privilege levels, numbered from 0 to 3. The greater numbers mean lesser privileges. Most operating systems running on x86 architecture only use two privilege levels where the operating system kernel and its services execute at privilege level 0 and the applications execute in user mode at privilege level 3. CET considers programs executing at privilege level 0, 1 and 2 to be equally privileged and supervisory programs and thus when CET is enabled for supervisor mode it is active at privilege levels lower than 3. In the x86 architecture, call gates facilitate controlled transfer of program control between different privilege levels and when a call gate is used to transfer control to a more privileged level, the processor automatically switches to the data stack for the

destination privilege level using the pointers to the privilege level 0, 1 and 2 stacks stored in the Task-State Segment (TSS) data structure of the current running task.

Intel 64 bit architecture supports an interrupt stack table (IST) to provide a method for specific interrupts (such as NMI, double-fault, and machine check) to always execute with a known good stack. The IST mechanism provides up to seven IST pointers that can be selected using a 3-bit IST index from the interrupt-gate-descriptor (IDT), which when not 0, is used to select the pointer to the data stack to switch to when delivering those specific interrupts.

CET extends this automatic stack switching to also switch the *SSP* when CET is enabled in supervisor mode. The *SSP* for the privilege level 0, 1 or 2 is obtained from one of following new model specific registers (MSRs) depending on the target privilege level:

- *IA32_PL2_SSP*, if transitioning to ring 2
- *IA32_PL1_SSP*, if transitioning to ring 1
- *IA32_PL0_SSP*, if transitioning to ring 0

The collection of these three MSRs is referred to as *IA32_PL0/1/2_MSR* in following sections. When a stack switch occurs through IST mechanism and CET is enabled at privilege level of the interrupt handler, a new *SSP* is selected using the IST index from a table of *SSPs* pointed to by the *IA32_INTERRUPT_SSP_TABLE_ADDR* MSR defined by CET. CET treats the OS kernel as being in the trust boundary of the user mode programs. Thus CET does not attempt to restrict any control transfers initiated by the OS to user mode programs. In x86 architecture the ring 0 is the most privileged ring and can invoke privileged instructions. Rings 1 and 2 cannot invoke privileged instructions however they have same memory access privileges as ring 0. Since they have equal privilege for memory accesses the CET architecture treats them as being in the "same privilege class". The architecture therefore enforces the control transfers between these privileged rings through shadow stacks and indirect branch tracking when CET is enabled for supervisor mode.

2.3. CALL operation

In the Intel 64 and IA-32 architecture (x86 architecture), the near CALL instruction allows control transfers to local procedures within the current code segment. The far call allows control transfer to procedures in a different code segment and can be used to access operating system procedures. A far call also allows transitioning to a 32-bit code segment to allow legacy (32 bit) binary to co-exist with 64-bit binary in 64-bit mode. The near CALL instruction pushes the Return Instruction Pointer on the data stack. When CET is enabled, the near CALL additionally pushes the Return Instruction Pointer on the shadow stack. The far CALL instruction, or an interrupt or exception flow, pushes the Code Segment (*CS*) selector and the Return Instruction Pointer on the data stack of the called procedure. When CET is enabled, the far CALL additionally pushes the *CS*, the Linear Instruction Pointer (LIP is computed by the CPU as the base of the Code Segment descriptor plus the logical address value of the Return Instruction Pointer) and the *SSP* at the time of initiating the far transfer on the shadow stack of the called procedure. The pushes on the shadow stack are always performed as 8 byte pushes. Pushing the previous *SSP* on the shadow stack prevents a far CALL from being paired with a near return as the addresses on the shadow stack are non-executable (enforcing type safety). For these far transfers, the choice to store the Linear Instruction Pointer on the shadow stack instead of the logical address Return Instruction Pointer was made to ensure detection of conditions where the base address of the Code Segment descriptor may have been changed between a far CALL and the matching far RET. When the far transfer to higher privilege level originates in user mode i.e. privilege level 3, the processor switches both - the data and shadow stack to that of the new privilege level. The user mode shadow stack pointer is saved to into a new MSR called *IA32_PL3_SSP*. The processor does not push any return addresses on the new supervisor shadow stack. This follows the trust model where the user level programs have the supervisor in their trust boundary, so the OS can change the address to subsequently return to or the *SSP* of the user space program. If the far transfer is to 32-bit mode the processor causes a general protection fault if the *SSP* is not in the lower 4 GB of the linear address space. By faulting and not implicitly truncating the *SSP* to 32 bits the CET architecture avoids any unintended or malicious aliasing to another shadow stack. A far transfer in the x86 architecture may or may not involve a privilege change. When there is a privilege change it is associated with a stack switch and the processor requires the new stack to be 8 byte aligned. When there is no privilege change, the processor prior to pushing the return address information on the shadow stack aligns it to the next 8 byte boundary and zeroes out any alignment hole created to avoid unknown data from appearing on the shadow stack. The section 2.7 discusses stack switching on privilege changes. Alignment of shadow stack to 8 byte boundaries and saving the return address information in 8 byte elements avoids type confusion when transitioning between 64-bit and 32-bit modes.

2.4. RET/IRET operation

The RET instruction allows near and far returns to match the near and far versions of the CALL instruction. The IRET instruction returns program control from an interrupt or exception handler to the interrupted procedure. When CET is enabled, the near RET instruction pops the return address from both the shadow stack and the data stack. If the return address values popped from the two stacks are not equal then the processor causes a control protection exception (#CP) with error code "NEAR-RET". When CET is enabled, the far RET and IRET (except when transition to user space) pops the return-SSP, LIP and the *CS* from the shadow stack. If the *CS* and LIP do not match the return address as determined by popping the *CS* and Return Instruction Pointer from the data stack, the processor causes a #CP exception with error code "FAR-RET/IRET". The error code provided with the resulting #CP exception helps identify the type of call frame that caused the fault. If the return was successful then the SSP is set to the return-SSP. If a RET or IRET instruction is used to return to user space i.e. to privilege level 3, the processor establishes the *SSP* for the user mode using contents of the *IA32_PL3_SSP* MSR. No return address verification is done. The OS is allowed to switch shadow

stacks and return to any address in user mode. This follows the trust model where the user level programs have the OS in their trust boundary.

2.5. Write protecting the shadow stack

Adversary model for CET assumes that attacker has capabilities to perform read and writes at will innumerable number of times (T_1 and T_3) using some memory safety bug. CET further assumes that attacker has computational capabilities (T_5) to make intelligent decisions on exercising memory writes. CET addresses attempts to corrupt the shadow stack through malicious writes by exploiting vulnerabilities like buffer overflows, use-after-free, etc. by extending the page tables such that pages mapped as shadow stack pages are not writeable by software use of memory store instructions. The CPU enforces that software writes to the shadow stack occur only in the context of a CALL instruction and new CET ISA for shadow stack management invoked by software. Control transfer instructions/flows and shadow stack management instructions perform loads/stores to the shadow stack. Such load/stores from control transfer instructions and shadow stack management instructions are termed as *shadow_stack_load* and *shadow_stack_store* (or collectively as *shadow_stack_accesses*; enforcing type safety in accesses) to distinguish them from load/store performed by other instructions like *MOV, XSAVES,* etc that are performed by software.

2.5.1. x86 paging protections. CET extends x86 paging architecture to allow pages to be mapped in linear address space as shadow stack pages. A page mapped as not-writeable-but-dirty i.e. W=0, D=1 is treated by the CPU as a shadow stack page. By using this software-unused encoding of writeable and dirty attributes we avoid introducing new paging attribute bits. The chosen page control bit encodings for shadow stack mappings also ensure that shadow stack pages are not writeable and hence naturally protected from unintended or malicious software writes. CET further enforces that *shadow_stack_accesses* must be to shadow stack regions by causing a page fault if the shadow stack addresses are not mapped to shadow stack pages. This helps detect any attempts to pivot the *SSP* to writeable memory or to overflow/underflow the *SSP* beyond the bounds of the current active shadow stack. CET also enforces that *shadow_stack_accesses* from supervisor mode must be to shadow stacks mapped as supervisor pages i.e. using a user shadow stack in supervisor mode is disallowed by the CPU. Lastly, CET enforces that paging write protection (*CR0.WP*) cannot be disabled when CET is enabled to prevent unintended writes to shadow stack by disabling paging write protection.

2.5.2. Second level page table protection. OS/supervisor shadow stacks can be write-protected using the extended page tables (EPT) established by a virtual machine manager (VMM) by using a new EPT attribute "*supervisor shadow stack*" to designate the (guest) physical pages used by the OS for shadow stacks as supervisor shadow stack pages. When this functionality is enabled *shadow_stack_accesses* to supervisor shadow stacks are only allowed to (guest) physical pages mapped as "*supervisor shadow stack pages*" under EPTs by the VMM. Shadow stack writes to pages mapped as "*supervisor shadow stack*" pages in EPT do not require the EPT to provide write permission. This allows a VMM to write protect OS/guest supervisor shadow stack pages from CPU initiated stores as well as device DMA accesses (when the EPT is shared by the IOMMU).

2.6. Shadow stack tokens

As stated earlier direct manipulation of *SSP* register using move instruction is not supported by CET. To allow multiple execution context within application programs and in operating system, CET provide mechanisms for saving and restoring shadow stack pointer (i.e. *SSP* register) to and from memory without compromising design goals and security properties. In order to establish a new shadow stack in *SSP* register while continuing to provide guarantees against memory safety bugs, the following properties are enforced by the CPU:

- **Secure storage**: Memory storing shadow stack pointer must be protected against memory safety errors (T_1 and T_3).
- **Immutability**: Even if an adversary is able to obtain a write primitive to this secure storage, they shouldn't be able to write any value of their choice. If an adversary changes the pointer value, using that value should result in a processor fault and notify the operating system of that violation.
- **One time use**: Pointer stored in memory to establish new shadow stack can be used once and further usage should result into a fault. This prevents any possibility of two execution contexts (e.g. two program threads) establishing same shadow stack.

As described earlier, CET shadow stack memory access-control satisfies the property of read-only permissions while still allowing stores using *shadow_stack_store* primitive in selected architectural flows. This access-control model allows CET to use shadow stack memory itself as secure storage for shadow stack pointers. To enforce immutability, CET enforces that shadow stack pointer itself should be a function of the address on which it is stored. As part of the save sequence, shadow stack is first aligned on 8 byte boundary and then shadow stack pointer is saved on it. To further harden type checking and one time usage property, CET uses lower 2 bits of the stored shadow stack value for keeping extra information to track state of pointer (see section 2.5 and 2.6 for usage of lower 2 bits). This results in saving the shadow stack pointer in a specific format - collectively these stored shadow stack pointer formats are called as '*shadow stack tokens*'.
Section 2.7 and 2.8 describes different form of shadow stack tokens and their usage in shadow stack switching mechanisms.

2.7. Processor initiated stack switch

The OS is required to program the *IA32_PL0/1/2_MSRs* to point to the bottom of the supervisor shadow stacks of the current task and to ensure that no two logical processors have the MSRs pointing to the same shadow stack. However, the operating system may context switch *IA32_PL0/1/2_SSP* MSRs and save them in memory where they are susceptible to being modified (adversary capabilities T_1 and T_3). Likewise shadow stacks pointers in

memory referenced by the *IA32_INTERRUPT_SSP_TABLE* MSR may be modified. The modifications may be an attempt to point these MSRs to an address beyond the bottom of the shadow stack and thereby create bad return addresses on the shadow stack. The modifications may also be an attempt to point them to the shadow stack that is active on another logical processor such that return addresses pushed by one logical processor are consumed by another (threats T_2 and T_4). To overcome the issues mentioned above, CET implements a token check mechanism to detect such modifications and ensure that the shadow stack starts out empty and that the same shadow stack cannot be activated simultaneously on two logical processors. Supervisor shadow stacks must be constructed with a token at the bottom of the shadow stack called the "*supervisor shadow stack token*". This token is a 64-bit value formatted as follows:

- Bit 63:3 – 8 byte aligned linear address of the token itself.
- Bit 2:1 – reserved. Must be zero.
- Bit 0 – Busy bit. The busy bit when 1 indicates that the corresponding shadow stack is active on some logical processor, thus enforcing the *one time use* property.

To switch stack on transition to higher privilege level, the processor performs the following steps:

- Loads the "supervisor shadow stack token" from the address in IA32_PL0/1/2_SSP.
- Verifies the busy bit and all reserved bits in the token is 0. This prevents a given shadow stack from being made active on two logical processors simultaneously.
- Verifies that the address programmed in the MSR matches the address in the "supervisor shadow stack token".
- If the checks 2 and 3 are successful then the busy bit in the token is set to 1 and the processor switches the *SSP* to the value specified in the *IA32_PL0/1/2_SSP* MSRs.

The load in step 1 and store in step 4 are done as shadow stack accesses to ensure that the address points to a page mapped as a shadow stack page. Step 3 ensures that the address in the MSR is pointing to the bottom of the shadow stack i.e. an empty shadow stack. This check relies on the property that an 8 byte aligned location on the shadow stack having a value that is the address of that 8 byte location never occurs on a shadow stack except when created by the OS by storing this "supervisor shadow stack token". The steps 1 through 4 are done as an atomic transaction to avoid TOCTOU issues. If the checks 2 or 3 fail then the busy bit is not set and a general protection (#GP) exception is caused.
Figure 1 illustrates this token check to make the shadow stack active. In this example, the IA32_PL0_SSP MSR points to address 0xFF8. The token check loads the 8 byte token at address 0xFF8 and verifies that busy bit is 0 and that the address in the token matches the address in the MSR. As the token check succeeds, the busy bit in the token is set to 1 and the SSP is now updated to point to 0xFF8 making this shadow stack active, Next push on this shadow stack saves at the address 0xFF0.

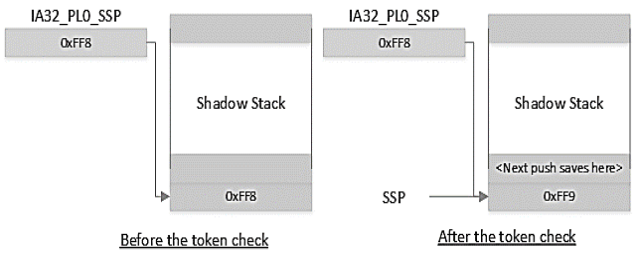

Figure 1: Processor initiated shadow stack switching

When the processor returns to a lower privilege level it switches to the shadow stack of the lower privilege level. The current active shadow stack is made free by the far RET/IRET instructions by performing following steps:

- Loads the "supervisor shadow stack token" from the address in SSP
- Checks if the busy bit is 1 and all reserved bits are 0
- Checks if the address programmed in "*supervisor shadow stack token*" matches *SSP*
- If the checks 2 and 3 are successful then clears the busy bit in the token

The load in step 1 and store in step 4 are done as a shadow stack accesses to ensure that the address points to a page mapped as a shadow stack page. The steps 1 through 4 are done as an atomic transaction to avoid TOCTOU issues. The checks 2 and 3 when successful indicate that the *SSP* is at the bottom of the shadow stack i.e. there are no valid call frames on the shadow stack. If there are valid call frames on the shadow stack then the shadow stack remains busy.

2.8. Shadow stack management instructions

Shadow stack management instructions provide controlled and safe ways to manipulate *SSP* to implement common software constructs like stack unwinding, thread switching, etc. The following descriptions group the instructions by their typical usage.

2.8.1. Stack unwinding.

Like the data stack, the shadow stack grows from high to low address and thus unwinding the shadow stack involves incrementing the *SSP*. To support unwinding the shadow stack, the RDSSP instruction may be used to read the contents of the *SSP* as needed in the program – for example by the setjmp function. To unwind to the snapshot, the INCSSP instruction can be used – for example by the longjmp function – to unwind the current *SSP* to the value recorded at the previous snapshot. Since the shadow stack only holds return addresses the number of bytes to unwind is usually small. For example, to unwind from a call depth of 100 functions the INCSSP instruction would be invoked with operand 100. Here are summary descriptions of INCSSP and RDSSP.

- INCSSP – increment the *SSP* by 'n * operand size of shadow stack', where n is an 8 bit operand. The instruction does a 'pop-and-discard' on the first and last frame in the range. The

'pop-and-discard' and the restriction of 'n' to be at most 255 prevents using INCSSP to roll off one shadow stack into another by skipping over an intervening guard page.

- RDSSP – instruction used to read the contents of the *SSP* register into a GPR.

2.8.2. Software initiated stack switching.

Stack switching is required when the OS scheduler schedules a new task and switches from the current task stack to the next task stack. Similar thread switching may be performed in user space to support user space thread schedulers and co-routines. The RSTORSSP and SAVPREVSSP instructions are provided to perform the stack switching in a controlled manner. When the scheduler switches away from an active shadow stack and later switches back to that shadow stack, CET ensures that the SSP established is same as at the time of switching away.

The shadow stack switching sequence is a two-step process; execute RSTORSSP to verify and switch to the new shadow stack, then execute SAVEPREVSSP to record a restore point on the old shadow stack. A restore point is recorded in the form of saving a "Shadow Stack Restore token" at the top of the old shadow stack. Alternatively, the OS can create the restore point when setting up a new shadow stack.

CET enforces there can be only one restore point valid on the shadow stack (one time use property of token) and if a restore point is valid on the shadow stack then that shadow stack is not active. CET further enforces that when a shadow stack is activated the SSP is restored to the last value of SSP when that shadow stack was previously active. CET further enforces that the restore point that records the last active SSP is protected from unintended writes. Lastly CET enforces that a restore point created in 32-bit or 64-bit mode can be restored only in the matching mode (enforcing type safety).

The RSTORSSP instruction verifies a "*Shadow Stack Restore*" token referenced by the memory operand of this instruction to determine a valid restore point on the new shadow stack. This "*Shadow stack restore token"* is a 64-bit value formatted as follows:

- Bit 63:2 – 4-byte aligned *SSP* for which this restore point was created. This *SSP* must be at an address that is 8 or 12 byte above the address where this token itself is found. The RSTORSSP instruction verifies this property.
- Bit 1 – reserved. Must be zero
- Bit 0 – Mode bit. If 0 then this shadow stack restore token can be used by RSTORSSP instruction in 32-bit mode. If 1 then this shadow stack restore token can be used by the RSTORSSP instruction in 64-bit mode.

The "*shadow stack restore token*" is created by the SAVEPREVSSP instruction (described later). The RSTORSSP instruction verifies the "*shadow stack restore token"* and switches the *SSP* as follows:

- Verifies that the memory operand of the instruction is an 8 byte aligned address.
- Loads the "*shadow stack restore token*" from the address in specified as the memory operand.
- Verifies reserved bits in the token are 0.
- Verifies that the *SSP* recorded in bits 63:2 of the token is 8 bytes or 12 bytes higher than the address of the token.
- Verifies that if the current mode of the machine is 64-bit then the bit 0 is 1 else it must be 0.
- If the checks 2, 3 and 4 succeed replaces the "*shadow stack restore token*" on shadow stack with a "*previous SSP token*" which records the *SSP* active when the RSTORSSP instruction was invoked
- Switches *SSP* to the value address of the token such that now the "*previous SSP token"* is at the top of the stack.

The load in step 2 and store in step 6 is done as *shadow_stack_accesses* to ensure that the address points to a page mapped as a shadow stack page. The steps 2 through 6 are done as an atomic transaction to avoid TOCTOU issues. The property verified by step 1 and 4 ensures the token is a valid token as the SAVEPREVSSP pushes the "*shadow stack restore token*" after alignment to the next 8 byte boundary.

The "*previous SSP token"* records the *SSP* that was active at the time the RSTORSSP instruction was invoked and is formatted as follows:

- Bit 63:2 – previous *SSP* pointing to the top of old shadow stack i.e. the *SSP* active when RSTORSSP was invoked
- Bit 1 – set to 1 to indicate this is a "*previous SSP token"*
- Bit 0 – Mode bit. If 0 then this "*previous SSP token"* can be used by SAVEPREVSSP in 32-bit mode. If 1 then this "*previous SSP token"* can be used by SAVEPREVSSP in 64-bit mode.

This is illustrated by the following example (Figure 2):

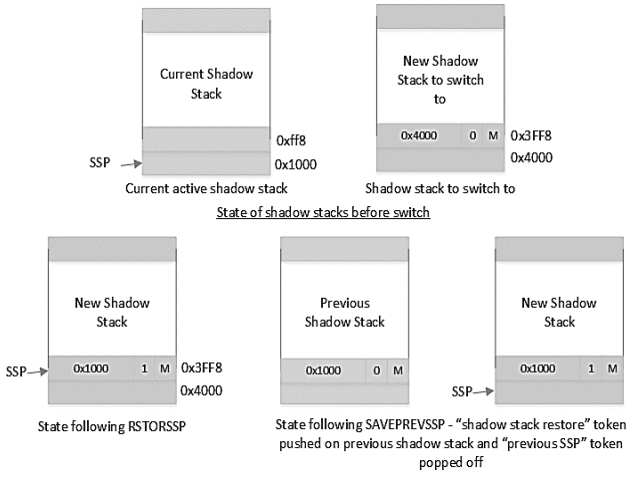

Figure 2: Software initiated shadow stack switching

In Figure 2, the SSP is currently pointing to the current active shadow stack and has a value of 0x1000. The target shadow stack has a "shadow stack restore token" at address 0x3FF8 and records

the new SSP to restore as 0x4000. The RSTORSSP instruction is invoked with the memory operand specifying the address of the "shadow stack restore token" as 0x3FF8. The RSTORSSP instruction verifies the mode of the machine against the mode M recorded in the token, verifies that the reserved bit at position 1 is 0 and that the address is in the token, 0x4000 in this example, is 8 or 12 bytes from the address of the token itself. Since these checks succeed, the SSP is now set to 0x3FF8 and the "shadow stack restore token" is replaced by the "previous SSP token". Subsequent to switching to the new shadow stack, a restore point can be created on the old shadow stack using SAVEPREVSSP instruction. The SAVEPREVSSP instruction uses the "previous SSP token" created by the RSTORSSP instruction to create a "shadow stack restore token" on the old shadow stack. The SAVEPREVSSP instruction does not take any operand but consumes a "previous SSP token" at the top of the shadow stack i.e. at the current SSP as follows:

- Verifies that the *SSP* 8 byte aligned address.
- Pops 8 bytes of "*previous SSP token"* from the shadow stack.
- Verifies that the bit 1 is set to 1.
- Verifies that if the current mode of the machine is 64-bit then the bit 0 is 1 else it must be 0.
- Aligns the previous *SSP* recorded in the "*previous SSP token"* to next 8 byte boundary and pushes a "*shadow stack restore token"* to the old shadow stack.

In this example, continuing with the state following the RSTORSSP, the SAVEPREVSSP instruction is invoked. The SAVEPREVSSP instruction finds the "*previous SSP token"* with the previous *SSP* recorded as 0x1000 and verifies it. Following this verification, the processor pushes a "*shadow stack restore token*" on the previous shadow stack at address 0xFF8. If a restore point on the old shadow stack is not needed, then the "*previous SSP token"* created by the RSTORSSP instruction on the current shadow stack can be popped using the INCSSP instruction.

2.8.3. Shadow stack fixup.

CET defines two instructions to enable software to fix-up the shadow stack contents if required. The first instruction WRUSS (Writes User Shadow Stacks) is a privileged instruction that can only be invoked by the OS. The OS may use WRUSS to, for example, create a bootstrap "*shadow stack restore token"* for a user mode thread or for actions like creating a call frame for signal delivery. A second instruction - WRSS – does a write to the Shadow Stack. WRSS is expected to be used only in specific instances to support a software construct (e.g. if the program implements an unusual control transfer using a push followed by a RET) for the short term before the software can be updated to not require such fix ups.

WRUSS can be used by the OS to write to user mode shadow stacks but not to supervisor mode shadow stacks. A page fault exception occurs if the address operand of the instruction does not reference a user mode shadow stack and prevent any attempts to maliciously modify the parameters of this instruction to point to a supervisor shadow stack.

WRSS can only write to user shadow stack when invoked in user mode and can only write to supervisor shadow stacks in supervisor mode. CET provides supervisory controls that allows an OS to enable this instruction for user and supervisor mode if the current user program or OS needs this function. For most applications it is expected that this instruction will be disabled and when disabled invocation of this instruction leads to an invalid opcode fault.

2.8.4. Fast system call support

The Intel 64 architecture defines SYSCALL and SYSENTER instructions to invoke an OS system call handler at privilege level 0 and switch to the OS data stack. When CET is enabled, these instructions save the user mode *SSP* to the *IA32_PL3_SSP* MSR and set the *SSP* to 0 (invalid). The OS returns to user mode following the system call handling using the SYSRET or SYSEXIT instructions. These instructions restore the user mode *SSP* from the *IA32_PL3_SSP* MSR. An OS that needs to make function calls from the system call handler must first activate a supervisor mode shadow stack because the *SSP* following SYSCALL/SYSENTER is 0 (invalid). CET provides the SETSSBSY instruction to activate the privilege level 0 shadow stack referenced by *IA32_PL0_SSP*. SETSSBSY instruction verifies the "*supervisor shadow stack token*" referenced by the *IA32_PL0_SSP* MSR and if verification is successful, makes the token busy and sets *SSP* to content of *IA32_PL0_SSP* MSR. If token verification fails, the processor will raise a #CP exception with error code "SETSSBSY". If a system call handler has activated a shadow stack, it must use CLRSSBSY instruction to deactivate this shadow stack. The CLRSSBSY instruction takes a memory operand that points to the "*supervisor shadow stack token*" of the stack to deactivate and if the token verifies, clears the busy bit in the token. If token verification fails, processor sets carry flag (CF) as error indicator. If the CF is set following CLRSSBSY instruction the OS should consider this a fatal error. The *SSP* following the CLRSSBSY instruction is set to 0 (invalid).

3. INDIRECT BRANCH TRACKING

To detect and prevent attempts to redirect control flow to unintended targets, CET added support for indirect branch tracking. Indirect branch tracking introduces new branch termination instructions: ENDBR32 for 32-bit programs and ENDBR64 for 64-bit programs. CET detects and prevents attempts to redirect control flow to unintended targets in the program by causing a #CP exception if the instruction at the target of an indirect call or jump targets is not a matching branch termination instruction.

The ENDBR32 and ENDBR64 opcodes are selected such that they are NOP instructions on Intel 64 processors that do not support CET. On processors supporting CET, these instructions are still NOP-like as they do not affect the execution state of the program, do not cause any additional register pressure and are minimally intrusive from power and performance perspective. This allows CET instrumented programs to execute on processors that do not support CET.

To track indirect call/jump for terminations, the processor implements two state machines; one for user mode and one for

supervisor mode. At reset the user and supervisor mode state machines are in IDLE state. When instructions other than indirect call/jump retire the state machine stays in the IDLE state. On an indirect call or jump instruction completion, the state machine transitions to WAIT_FOR_ENDBRANCH state. In this state, the state machine will cause a #CP fault with error code "ENDBRANCH" if the next instruction (i.e. the instruction at the target of the indirect call or jump) is not ENDBR64 in 64-bit mode or ENDBR32 in 32-bit mode. If the instruction is a proper ENDBRANCH, the state machine moves back to IDLE state.

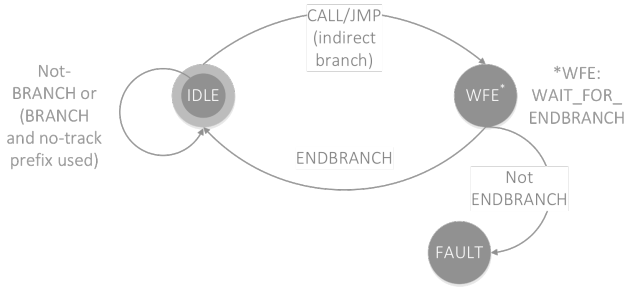

Figure 3: Software initiated shadow stack switching

The indirect branch tracking does not apply to relative call, relative jump or conditional jumps (Jcc) as these forms have their target encoded into the instruction and cannot be manipulated.

3.1. NO-TRACK prefixed jmp/call

For certain constructs such as switch-case for which the compiler has full control over the possible jump targets, (for example, because it ensured all possible case were validated), it is possible for the compiler to opt out of emitting an ENDBR32/ENDBR64 instruction at the target of these JMP by prefixing the JMP with a NO-TRACK prefix. Such prefixed indirect JMP do not require an ENDBRANCH instruction at their target and the state machine stays in IDLE state. Software may choose to restrict certain sensitive functions in program address space (e.g. exec, execv, etc.) to be called from only designated call sites in program. Such call sites can either use direct addresses or use the NO-TRACK prefixed call to these functions – for both the CPU will not require an ENDBRANCH instruction at the entry point of these sensitive functions. Not having an ENDBRANCH instruction at the entry point of these functions makes such functions strictly reachable via these designated call sites. Other indirect call sites trying to reach such sensitive functions will lead to #CP exception with error code "ENDBRANCH".

3.2. ENDBRANCH Opcode selection

The ENDBR32 opcode is F3 0F 1E FB and the ENDBR64 opcode is F3 0F 1E FA. The opcodes were selected to avoid cases where the last few bytes of an instruction and first few bytes of the next instruction could decode to an unintended ENDBRANCH instruction. A CET enabled compiler should not emit the 0F 1E FA or 0F 1E FB NOP in CET compiled code. If the last 2 bytes of an instruction are F3 0F then next two instructions must be "push DS (1Eh)" and "STI (FBh) or CLI (FAh)" to form an unintended ENDBRANCH instruction. If the last 3 bytes of an instruction are to be F3 0F 1E then the next instruction must be "STI (FBh) or CLI (FAh)" to form an unintended ENDBRANCH instruction. CLI/STI and PUSH DS are not typically compiler generated instructions. Push DS is not a valid instruction in 64-bit mode. If an instruction encodes an immediate that matches the ENDBR32/ENDBR64 instruction then the compiler/code generator should elide those using techniques like constant blinding.

4. SPECULATION SAFE PROPERTIES OF CET

CET enforces additional constraints to mitigate speculative execution side-channel attacks which leverage spectre [26] style branch target injection attacks.

Constraining execution at targets of RET

When CET shadow stack is enabled, instructions at the target of a RET instruction will not execute, even speculatively, if the RET addresses (both from data stack or shadow stack) are speculative-only or do not match, unless the target of the RET is also predicted (e.g. by some micro-architectural predictor due to a previous CALL before that address). A RET address would be speculative-only if it was modified by an older speculative-only store or was an older value than the most recent value stored to that address on logical processor.

Speculation constraint on missing ENDBRANCH

When the CET tracker is in WAIT_FOR_ENDBRANCH state, instruction execution will be limited or blocked, even speculatively, if the next instruction is not an ENDBRANCH. This means that when indirect branch tracking is enabled and not suppressed, the instructions at the target of a near indirect JMP/CALL without the no-track prefix will only speculatively execute if there is an ENDBRANCH at the target. Early implementations of CET may limit the speculative execution to a small number of instructions (less than 8, with no more than 5 loads) past a missing ENDBRANCH, while later implementations may completely block the speculative execution of instructions after a missing ENDBRANCH. This mechanism also limits or blocks speculation of the next sequential instructions after an indirect JMP/CALL; presuming the JMP/CALL puts the CET tracker into WAIT_FOR_ENDBRANCH state and the next sequential instruction is not an ENDBRANCH. Additional restrictions on speculative execution of code which has an ENDBRANCH present at the target of an indirect branch may be enforced via software instrumentation.

5. RESULTS AND DISCUSSION

5.1. Performance

The performance impact of shadow stacks was evaluated using a suite of microprocessor benchmark and application traces executed on a cycle accurate processor performance model. The CALL instruction model was updated to do the additional push on the

shadow stack and the RET instruction model updated to pop return address from shadow stack and compare against the return address from the data stack. The geometric mean of instruction-per-cycle (IPC) loss across workload traces is around 1.65%. The range of IPC loss ranged from 0.08% (HPC and multimedia kernels traces) and 2.71% (sysmark benchmark traces).

The performance impact of indirect branch tracking was evaluated by compiling C and C++ programs from the SPEC CPU 2006 C/C++ using a modified ICC compiler with CET support. As ENDBRANCH instructions execute as NOP on current shipping processors, (and will execute as NOP on future processors that support CET) these programs are executed with ENDBRANCH instrumentation on Core i7-6500U Processor to measure the performance impact - No perceptible slowdown was measured on average.

5.2. Security Metrics

The shadow stack restricts the flexibility available in creating ROP gadget chains by enforcing matching calls and returns and also enforcing a LIFO order on the returns. The shadow stack being write protected blocks attempts to inject return address frames on the shadow stack through arbitrary writes (thwarting adversary capabilities T_1 and T_3). The shadow stack pointer register not being directly writeable and paging checks that require the page referenced by call and returns to be mapped as a shadow stack page blocks attempts to pivot the shadow stack to writeable memory or to another shadow stack. Re-using old call frames on the shadow stack is not possible as the only instructions provided are to unwind the shadow stack through INCSSP.

With indirect branch tracking, COP/JOP gadgets are now limited to only calling or jumping to indirect callable functions, as only such functions would have an ENDBRANCH instruction. The exploit author will also need to precisely control the parameters needed to be passed to each of the functions in the chain. Likewise since entire functions are called and the use of "unintended gadgets" is blocked by the indirect branch tracking, the parameters to such functions will need to be carefully controlled to not have a return in the function path as a function that was jumped to but returned from will be fatal to the gadget chain due to the shadow stack enforcement. Requiring that JOP/COP chains to call or jump to the entry point of functions also constrains the attackers ability to retain control on the stack and registers as the x86 calling convention requires the called procedure to restore all of the registers and so they begin with pushing the registers on the stack and end by popping them off. Not being able to exploit function tails to do register restores creates an impediment in gadget chaining. Exploit techniques like call-preceded ROP [4] are effectively blocked by the shadow stack and indirect branch tracking. Characteristics of the x86 ISA allows finding sufficient byte sequences [18] that decode to *jmp* through register instruction. However unlike ret, the indirect *jmp* through register is much less frequent in programs [19]. Indirect branch tracking significantly restricts the gadget catalog by requiring that gadgets must be of the *endbranch; jmp *y* form and be valid instruction sequences in the program. Chaining of these gadgets through an update-load-branch gadget [19, 3] of the form *endbranch; pop x; jmp *x* requires the property that the *register y* used to link back to the dispatcher be preserved, which further restricts choice from the restricted gadget catalog. Likewise, calling functions is also restricted to functions that have their address taken and the invocation has to be at the function entry point placing further constraints on control of stack and register contents. With the CPU providing the indirect branch tracking and return address protection, software and toolchains can further augment protection with language and platform specific policies and restrictions on control flow enforcement to increase its precision. One example policy could be to restrict the indirect calls to only land on functions that have the same prototype as intended by the call site [20, 17, 22]. With this policy a call site may look like: *mov $0xaabbccdd, %rax; call *%rbx* and a hash check performed in the prolog of the address-taken functions as: *endbranch; cmp $0xaabbccdd, %rax; jne error*. Other policies may be to restrict sensitive kernel functions to core OS and not drivers, restricting sensitive functions to be invoked only from specific call sites, etc.

Average indirect target reduction

Average indirect target reduction (AIR) [9] is a metric proposed by Zhang et. al. to measure strength of control flow integrity (CFI) of a program and represents set of reachable program addresses via indirect control transfer sites in program. We use the AIR metric as a measure of the improvement to a program using CET using below equation.

$$\sum_{i=1}^{n}(1 - |T_i|/S)$$

Here *n* is the total number of indirect branch transfer sites in the program, *S* represents set of program addresses which all the indirect branch transfer sites can direct control flow with no CFI protections. And *Ti* is represents set of program addresses to which i^{th} indirect branch transfer site can direct control flow with CFI protections. On x86, indirect control transfers can target any byte in program, thus *S* is program code size. Lower AIR value represents bigger (weak CFI) set of reachable addresses via indirect control transfer sites while higher AIR value represents a smaller (strong CFI) set of reachable addresses via indirect control transfer sites.

With CET enabled, the ret instruction can target exactly one target in the program which is the return address at top of the shadow stack and the call/jmp indirect can only target an endbranch. For the SPEC CPU 2006 C/C++ benchmark the average AIR metric was computed as 99.98% and for individual programs ranges from 99.93% (xalancbmk) to 99.99%.

Linux Kernel Gadget Analysis

We analyzed the Linux kernel (v4.9.9) binary for available gadgets using the ROPgadget tool [21]. The ROPgadget tool searches binaries for gadgets to facilitate research related to ROP exploits. The Linux binary we used was a default configuration kernel build of size 25 MB. We restricted our ROPgadget tool scan up to a gadget depth of 10 bytes yielding a sum of 197241 gadgets – we expect the usable gadget count to be more than that sum since we restricted our search space to a depth of 10 bytes due to limitations of the tool. The distribution of the counts for different gadget sizes

is shown in figure 4. Gadgets harvested via the ROPgadget tool ends in an indirect branch and are linkable/chainable. In contrast, in a CET-enabled binary, the exploit writer is restricted to using exported functions that have an ENDBRANCH and returning to the last address on the shadow stack. In the Linux kernel binary analyzed, 18412 exported functions were found. These exported functions need to be chained using an indirect call/jump and not through malicious use of ret. The measured average size of the kernel exported functions is 214 bytes and indicate the increase in complexity of using COP due to larger side-effects.

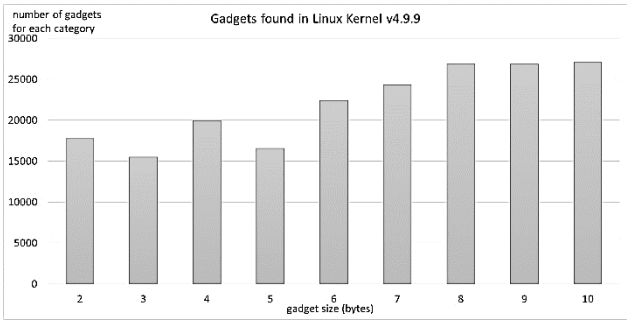

Figure 4: Gadgets found in Linux Kernel v4.9.9

We also analyzed the 18412 exported functions for the presence of an outgoing indirect CALL/JMP from those functions, and possible dispatch loops - the presence of a loop around an outgoing indirect branch that allows for functions to be chained. We found 2988 functions with outgoing indirect CALLs and none with indirect JMP. Of the 2988 exported functions with forward links, we found 148 functions that have at least one dispatch loop. This elimination of unintended gadgets and small number of exported functions that can be linked indicates that the attack surface can now be analyzed systematically to eliminate un-needed cases and address un-safe code constructions via redesign or focused checks via known software techniques.

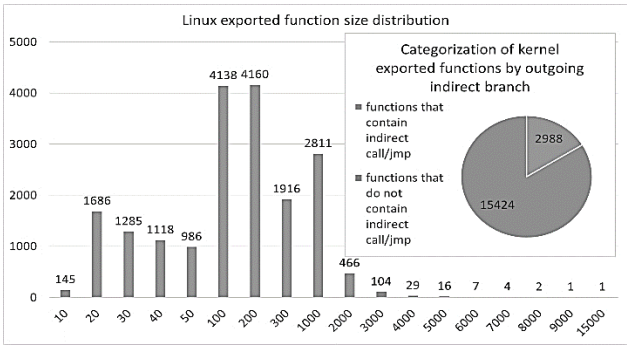

Figure 5: Linux exported function size distribution

5.3. Related work

Shadow stacks have been proposed as an effective means for return address protection. Dang et. al. [8] provide a survey of the various shadow stack techniques [8] using software instrumentation and the associated performance overheads. Pointer authentication Code (PAC) [23] and CCFI [24] have proposed using cryptographic message authentication code (MACs) to protect control flow elements such as return addresses. Safestack separates the program stack into two regions to protect return addresses [25]. Davi et. al. propose HAFIX [12] that uses hidden label stack based hardware implementation to restrict returns to active call sites. Lee et. al. propose a Secure Return Address Stack (SRAS) [14] that modifies call and return instructions to implement a secure stack in hardware and use pinned physical memory as a backing store. CET approach to shadow stack has parallels to the SRAS scheme but unlike SRAS, it does not implement a hidden shadow stack and supports shadow stack in linear address space of program. A desirable property of the shadow stack is to protect it from unintended writes. Shadow stack schemes using software instrumentation have relied on information hiding [11] to prevent writes to the shadow stack whereas hardware schemes have been proposed [12, 14] using on-chip memory for the shadow stack. CET extends x86 paging and EPT architecture to allow the OS and VMM to write-protect the shadow stacks. While protecting the return addresses using shadow stacks it is also important to be able to preserve the last-in-first-out (LIFO) [12, 13] property of control flow. CET defines a LIFO shadow stack and defines instructions to enable non-LIFO software constructs in a safe manner. An indirect call or jump can target any executable byte in the program and given the dense encoding of the x86 ISA the byte stream thus targeted may be interpreted as a valid sequence of instructions with high probability. Control-flow integrity schemes have tried to address this issue by introducing instrumentation to check if the target of the indirect call or jump is a valid target. Abadi et. al. [6] propose using prefetch instruction to embed an ID at valid indirect call/jump targets and inject a code sequence prior to the indirect call/jump to check the ID. Microsoft Control Flow Guard (CFG) [7] introduces a bitmap where each bit indicates whether there is a start of a function in the 16 bytes of process address space corresponding to that bit. A guard check function is invoked prior to an indirect call to test the CFG bitmap to determine if the target is a valid target. Hardware Control Flow Integrity proposal [5] proposes a pair of new instructions called "jump landing point" (JLP) and "call landing point" (CLP) that can be used to mark destinations of control flow branches in a program. LLVM Indirect Function Call Check (IFCC) [17] generates jump tables for indirect-call targets and adds code at indirect-call sites to transform function pointers such that they point to a jump table entry. Schuster et. al [15] have proposed defenses by restricting the invocation of sensitive functions to specific call sites in the program. CET provides the branch terminating instructions (ENDBR32/64) to enforce instruction alignment and restrict control transfers to valid indirect call targets in the program with low overhead and enables complimentary software and language specific policies to be anchored around the hardware enforcement.

6. CONCLUSIONS

CET provides the first general-purpose processor implementation of robust technologies for preventing control-flow subversion using techniques like ROP, and provides software with capabilities

to restrict COP/JOP attacks. CET design strives for minimal performance and memory overheads, while meeting strong security and compatibility objectives. In this paper, we perform an analysis of the enforcement of the security objectives for CET. In future work, we aim to evaluate how the CET ISA can be leveraged by software for further strengthening CFI properties for specific software domains.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their review and feedback.

REFERENCES

[1] Intel® 64 and IA-32 Architectures Software Developer Manuals. https://software.intel.com/en-us/articles/intel-sdm

[2] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. 2012. Return-oriented programming: Systems, languages, and applications. ACM Transactions on Information and System Security (TISSEC).

[3] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security.

[4] N. Carlini and D. Wagner. 2014. ROP is Still Dangerous: Breaking Modern Defenses. In 23rd USENIX Security Symposium (USENIX Security 14).

[5] Systems and security services analysis office. 2015. Hardware Control Flow Integrity (CFI) for an IT ecosystem. https://github.com/iadgov/Control-Flow-Integrity/tree/master/paper.

[6] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. 2009. Control-flow integrity principles, implementations, and applications. ACM Transactions on Information and System Security.

[7] Control Flow Guard. https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx

[8] T. H. Dang, P. Maniatis, and D. Wagner. 2015. The performance cost of shadow stacks and stack canaries. In ACM Symposium on Information, Computer and Communications Security, ASIACCS '15.

[9] M. Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In USENIX Security.

[10] Intel® 64 and IA-32 Architectures Optimization Reference Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[11] A. Oikonomopoulos, C. Giuffrida, E. Athanasopoulos, and H. Bos. 2016. Poking holes into information hiding. In USENIX SEC.

[12] L. Davi, P. Koeberl, and A.-R. Sadeghi. 2014. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In Annual Design Automation Conference - Special Session: Trusted Mobile Embedded Computing, DAC '14.

[13] M. Theodorides and D. Wagner. 2017. Breaking Active-Set Backward-Edge CFI. Proceedings of the IEEE International Symposium on Hardware Oriented Security and Trust (HOST '17)

[14] R. B. Lee, D. K. Karig, J. P. McGregor, and Z. Shi. 2003. Enlisting Hardware Architecture to Thwart Malicious Code Injection, Proceedings of the International Conference on Security in Pervasive Computing, Boppard, Germany.

[15] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. 2015. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications, in IEEE Symposium on Security and Privacy (S&P).

[16] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. 2005. Control-flow integrity. In Proceedings of the 12th ACM conference on Computer and communications security.

[17] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike. 2014. Enforcing forward-edge control-flow integrity in GCC & LLVM. In USENIX conference on Security.

[18] S. Checkoway and H. Shacham. 2010. Escape from return-oriented programming: Return-oriented programming without returns (on the x86). Technical Report CS2010-0954, UC San Diego.

[19] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. 2010. Return-oriented programming without returns. In ACM Conference on Computer and Communications Security (CCS).

[20] PaX Team. 2015. RAP: RIP ROP https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf

[21] Salwan, J. Ropgadget https://github.com/JonathanSalwan/ROPgadget

[22] Victor van der Veen, Enes Goktas, Moritz Contag, Andre Pawlowski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A Tough call: Mitigating Advanced Code-Reuse Attacks At The Binary Level. In 2016 IEEE Symposium on Security and Privacy.

[23] Pointer Authentication on ARMv8.3. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf

[24] Mashtizadeh, A. J., Bittau, A., Mazieres, D., and Boneh, D. 2014. Cryptographically enforced control flow integrity. In arXiv:1408.1451[cs.CR].

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

[26] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz and Yuval Yarom. 2019. In Proceedings of the 40th IEEE Symposium on Security and Privacy.

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