Taming Transactions: Towards Hardware-Assisted Control Flow Integrity Using Transactional Memory

Marius Muench[1](✉), Fabio Pagani[1], Yan Shoshitaishvili[2], Christopher Kruegel[2], Giovanni Vigna[2], and Davide Balzarotti[1]

[1] Eurecom, Sophia Antipolis, France
{marius.muench,fabio.pagani,davide.balzarotti}@eurecom.fr
[2] University of California, Santa Barbara, USA
{yans,chris,vigna}@cs.ucsb.edu

Abstract. Control Flow Integrity (CFI) is a promising defense technique against code-reuse attacks. While proposals to use hardware features to support CFI already exist, there is still a growing demand for an architectural CFI support on commodity hardware. To tackle this problem, in this paper we demonstrate that the Transactional Synchronization Extensions (TSX) recently introduced by Intel in the x86-64 instruction set can be used to support CFI.

The main idea of our approach is to map control flow transitions into transactions. This way, violations of the intended control flow graphs would then trigger transactional aborts, which constitutes the core of our TSX-based CFI solution. To prove the feasibility of our technique, we designed and implemented two coarse-grained CFI proof-of-concept implementations using the new TSX features. In particular, we show how hardware-supported transactions can be used to enforce both loose CFI (which does not need to extract the control flow graph in advance) and strict CFI (which requires pre-computed labels to achieve a better precision). All solutions are based on a compile-time instrumentation.

We evaluate the effectiveness and overhead of our implementations to demonstrate that a TSX-based implementation contains useful concepts for architectural control flow integrity support.

Keywords: Control flow integrity · Transactional memory · Intel® TSX · Binary hardening · Software security

1 Introduction

One serious security problem that continues to haunt security researchers and victims alike is the presence of memory corruption vulnerabilities, which can lead to the *arbitrary execution* of code specified by an attacker. Because these attacks can have serious consequences for the security of our lives and our society, countermeasures against classical stack- and heap based *code-injection attacks*

© Springer International Publishing Switzerland 2016
F. Monrose et al. (Eds.): RAID 2016, LNCS 9854, pp. 24–48, 2016.
DOI: 10.1007/978-3-319-45719-2_2

are widely deployed together with general security mechanisms in modern computer systems. For instance, Operating Systems ship with Address Space Layout Randomization (ASLR) and executable space protection like Exec Shield [40] or Data Execution Prevention (DEP) [2]. Additionally, modern compilers are able to harden applications against specific classes of attacks. For example, stack canaries protect against buffer overflows, and Relocation Read-Only (RELRO) protects against Global Offset Table (GOT) overwrite attacks. In combination, these countermeasures have nearly eliminated code-injection attacks.

However, even with all of these mechanisms in place, *code-reuse attacks* are still feasible. Hereby, the attacker reuses parts of the existing codebase of an application to achieve his goal. Generally, attackers accomplish this by corrupting data within the program and overwriting the target of an indirect jump (for example, by creating a fake stack with fake return values), thus hijacking the program execution. Any *indirect control flow transfer* (that, unlike a direct control flow transfer, can be influenced by values in program memory and CPU registers) is potentially vulnerable to this hijacking.

One line of defense against this type of attacks consist of checking the correctness of indirect control flow transfers before they are executed, by using a technique called *Control Flow Integrity* (CFI) [1]. In essence, CFI prohibits malicious redirections of a program's control flow by white-listing the possible targets of indirect transfers. If a change of control flow resolves to anything but an allowed target, the system would assume that there is an ongoing attack and terminate the program. Therefore, the goal of CFI is to ensure that, even if an attacker can gain control of the target of an indirect jump, her possible targets for control flow redirection are very limited and confined to the expected behavior of the program. Many CFI implementations [1,4,28,41–43], countermeasures to CFI implementations [6,9,16,17,19], and defenses against these countermeasures [25,26,31,34,38,39] have been proposed in recent years. Most of these studies have focused on the recovery of accurate control flow graphs (to understand the valid targets for indirect control flow transfers), on the binary instrumentation (to add CFI to existing binaries), and on reducing the performance overhead of the solution. Despite the importance of a hardware-supported CFI was already envisioned its original proposal [1], not much work has focused on how control flow integrity can be enforced by using features available in commodity hardware.

In this paper we present an application of the *Transactional Memory*-enabling instruction set extension (*TSX*), recently released by Intel for their Haswell processors, to provide hardware support for the implementation of control flow integrity. In particular, we propose a novel design that uses TSX instructions to ensure control flow integrity and we present a novel CFI implementation that introduces several interesting challenges for attackers. TSX-based CFI can provide, in hardware, new constraints on the attacker capabilities and a set of interesting protection features. In fact, aside from ensuring control flow integrity, our solution prevents an attacker from executing any system call after a hijacked indirect control flow transfer and introduces the ability to "reset" a program to its state prior to the hijacked control flow when the presence of an attacker is detected. These are powerful, previously absent capabilities, that we believe can significantly raise the bar for attackers in terms of control flow hijacking attacks.

In summary, we make the following contributions:

Design. We design a novel approach to implement control flow integrity, using the TSX instruction set extension recently released by Intel. Aside from simply providing CFI, this approach also provides a level of protection against unwanted invocation of system calls.

Implementation. We present two proof-of-concept implementations to show how TSX can be used to enforce both loose and strict CFI solutions.

Evaluation. We perform a thorough evaluation of our TSX-based CFI implementations, detailing overhead, side-effects, and security gains.

2 Control Flow Integrity

In recent years, researchers have proposed several solutions to enhance programs with CFI policies. Control flow integrity policies comes in two main forms, depending on how *restrictive* they are in specifying and enforcing the possible control flow.

In strict, or “fine-grained” CFI, the minimum set for allowable targets for any indirect control flow transfer is identified. For example, a return from a function would only be allowed to return to callers that could (legitimately) call the function. Before the execution, the target of every indirect transfer must be determined, and at runtime, these targets are verified whenever a control flow transfer takes place. A common method to implement such a strict form of CFI is labeling. Labels are assigned to the edges of a control flow transfer and are checked whenever the transfer occurs. Before a transfer is allowed, a verification procedure ensures that this transfer resolves to a target with a valid label.

Strict CFI is difficult to implement correctly, since the targets of indirect control flow transfers must be determined statically. Thus, researchers proposed a weaker form of CFI, called loose or “coarse-grained” CFI. This approximate technique segregates indirect control flow transfers by category and enforces policies on each category. For instance, loose CFI mandates that a transfer initiated by a *ret* instruction should always resolve to a location directly after a *call* instruction. Likewise, a call instruction should always transfer the control flow to the beginning of a function. Recent CFI implementations improve loose CFI by segregating control flow transfers into less coarse categories. Typically, different types of control transfers are given a different label [42,43].

BinCFI [43], for instance, uses static disassembly techniques, code instrumentation, and binary rewriting to identify all indirect control flow transfers and then enhances the program with coarse-grained CFI. Likewise O-CFI [26] uses static binary rewriting to introduce coarse-grained CFI in combination with fine-grained code randomization as protection against novel attacks. Another implementation requiring the relocation information of a binary, and thus slightly more knowledge, is CCFIR [42]. It introduces a springboard section and indirect control flow transfers are only allowed to target the springboard, which then redirects to the actual target. Contrary to these three implementations, which provide coarse-grained CFI and utilize static instrumentation, vfGuard [34] recovers C++ semantics of a given binary and uses Pin [3] to dynamically enforce strict CFI. Another example for dynamic CFI enforcement is Lockdown [33], which adds control flow guards for indirect calls and jumps and a shadow-stack to protect function returns via dynamic binary translation.

A completely different approach is to integrate CFI at compile time, which has the advantage of avoiding many complex issues related to resolving the targets of indirect control flow transfers (since static analysis of the source code before compilation can be used to provide this information) and removing the need to instrument or rewrite binaries. Niu et al. [28], for instance, introduced Monitor Integrity Protection, a coarse-grained form of CFI which aligns instruction to chunks and enforce that indirect jumps are targeting the beginning of a chunk with the goal to enforce low-level inlined reference monitor code integrity. Another example is SafeDispatch [24], a compiler based on Clang++/LLVM that adds protection for C++ dynamic dispatches.

All comprehensive compiler-based fine-grained CFI solutions need to deal with a common problem: shared libraries. Modern programs make often use of shared or dynamic loaded libraries. This problem is addressed by Niu et al. [29], who introduced modular CFI based on ID tables representing the actual CFG which is constructed during link-time. In between, production compilers could be enhanced to support compilation for binaries with fine-grained CFI policies. Tice et al. [38], for example, use vtable verification for virtual calls and indirect function-call checking to add practical CFI instrumentation to both GCC and LLVM.

A hybrid approach, combining both compile- and runtime instrumentation, is presented by πCFI [31]. In this case, programs are initialized with an empty CFG, which gets populated at runtime based on the provided input.

Recent research has expanded CFI beyond traditionally-compiled code on desktop systems. For example, just-in-time compilation can be enhanced with CFI policies, as shown in the case of RockJIT for JavaScript [30]. Furthermore, it has been shown that even entire commodity kernels can be instrumented to enforce CFI policies, as demonstrated in [12,18]. Moreover, MoCFI [13], a CFI framework for smartphones that uses static analysis to extract the CFG of binary files and utilizes library injection and in-memory patching to enforce CFI during runtime, shows that smartphones can also benefit from CFI.

Hardware support for CFI. The vast majority of CFI implementations employ software mechanisms for enforcing the integrity of control flow transfers [1,4,12,28,42,43]. However, a few attempts have been made to implement CFI using existing hardware features. CFIMon, for instance, utilizes Intel's Branch Trace Store, in combination with performance monitoring units, to detect control flow violations on-the-fly [41]. Likewise, kBouncer employs Intel's Last Branch Recording to mitigate ROP exploits without modification of the program [32] and PathArmor [39] uses the same hardware feature to enforce context-sensitive CFI. Unfortunately, those systems suffer from the fact that the Last Branch Record in its current implementation only records up to 16 branches. Our proposed method of using TSX to achieve CFI complements software-based CFI approaches by providing them with a mechanism to do the actual *enforcement* of CFI. Generally, it can work with any label-based CFI scheme, and replaces software-enforced control flow checking with a hardware-based solution.

Explicit architecture support of control flow integrity has been proposed by Budiu et al. [5]. In their proposal, new instructions are added for labeling targets of control flow transfer and for automated verification of these labels during a transfer. Davi et al. [15] have pointed out that this approach is likely to generate coarse-grained CFI policies and presented a different architecture for fine-grained CFI, based on two new instructions for function calls and returns as well as heuristics for validating indirect jumps.

Two recent approaches that proposed hardware-based, label-based CFI systems are HAFIX [14] and HCFI [7]. HAFIX enforces backward-edge CFI by extending the instruction set architecture of the Intel Siskiyou Peak and the LEON3 synthesizable processors. Similarly, HCFI extends the ISA of a SPARC SoC and utilizes an additional shadow stack to enforce both forward- and backward-edge CFI. Another hardware-based approach is presented by Clercq et al. [8], in which instructions reside encrypted in memory and are decrypted by the architectural CFI features in the instruction cache. This architectural features are implemented in a LEON3 processor and decryption errors occur when invalid branch targets are taken. While all these systems are good examples of hardware-based control flow integrity, they rely on custom hardware, rarely shipped in commodity computers. Our proposed approach, on the other hand, leverages a functionality that is *already* deployed in consumer CPUs.

An equivalent approach that uses recently introduced hardware features to enforce CFI was developed in parallel to our work by Mashtizadeh et al. in CCFI [25]. CCFI uses Intel's AES-NI extensions to construct, store and verify cryptographic MACs for pointers being used for control flow transfers, while the cryptographic key is held in compiler reserved registers, invisible to the attacker. This solution provides strong security guarantees, but it faces additional challenges not present in our approach, which result in an increased complexity. First, the introduced MACs for stack and heap addresses can suffer from replay attacks, in which an attacker leaks and uses a previous constructed MAC to change the control flow. To prevent this attack, additional heap- and stack-randomization need to be deployed. Furthermore, in certain corner cases, the compiler does not recognize function pointers which would lead to MAC failures and subsequent program termination. Although a static analyses pass for clang to detect these cases is provided, additional work by the developer of a software is required. Another minor problem is that the compiler reserved registers to store the cryptographic key are a subset of the registers introduced by Intel's SIMD extension. Thus, applications which are heavily using this extensions would experience additional overhead.

3 Transactional Memory

Transactional memory is a concept used in concurrent programming to describe a technique that allows synchronized and efficient access to data structures in a concurrent environment without the need of mutual exclusion [22]. Transactional memory introduces the concept of *transactions*, finite sequences of machine instruction that are *serializable* and *atomic*. Serializability means that different transactions appear as if they are executed serially, and therefore that different transactions do not interleave with each other. Atomicity, on the other hand, refers to the changes made to the shared memory: upon completion of a transaction, it either *commits* or it *aborts*. A commit makes all changes to the shared memory visible to other processors, while an abort discards the changes. Hence, the changes made to shared memory by one transaction are either fully represented in the memory space of the program or completely undone.

3.1 Transactional Synchronization Extensions

A selected subset of Intel's recent Haswell processors were manufactured with the Transactional Synchronization Extension (TSX) [36]. This extension enhances the x86-64 instruction set architecture by adding transactional memory features. Intel's TSX allows a programmer to specify code regions for transactional execution and provides two distinct interfaces, Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM) [10], that offer different functionality to users of transactional memory.

3.2 Hardware Lock Elision

HLE improves performance of concurrent code through the elision of hardware locks. Two new instruction prefixes are introduced to be used in front of instructions which normally would use software locks for synchronization:

XACQUIRE: The `XACQUIRE` prefix is used in front of an instruction which acquires a lock to a critical memory region. It marks the beginning of a transaction but instead of adding the shared memory to the processor's read or write set, the lock itself is added to the transaction's read set. For the acquiring processor, it appears as if it has acquired the lock, while for other processors the lock appears to be unchanged. Thus, other processors can read the lock without causing a conflict and, therefore, concurrently enter into the critical section. Although no data is *actually* written to the lock, the hardware ensures that conflicts on shared data will cause a transactional abort.

XRELEASE: The `XRELEASE` prefix is used in front of an instruction which releases a lock and ends a transaction. Normally, the release of a lock would involve a write to the lock. Instead, the system verifies that the instruction following the `XRELEASE` prefix restores the value of the lock to the value that it had before the `XACQUIRE` prefixed instruction. If this is the case, the processor tries to commit the transaction.

If a transaction fails due to a conflicting write in the shared data or the associated lock, all changes of the transaction are rolled back and the critical section is re-executed - this time using the lock in the classical manner. The advantage of HLE is that multiple threads can enter and execute critical sections protected by the same lock as long as no simultaneous operations on shared data are causing conflicts.

Additionally, HLE provides backward compatibility in the instruction set through a clever usage of instruction prefixes: processors without HLE support simply ignore the `XACQUIRE` and `XRELEASE` prefixes for all instructions which can be prefixed by `XACQUIRE` and `XRELEASE` and, thus, execute the critical code section with traditional locking.

3.3 Restricted Transactional Memory

RTM is a more flexible interface for marking code regions for transactional execution, without backward compatibility. This extension introduces three new instructions:

XBEGIN: The `XBEGIN` instruction is used to enter a transaction. Within a transaction, all accessed memory is added to the transaction's read set and all modified memory is added to the transaction's write set. The `XBEGIN` instruction must be followed by a 16- or 32-bit relative address to specify a fall-back path which gets executed when the transaction's commit fails or an explicit transactional abort occurs.

XEND: The `XEND` instruction ends a transaction and attempts to commit all changes. Should the commit fail, the fall-back path specified in the `XBEGIN` instruction is executed.

XABORT: The `XABORT` instruction is used to issue an abort for the transaction, rolling back all changes made by the transaction and executing the fall-back path. The `XABORT` instruction has to be followed by an 8-bit immediate as status code. This gives the programmer the possibility to specify a reason for issuing the abort.

The `RAX` register is used to indicate the reason for the execution of the fall-back path when a transaction abort occurs. The value of this register is not relevant for our purposes but, as we will see, the fact that it gets clobbered is inconvenient.

3.4 TSX Minutia

Intel's TSX provides another instruction, which can be used in both RTM and HLE based transactional execution paths:

XTEST: The `XTEST` instruction checks whether the processor is executing in a transactional state due to a HLE or RTM transaction. If `XTEST` is executed inside a transaction, the Zero Flag (`ZF`) is set to 0. Otherwise, it is set to 1.

Furthermore, both RTM and HLE are capable of transactional nesting and instruction-based aborts: While serializability of two distinct transactions is still ensured, both RTM and HLE allow the execution of transactions within transactions. The processor specific variables `MAX_RTM_NEST_COUNT` and `MAX_HLE_NEST_COUNT` are limiting this nesting. The nesting of a HLE transaction inside a RTM transaction or the nesting of RTM inside HLE remains undefined because both interfaces are accessing the same hardware capabilities.

Additionally, certain instructions cause a transaction to abort, regardless of how the transaction was initiated or what data has been written or read. Besides `XABORT`, the instructions `CPUID` and `PAUSE` cause a transactional abort in all situations. Depending on the TSX implementation, other instructions can trigger an abort as well. Among those are instructions for updating non-status parts of the `EFLAGS` register, interrupts, ring transitions, processor state saves, and instructions for updating the segment registers. A side-effect of the instruction-based aborts is context switch sensitivity. Several instructions, which can cause aborts depending on the specific implementation, are used by the kernel to perform context switches. As a consequence, transactions are aborted upon context switches.

3.5 Suitability for Software Security

TSX has already been analyzed for its possible application to software security. For example, Muttik et al. [27] pointed out that TSX can be used to detect malicious changes in memory by monitoring OS memory from a hypervisor and using transactional memory to automatically roll back these malicious changes. Furthermore, recent research by Guan et al. [21] proposes Mimosa, a system to protect private keys against memory disclosure attacks using hardware transactional memory features. Private keys are held in memory and decrypt or sign messages only within a transaction. Atomicity (as described in Sect. 3) causes the transaction to abort when a concurrent, malicious process tries to access the decrypted private key.

3.6 TSX Application for Control Flow Integrity

By studying the implementation of Intel's TSX, we realized that it can be leveraged as prototype for hardware-assisted CFI. Our intuition is that we can enter a transactional execution state before a control flow transfer and commit the transaction after the control flow transfer is done. In this manner, RTM can be used to implement loose CFI without checking labels in software. This is similar to the idea of *control flow locking* [4], which involves a write to a lock before an indirect control flow transfer and an unlock operation after the transfer.

Furthermore, HLE can be used to implement labels, allowing both loose CFI and strict CFI. This is based on the fact that the memory changed by a `XACQUIRE` instruction to enter a transaction has to be restored to its original value with the `XRELEASE` instruction in order to successful commit the transaction. By carefully choosing the memory location and value, we can ensure that redirected control flow will cause a transaction to abort, which will then be detected.

Besides basic CFI functionalities, the current implementation of TSX provides additional protection against current code-reuse attacks. Return Oriented Programming (ROP), for instance, relies on the fact that a set of so called Gadgets, each ending with a return instruction, can be chained together to form a more complex piece of code. In our TSX-based CFI, every return instruction is preceded by either a RTM or HLE instruction to begin a transaction. Thus, the number of gadgets that can be chained is limited by the corresponding `MAX_NEST_COUNT` for transactional nesting. Recent research has shown that restricting the maximum length of ROP gadget chains makes exploitation significantly harder, but attackers can still work around it [20]. However, TSX-based CFI adds another challenge for an attacker. In fact, many instructions that are typically used during an exploit (including system calls) trigger transactional aborts. Since for most exploits an interaction with the kernel is required, the attacker would need to find a way to escape from the transaction before the exploit can work.

4 Achieving CFI with TSX

Building up on the ideas described in Sect. 3.6, we designed an approach for providing control flow integrity using Intel's Transactional Memory Extensions (TSX). It is important to note that the techniques we discuss in this section can be adopted by any existing CFI techniques to ensure the integrity of control flow transfers, as well as to provide the additional protections afforded by TSX-CFI. Thus, we focus on the mechanism to detect the hijacking of the control flow, rather than on implementation details of CFI. Specifically, we expect that other techniques (such as [17,38,42,43]) can be leveraged to generate labels for strict CFI, which includes the computation of the valid targets for indirect control flow transfers.

In this section we discuss the implementation of both loose and strict CFI techniques. As with other loose CFI designs, our solution trades limited protection for simplicity in the implementation and deployment (i.e., the exact jump targets of every instruction do not have to be determined). On the other end, our reference strict CFI design provides stronger guarantees, with the requirement of a more complex analysis to identify jump targets.

An important difference between TSX CFI and traditional CFI is that TSX CFI *does not* prevent the attacker from hijacking the program's control flow. Instead, it simply ensures that any indirect control flow transfer in the program that can be hijacked by an attacker happens *inside* a TSX transaction. As a result, the control flow hijacking will eventually cause the transaction to abort, essentially *rewinding* the program to the clean state right before the control flow was hijacked and redirecting the execution into our fall-back path, which can use more sophisticated and time-consuming techniques to verify the presence of an attack and abort the program.

4.1 Transaction Protection

A core tenet of performing CFI with TSX is that many instructions, including system calls, cannot be performed inside a memory transaction. Thus, the underlying principle of our approach is that we enter into a transaction before attempting an indirect jump and exit from it at the jump target. These transactions are very short – in the normal case, the transaction starts, the indirect control flow occurs, and the transaction ends. If an attacker is able to manipulate the target of this instruction, and redirects it to an instruction that does not end the transaction, the transaction will fail for one of several reasons:

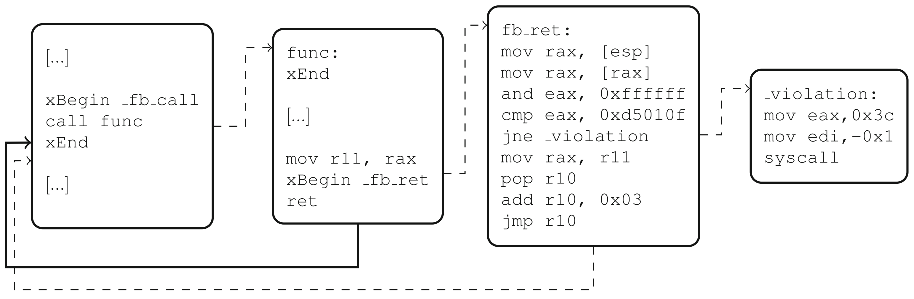
Context switch. The execution of the program is suspended by the kernel to run another process.

Instruction-based aborts. The execution of a transaction-aborting instructions (e.g., a system call).

TSX nesting limit. A transaction is nested in X other transactions, where X is the transaction nesting limit of the CPU.

Each TSX failure case presents a different challenge to an attacker. The context switch failure case limits the amount of code that an attacker can execute without closing the transaction, instruction based aborts makes it impossible to execute certain instructions like system calls while inside a transaction, and the TSX nesting limit puts a bound on the length of an attacker's ROP chain. This latter effect is very interesting: since we initiate a transaction before each indirect control flow transfer, an attacker that chains ROP gadgets in the traditional way will enter an extra nested transaction with each gadget. Since the nesting depth is limited (on most processors, this depth is 16), an attacker will quickly hit the transaction nesting limit, and, thus, cause a transactional abort. Furthermore, to be successful, an attacker must first *successfully exit* all entered transactions, in the reverse order of entering them, before operations such as system calls can be carried out. We want to emphasize that the nesting limit poses problems only for an attacker and not for benign applications. In fact, our implementation encloses only the control flow transfer instructions within transactions, and not the entire target function. For example, the transaction is opened just before a `call` instruction, and closed with the first instruction of the call destination.

When a transaction *aborts*, two things occur. First, the actions taken by the attacker while inside the transaction are *discarded* by the processor. Second, a *fall-back path* is triggered to check for the presence of an attacker (i.e., by verifying that the control flow transfer is going to a legal target). This is done because, aside from a control flow hijack, a context switch (for example, when the process is suspended to allow another process to run) will also abort a transaction. While this complicates our fall-back path, it introduces another challenge to the attacker: they must escape our transaction *quickly*, before the process is swapped out and the transaction is aborted.



```
[...]

xBegin _fb_call
call func
xEnd

[...]
```

```
func:
xEnd

[...]

mov r11, rax
xBegin _fb_ret
ret
```

```
fb_ret:
mov rax, [esp]
mov rax, [rax]
and eax, 0xffffff
cmp eax, 0xd5010f
jne _violation
mov rax, r11
pop r10
add r10, 0x03
jmp r10
```

```
_violation:
mov eax,0x3c
mov edi,-0x1
syscall
```

Fig. 1. Control flow of a function returning for RTM-based CFI

4.2 RTM and Loose CFI

We leverage Restricted Transactional Memory (RTM) to provide an implementation of loose CFI. To ensure that every indirect control flow transfer goes to a valid target, a transaction is started before each transfer and ended at the target site. For example, every function return is preceded by a XBEGIN instruction, while every function call is followed by a XEND instruction. Thus, a transaction will be started, the function will return, and the transaction will then be completed. As long the return address used by the return instruction is not manipulated, the transaction can only fail due to a context switch. The idea is visualized in Fig. 1, using the example of a function return.

In a failure case, the fall-back path specified in the XBEGIN transaction will be executed. Since RAX is used to indicate the reason for the fall-back path execution, we copy its value into an unused scratch register before entering a transaction. This enables us to restore the original function return value, which is also passed in RAX, in the case that the fall-back path gets executed due to a context switch during a benign control flow transfer. This can happen for two reasons: when the transaction is interrupted by chance because of a context switch initiated by the kernel, and when the control flow is hijacked by an attack. Thus, the fall-back path itself has to verify that the target of the control flow transfer is still pointing to a memory location containing the opcodes for an XEND instruction. Since different kinds of indirect control flow transfers determine the target of a transfer differently, several fall-back paths are required. In the case of function returns, the target (i.e., the return address) is on the stack, and can be dereferenced via RSP. Certain indirect jumps and calls, on the other hand, use a general purpose register to specify the target of the transaction. Thus, the fall-back path has to deference the content of the specific register. The only exception is provided by the CALL RAX and JMP RAX instructions because RAX gets overwritten upon entering the fall-back path. Naturally, instead of RAX, the local copy in the scratch register has to be dereferenced. Furthermore, if the control flow transfer is initiated by a call instruction, it is also necessary to save its origin inside another scratch-register. If the fall-back path can not detect the presence of an attacker, it can push the saved origin and jump to the target, effectively emulating the call. If the fall-back path does not detect the presence of a `XEND` instruction at the transfer's target, the presence of an attacker is assumed, and a CFI violation policy is triggered. This, naturally, terminates the program. If the presence of an attacker cannot be determined, the original value of `RAX` is restored, and the control flow is transferred to the target of the indirect jump.

Provided Protection. While the RTM implementation is very straightforward, it can only reason about a single set of *jump origins* and *jump targets*. That is, if an attacker hijacks the control flow, the presence of RTM CFI forces her to terminate the transaction at a valid jump target. However, with the exception of certain actions that are prohibited within a transaction (discussed in Sect. 4.1), an attacker can carry out any modification of memory (for example, by initiating a ROP chain) and then transfer the control flow back to a valid jump target, which will, in turn, terminate the transaction.

In essence, RTM provides weak security guarantees, but it is an important building block towards TSX CFI, and a useful tool to later measure the performance impact of our techniques. HLE, on the other hand, builds on these building blocks to provide security guarantees for TSX-assisted CFI.

4.3 HLE and Strict CFI

With strict CFI, every indirect control flow transfer is only allowed to proceed along strict well-defined paths. For example, functions may only return to their callers, and indirect jumps can only go to targets that were intended for them by the programmer. One way to implement such a policy is by using *labels*. With labels, every control flow transfer is assigned a label that it shares with the valid targets of that control flow transfer. When the transfer is triggered, the CFI policy ensures that the label at the source address matches the label at the destination address, terminating the program if this is not the case.

Intel's Hardware Lock Elision provides functionality that can be leveraged to implement such labeled CFI. Specifically, HLE elides a memory write to a memory location that represents the lock itself. We will term this location the *lock location*, and the value at the lock location the *unlock value*. A transaction is entered by performing a write to the lock location (termed a *locking write*), with the write instruction prepended by `XACQUIRE`, and is successfully terminated when the unlock value is restored by a write instruction prepended by `XRELEASE` (termed an *unlocking write*). We call the value that resides at the lock location during a transaction a *lock value*. For a transaction to commit successfully, the value written to the lock location during an `XRELEASE` *must* be the unlock value.

Our idea is to introduce labels by carefully choosing the (numeric) value used during the locking and unlocking write operations. The lock location is chosen as an offset on the stack, and we implement the locking write by simply adding the label value to that location. In turn, the unlocking write consists in subtracting the label, thus restoring the unlock value and successfully committing the transaction at the intended target of this control flow transfer. As with RTM,

Listing 1. HLE-based CFI

```
[...]
  call func
  xrelease lock sub [rsp], 0xcf1bee
[...]

func:
[...]
  xacquire lock add [rsp-0x8], 0xcf1bee
  xtest
  jnz __inside_transaction:
  mov r11, 0xcf1bee
  jmp __hle_cfi_fb_ret
__inside_transaction:
  ret
```

a transaction abort signals a potential attack. However, some additional details must be considered when enforcing HLE-based CFI. HLE has no mechanism to detect the *reason* why a transaction failed. While this has the benefit of not clobbering `RAX` (unlike RTM), it comes with a cost: HLE has no capability to execute a fall-back path on a transaction abort. Instead, HLE simply re-executes the critical section *without* eliding the lock write. Intel's intention is that, if the elided lock fails, a software-locking mechanism would simply take over. Thus, a *virtual fall-back path* has to be injected for HLE-protected control flow transfers. This can be done with the `XTEST` instruction, which identifies whether the process is currently in a transactional context. Therefore, a failed or aborted transaction can easily be detected by executing `XTEST` after entering the critical section. When an unsuccessful transaction is detected, a jump to the virtual fall-back path can be issued manually.

The fall-back path itself is similar to the fall-back path of RTM CFI. The only difference is that the fall-back path checks for a label in the code that would be executed after the indirect control flow completes. As with RTM, we cannot simply assume the presence of an attacker on transaction abort, because any context switch into the kernel would also trigger a transactional abort. Thus, the fall-back path is necessary.

An example showing an instrumented return using HLE is presented, for clarity, in Listing 1. A careful reader will notice that the lock location is actually different between the `XACQUIRE` and `XRELEASE` instructions. In reality, the lock location is the same: since the `RET` instruction itself modifies the stack pointer (by popping the 8-byte return address), the offset must be different by exactly 8 bytes after the `RET` executes.

Provided Protection. HLE extends the simple transactions provided by RTM with the ability to *label* indirect control flow transfers, allowing HLE CFI to ensure that indirect control flow transfers must proceed to a valid target and not just to *any* target. Likewise, the fact that indirect control flow transfers take place within a TSX transaction ensures that the execution flow cannot be hijacked and rerouted to system calls. Besides that, HLE introduces novel, interesting capabilities in control flow transfer protection: aside from ensuring that the transaction ends on a valid jump target, the use of HLE also mandates that, between the beginning and end of a transaction, the *value* of the stack pointer must be equal to itself plus the offset introduced by the instruction issuing the control flow transfer. This is implicit as part of its operation because a location on the stack is used as the lock location. To end a transaction successfully, this exact location must be written to, and the exact same value (the unlock value) that it had before the transaction began must be restored. If the stack pointer is unexpectedly modified during the transaction (for example, if the attacker hijacked the control flow and initiated a ROP chain), the unlock value will not be restored, since another location will be written to, instead. This, in turn, will cause the transaction to fail, the attacker's actions to be rewound, and the attacker to be detected.

Thus, HLE-supported CFI provides a formidable protection against control flow hijack attacks.

5 Implementation

We implemented our proposed TSX CFI design in a reference prototype to demonstrate that TSX-based CFI can be used to enforce CFI policies and to understand the overhead of such protection. This implementation is being released as open source, in the hope that it will be useful to other security researchers.

Because we did not possess a binary analysis system capable of constructing an accurate control flow graph from arbitrary binary applications, we implemented our approach at the source code level by instrumenting the compiler (specifically, we added a pass to the LLVM backend). While we consider binary analysis and rewriting outside of the scope of our work, existing tools have solved this problem [26,42,43], and their solutions could be reused directly to implement TSX-based CFI directly on binaries.

5.1 Integration Approach

We chose to implement our reference prototype as a backend compilation pass in LLVM. Our prototype combines a preprocessing-engine, the clang compiler with the TSX-CFI backend pass, and a postprocessing engine. The preprocessor performs a linear search over the provided source-code to detect and instrument

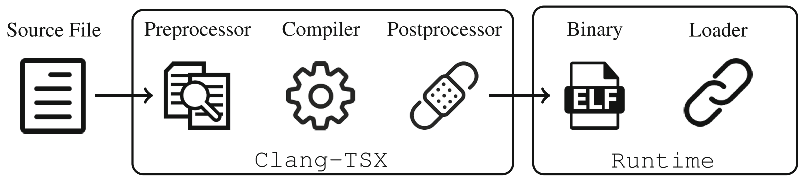


Fig. 2. Overview of the TSX CFI implementation

inline assembly, since it would be translated directly to binary code without being analyzed by the backend. The compiler produces an actual binary, where every function entry and every instruction issuing an *inter-functional control flow transfer* is instrumented (i.e. direct calls, indirect calls, function returns and tail-calls). During compilation, the TSX-CFI backend is unable to tell whether the targets of direct calls are located inside the binary or inside an external library, as this information is only visible at link time. However, since external calls are resolved via the *Procedure Linkage Table (PLT)*, these calls still require protection against control flow hijacks. Thus, the LLVM backend emits only no-op instructions for direct calls. These are fixed up by the postprocessor, which adds the protection for calls preformed via the PLT. Figure 2 shows the general overview of our implementation.

5.2 Implementation Details

As the implementation of our prototype is fairly intricate, we provide this section to introduce the interested reader with specific implementation aspects.

Selective Function Entry. Direct calls cannot be hijacked by attackers and, thus, do not need to be protected. However, this poses a problem: if a function is called by both direct *and* indirect calls throughout the binary, only the indirect calls should take place inside a transaction. To facilitate this, the post-processing engine modifies direct calls to a function to bypass the TSX transaction commit instruction. This does not reduce the security of the indirect jump because, even if the attacker redirects the jump to skip the transaction commit, he will still be stuck inside the transaction.

Lazy Binding and Full Relro. Our prototype supports the compilation of both binaries resolving external function during runtime (*Lazy Binding*) and binaries resolving all functions during load time. In *Full-Relro* the *Global Offset Table*, which stores the location of resolved functions, is marked as read-only. In this case, calls via PLT can only resolve to their intended target, which makes it impossible for an attacker to hijack those external function. Thus, these calls do not need protection. To optimize this case, we customized the dynamic loader to store pointers to the instruction *after* the commit instruction in the GOT, similar to the case of direct jumps to functions within the binary.

5.3 Limitations

Our prototype suffers from two classes of limitations, due to the fact that we lack kernel support and to our choice of using the clang compiler. While these problems are limiting the applicability of our current prototype for real-world applications, they do not pose any conceptual problem for TSX CFI. In the rest of this section, we address these limitations and describe how they could be solved in a future implementation.

Standard C Library. The standard C library for GNU/Linux systems, *glibc*, cannot be compiled with clang, since it requires functionality explicitly provided by *gcc*. In turn, it is not possible to create a TSX CFI instrumented *glibc* with our reference implementation. However, in order to provide a holistic TSX CFI solution, an instrumented standard C library is required. For this purpose, we instrumented musl, a free and lightweight standard C library. Obviously, this can be a problem for programs that are expecting the presence of `glibc`, for which we frequently observe crashes due to the different locale subsystems. We verified that these crashes are purely based on the incompatibility between the standard C libraries and are not introduced by our TSX CFI prototype. This issue can be solved by adding a `gcc` TSX CFI extension.

Virtual Dynamic Shared Object. Another interesting side-effect of our TSX CFI prototype is that we had to disable *Virtual Dynamic Shared Object (vDSO)* support. This object is provided by the kernel and mapped in the address space of every process, mainly to speed up syscalls, such as *gettimeofday()*, which are normally called via the standard library. Since vDSO is entered using a call instruction, an instrumented version of this object would be required for TSX CFI, which would require changes to the kernel and break the operation of uninstrumented programs. Therefore, the usage of vDSO in TSX-CFI instrumented programs is disabled for compatibility with unprotected programs. As solution, a holistic approach including kernel support for TSX CFI would be required.

Signal Handlers. Programs can register signal handlers that are executed upon the reception of given signals. However, in this case it is the OS kernel that redirects the control flow to the handler, and therefore a transaction is not entered when the signal handler is called. A possible solution would be to instrument the libc to alter the signal handlers pointers to use the un-instrumented function entry address. Another solution would be to instrument the kernel itself, similar to the case of vDSO.

Setjmp/Longjmp. The setjmp and longjmp interfaces are used to perform non-local gotos. While our implementation does not instrument non-local gotos, they still represent a class of indirect control-flow transfers. To cope with them, an advanced analysis engine for recovering the CFG would be required to retrieve the possible control flow targets. Nevertheless, the indirect transfer itself can easily be protected with transactions once the possible targets are known.

6 Evaluation

We evaluated our implementation to determine the practicality of TSX-based CFI. As we discuss in Sect. 4, we view our approach as a general way to implement the protection of indirect control flow transfers and expect that it will be leveraged by complete CFI approaches for that functionality. As such, the goal of this section is to provide an understanding of the overhead that TSX-protected control flow transfers induce in the runtime of actual application.

We performed our experiments on a diverse set of applications to evaluate the impact of TSX-CFI on different workflows. For this evaluation, we chose `GNU coreutils` (the collection of standard Linux binaries such `ls`), `bzip2` (a common compression utility), `lame` (an audio encoder with the main purpose of encoding WAV audio files to MP3), and `openssl` (the general-purpose cryptography command line tool).

6.1 Experiments

We measured the performance of TSX-based CFI on a Intel Core i7 Haswell processor that supports TSX operations. To measure the overall performance overhead of our TSX CFI implementation, we selected tasks for the instrumented programs and calculated the average execution time over a series of 20 executions, using both HLE and RTM-provided indirect control flow protections. We chose the following programs and tasks:

coreutils: Execution of the 580 tests provided in the test-suite for the various utilities.
bzip2: Compression of a 200 megabyte file.
lame: Conversion of a 60 megabyte WAV file to a MP3 file with a bit rate of 128 Kbps.
openssl: Encryption of a 4 gigabyte file of random data with AES-256-CBC.

All of the experiments were executed with our instrumented version of musl, in which we also protected all indirect control flow transitions. The bzip2, lame, and openssl experiments ran without issue. However, of the 580 test cases provided by coreutils, 47 failed to run with our prototype, due to the differences of the locale subsystem of musl, and glibc, as described in Sect. 5.3. In the case of TSX-CFI implemented with RTM, three additional coreutils test cases failed with segmentation faults. Investigation into these segmentation faults revealed that they occurred due to program-defined signal handlers: some signal handlers would redirect the control flow of the process *without* entering a transaction, resulting in the execution of an `XEND` instruction outside of a transaction, which crashes the process.

6.2 Performance Overhead

We averaged the runtime of 20 executions on each experimental binary (in the case of the coreutils, we averaged the total time of 20 executions of all coreutils test cases). Due to the distinct workload carried out by the applications, we expected to see different overhead with our CFI implementation, and this was, indeed, the case as shown in Fig. 3. For example, `lame`, `bzip2` and `openssl`, which spend most of their time in long-running loops for encoding, compression, or encryption, and call many helper functions in these loops, result in an overhead of up to 34 %. On the other hand, most tools inside coreutils spend much time

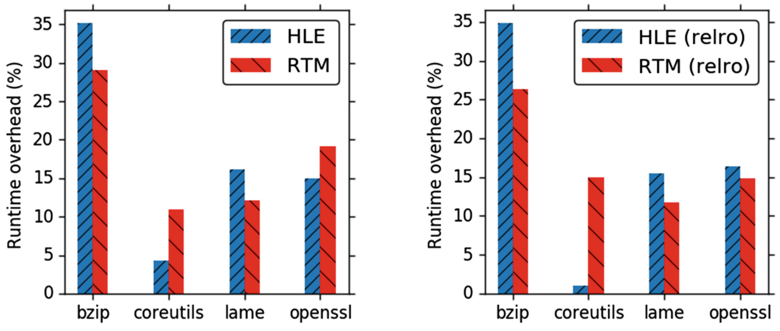


Fig. 3. Average runtime overhead

Table 1. Number of issued and aborted transactions.

Program	#Executed	#Aborted
bzip2	565711783	1781
lame	493580143	247
openssl	546743640	1088

Table 2. Size of instrumented programs, in Kilobytes.

Program	None	RTM	HLE	RTM-relro	HLE-relro
coreutils	85	101	117	90	99
bzip2	223	256	271	247	250
lame	401	459	523	422	450
openssl	2536	3362	4313	2817	3196
musl-libc	725	767	835	839	979

interacting with the host OS through system calls, resulting in a small overhead of up to 5 % in the case of HLE, which reflects our expectations.

In the coreutils case, the majority of the overhead from HLE came from the lazy binding of library functions. Again, this is consistent with what we expect: the coreutils binaries are a mostly short-running utilities and use many library functions, leading to the (CFI-protected) symbol resolution process to be called relatively frequently. Enabling RELRO (turning off lazy binding) for these binaries results in a drastic decrease of runtime overhead with HLE, to just 1 %. On the other binaries with less system interaction (lame, bzip2, and openssl), the difference is negligible.

6.3 Transaction Aborts

When a transaction fails, execution is diverted to the *fall-back* path, which checks whether the process has been exploited. However, as we discuss in Sect. 4.1, there are several reasons, other than exploitation, that can cause a transaction to abort.

To understand how frequently this occurs during normal operation, we evaluated the number of transactions that are attempted and the number that aborted. To do so, we utilized the capabilities of Intel's Software Development Emulator to measure the amount of executed transactions. Unfortunately, the emulator does not report aborted transactions caused by the environment (i.e., context switches).

Thus, we computed this number by instrumenting the aborted transaction fall-back path to track a counter of the number of times it was executed. The results of this measurement are presented in Table 1. From these results, we see that the rate of transaction failures is almost negligible. Thus, the most significant part of overhead that we experience with TSX-CFI is induced by continuously entering and leaving successful transactions. Ritson et Barnes [37] observed that invoking a transaction costs approximately 43 clock cycles, which, given the high number of executed transaction, results in the observed overhead.

6.4 Space Overhead

It is important to measure the space overhead of a program being protected by any CFI approach, both in terms of memory usage and program size.

While CFI, when implemented with TSX, suffers no additional memory usage overhead, the size of the program is increased due to the addition of both the TSX instructions themselves and also that of the fall-back paths. The size overhead, as shown in Table 2, depends on the TSX method that is used to enforce CFI and on the number of protected transitions inside the program. We calculated this overhead for the applications themselves (in the case of the coreutils, we used the arithmetic mean of the overhead for individual binaries) and the standard library.

We feel that, especially with modern systems, the low space overhead introduced by our implementation is quite acceptable for a CFI implementation.

7 Discussion

The use of TSX for control flow integrity brings interesting possibilities, but it also introduces several challenges. This section discusses the challenges, protections, and possible future research directions in the area of TSX-based CFI.

7.1 TSX Performance

As described in the evaluation section, the simple act of entering a TSX transaction incurs an immediate performance penalty. However, some different directions can be explored to reduce this overhead in the future:

Hardware improvements. TSX is a very young technology, and it is very likely that performance optimizations will be implemented in future Intel processors. While little can directly be done by security researchers to bring this about, the usefulness of TSX for things other than its actual intended application (i.e., this CFI approach or the protection of cryptographic keys in memory [21]) might make TSX a high-profile target for optimization.

Virtual Transactional Memory. TSX transactions are aborted whenever a context switch occurs. These transaction aborts have a strong impact on the performance, since they force our solution to use complex fall-back paths to check for the presence of an attack. These fall-back paths introduce runtime and space overhead, but are unavoidable with context-switch-based transaction aborts.

One approach toward eliminating this overhead is to allow a transaction to pause in the event of a process being paused, and resume when the process is resumed. In fact, designs for *virtual transactional memory* have been proposed [35] that would allow transactions to survive across context switches (specifically, pauses and resumes of a running process). If these techniques are adopted in the future, they could greatly improve the performance of TSX CFI.

Selective protection. Not every part of a program is equally vulnerable to a control flow hijack, and those functions that are not (and do not utilize any other functionality that must be protected) may not need CFI protection. A similar trade-off is seen in the application of *stack canaries* to prevent buffer overflows, where functions that lack any overflowable buffers do not receive stack canaries [11]. Performance could be greatly improved by leaving such "low-risk" functions similarly unprotected by TSX CFI. Similarly, protection could focus on specific types of control flow transfers. For example, function returns can be protected through the use of a shadow stack or a similar, less expensive approach, leaving the more expensive TSX protections for indirect calls and jumps, for which fewer alternative protection mechanisms exist.

While it is hard to speculate on the future of TSX, it is clear that it is an important capability, not only in the field of concurrent systems, but also in computer security. It seems quite likely that additional effort will be put into its optimization and the addressing of its limitations.

7.2 Protection Strength

As we discuss in Sect. 4.1, TSX-based CFI works by ensuring that, if an attacker manages to hijack the control flow of a program, he will find himself inside a TSX transaction. These transactions severely limit what an attacker can do, and if the attacker violates the restrictions the transaction is aborted and the process is rewound to the state *before* the control flow was hijacked. When this occurs, a fall-back path is triggered, checking for the presence of an attacker (by verifying whether the pending control flow transfer is targeting a legal location) and aborting the program if an attack is detected.

Thus, to perform useful actions, an attacker is forced to find a way to escape from the TSX transaction, using one of the following two options: (1) The attacker can jump to some previously-injected shellcode that commits the transactions and gives the attacker control or (2) the attacker can execute several ROP gadgets inside the transaction, influence the program state, then jump to the actual legal target of the initial protected control flow transfer.

Both options introduce challenges for the attacker. The first option is already mitigated by existing countermeasures against code-injection attacks, such as Data Execution Prevention or the *No-eXecute* bit, that are widely deployed in modern systems to prevent injected data from being executed. Attackers bypass these protections by diverting the control flow to execute a system call, such as `mprotect()`, that allows the injected data to be executed. However, this process involves the execution of a system call, which is not allowed inside a transaction. Thus, the attacker is presented with a chicken-and-egg problem: in order to commit the transaction, he must execute a system call, and in order to execute a system call, he must commit the transaction.

The second option is a possible, if seemingly infeasible way to escape a transaction. An attacker could hijack the control flow and, without aborting the transaction, utilize a small ROP chain to perform some action before jumping to the intended target of the hijacked control flow and letting the transaction commit happen. The attacker would then perform actions in this ROP chain, being careful not to violate the restrictions placed on him by the transaction. For example, these actions can include influencing sensitive data structures in the program. Although this certainly empowers the attacker in comparison to other CFI solutions, in practice, carrying out this attack is extremely difficult, especially for the HLE based CFI approach. Specifically, the stack pointer *and* the lock value must not unexpectedly change values during the control flow transfer. Thus, an attacker must execute this attack *without* altering the stack pointer or the lock value across the transaction. Additionally, this chain must be fairly short: a context switch during ROP chain execution will lead to an aborted transaction and the detection of the attacker. To make matters worse (for the attacker), using any protected indirect control flow transfer will cause the initiation of additional transactions, all of which the attacker must escape (in reverse order of initiation, and without modifying the stack pointer or lock value) before escaping the original transaction. We feel that, in practice, these restrictions make such an attack infeasible.

7.3 Comparison with Other Techniques

Our approach introduces a higher overhead when compared to other recent CFI enforcement schemes which do not require dedicated hardware features, such as [26,29,39,42]. While this is surely a drawback of the presented implementation, we believe that it is too early to disregard TSX CFI as unusable, since Hardware Transactional Memory itself is a new CPU feature and performance speed-ups are feasible in further iterations. However, our main goal is to explore the suitability of the new hardware transactional memory features for control flow integrity purposes. We hope that our study can provide useful insights on how hardware-assisted CFI could look like and that it can help other researchers in the field to design future CFI implementations.

Moreover, we were happy to see that Intel recently released its Control-flow Enforcement Technology Preview (CET) [23], in parallel to this paper, showing the demand of hardware manufacturers to provide architectural CFI support. CET is meant to advance the processor with features against ROP attacks. In more detail, a shadow stack is used to protect function returns and indirect branch tracking for protecting indirect control flow transfers. The latter technique introduces a new instruction, `ENDBRANCH`, which needs to be executed after the occurrence of an indirect control flow transfer. Since CET in its current state is only a preview and is not available for consumer hardware yet, we can not compare its performance to TSX CFI. However, it is notable that the CET's indirect branch tracking is similar to our RTM based approach: In both cases the processor is set to a state waiting for a certain instruction to specify the end of a control flow transfer; In TSX CFI this state is explicitly forced by opening a transaction, while CET introduces a new `WAIT_FOR_ENDBRANCH` state, which is implicitly imposed to the processor upon executing an indirect call or jump. While the shadow stack provides stronger security guarantees and could easily replace TSX CFI for backward-edges in a future implementation, the deployment of labels like presented in our HLE based CFI approach yields a finer granularity than CET's indirect branch tracking.

7.4 Additional Capabilities - Future Work

While not related directly to CFI, TSX has other potential applications that are interesting. A possible application is to ensure the integrity of certain sensitive memory regions or registers over the course of the execution of some functionality deemed to be "dangerous" (i.e., a strcpy known to contain user input). For example, a HLE transaction could be entered by subtracting 0 from the sensitive memory region, the functionality could be carried out, and the transaction would be committed by subtracting 0 again. If the contents of the sensitive memory region were different (i.e., due to an attack) at the end of the transaction from their value at the beginning, the transaction will abort. Registers can, likewise, be protected by XORing them to memory as part of initiating the transaction and XORing them to memory again to commit the transaction.

If virtual transactional memory is adopted, these approaches can be utilized to protect data in relatively complicated program functionality, as long as no system calls are performed.

8 Conclusion

In this paper, we proposed a technique to enhance control flow integrity by leveraging new hardware capabilities, intended to support transactional memory.

Our design provides two distinct levels of CFI protection: unlabeled CFI and labeled strict CFI. In a TSX-based CFI system, every indirect control flow transfer occurs inside a transactional memory transaction. If such a control flow transfer is hijacked by an attacker, the attacker will find himself inside the transaction, with severely limited capabilities. Eventually, this transaction will be aborted, which will roll back all of the changes to memory or registers made by the attacker and lead to the attacker's detection. As a side-effect, our technique can protect the values of the stack pointer as part of its operation. If an attacker modifies this register, for instance during a code-reuse attack, and attempts to commit the transaction, the transaction will fail.

We implemented a proof-of-concept prototype of TSX-supported CFI and used it to evaluate the runtime and size overhead of instrumented programs. The evaluation of our approach showed that induced overhead in performance is mediocre compared with other recent CFI solutions, with a very modest program size overhead and no other memory usage increase. While the overhead is higher in comparison to other CFI approaches, we discuss possibilities for speed-up, and the potential of future developments to enable faster TSX-supported CFI.

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: Proceedings of the 12th ACM Conference on Computer and Communications Security. ACM (2005)
2. Andersen, S., Abella, V.: Data execution prevention. Changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies (2004)
3. Berkowits, S.: Pin-a dynamic binary instrumentation tool (2012)
4. Bletsch, T., Jiang, X., Freeh, V.: Mitigating code-reuse attacks with control-flow locking. In: Proceedings of the 27th Annual Computer Security Applications Conference. ACM (2011)
5. Budiu, M., Erlingsson, U., Abadi, M.: Architectural support for software-based protection. In: Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability. ACM (2006)
6. Carlini, N., Barresi, A., Payer, M., Wagner, D., Gross, T.R.: Control-flow bending: on the effectiveness of control-flow integrity. In: 24th USENIX Security Symposium (2015)
7. Christoulakis, N., Christou, G., Athanasopoulos, E., Ioannidis, S.: HCFI: hardware-enforced control-flow integrity. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy. ACM (2016)
8. de Clercq, R., De Keulenaer, R., Coppens, B., Yang, B., Maene, P., de Bosschere, K., Preneel, B., de Sutter, B., Verbauwhede, I.: SOFIA: software and control flow integrity architecture. In: Design, Automation & Test in Europe Conference & Exhibition (DATE) (2016)
9. Conti, M., Crane, S., Davi, L., Franz, M., Larsen, P., Negro, M., Liebchen, C., Qunaibit, M., Sadeghi, A.R.: Losing control: on the effectiveness of control-flow integrity under stack attacks. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM (2015)
10. Intel Corporation: Intel Architecture Instruction Set Extensions Programming Reference (2012)
11. Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., Hinton, H.: Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In: USENIX Security, vol. 98 (1998)
12. Criswell, J., Dautenhahn, N., Adve, V.: KCoFI: complete control-flow integrity for commodity operating system kernels. In: IEEE Symposium on Security and Privacy. IEEE (2014)
13. Davi, L., Dmitrienko, A., Egele, M., Fischer, T., Holz, T., Hund, R., Nürnberger, S., Sadeghi, A.R.: MoCFI: A framework to mitigate control-flow attacks on smartphones. In: NDSS (2012)

14. Davi, L., Hanreich, M., Paul, D., Sadeghi, A.R., Koeberl, P., Sullivan, D., Arias, O., Jin, Y.: HAFIX: hardware-assisted flow integrity extension. In: Proceedings of the 52nd Annual Design Automation Conference. ACM (2015)
15. Davi, L., Koeberl, P., Sadeghi, A.R.: Hardware-assisted fine-grained control-flow integrity: towards efficient protection of embedded systems against software exploitation. In: The 51st Annual Design Automation Conference on Design Automation Conference. ACM (2014)
16. Davi, L., Lehmann, D., Sadeghi, A.R., Monrose, F.: Stitching the gadgets: on the ineffectiveness of coarse-grained control-flow integrity protection. In: 23rd USENIX Security Symposium (2014)
17. Evans, I., Long, F., Otgonbaatar, U., Shrobe, H., Rinard, M., Okhravi, H., Sidiroglou-Douskos, S.: Control jujutsu: on the weaknesses of fine-grained control flow integrity. In: 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM (2015)
18. Ge, X., Talele, N., Payer, M., Jaeger, T.: Fine-grained control-flow integrity for kernel software. In: 1st IEEE European Symposium on Security and Privacy. IEEE (2016)
19. Goktas, E., Athanasopoulos, E., Bos, H., Portokalidis, G.: Out of control: overcoming control-flow integrity. In: IEEE Symposium on Security and Privacy. IEEE (2014)
20. Göktaş, E., Athanasopoulos, E., Polychronakis, M., Bos, H., Portokalidis, G.: Size does matter: why using gadget-chain length to prevent code-reuse attacks is hard. In: 23rd USENIX Symposium (2014)
21. Guan, L., Lin, J., Luo, B., Jing, J., Wang, J.: Protecting private keys against memory disclosure attacks using hardware transactional memory. In: IEEE Symposium on Security and Privacy. IEEE (2015)
22. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures, vol. 21, pp. 289–300 (1993)
23. Intel: Control-Flow Enforcement Technology Review (Revision 1.0), June 2016
24. Jang, D., Tatlock, Z., Lerner, S.: Safedispatch: securing C++ virtual calls from memory corruption attacks. In: Symposium on Network and Distributed System Security (NDSS) (2014)
25. Mashtizadeh, A.J., Bittau, A., Boneh, D., Mazières, D.: CCFI: cryptographically enforced control flow integrity. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM (2015)
26. Mohan, V., Larsen, P., Brunthaler, S., Hamlen, K., Franz, M.: Opaque control-flow integrity. In: NDSS (2015)
27. Muttik, I., Nazshtut, A., Dementiev, R.: Creating a spider goat: using transactional memory support for security (2014)
28. Niu, B., Tan, G.: Monitor integrity protection with space efficiency and separate compilation. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. ACM (2013)
29. Niu, B., Tan, G.: Modular control-flow integrity. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM (2014)
30. Niu, B., Tan, G.: RockJIT: securing just-in-time compilation using modular control-flow integrity. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM (2014)
31. Niu, B., Tan, G.: Per-input control-flow integrity. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM (2015)

32. Pappas, V., Polychronakis, M., Keromytis, A.D.: Transparent ROP exploit mitigation using indirect branch tracing. In: 22nd USENIX Security Symposium (2013)
33. Payer, M., Barresi, A., Gross, T.R.: Fine-grained control-flow integrity through binary hardening. In: Almgren, M., Gulisano, V., Maggi, F. (eds.) DIMVA 2015. LNCS, vol. 9148, pp. 144–164. Springer, Heidelberg (2015)
34. Prakash, A., Hu, X., Yin, H.: vfGuard: strict protection for virtual function calls in cots C++ binaries. In: NDSS (2015)
35. Rajwar, R., Herlihy, M., Lai, K.: Virtualizing transactional memory. In: 32nd International Symposium on Computer Architecture (ISCA 2005). IEEE (2005)
36. Reinders, J.: Transactional synchronization in Haswell, February 2012
37. Ritson, C.G., Barnes, F.: An evaluation of intels restricted transactional memory for CPAS. In: Communicating Process Architectures (2013)
38. Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., Pike, G.: Enforcing forward-edge control-flow integrity in GCC & LLVM. In: 23rd USENIX Security Symposium (2014)
39. van der Veen, V., Andriesse, D., Göktaş, E., Gras, B., Sambuc, L., Slowinska, A., Bos, H., Giuffrida, C.: Practical context-sensitive CFI. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM (2015)
40. van de Ven, A.: New security enhancements in red hat enterprise linux (2004)
41. Xia, Y., Liu, Y., Chen, H., Zang, B.: CFIMon: detecting violation of control flow integrity using performance counters. In: 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE (2012)
42. Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., Zou, W.: Practical control flow integrity and randomization for binary executables. In: IEEE Symposium on Security and Privacy. IEEE (2013)
43. Zhang, M., Sekar, R.: Control flow integrity for cots binaries. In: 22nd USENIX Security Symposium (2013)