Efficient and Compatible CFI for x86-64 binaries

A Thesis presented

by

Rohit Chouhan

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Master of Science

in

Computer Science

Stony Brook University

May 2023

Stony Brook University

The Graduate School

Rohit Chouhan

We, the thesis committee for the above candidate for the

Master of Science degree, hereby recommend

acceptance of this thesis

R. Sekar
Professor, Computer Science

Michalis Polychronakis
Associate Professor, Computer Science

Dongyoon Lee
Assistant Professor, Computer Science

This thesis is accepted by the Graduate School

Celia Marshik

Interim Dean of the Graduate School

Abstract of the Thesis

Efficient and Compatible CFI for x86-64 binaries

by

Rohit Chouhan

Master of Science

in

Computer Science

Stony Brook University

2023

Control-Flow Integrity (CFI) is a low-level security policy that can serve as the basis for secure and non-bypassable instrumentation. Protecting backward-edge (returns) through a coarse-grained CFI is insufficient, as previous studies have shown that it can be easily bypassed. Shadow stacks provide fine-grained protection, but binary-instrumentation based shadow stack protection suffers from significant performance overheads.

In this thesis, we present a new binary-instrumentation based CFI technique. It combines coarse-grained CFI for forward edges (indirect calls and jumps) with a shadow stack for protecting backward edges. For forward-edge protection, we compare two alternative techniques, one based on runtime address translation and another based on statically replacing code pointers with array indices. For backward-edge protection, we develop a novel way to piggyback an efficient shadow stack on top of the stack canary mechanism that is already included in most binaries. This combination can achieve a combined performance overhead of about 7% on the SPEC 2006 benchmark suite.

Dedication

I dedicate this thesis to my mother, Archana, the kindest and most selfless person I know.

To my father, Jaydeep, for his unwavering trust and support in all of my endevours.

Finally, I would like to dedicate this thesis to my grandmother for teaching me the joy of giving, to my maternal grandfather, for inspiring me to be a good human being. I also extend my dedication to my brother for his support and to God for helping me all along the way.

CONTENTS

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Professor R. Sekar, for their invaluable advice, guidance, and support throughout my Masters degree. Their expertise and mentorship have been instrumental in shaping the success of this research.

I would also like to extend my heartfelt thanks to Soumyakant Priyadarshan, who has been a great mentor to me. Working alongside Soumyakant has been a truly rewarding experience, and their guidance and insights have contributed significantly to my growth as a researcher.

Additionally, I am grateful to my colleagues Huan and Hanke for their encouragement and support throughout this journey. Their camaraderie and willingness to lend a helping hand have made this experience more enjoyable and fulfilling.

1 Introduction

Control-Flow Integrity (CFI) [1, 2, 3, 52, 8, 51, 54, 25] is a low-level security mechanism that prevents unintended transfer of control flow. It is achieved by enforcing a set of rules that do not allow a program to transfer control flow to unintended targets. CFI can mitigate the threat from code-reuse attacks such as Return-Oriented Programming (ROP) [37] and Jump-Oriented Programming (JOP) [10, 4], which typically involves corrupting a code pointer to divert the control flow of the program to chained gadgets (set of instructions) in the address space of the program. CFI also lays a foundation for secure and non-bypassable instrumentation [53, 50] enabling other protection mechanisms such as Software Fault Isolation (SFI) to be built on top.

A complete CFI solution will protect both forward edges such as indirect calls and backward edges e.g. returns. Coarse-grained forward-edge policies are effective in stopping a majority of attacks on forward edges. However, control-flow hijacking attacks like ROP that target backward edges are still viable even with a coarse-grained backward-edge policy in place [8, 13, 18]. A precise mechanism like shadow stack [7, 34, 12, 40, 15, 14, 30, 29, 26] is needed to effectively mitigate attacks on backward edges. A shadow stack guarantees the integrity of a return address by storing a second copy of the return address in an isolated memory region. At every call instruction, a copy of the return address is pushed onto the shadow stack. Then, at every return instruction, the return address on the top of the stack is compared with the second copy stored on the shadow stack. On a mismatch, the program is aborted to prevent a control-flow hijack. Although the idea of shadow stack has been around for a long time, its wide-scale deployment is limited due to performance overhead and compatibility issues with features such as exception handling and longjmp.

We propose a novel binary-instrumentation based shadow stack design that makes use of the stack canary slot on the program stack to maintain a direct pointer to the shadow stack. The design of our shadow stack is guided by factors such as compatibility, performance, and modularity. We measure the performance of our shadow stack using SPEC CPU 2006 benchmark suite. Our shadow stack does not require any changes to glibc or other libraries.

To be an effective defense mechanism a shadow stack has to be combined with a forward-edge CFI. Therefore, we build our shadow stack on top of a coarse-grained forward-edge CFI. Our approach works by keeping instrumented code in a new code segment different original code segment. This means that pointer constants in the new code would still point to the original code. Since there is no reliable way to distinguish code pointers from pointer constants, we follow an approach similar to BinCFI [54]. By waiting until a value is used as a target in an Indirect Control-Flow (ICF) transfer and then performing the fix-up at the runtime by finding the new code address corresponding to the original address. This is achieved by maintaining a table that maps the original address to its corresponding new address. This is called runtime address-translation and it is useful for the following reasons:

1. Ensures CFI by allowing ICF transfers to only go to predetermined targets from the lookup table.
2. It ensures correct instrumentation of binaries by doing runtime translation instead of

static translation. Static translation depends on the identification of code pointers, which is not 100% accurate.

Address translation based schemes usually make use of a hash table to map the old code address to the new code address. Hence, the performance of address translation based schemes would depend on the performance of its hash table lookup. However, a systematic study on the performance limits of address translation has not been done. Beyond proposing our novel shadow stack design, we also investigate ways to improve the performance of address translation technique to help make address translation based CFI implementations more efficient. We also compare address translation with a faster but less robust array index-based scheme that relies on static code pointer modification.

We make the following contributions in this thesis:

- We significantly improve the performance of binary-based shadow stack implementation by piggybacking over the canary instrumentation that is already part of most compiled binaries. In addition, our scheme avoids the compatibility challenges faced in shadow stack implementations, such as those due to exception handling and long jumps.
- We compare an index-based CFI scheme with one based on the address translation technique used in binary instrumentation systems such as DynamoRio [6], Pin [31] and BinCFI [54]. The index-based approach requires all code pointers to be statically transformed into indices, which are then used at runtime to index into a table of actual targets. In this thesis, we explore the limits of performance that can be achieved using address translation and compare it with that of the index-based approach.

2 Related work

2.1 Control-Flow Integrity

Ever since its introduction by Abadi et al. [1], CFI has been a fundamental security feature that restricts the control-flow of an application to desired paths. It works by limiting the targets that an Indirect Control Flow (ICF) transfer can take. CFI also enables the implementation of other security mechanisms such as Software Fault Isolation (SFI).

PittSFIeld [23], Native Client [49], and other works [21, 32] that enforce CFI often rely on compiler-provided information, e.g., relocation information. BinCFI [54] brings a binary-level coarse-grained CFI for stripped COTS binaries on 32-bit Linux. Since, it uses binary instrumentation, newly added instructions will cause the address of all code pointers to change. To fix this, code pointers need to be changed to their new addresses. BinCFI uses an address translation [6] based approach for translating code pointers by doing a fixup at runtime. This translation is performed by using a table that maps the original address to the new address. BinCFI does not rely on any compiler-provided information and can work with stripped binaries.

BinCC [44] builds on top of BinCFI to improve upon the tightness of CFI policy. It does so by dividing the code into mutually exclusive continents and then applying strict policies for both inter and intra-continent transfer.

CCFIR [52] is a binary-level CFI solution for Windows binaries. It works by redirecting all indirect control-flow transfers through a springboard section and limiting indirect control-flow transfers to it. Typearmor [43] focuses on reducing the number of possible targets for an indirect call site. It is a binary-based solution and works by doing a static analysis to infer the type of a function, i.e., the type of its arguments and return values. This information is matched with the calling context for an indirect call, enabling the target set to be narrowed to those functions whose argument types match those at the calling site.

Works such as CFCI [29, 55] deploy fine-grained CFI but it is shown that even fine-grained CFI can be bypassed [8].

2.2 Shadow stack

Since coarse-grained policies are insufficient in protecting backward edges [8, 13, 9]. The idea of using a precise mechanism such as a shadow stack for defending against ROP attacks has been for a long time. Shadow stack was first proposed as a defense against stack-smashing attacks [17, 33].

Compiler-based [41, 7] as well as binary instrumentation [33, 34, 53] based implementations have been presented in the past. Dynamic binary instrumentation [5, 6, 16, 21, 22, 24, 36] based approaches like ROPdefender [14] handled compatibility issues like C++ exception handling and use of non-standard return in lazy binding but suffered from significant runtime overhead.

PSI [53] shows a very simple implementation of a shadow stack using static binary instrumentation [19, 35, 39, 54]. It shows an elegant approach for initializing shadow stacks for all the threads by checking the validity of the shadow stack before pushing a value on it. This approach does not require any library changes. The paper by Qiao et al. [34] builds a complete ROP defense solution on top of PSI. It achieves a very good performance while maintaining compatibility with non-standard calls/returns, threading mechanisms, etc.

Dang et al. [12] introduce parallel shadow stack design which is proposed as a faster alternative to traditional shadow stacks. It eliminates the need to maintain a shadow stack pointer by storing the shadow stack at a fixed offset from the program stack. Authors note that their approach is not intended for real-world deployment but to measure the performance cost of a shadow stack.

Burow et al. [7] summarizes all the possible shadow stack mechanism based on factors such as performance, security, and compatibility. It proposes an LLVM-based compact shadow stack implementation that makes use of a dedicated register to store the shadow stack pointer. Multi-threading support is achieved by preloading a support library. In the case of setjmp/longjmp use, synchronization is maintained by popping the shadow stack until the value at the top of the shadow stack matches the return address on the program stack. It also shows that shadow stack integrity protection remains an open challenge as none of the hardware-based techniques they tested provided low enough overheads to be ready for real-world deployment and the authors suggested using information hiding as protection mechanism.

3 Backward-edge CFI

Seminal work by Aleph One [27] demonstrated how an attacker can overflow a local variable on the stack to overwrite the return address and hijack the control-flow of the program. To detect buffer overflow modern compilers place a special value called the stack canary [11] between the return address and local variables. A contiguous buffer overflow would also overwrite the stack canary, which is checked before a function returns. Unfortunately, information leak attacks can bypass stack canaries easily, subverting the defense checks. Nevertheless, canaries have seen a wide-scale deployment due to low overheads of less than 1%. To encourage the adoption of stronger backward edge protection, the mechanism should only incur comparable overheads.

Shadow stack [10] is a fine-grained mechanism used to enforce the integrity of return addresses by keeping a second copy in an isolated memory region. Before returning from a function the return address is checked against a second copy stored on the shadow stack. Since it is hard for attackers to simultaneously corrupt the return address at two locations, shadow stacks can provide stronger protection than canaries.

Some of the key challenges in the implementation of shadow stack as mentioned by previous works [34, 7, 12] are:

- Non-standard use of call and return instructions: Functions calls and returns do not always maintain symmetry. This can happen with longjmp, exception handling, etc.
- Compatibility with multi-threaded programs: Each thread maintains its stack, so a shadow stack needs to be maintained for each of the program threads.
- Integrity of the shadow stack memory pages: Shadow stack pages should be protected so that the attacker cannot modify them.
- Performance of shadow stack protected program.

Many implementations of shadow stacks have been presented over the past years. Previous works [7, 34] use repeated popping to bring the shadow stack back in synchronization with the program stack when the stack unwinds multiple frames and call-ret symmetry is violated. We propose a new technique for maintaining the synchronization between the two stacks, without needing iteratively pop the shadow stack. We piggyback the canary instructions and canary slot on the program stack to store the address of the return address copy of the stack frame.

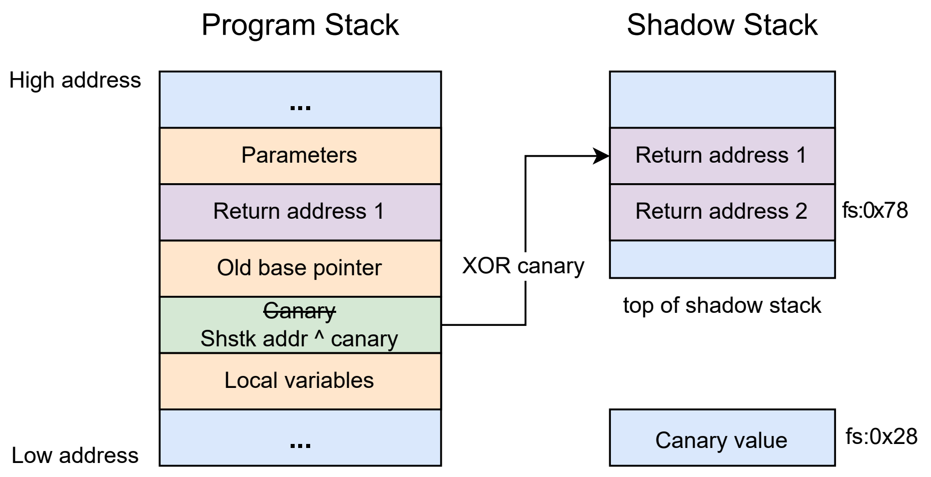

Figure 3.1: Program stack setup with a pointer to corresponding shadow stack entry.

3.1 SHADOW STACK DESIGN

Design choices for a shadow stack determine important characteristics such as compatibility, integrity, and performance. We describe the major design decisions of our shadow stack below.

SYNCHRONIZATION: In x86, a call instruction pushes the return address on the program stack and a corresponding ret instruction pops the return address and jumps to it. This call-ret symmetry is violated in cases where setjmp/longjmp and C++ exception handling is used – causing the stack to unwind multiple frames. This leads to the problem of the shadow stack going out of synchronization with the program stack, so the return address stored on the program stack does not match the return address on top of the shadow stack. Previous works usually solve this issue by repeatedly popping the shadow stack until the top entry matches to program stack return address. This approach is imprecise, and, can be confused in the presence of recursion. To solve this, one can also push the program stack pointer along with the return address to have a unique identification of the correct stack frame to unwind the shadow stack.

We propose a solution to this problem by making use of the canary slot to link a stack frame with the location in the shadow stack that holds the return address of that frame. At function return, this pointer can be used to synchronize the shadow stack with the program stack after an operation such as a longjmp. Figure 3.1 shows this setup.

COMPATIBILITY: With every call instruction, the return address has to be pushed to both the program stack and the shadow stack, and before every return instruction the return address on the program stack needs to be compared with the return address stored on the shadow stack. Instrumentation for saving the return address on the shadow stack could be done either at the call site or in the function prologue. Instrumenting the call site breaks

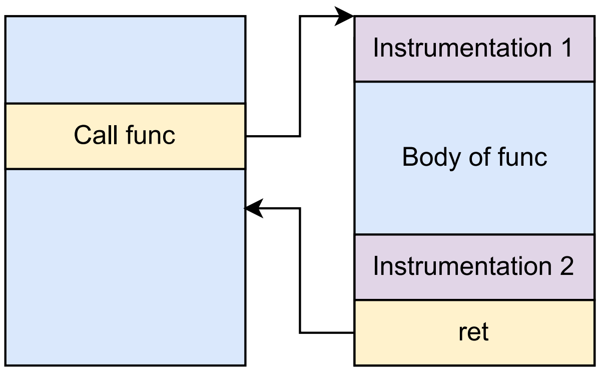

Figure 3.2: Callee-only instrumentation scheme.

the call-ret symmetry in cases when there is an atypical use of call instruction, e.g., using a call to compute code-relative address on 32-bit systems. By choosing to do all the instrumentation inside a callee, stack call-ret symmetry is always ensured. The instrumentation locations are shown in Figure 3.2. This also enables mixing unprotected functions with protected functions because a call would not have to distinguish between a shadow stack protected and a non-protected function. Furthermore, we can selectively instrument functions for shadow stack protection, assisting with incremental deployability.

We instrument function prologues with a shadow stack push instruction and function epilogue with a shadow stack pop instruction. For ensuring compatibility with multi-threaded programs, we initialize one shadow stack for each thread. The shadow stack pointer for each shadow stack is stored in thread-local storage (TLS).

INTEGRITY: A shadow stack is essentially a read/write metadata for inline checks for backward-edge CFI. If an attacker can manipulate values on the shadow stack, our checks would be rendered useless. Hence, we need a mechanism to maintain the integrity of our shadow stack. Two techniques have been used in the literature:

- Information hiding: This is an attractive option for performance but is less secure. Integrity of the shadow stack depends on keeping its location hidden from an attacker.
- Write protection of shadow stack pages: This offers much better security at the cost of increased overheads.

Burow et al. [7] shows that even hardware-supported integrity mechanisms such as Intel Memory Protection Keys suffers high overheads. Hence, we have decided to go with information hiding.

To protect the address of the shadow stack stored on the program stack, we **xor** it with the stack canary value and then store it at the canary slot on the program stack. We store the address of the top of the shadow stack in TLS. Essentially, knowing this gives someone permission to read/write from the shadow stack. Refer to Figure 3.1.

3.1.1 Security analysis

Threat model: We assume that the attacker can read or write arbitrary memory locations, but does not know the location of TLS. The target system is running a benign but vulnerable program with memory vulnerabilities. Defenses like DEP and ASLR [42] are enabled and stack canaries [11] are no longer deployed since we have repurposed their code for our shadow stack implementation. An attacker is interested in corrupting a code pointer specifically by targeting a backward-edge by overwriting the return address on the stack.

Shadow stack integrity: The protection offered by the shadow stack mechanism depends on the integrity of the shadow stack values. An attacker should not be able to control shadow stack values. There are two approaches taken by past works to protect the shadow stack. (i) Write protected shadow stack (ii) Hidden shadow stack (information hiding). In the case of a write-protected shadow stack, the memory pages containing the shadow stack are unwritable and only certain privileged instructions are allowed to write to the shadow stack. With information hiding, knowing the location of the shadow stack gives access to read and write to it. It is faster since there is no permission switch but offers less security. We have opted to go with information hiding as the technique to protect the integrity of our shadow stack. We store the shadow stack address **xor**ed with the stack canary value on the program stack. So, even if the attacker can leak the stored value on the stack, it would not be of any help without knowing the canary value. Since the attacker does not know the location of TLS, it is very hard to get the value of the canary from TLS.

TOCTTOU attacks: Time Of Check To Time Of Use (TOCTTOU) attacks are a type of security vulnerability where the attacker takes advantage of a window of time between the check and use of a resource. There are two TOCTTOU opportunity windows in our proposed design:

1. In x86 call instruction pushes the address of the next instruction onto the stack. This can be modified by the attacker before it is picked up by our instrumentation in function prologue.
2. Another window is at function return, the return address may be correct at the time of the shadow stack epilogue check but can be modified by the attacker before the function returns.

   However, this vulnerability is very difficult to exploit as mentioned by Zhang et al. [51]. To carry out a TOCTTOU attack the attacker needs to be in control of a thread that is racing with a thread of a vulnerable program. Since this is not possible with our assumed threat model we leave it out as a possible attack.

3.1.2 Implementation

Initializing the shadow stacks: To maintain compatibility with multi-threaded programs we initialize a shadow stack for each thread. A new shadow stack should be initial-

```
    mov %fs:0x28,%rax
    mov %rax,0x28(%rsp)
```

```
    xor %fs:0x28,%rcx
    jne stack_chk_fail
```

Figure 3.3: Original canary prologue (left) and canary epilogue (right).

ized whenever a new thread is spawned. This could be achieved by potentially changing glibc. Since our focus is on ensuring maximum compatibility while being efficient we select a less efficient but more robust option to check the status of the shadow stack at shadow stack insertion code, an approach similar to PSI [53]. A **cmp** instruction is used to check the initialization status of the shadow stack. If not initialized the shadow stack is initialized and the memory address of the shadow stack is stored in the TLS slot reserved for the hardware-based shadow stack (currently unused).

REPURPOSING CANARY CODE: Proposed shadow stack design depends upon the shadow stack address saved on the program stack at the canary slot to maintain synchronization between the stacks. By default, GCC does not put canary checks in all the functions, and finding a usable slot on the stack can be very difficult for complex binaries. Hence, to expedite and simplify implementation we compiled binaries with canary checks for all functions using the flag *-fstack-protector-all.*

Stack canary related code in a function is generally divided into two parts: (i) canary prologue (ii) canary epilogue. As shown in Figure 3.3 canary prologue is reading the canary value from TLS and storing it on the stack. The epilogue performs an equality check, usually by performing a **xor**, and aborts the program if the values do not match. This is analogous to what we want to achieve with shadow stacks. Hence, canary code can essentially be piggybacked without shadow stack instrumentation.

CANARY PROLOGUE INSTRUMENTATION: Canary prologue is responsible for storing the canary value on the stack. This is replaced by a sequence of instructions that first checks for the initialization of the shadow stack and then copies the return address on it as shown in Figure 3.4. We do static analysis to find the return address on the stack. The instruction following the added instructions moves the xored shadow stack address at the canary slot on the program stack.

CANARY EPILOGUE INSTRUMENTATION: We describe a canary epilogue as the set of instructions responsible for checking the equality of the canary value on the stack and with the canary value stored in TLS. This site is instrumented to add instructions for the synchronization of stacks by setting the top of the shadow stack equal to the shadow stack address stored at the canary slot. The canary check is disabled by doing **xor** of equal values to set the flag registers right. The pseudo-assembly for this is shown in Figure 3.4.

```
cmp $0,%fs:0x78
jne .shstk_ok
call .init_shstk
.shstk_ok:
add $8,%fs:0x78
push extra_reg
mov 0x28(%rsp),extra_reg
mov %fs:0x78,canary_reg
mov extra_reg,(canary_reg)
xor %fs:0x28,canary_reg
pop extra_reg
```

```
xor %fs:0x28,canary_reg
mov canary_reg,%fs:0x78
xor canary_reg, canary_reg
```

Figure 3.4: Instrumented canary prologue (left) and canary epilogue (right).

```
push %rdi
push %rsi
mov 16(%rsp),%rdi
mov %fs:0x78,%rsi
sub $8,%fs:0x78
cmp (%rsi),%rdi
jne shstk_abort
pop %rsi
pop %rdi
```

Figure 3.5: Instrumented function return.

FUNCTION RETURN INSTRUMENTATION: At function returns, we add instructions to check if the return address on the program stack matches the return address stored on the shadow stack, as shown in Figure 3.5. If the return addresses do not match the function, *abort_shstk* is called which prints some debugging information. No additional shadow stack unwinding or synchronization operations are needed here since we already did that in the canary epilogue instrumentation preceding the *return* instruction.

Instrumentation for synchronization of the shadow stack and checking the integrity of the return address is split into two parts to eliminate the requirement of doing a static analysis in the canary epilogue for finding the location the of return address on the program stack. Since, at a return instruction the stack pointer would be pointing to the return address.

Our shadow stack implementation works with a modest overhead of 4.92% on SPEC CPU 2006 binaries. We present a detailed evaluation in Chapter 5.

3.2 Intel CET shadow stack

Intel Control-Flow Enforcement Technology (CET) [38] is a hardware-supported defense mechanism against JOP and ROP attacks. It is supported starting from 11th generation Intel processor and requires compiler, glibc, and Kernel support which is not available at the moment. Full support for hardware features usually takes time, and their deployment is often slowed down because of compatibility issues. CET has two components:

1. Indirect Branch Tracking (IBT)
2. Shadow Stack (SHSTK)

IBT is a coarse-grained forward-edge hardware-based check. All valid indirect targets start with a newly introduced instruction – *endbr32/64*. A compiler is responsible for adding *endbr* instructions in a binary to make it IBT compatible. SHSTK is a hardware-based shadow stack implementation that implements shadow stack using hardware and operating system support. When CET SHSTK is enabled, **call** and **ret** instructions automatically push and pop the return address from a protected shadow stack in the memory. The shadow stack is marked as a special type of page by the operating system. Access to the shadow stack memory pages is only possible by CPU through **call** and **ret** instructions. In some special cases, an instruction **wrss** can be enabled and used to write to the CET shadow stack.

3.2.1 Problems

Problems with Intel CET that motivate a software-based shadow stack implementation are summarized below:

- Intel CET is only supported on processors starting from 11th generation. This leaves older hardware without any support.
- It requires a compatible operating system to work – Linux does not fully support Intel CET yet.
- It requires support for glibc and GCC to be compatible with features like longjmp and C++ exception handling.

To test the current state of CET, we installed a patched Kernel on a 12th generation Intel machine. Even with supporting a kernel, shadow stack is not enabled by default. According to Linux Kernel CET support design, this task of requesting shadow stack should be done by glibc for supported applications Since GCC and glibc do not support the latest Linux Kernel CET design, shadow stack does not work out of the box. For testing, we modified the source code of C and C++ SPEC CPU 2006 programs to issue an arch_prctl call to request shadow stack at the beginning of their main function. While being very efficient with just 0.41% overhead it suffers from compatibility issues with programs that use longjmp and exception handling. Intel CET shadow stack performance numbers are presented in Chapter 5.

4 Forward-edge CFI

Protecting forward-edge control-flow transfer instructions like indirect calls and jumps are called forward-edge CFI. It is a precursor to having an effective shadow stack defense. If an attacker is simply able to bypass the shadow stack checks, the mechanism does not provide any security. In this section, we will focus on improving the performance of the forward-edge CFI scheme. Specifically, we explore different hashing schemes and setups for improving the hash table lookup performance. We also introduce an index-based scheme for CFI, which aims to reduce overhead while ensuring CFI. A study like this has been missing in the address translation based CFI literature [6, 54, 44], and we believe that our findings would benefit future address translation-based works.

4.1 Address translation based CFI

In an instrumented binary new code is added, which causes the location of the instructions to shift. Therefore, code pointers need to be updated to their new address, this process is called address translation by the literature. During static analysis, one cannot be sure whether an integer constant is a code pointer or some other type of constant. Therefore, pointer fix-ups must be done at runtime. A typical way to do this, as shown in DynamoRIO [6] and BinCFI [54], is to wait until the constant is used in an Indirect Control Flow (ICF) transfer instruction. At that time, a lookup is performed and the target value is replaced by the corresponding new address from the lookup table. Since a target is now restricted to entries stored in the lookup table, this scheme ensures CFI. However, any additional runtime operation adds to the overhead of the running program, making the performance of address translation critical to the performance of the instrumented binary. BinCFI and BinCC [44] use a hash table to implement a lookup table. However, there has been no systematic study on the performance of address translation with different hash table schemes and setups. Performance overhead is often the biggest factor that determines the deployment of a hardening mechanism. We aim to test the limits of address translation performance without sacrificing compatibility. We evaluate different hashing schemes and hash table setups in hopes of finding a more efficient approach.

4.1.1 Hashing algorithm

The choice of hashing scheme is an important factor in the performance of the address translation routine. Different collision avoidance schemes have different insertion, search, and deletion costs. The choice of the hashing scheme depends on the use case.

Separate chaining stores collided keys separately in a linked list. It works well with bad hash functions and high load factors without needing to be resized but is not very cache efficient. On the other hand, open addressing [47] requires extra care with the choice of hash function and load factor but provides better cache performance than separate chaining. For our use case, open addressing would work better since we require to have fast a lookup performance. In the open addressing scheme a collision is resolved by probing the next slots. Some of the most popular probing techniques are linear probing, quadratic probing, and double hashing.

In linear probing [46] collisions are handled by scanning the next slots in the hash table sequentially. Linear probing suffers from the problem of primary clustering where many consecutive elements form a group increasing the time taken to find a slot (the number of probes increase). Quadratic probing [48] is an upgrade over linear probing as it operates by adding successive values of an arbitrary quadratic polynomial until an open slot is found. It can avoid the problem of primary clustering. Double hashing [45] makes use of two hash functions, one hash function is used to calculate the initial value, and the second hash function is used to calculate the step size in the probing sequence. It is the most effective form of probing producing uniform distribution of records throughout the hash table. However, calculating two hashes per lookup is expensive and makes the implementation complicated. Quadratic probing strikes a good balance between ease of use and performance, so it promises to be a good option for our use case. However, in the worst case, quadratic probing can have a O(n) lookup cost. To improve upon this, we test Cuckoo hashing [28], which guarantees a worst-case cost of just 2 lookups, it achieved this by storing keys in two separate tables. The scheme guarantees that every key is stored in its ideal slot in one of the two tables. The ideal slots are calculated using two different hash functions. Cuckoo hashing does not probe any slots other than ideal slots for a given key. Quadratic probing and cuckoo hashing are the candidates for our efficient hashing scheme. Hence, we test their performance to see if cuckoo hashing is an improvement over quadratic probing.

Hypothesis: Cuckoo hashing should perform better than quadratic probing.

IMPLEMENTATION OF QUADRATIC PROBING: Implementation of quadratic probing is shown in Figure 4.1. Quadratic probing is implemented with the following probing function:

$p(i) = 0.5i(i+1) mod N$

Where N is the size of the hash table, this can also be computed incrementally by -

$p(i) = (p(i-1)+i) mod N$

In other words, we can simply add i to the last slot that was tried, with i going from 1 to tabsize - 1. When the size of hash table N is a power of 2, the period of probing becomes equal to the size of the table [20].

IMPLEMENTATION OF CUCKOO HASHING: Cuckoo hashing implementation is shown in the Figure 4.2. The insertion works in the following manner:

1. Check if the first slot is empty, if it is empty the key is stored there.
2. If the first slot was already occupied the stored element is evicted and the key is stored there.
3. The evicted item is then inserted into the second table by following the same procedure.
4. This process continues until an empty position is found to store the key.

```
function insert(x) is
  if lookup(x) then
    return
  end if
  if T[h(x)] is empty then
    T[h(x)] = x
    return
  end if
  else
    loop 1 to T_SIZE - 1
      idx = (idx + 1) &
          (T_SIZE - 1)
      if T[h(idx) is empty]
          then
        T[h(idx)] = x
        return
      end if
    end loop
end function
```

```
function lookup(x) is
  if T[h(x)] == x
    return
  end if
  else
    loop 1 to T_SIZE - 1
      idx = (idx + 1) &
          (T_SIZE - 1)
      if T[h(idx)] == x then
        return
      end if
    end loop
end function
```

Figure 4.1: (Left) Quadatic probing insert function and (Right) lookup function.

It is possible to run into infinite loops where a cycle of evictions is repeated and the hashing is not able to generate a hash table. To exit from that case, the iteration loop is only repeated a MAX_LOOP number of times. A cuckoo hash might require multiple rehashes before it can create a hash table. The choice of hashing function and the load factor also determines the number of rehashes required by cuckoo hashing. We evaluate the performance of insertions and lookup between the two schemes.

EXPERIMENTAL SETUP: An experiment is set up to perform a large number of insertions and lookups using cuckoo hashing and quadratic probing. The keys used are pointer constants from SPEC binaries that a translation table would contain. We conduct tests with two different load factors of 0.4 and 0.8. We test with two different hashing functions, one with a 1-degree of independence and the other with a 2-degree of independence. The hash functions used are shown in Figure 4.3.

RESULTS: It is observed that with a load factor of 0.4 and a simpler hashing function with 1-degree of independence cuckoo hashing failed to generate a hash table in the first attempt. When switching to a hashing function with 2 degrees of independence cuckoo hashing was able to generate a hash table but it takes 20% more time than quadratic probing. With a load factor of 0.8 cuckoo hashing fails to work with both hashing functions while quadratic probing still works robustly. Cuckoo hashing requires more hashes per insert,

```
function insert(x) is
  if lookup(x) then return
  end if
  loop MAX-Loop times
    if T1[h1(x)] is empty
        then
      T1[h1(x)] = x
      return
    end if
    swap(x, T1[h1(x)])
    if T2[h2(x)] is empty
        then
      T2[h2(x)] = x
      return
    end if
    swap(x, T2[h2(x)])
  end loop
  rehash(x)
  insert(x)
end function
```

```
function lookup(x) is
  return T1[h1(x)] == x or
      T2[h2(x)] == x
end function
```

Figure 4.2: (Left) Cuckoo insert function and (Right) lookup function.

```
(key * random) &
    hash_table_size
```

```
key(key * random1 + random2)
    & hash_table_size
```

Figure 4.3: (Left) 1-degree independence hash function and (Right) 2-degree independence hash function.

due to its eviction-based strategy. On the other hand, our quadratic probing based approach can do probing with a simpler iterative approach as described in the last section. Refer to Table 4.1 for results.

Even with lookups cuckoo hashing takes 24% more time than quadratic probing. Refer to Table 4.2. This is because cuckoo hashing lookup will require 1.5 hashes per lookup (assuming keys are evenly distributed between two tables). Also, the memory accesses to these buckets are unlikely to fit on a single cache line. Compare this to quadratic probing where only one hash is required and potential probes are likely to be in the current cache line – cuckoo hashing is not very cache efficient.

CONCLUSION: Cuckoo hashing looks like a better approach on paper but its unreliable insertion and cache inefficiency make it worse than quadratic hashing for use in an address translation table. Hence, we recommend using quadratic probing with a simple 1-degree

Hash function	qprobe | LF=0.4	cuckoo | LF=0.4	qprobe | LF=0.8	cuckoo | LF=0.8
1-degree	5.26	failed	4.95	failed
2-degree	5.94	7.14	5.46	failed

Table 4.1: Insertion cost of cuckoo hashing and quadratic probing. LF stands for load factor.

Hash scheme	Time
Qprobe	6.03
Cuckoo	7.52
% Overhead	24.70

Table 4.2: Lookup cost of cuckoo hashing and quadratic probing.

of independence hashing function.

4.1.2 Hash table setup

In BinCFI [54], the translation function first performs a check to determine whether the target is within the current module. If so, the lookup is performed on the current module translation table. If the target is from a different module, it goes through a two-stage process. In the first stage, a global table is used to dispatch the lookup on the correct module. The top-level table in BinCFI is implemented as an array because on 32-bit systems, only 1M pages are possible and the lookup table entry would contain a mapping from a page address to a particular module of the running program. Shared libraries are page-aligned therefore they would at least be one page apart, this ensures that a page cannot be part of two modules.

We cannot create this type of array top-level hash table on 64-bit systems, since the number of possible pages is 2^{52}, far more than the maximum addressable range of 2^{48} on 64-bit systems. So, Instead of maintaining a top-level array like BinCFI, another hash table needs to be created. Thus, address translation for an inter-module target would require two lookups on two different hash tables.

An alternative to this setup is to create a single hash table with all targets. A lookup on the single hash table should save one memory access and one hash table lookup. However, when most of the transfers are intra-module and the number of pointers in the current module is small relative to the total number of pointers of the binary, then lookup on a smaller table in a two-level scheme can be faster than a one-level setup.

Hypothesis: One-level hash table setup should be faster than a two-level setup.

Implementation: In a one-level hash table, only a single hash table is created for all the modules of a binary, the table is created at runtime when the program is being loaded by the loader. For a two-level hash table, different hash tables are statically created for different modules of a program. A top-level hash table maps page addresses to respective modules.

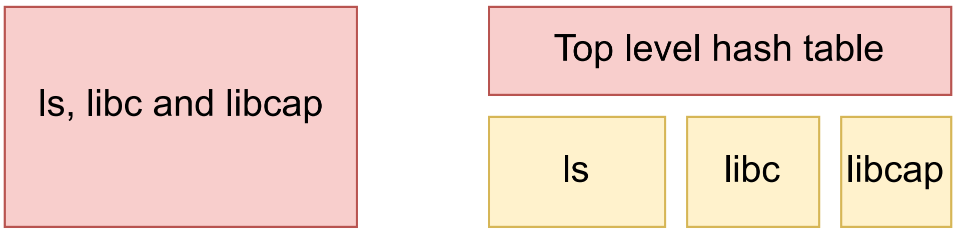

Figure 4.4: (Left) One-level hash table setup and (Right) two-level hash table setup.

After lookup on the top-level hash table, another lookup is then performed on the per-module hash table. The setup of both hash table schemes is shown in Figure 4.4. One-level hash table for a binary such as **ls** would have entries for all the pointers in a single table. For a two-level hash setup, there are tables for each module and a top-level hash table to dispatch the lookup on a per-module hash table.

EXPERIMENTAL SETUP: To compare the performance of a one-level hash table with a two-level hash table setup we created an experiment that uses pointers from SPEC binaries and their libraries. First, we generate the hash table(s) for both of the schemes and then perform lookups 1000 times the number of total entries. This is to ensure that the cost to insert is not a factor when comparing the performance difference between lookups. We test the two schemes with different ratios of inter-module to intra-module pointers and two different load factors of 0.5 and 0.8. The idea behind using a high number of intra-module pointer lookups as compared to the total number of lookups is, during a typical run of a binary a majority of ICF targets are usually within the module. Specifically, we test configuration with 90% to 100% of intra-module targets. The first column of Table 4.3 show that.

RESULTS: As we can see in Table 4.3 that one-level hash table is faster than a two-level table in all the cases. As the fraction of intra-module pointers increases the performance of both the schemes improves. For a one-level setup, this increase in performance is attributed to decreased collision/lookup, and for a two-level setup, this is because the lookup only needs to be performed on a single hash table. Surprisingly, both of the schemes perform better with a higher load factor. We find that this is due to better caching of a smaller hash table dominating the effect of increased collisions/lookup. In case, where all the targets are intra-module the time taken by the one-level scheme should be equal to the time taken by the two-level scheme but the numbers in the fourth row do not support our assumption. This is because when using a similar load factor for both the schemes, say 0.8. The size of a hash table in the case of a one-level table would be larger than that of the size of a two-level table for the main module because in a one-level scheme, a single table would contain all the pointer targets. One-level setup with a load factor of 0.8 and a two-level setup with a load factor of 0.5 achieves the same number of collisions/lookup and the same size of hash table but still shows very different times. We found that the comparison made in the two-level setup that dispatches the lookup to the current module hash table is

Intra/Total	1-level | LF=0.5	1-level | LF=0.8	2-level | LF=0.5	2-level | LF=0.8
0.9	9.48	8	13.02	10.7
0.95	9.13	7.82	11.81	9.98
0.99	9.07	7.67	10.4	8.77
1	9.03	7.62	10	8.53

Table 4.3: Comparison of one-level and two-level hash table implementation. LF stands for load factor.

```
if isEncoded(ptr) then
  // decoding logic
  // ...
else
  addr_trans(ptr)
end if
```

Figure 4.5: Index-based address translation.

responsible for the observed overhead. On removing the comparison, we found the performance to be similar.

CONCLUSION: One-level hash table implementation is a better option as it performs better than a two-level setup for both load factors.

4.2 INDEX-BASED CFI

We propose an alternative to address translation that avoids a table lookup while enforcing CFI. We identify code pointers and encode them with their index in the translation table. In particular, we have a per-module translation table called Local Translation Table (LTT) and a Global Translation Table (GTT). GTT maintains information like LTT address, the load address of the module, size, etc. We modify the address translation function to perform a check to see if the pointer is encoded. For an encoded pointer, decoding it would give global and local indices in GTT and LTT respectively, which can be used to get to the LTT entry which stores the respective new pointer. Instead of performing hash table lookup, the operation now reduces to two array indexing operations.

Encoding a code pointer depends upon the correct identification of code pointers. Static analysis cannot guarantee error-free code pointer classification. Position Independent Executable (PIE) binaries on x86-64 Linux have relocation information that could be used to identify all pointer constants, but finding a subset of code pointers from it relies on some assumptions such as the absence of data in the middle of the code. So, this approach is vulnerable to both false-positives and false-negative in code pointer classification.

4.2.1 Inlining of decoding routine

As discussed above, in index-based CFI we encode the code pointers with LTT and GTT array indices. Since PIE binaries are widely used today, we can encode almost all of the pointers. Instead of keeping the decoding routine in the address translation function we choose to inline it at all the indirect calls. This would save a call to the translation function and improve performance. The pseudocode of instrumentation at an ICF with inlined decoding function would look like Figure 4.5. Our index-based scheme with inlined decoding routine can reduce the overhead of address translation by 67%. A discussion of the evaluation is done in Chapter 5.

4.3 Implementation

Binary Instrumentation: Our instrumentation works by leaving the original code in place and storing the instrumented code in its own section. The original code section is marked as read-only while the instrumented code section is made executable. An LTT is created inside every instrumented binary. This table contains entries of all possible code-pointers and their corresponding new pointer.

Every ICF transfer site is instrumented to jump to the translation function which is replicated inside each instrumented module. In the case of address translation based CFI the translation function performs a lookup on the one-level hash table which returns the GTT and LTT indices of the code-pointer. Following the indices to the LTT table entry where a trampoline to the new code-pointer is stored the translation process is completed. When an index-based scheme is used the encoded pointers are decoded to get its GTT and LTT indices, which would directly get us to the trampoline for the pointer.

Jump tables are analyzed statically and fixed up with the new code address. This is done by making a copy of the jump table and modifying the code that used original jump table to use this copy. The contents of this new jump table are changed so that it points to the new code location.

Loader changes: To keep our implementation completely transparent, instrumented binaries are modified to use our customized loader while regular binaries use the default system loader. We made our changes on top of the default loader for Ubuntu 20.04 (GNU libc-2.31). Additional tasks performed by the loader can be summarized as:

- Creation of GTT: Loader will create a global table called GTT at runtime. GTT contains information on the modules loaded by the loader.
- Creation of one-level hash table: The loader is responsible for creating the one-level hash table at the runtime. The hash table is re-hashed when inserting pointers from a new module causes the load factor to go above a set limit. This can happen during the initial phase when a program is being set up the loader or when a shared library is opened at runtime using dlopen.

- Encoding the pointer for the index-based scheme: The loader is responsible for relocating pointers by adding the module base address to the pointer offset value. For identified code pointers we change their relocation type to a new type. The loader on encountering this new relocation type encodes the pointer according to its array indices instead of performing relocation.
- Loading instrumented modules: The default search process is changed to always load instrumented version of shared libraries for an instrumented program.

5 Evaluation

All the testing is done on a desktop with 12th generation Intel Core i7 with 16GB RAM, running Ubuntu 20.04. We test the performance of address translation and index-based encoding scheme by instrumenting SPEC binaries and comparing their overheads with the base timings. With just address translation enabled the average overhead is **7.27%**. When index-based encoding is used with inlined decoding routine, the overhead reduces to **2.35%**. Figure 5.1 shows the overhead of the address translation scheme next to the overhead of index-based scheme with inlined decoding routine. Index-based encoding scheme improves the performance by more than 2x.

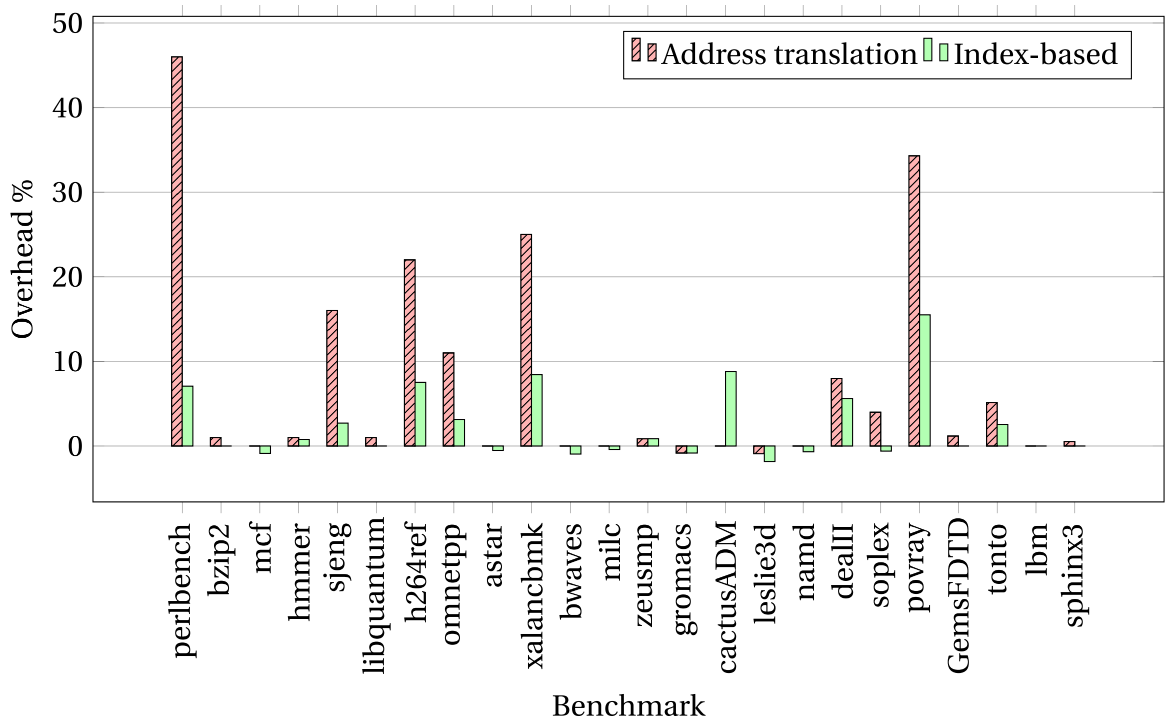

Figure 5.1: Overhead comparison of address translation and index-base CFI.

Next, we conduct tests on our proposed shadow stack design to evaluate the following:

- The performance overhead of our shadow stack.

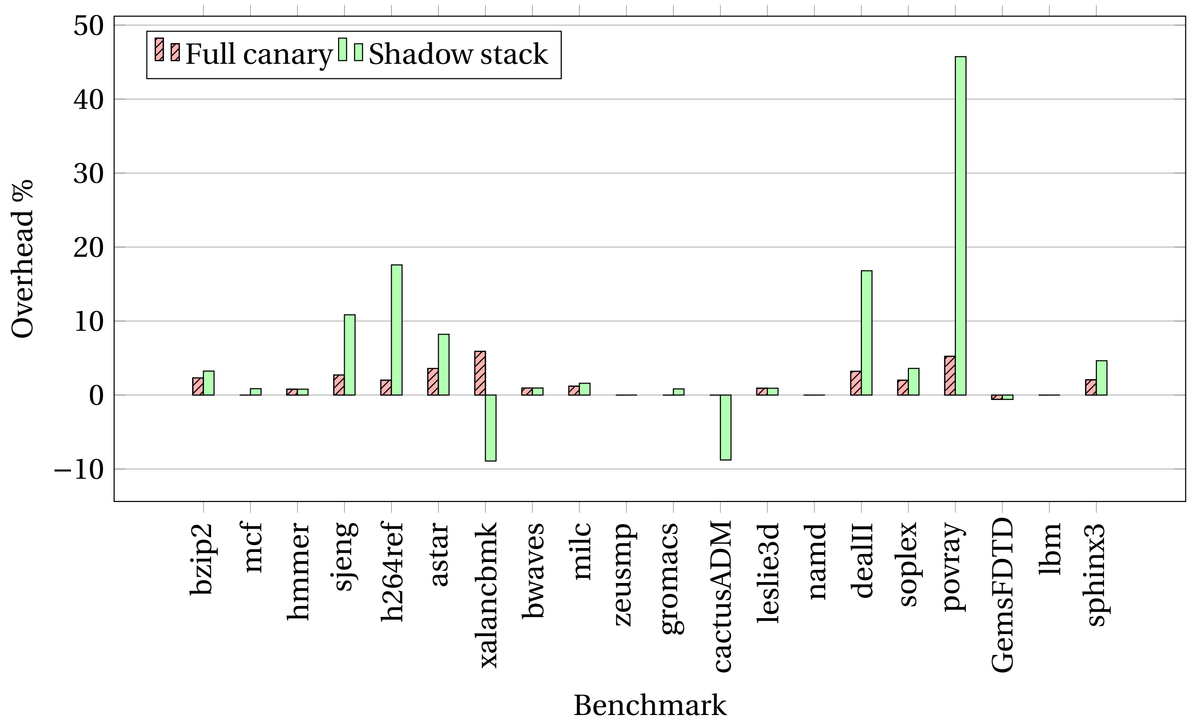

Figure 5.2: Overhead of our proposed shadow stack with overhead of full canary protection.

- Overhead compared to the overhead of using canary protection for all functions by using *-fstack-protector-all*.
- Performance of Intel CET shadow stack.
- Code size increase due to shadow stack instrumentation.

We see that the average performance overhead of our shadow stack is **4.92%** over the default SPEC binaries. Povray incurs an extremely high overhead of **45.74%**, excluding *povray* the overhead is **2.77%**. The overhead of enabling stack canary protection for all functions is **1.61%**. Figure 5.2 shows the full canary protection overhead plotted alongside our shadow stack overhead for various SPEC programs.

To test the performance and compatibility of the CET shadow stack, we installed the latest patch, based on Kernel 6.1.13 on a 12^{th} generation Intel processor machine and used SPEC CPU 2006 to calculate overhead introduced when CET shadow stack is enabled. As expected, we were only able to get a subset of C/C++ SPEC binaries running. This is due to a lack of support from glibc and GCC which is necessary for supporting language features like longjmp/setjmp and C++ exception handling.

Intel CET, while having a very low overhead of **0.41%**, in its current state does not work with programs with longjmp and exception handling 5.3. In our testing, we found that Intel CET aborted all the SPEC binaries that made use of C++ exceptions or setjmp/longjmp.

The total overhead of full CFI protection including index-based forward-edge CFI (inlined decoding) and shadow stack is **7.09%**. This is shown in Figure 5.4.

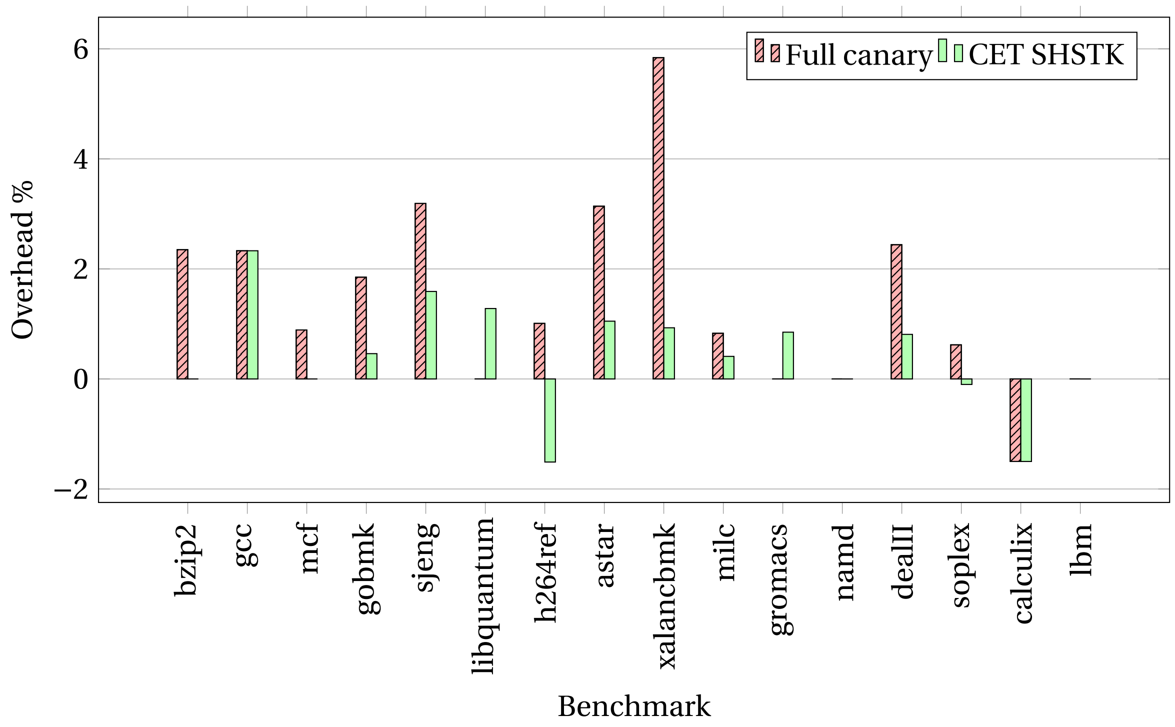

Figure 5.3: Intel CET shadow stack overhead compared with full canary protection overhead.

The average size increase of code section size for SPEC binaries when full CFI protection is enabled with index-based forward-edge CFI (inlined decoding) and the shadow stack is **2.15x**. Size increase for different binaries is shown in Figure 5.5.

6 CONCLUSION

In this thesis, we presented a novel binary instrumentation based shadow stack which repurposes stack canary code and stack canary slot to solve the compatibility challenges that arise due to the use of long jumps and exception handling. Our evaluations show that our shadow stack implementation is very efficient with an overhead of only 4.92%.

We also conducted a study on hashing schemes and hash table setups to determine the best configuration for address translation. Our experiments showed that a one-level hash table setup with quadratic probing gives the best performance. Finally, with our index-based scheme we were able to achieve an overhead of only 2.35% for forward-edge CFI. A complete CFI implementation using index-based forward-edge CFI along with our shadow stack was achieved at a modest overhead of 7.09%.

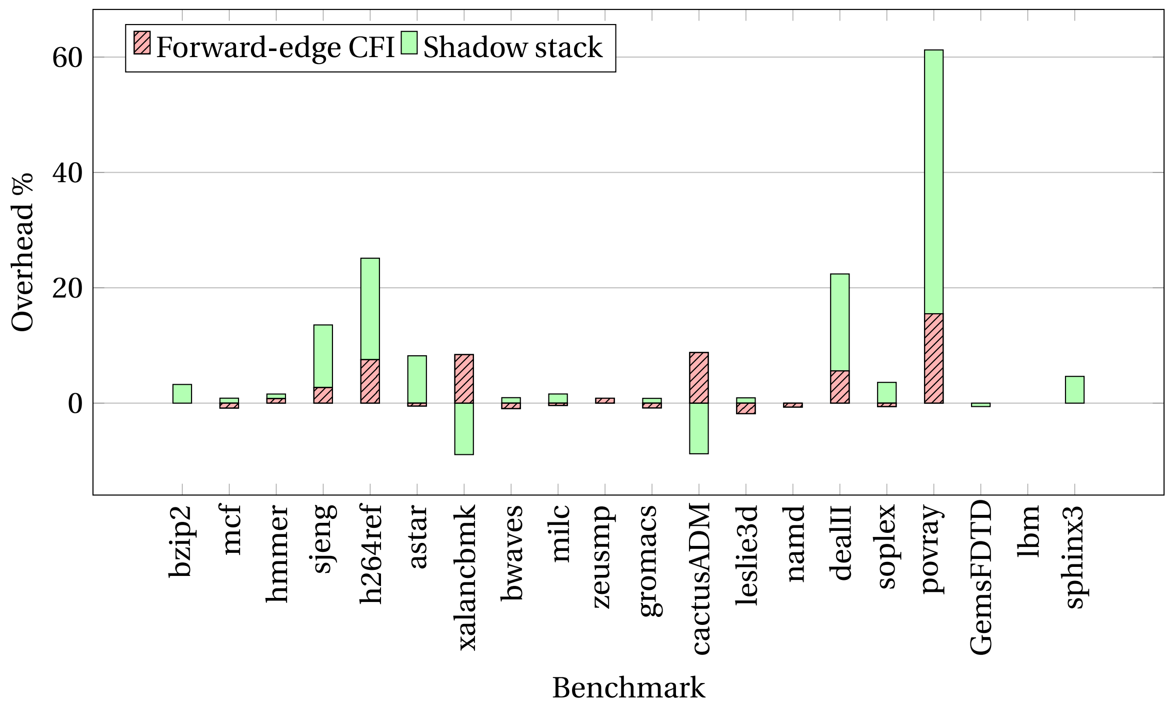

Figure 5.4: Overhead of our shadow stack scheme including forward-edge CFI overhead and using index-based scheme (inlined decoding).

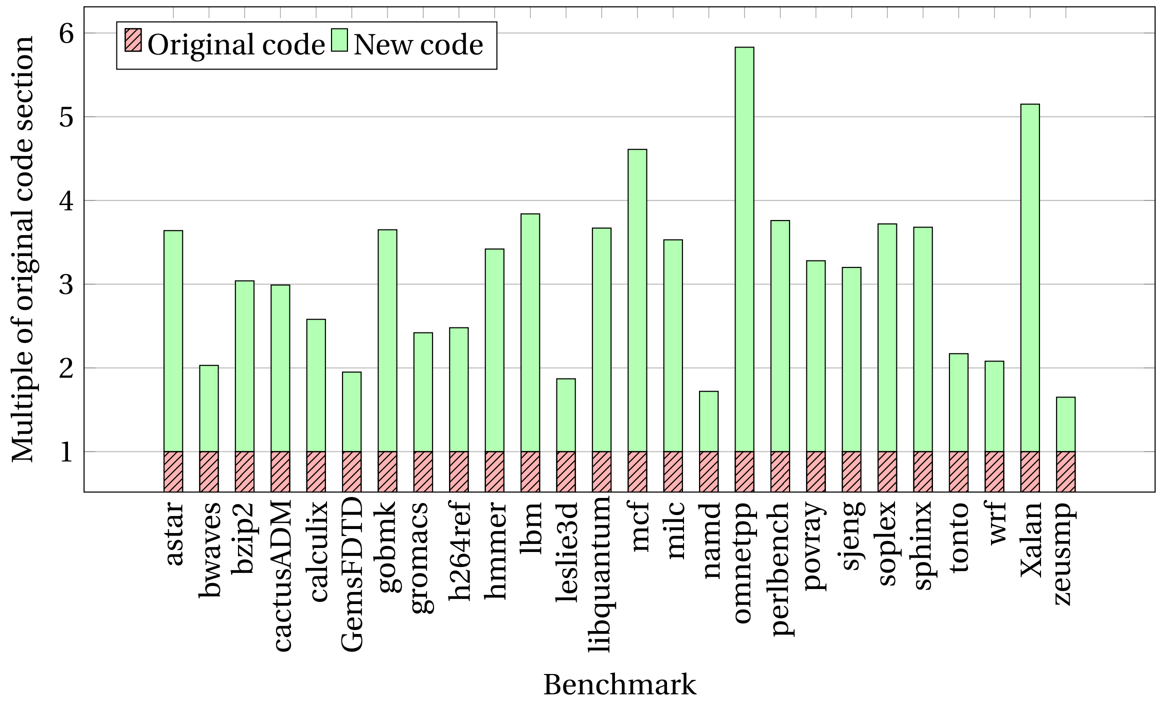

Figure 5.5: Size of the instrumented code section with index-based (inlined decoding) forward-edge CFI along with shadow stack compared to the size of original code section.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Cfi: Principles, implementations, and applications. In *ACM CCS*, 2005.

[2] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM TISSEC*, 2009.

[3] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. In *ACSAC*, 2011.

[4] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *ASIACCS*, 2011.

[5] E. Borin, C. Wang, Y. Wu, and G. Araujo. Software-based transparent and comprehensive control-flow error detection. In *Code Generation and Optimization*, 2006.

[6] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Code Generation and Optimization*, 2003.

[7] N. Burow, X. Zhang, and M. Payer. Sok: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 985–999. IEEE, 2019.

[8] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security Symposium*, 2015.

[9] N. Carlini and D. Wagner. {ROP} is still dangerous: Breaking modern defenses. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014.

[10] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *ACM CCS*, 2010.

[11] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[12] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, 2015.

[13] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security*, 2014.

[14] L. Davi, A.-R. Sadeghi, and M. Winandy. ROPdefender: a detection tool to defend against return-oriented programming attacks. In *ASIACCS*, 2011.

[15] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. Xfi: Software guards for system address spaces. In *Operating systems design and implementation*, 2006.

[16] B. Ford and R. Cox. Vx32: lightweight user-level sandboxing on the x86. In *USENIX ATC*, 2008.

[17] M. Frantzen and M. Shuey. Stackghost: Hardware facilitated stack protection. In *USENIX Security Symposium*, 2001.

[18] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *USENIX Security*, 2014.

[19] L. C. Harris and B. P. Miller. Practical analysis of stripped binary code. *ACM SIGARCH*, 2005.

[20] F. R. A. Hopgood and J. Davenport. The quadratic hash method when the table size is a power of 2. *The Computer Journal*, 1972.

[21] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *USENIX Security Symposium*, 2002.

[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Programming language design and implementation*, 2005.

[23] S. McCamant and G. Morrisett. Evaluating sfi for a cisc architecture. In *USENIX Security Symposium*, 2006.

[24] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Programming language design and implementation*, 2007.

[25] B. Niu and G. Tan. Rockjit: Securing just-in-time compilation using modular control-flow integrity. In *ACM CCS*, 2014.

[26] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In *ACSAC*, 2010.

[27] A. One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49), 1996.

[28] R. Pagh et al. Cuckoo hashing for undergraduates. *IT University of Copenhagen*, 6, 2006.

[29] M. Payer, A. Barresi, and T. R. Gross. Fine-grained control-flow integrity through binary hardening. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2015.

[30] M. Payer, T. Hartmann, and T. R. Gross. Safe loading - a foundation for secure execution of untrusted programs. In *IEEE S&P*, 2012.

[31] Pin - a dynamic binary instrumentation tool. `http://pintool.org/`.

[32] A. Prakash, H. Yin, and Z. Liang. Enforcing system-wide control flow integrity for exploit detection and diagnosis. In *ACM CCS*, 2013.

[33] M. Prasad and T.-c. Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *USENIX Annual Technical Conference, General Track*, 2003.

[34] R. Qiao, M. Zhang, and R. Sekar. A principled approach for rop defense. In *Annual Computer Security Applications Conference*, 2015.

[35] R. Rohleder. Hands-on ghidra-a tutorial about the software reverse engineering framework. In *Proceedings of the 3rd ACM Workshop on Software Protection*, 2019.

[36] K. Scott and J. Davidson. Strata: A software dynamic translation infrastructure. In *IEEE Workshop on Binary Translation*, 2001.

[37] H. Shacham et al. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM CCS*, 2007.

[38] V. Shanbhogue, D. Gupta, and R. Sahita. 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*, 2019.

[39] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Security and Privacy (SP)*, 2016.

[40] S. Sinnadurai, Q. Zhao, and W. fai Wong. Transparent runtime shadow stack: Protection against malicious return address modifications, 2008.

[41] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *IEEE Security and Privacy*, 2013.

[42] P. team. Address space layout randomization. http://pax.grsecurity.net/docs/aslr.txt, 2001.

[43] V. Van Der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Security and Privacy (SP)*, 2016.

[44] M. Wang, H. Yin, A. V. Bhaskar, P. Su, and D. Feng. Binary code continent: Finer-grained control flow integrity for stripped binaries. In *Proceedings of the 31st Annual Computer Security Applications Conference*, 2015.

[45] Wikipedia. Double hashing — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Double_hashing`, 2023. [Online; accessed 11-May-2023].

[46] Wikipedia. Linear probing — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Linear_probing`, 2023. [Online; accessed 11-May-2023].

[47] Wikipedia. Open addressing — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Open_addressing`, 2023. [Online; accessed 11-May-2023].

[48] Wikipedia. Quadratic probing — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Quadratic_probing`, 2023. [Online; accessed 11-May-2023].

[49] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, 2009.

[50] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *ACM CCS*, 2011.

[51] C. Zhang, T. Wei, Z. Chen, L. Duan, S. McCamant, and L. Szekeres. Protecting function pointers in binary. In *ASIACCS*. ACM, 2013.

[52] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *IEEE Security and Privacy*, 2013.

[53] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar. A platform for secure static binary instrumentation. *ACM VEE*, 2014.

[54] M. Zhang and R. Sekar. Control flow integrity for cots binaries. In *USENIX Security*, 2013.

[55] M. Zhang and R. Sekar. Control flow and code integrity for cots binaries: An effective defense against real-world rop attacks. In *ACSAC*, 2015.