Stateful Forward-Edge CFI Enforcement with Intel MPX

Jun Zhang[1,3], Rui Hou[2], Wei Song[2], Zhiyuan Zhan[2,3], Boyan Zhao[1,3], Mingyu Chen[1,3], and Dan Meng[2]

[1] State Key Laboratory of Computer Architecture, ICT, CAS, Beijing, China
[2] Institute of Information Engineering, CAS, Beijing, China
[3] University of Chinese Academy of Sciences, Beijing, China

Abstract. This paper presents a stateful forward-edge CFI mechanism based on a novel use of the Intel Memory Protection Extensions (MPX) technology. To enforce stateful CFI policies, we protect against malicious modification of pointers on the dereference pathes of indirect jumps or function calls by saving these pointers into shadow memory. Intel MPX, which stores pointer's bounds into shadow memory, offers the capability of managing the copy for these indirect dereferenced pointers. There are two challenges in applying MPX to forward-edge CFI enforcement. First, as MPX is designed to protect against every pointers that may incurs memory errors, MPX incurs unacceptable runtime overhead. Second, the MPX defense has holes when maintaining interoperability with legacy code. We address these challenges by only protecting the pointers on the dereference pathes of indirect function calls and jumps, and making a further check on the loaded pointer value. We have implemented our mechanism on the LLVM compiler and evaluated it on a commodity Intel Skylake machine with MPX support. Evaluation results show that our mechanism is effective in enforcing forward-edge CFI, while incurring acceptable performance overhead.

Keywords: Code-reuse attacks · Control-flow integrity · Shadow stack · Shadow memory · MPX · LLVM.

1 Introduction

Code-reuse attacks (CRA) [1,2,3,4,5] exploit memory corruption vulnerabilities to redirect the intended control-flow of applications to unintended but valid code sequences. As these attacks require no code injection, they can defeat the defenses in mainstream computing devices [6,7], such as StackGuard [8], DEP [9] and ASLR [10]. Control-flow integrity (CFI) [11,12] is considered as a general and promising method to prevent code-reuse attacks. CFI restricts the control transfers along the edges of the programs's predefined Control-Flow Graph (CFG), which is constructed by statically analyzing either the source code or the binary of a given program. The control-transfers caused by indirect jumps and function calls are corresponding to forward-edge control-flow. Backward-edge control-flow represents transfers caused by `ret` instructions.

Shadow stack is considered as an essential mechanism to enforce stateful backward-edge CFI policies [11,13]. It keeps track of the function calls by storing the return addresses in a dedicated protected memory region. Most of the forward-edge CFI enforcement technologies follow a two-phase process. During the analysis phase, all the legal target(s) of each indirect control-transfer are abstracted from the protected program's CFG. The enforcement phase ensures that each control-transfer target belongs to the legal targets set. However, even the context/field sensitive static analysis still over-approximates the targets of indirect control-transfers [13,14,15]. Recent researches show that just the intended legal targets are enough for a successful attack [13,14,15]. The weakness of current forward-edge CFI mechanisms is that conformance to the CFG is a stateless policy [13]. To conduct control-flow hijack attacks without violate the CFG restriction, attackers still have to maliciously overwrite (craft) the targets of indirect control-transfers [13,15]. Malicious modifications can be detected by verifying the runtime control-flow information [16,17].

In this paper, we introduce a novel stateful forward-edge CFI mechanism. Unlike the traditional CFI mechanisms, which check only whether each control-transfer target belongs to legal targets set [11,12,18,19,20,21], our mechanism checks the integrity of all pointers on the dereference pathes of indirect jumps and function calls. We call the pointers on the dereference pathes of indirect jumps and function calls as control-transfer related pointers. To support this method, we protect against malicious modification on control-transfer related pointers by saving these pointers in a disjoint shadow memory[1] when they are stored into memory. When a control-transfer related pointer is dereferenced, its copy is loaded from the shadow memory and compared with itself. If the integrity check passes, no action is taken; if the check fails, the program control transfers to the error handler. This process is similar to shadow stack. To facility the copy management and integrity checking, we implement our mechanism based on a new, commercially available hardware feature called Memory Protection Extensions (MPX) on Intel CPUs [25,26,27]. In MPX, every pointer stored in memory has its associated bounds stored in a shadow memory, which is only accessible via `bndstx` and `bndldx` instructions.

In particular, we make the following contributions:

- We design a stateful forward-edge CFI mechanism, which protects the control-transfer related pointers by saving a copy into shadow memory. When a control-transfer related pointer is dereferenced, the copy is used to check its integrity similar to the shadow stack.
- Intel MPX is reused to manage the copies of control-transfer related pointers. We implement our mechanism on the LLVM compiler framework. A compiler pass is developed to identify the control-transfer related pointers and instrument integrity check codes for them. A runtime library is developed to facility the MPX hardware initialization and check code instrumentation.
- We evaluated our mechanism on a commodity Intel Skylake machine with MPX support. The evaluation shows that our mechanism is effective in en-

[1] Shadow memory is a memory space paralleling the normal data space [22,23,24].

forcing stateful forward-edge CFI, while incurring acceptable performance overhead.

2 Intel MPX

Intel MPX [25,26,27] was first announced in 2013 and became available as part of the Skylake microarchitecture in late 2015. The purpose of Intel MPX is to protect against memory errors and attacks. When Intel MPX protection is applied, bounds-check codes are inserted to detect out-of-bounds accesses. To realize this goal, each level of the hardware-software stacks is modified to support the Intel MPX technology.

At the hardware level, new MPX instructions [26] are introduced to facilitate the bounds operations. These instructions are summarized in Table 1. To reduce the register pressure on the general-purpose registers (GPRs), MPX introduces a set of 128-bit bounds registers. The current Intel Skylake architecture provides four bounds registers named `bnd0-bnd3`. Each of the bounds registers stores a lower 64-bit bound in bits 0-63 and an upper bounds in bits 64-127. MPX also introduces `#BR` exception to facilitate the exceptions thrown by the bounds operations.

Table 1: Intel MPX instruction summary

Intel MPX Instruction	Description
`bndmk bndx, m`	create LowerBound and UpperBound
`bndcl bndx, r/m`	check the pointer value in GPR/memory against the lower
`bndcu bndx, r/m`	check the pointer value in GPR/memory against the upper
`bndmov bndx, bndx/m`	move pointer bounds from bnd/memory to bnd
`bndmov bndx/m, bndx`	move pointer bounds from bnd to bnd/memory
`bndldx bndx, mib`	load pointer bounds from memory
`bndstx bndx, mib`	store pointer bounds to memory

The memory of bounds and `#BR` exceptions are managed by the OS. Bounds are stored in shadow memory, which is dynamically allocated by the OS in a similar way of paging. Each pointer has an entry in a Bounds Table (BT), which is comparable to a page table. The addresses of BTs are stored in a Bounds Directory (BD), which corresponds to a page directory in analogy. As the bounds registers are not enough for real-world programs, bounds have to be stored/loaded to/from BT by the `bndstx/bndldx` instructions. When a BT does not exist, the CPU raises `#BR` and traps into the OS. Then the OS allocates a new BT for the bounds. Furthermore, the OS is in charge of bounds check violation.

At the compiler level, new MPX transformation passes are added to insert MPX instructions to create, propagate, store and check bounds. Additional

runtime libraries provide initialization/finalization routines, statistics and debug info, and wrappers for functions from standard C libraries [29]. Until now, both GCC and ICC compilers have native support for Intel MPX [25,27]. The LLVM compiler only adds the MPX instructions and bounds registers to its Backend [30].

There are at least two challenges in applying MPX to implement our mechanism. First, MPX is designed to protect every pointers that may incur memory errors. To enforce our mechanism, we have to identity the control-transfer related pointers before the instrumentation. Second, MPX utilizes the `bndldx` instruction to load bounds from the BT. When the content of the index register of `bndldx` instruction matches with the pointer value stored along with bounds in the BT, the destination MPX register is updated with the loaded bounds. However, if a mismatch is detected, the destination MPX register is updated as always-true (INIT) [25,26,27]. This creates holes in MPX defense. Thus, we need to address the problem of how to check the integrity of control-transfer related pointers based on the loaded bounds.

3 Threat Model

In this paper, we only focus on user-space forward-edge CFI and assume that the backward-edge CFI has been efficiently enforced by previous solutions. Since bounds memory and `#BR` exceptions are managed by the OS, we assume adversaries have no control over the OS kernel. This assumption prevents adversaries from directly tampering with our enforced protection. We assume that (1) attackers can not control the program loading process; (2) the system deploys the memory protection, which forbids code section and read only data to get written at run time, and forbids a memory region to be writable and executable at the same time. These assumptions ensure the integrity of the loaded program and prevent code injection attacks. We assume attackers can arbitrary read application's code, and has the full control over the program's stack and heap. In other words, attackers have the following capabilities: (1) attackers can launch information attacks and defeat the protection of ALSR; (2) they can corrupt control data such as return address and function pointers. Our assumptions are as strong and realistic as prior work in this area.

4 Stateful Forward-edge CFI

The goal of this paper is to enforce stateful forward-edge CFI mechanism, which is similar to shadow stack [11,13] and incurs acceptable runtime overhead. In this section we set up a stateful forward-edge CFI model, and discuss the enforcement method based on this model.

To check the integrity of forward-edge control-flow, we need to understand the low level process of control-flow transfers caused by indirect jumps and function calls. A function call through pointer dereference is shown in Figure 1(a). The source code is in black and the disassembly is in green. At line 8, a pointer,

which is a return value from *malloc*, is assigned to *heap_struct*. At lines 11-12, the execution makes *sptr_p* point to the address of *heap_struct*. At lines 15-17, the address of function *func* is assigned to a structure member *sfunc_ptr*, which is found by dereferencing pointer *sptr_p* twice. At that program point, the pointer relationships holding between the variables and functions are illustrated in Figure 1(b). At lines 19-23, function *func* is called by dereferencing pointer *sptr_p*. We call this dereference path as a Dereferenced-Pointers-Flow (DPF), which is analogous to a linked list. DPF consists of a series of elements (such as structures, arrays, pointers). Each contains (or is) a pointer to a element containing its successor. We call these pointers as control-transfer related pointers. The last level control-transfer related pointer points to a function or a address.

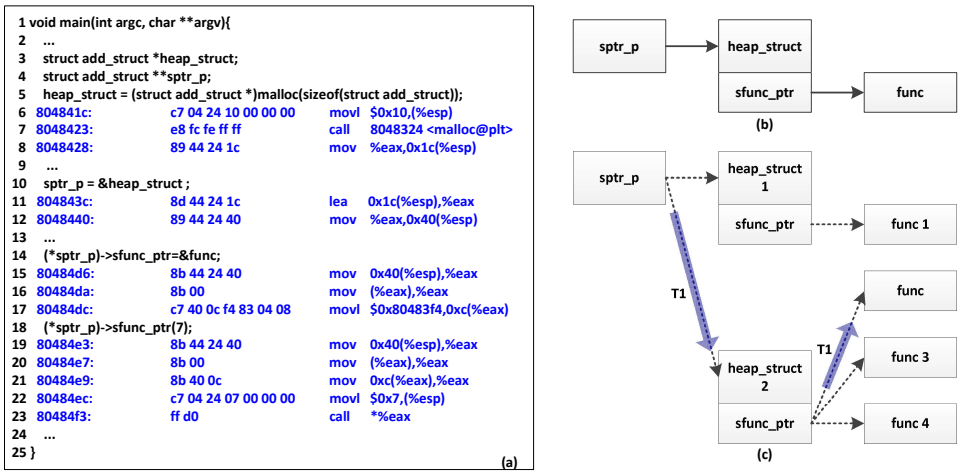


Fig. 1: Stateful forward-edge CFI model

For the whole program, the pointer relationships related to pointer *sptr_p* can be abstracted by statically analyzing. As shown in Figure 1(c), the relationships can be represented as a tree. Every node contains (or is) a control-transfer related pointer. The root node is pointer *sptr_p*, and the leaf nodes are functions with the same type. There are multiple pathes (indicated as dotted lines) from *sptr_p* to the leaf nodes. But there are only one DPF (indicated by the shadow blue arrow) at moment *T1*. If we can make sure that every pointer on the DPF is trusted, we call this forward control-flow is integrity. As shown in Figure 1(a), the DPF is selected by assigning proper value (e.g., location of a function, return pointer from *malloc*, or one address in the stack) to the control-transfer related pointer. If any pointer in the code-pointer tree is overwritten by attackers, the pointer dereference will use another DPF, and the control-flow transfers to target controlled by attackers. We come exactly to the conclusion that the correctness of function call or jump through a pointer dereference depends on the integrity

of the DPF at a moment. A pointer dereference satisfies the integrity property iff its value equals to the last legal update. We say an indirect control-transfer satisfies the CFI property iff the DPFs are protected. If all DPFs are protected, it is sufficient to prevent forward-edge control-flow hijack attacks.

For fine-grained CFI (such as IFCC and VTV [20]) mechanisms, they prevent control-flow hijack attacks by ensuring that the target address of each indirect branch is within the predefined targets set. The targets sets are computed by static program analysis. Thus *func1-func4* are all valid targets for the control transfer at line 23 in Figure 1 at a moment. Actually, there are only one dereference path at a moment. For example, when the program in Figure 1(a) executes at line 23, there is only one DPF as shown in Figure 1(c) at moment *T1*. The false negative of fine-grained CFI mechanisms can be attributed to their stateless target checking. In other words, the target of a control transfer depends on the DPF which is selected by the control-transfer related pointers at a moment.

5 Implementation

We implement our stateful forward-edge CFI mechanism on the LLVM compiler framework [30]. As shown in Figuare 2, we add an optimization pass (DFI pass) during the optimization stage, and link the object codes with the runtime library at the link stage.

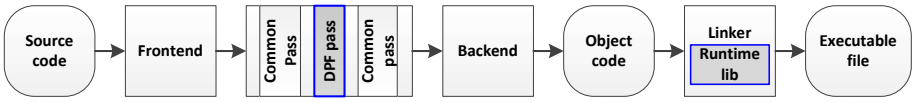


Fig. 2: The process of our stateful forward-edge CFI mechanism implementation. It first identifies the DPF nodes and inserts integrity checking codes by the DPF pass, and finally links the object codes with the runtime library.

Integrity check based on MPX instructions: As shown in Figure 3(a), the function `bound_set` creates bounds at line 4. Since we set the base register of `bndmk` instruction as `ptr_value`, `ptr_value` is stored in the lower bound `bnd0.LB`. As shown in Figure 3(b), when we call `bound_assert` to check the integrity of `ptr_value` loaded from `ptr`, we firstly load its bounds to `bnd0` at line 15. Then, we move the bonds from `bnd0` to the memory space indexed by the pointer `ptr_tmp` at line 17, and assign the lower bound to `ptr_rst` at line 18. Finally, we compare the loaded pointer value `ptr_value` with the lower bound at line 22. If a mismatch is detected between them, the control transfers to the `error_lable()` function.

Runtime library: As described in the above paragraph, the `bound_set()` function and `bound_assert()` function are added as intrinsic function calls. We implement these functions into a runtime library. Besides these integrity checking

```
1 __MPX_INLINE void __llvm__bound_set (void **ptr, void *ptr_value){
2   unitprt_t offset;
3   offset = 4;
4   __asm__ __volatile__ ( "bndmk  (% 2, % 1),   %% bnd0\n\t"
5                          "bndstx  %% bnd0,   (% 0, % 2)"
6                          :
7                          : "r"  (ptr), "r"  (offset), "r"  (ptr_value)
8                          : "% bnd0" )
9 }                                                                    (a)
```

```
10 __MPX_INLINE void __llvm__bound_assert (void **ptr, void *ptr_value){
11   __llvm__bounds  bounds;
12   __llvm__bounds* ptr_tmp;
13   ptr_tmp = &bounds;
14   int ptr_rst;
15    __asm__ __volatile__ ( "bndldx  (% 1, % 2),   %% bnd0\n\t"
16                           "mov       %3,          %% rax\n\t"
17                           "bndmov  %%bnd0,  (%% rax)\n\t"
18                           "mov      (%% rax),    %0"
19                           : "=r" (ptr_rst)
20                           : "r"  (ptr), "r"  (ptr_value), "r"  (ptr_tmp)
21                           : "% bnd0" )
22   if(ptr_rst != ptr_value) error_lable();                           (b)
23 }
```

Fig. 3: Integrity checks based on MPX instructions.

functions, we also add some functions to initialize the MPX hardware at program startup. These functions are migrated from the `libmpx` library of GCC compiler. We compile these functions into a object file and link with this object file at the link stage.

DPF pass: We implemented the static analysis and instrumentation as an optimization pass. The optimization pass operates on the LLVM intermediate representation (IR), which is a low-level strongly-typed language-independent program representation tailored for static analyses and optimization purpose [30]. The LLVM IR is generated from the C/C++ source code by clang, which preserves most of the type information that is required in our analysis. When our stateful mechanism is applied, the DPF pass works as the following: (1) DPF pass performs type based static analysis to identify any pointers that are control-transfer related. As shown in Figure 1, control-transfer related pointers are pointers to functions, pointers to `struct` or other composite types which contain control-transfer related pointers. This method is similar to CPI [41]. (2) Once the control-transfer related pointers are identified, the DPF pass creates appropriate function calls to the intrinsic functions. When a value is assigned a control-transfer related pointers, a call to `bound_set` is created before the `store` instruction. Function `bound_set` saves the pointer's value in the shadow memory in the form of bounds. When a control-transfer related pointers is used[2], a call to `bound_assert` is created before this instruction. Function `bound_assert` check the pointer's integrity before being used. An example of instrumented codes are shown in 4.

[2] The control-transfer related pointers can be used to call functions, used as function parameters, used to load pointers and so on.

```
1 void main(){
2   ...
3   struct students    *p_to_s1;
4   struct students   **ptr_ps1;
5   p_to_s1 = (struct students *)malloc(sizeof(struct students));
6      bf 18 00 00 00                mov   $0x18,%edi
7      e8 94 fe ff ff                callq  4004c0 <malloc@plt>
8      e8 cb 17 00 00                callq  4027f0 <__llvm__bound_set>
9      48 89 45 e8                   mov   %rax,-0x18(%rbp)
10    ...
11  ptr_ps1 = &p_to_s1 ;
12     e8 cb 17 00 00                callq  4027f0 <__llvm__bound_assert>
13     48 8d 45 e8                   lea   -0x18(%rbp),%rax
14     e8 cb 17 00 00                callq  4027f0 <__llvm__bound_set>
15     48 89 45 f0                   mov   %rax,-0x10(%rbp)
16  ...
17  (*ptr_ps1)->func_ptr = &func;
18     e8 cb 17 00 00                callq  4027f0 <__llvm__bound_assert>
19     48 8b 45 f0                   mov   -0x10(%rbp),%rax
20     e8 cb 17 00 00                callq  4027f0 <__llvm__bound_assert>
21     48 8b 00                      mov   (%rax),%rax
22     e8 cb 17 00 00                callq  4027f0 <__llvm__bound_set>
23     48 c7 40 10 d6 05 40          movq   $0x4005d6,0x10(%rax)
24  ...
25  (*ptr_ps1)->func_ptr(a, b);
26     e8 cb 17 00 00                callq  4027f0 <__llvm__bound_assert>
27     48 8b 45 f0                   mov   -0x10(%rbp),%rax
28     cb 17 00 00                   callq  4027f0 <__llvm__bound_assert>
29     48 8b 00                      mov   (%rax),%rax
30     e8 cb 17 00 00                callq  4027f0 <__llvm__bound_assert>
31     48 8b 40 10                   mov   0x10(%rax),%rax
32  ...
33     ff d0                         callq  *%rax
34  ...
35 }
```

Fig. 4: An example of our stateful forward-edge CFI enforcement.

6 Evaluation

6.1 Effectiveness Evaluation

To evaluate our mechanism's effectiveness, we use the RIPE benchmark [31] which is developed to provide a standard way of testing the coverage of a defense mechanism against memory errors. This program contains 850 attack forms. Our experiment is on the Ubuntu 16.04. To make more attacks work, we disabled the ASLR and compiled it without stack protection and data execution protection. Even though, many exploits failed because of built-in system protection mechanisms, such as changes in the runtime layout, as well as compatibility issues due to the usage of newer-version libraries. At last, 64 attacks works. These attacks can be divided into forward-edge control flow hijacks and backward-edge control flow hijacks. After implementing our stateful forward-edge CFI mechanism, only 6 attacks work. These attacks belong to backward-edge hijack attacks. It is shown that our mechanism is effective in forward-edge control flow enforcement.

6.2 Performance Evaluation

To evaluate the performance overhead of our protection mechanism, five applications are selected from the SPEC CPU2006 benchmark suit [32]. As shown in Table 2, these applications have different fractions of instrumented memory operations. Their allocated bounds tables and instruction overhead are also shown in Table 2. These information is obtained by the profiler tool Perf [33]. We

re-compile these applications with Low Level Virtual Machine (LLVM) [30] to apply our stateful protection.

We ran our experiments on an Intel Xeon(R) E3-1280 v5 with 8 cores 3.7GHz in 64-bit mode with 64GB DRAM. As shown in Figure 5a, the y-axis shows that the runtime overhead normalized to the baseline, i.e., the native applications without protection. In average, our protection mechanism incurs 9.1% runtime degradation. The worst-case is 28.1% for `h264ref`. On the one hand, the performance overhead can be attributed to the increase in number of instructions executed in a protected application. Comparing Figure 5a and the IO column in Table 2, there is a strong correlation between them. As expected, `hmmer`, which has the least instructions increase, has ignorable performance overhead. `h264ref`, which has the most instructions increase, has the worst performance overhead. On the other hand, the performance overhead can be partially attributed to the lower hit rate. Figure 5b shows the impact of our instrumentation on the data cache hit rate. As seen from the figure, most of protected applications have lower data cache hit rate. The exception is `hmmer`, which has ignorable instrumented memory operations.

Table 2: Statistics for the selected applications: FMON represents the fraction of memory operations instrumented; NBT represents the bounds tables allocated for each application; IO represents the instruction overhead normalized to the baseline.

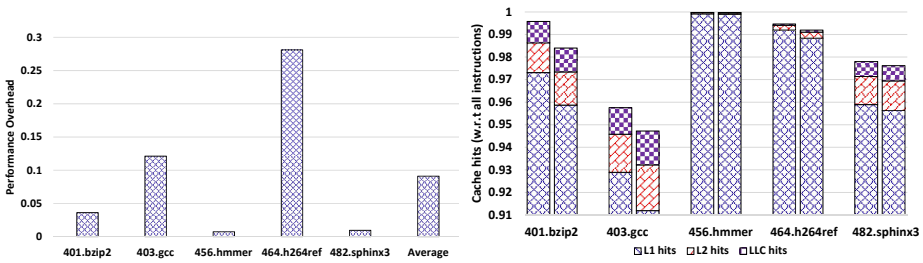
	FMOI	NBT	IO
401.bzip2	0.25%	1	9.49%
403.gcc	2.54%	129	17.12%
456.hmmer	≈ 0	1	≈ 0
464.h264ref	2.42%	18	33.83%
482.sphinx3	0.06%	2	0.20%

7 Related Work

7.1 Control-Flow Integrity

CFI is proposed by Abadi *et al.* in 2005 [11]. It restricts the control-transfers along the edge of the program's predefined CFG. The initial implementation of CFI instruments software with runtime label checks to ensure the source and destination of indirect control transfer have the same label. As frequently called function might have a large set of valid target addresses, CFI is generally coupled with a protected shadow stack to ensure backward-edge CFI [13]. Researchers mainly focus on two CFI enforcement techniques: software-based and hardware-assisted mechansims.

Software-based approaches. Software-based CFI approaches enforce the CFI policies by instrument the source code or legacy binaries. This can be done



(a) (b)

Fig. 5: (a) Performance overhead of our stateful forward-edge CFI mechanism. (b) CPU cache behavior of baseline (bar on the left) and our stateful forward-edge CFI mecahnism (bar on the right).

as part of a compiler optimization pass or binary rewriting. For the compiler-based approaches [20,21,16,35,36,37], the type information is used to abstract the indirect control transfer targets. Now, the LLVM includes an implementation of a number of CFI schemes [34]. Ge *et al.* [21] leveraged LLVM to enforce fine-grained CFI for FreeBSD and MINIX kernels. The binary rewriting approaches [11,18,19,38,39,40] derive the CFI policy directly form binaries and insert checks for CFI policies enforcement. While software-based approaches are effective in enforcing CFI, they have to make a tradeoff between efficiency and precision.

Hardware-assisted protection. To reduce the performance overhead of software-based approaches, several hardware-assisted CFI approaches have been designed. New CFI instructions and hardware-based shadow stack are introduce to accelerate label checking on each indirect branch [42,42,43,44]. Intel have added the CFI instructions and shadow stack into their Instruction Set Architecture (ISA) [45]. kBounder [46] and PathARmor [47] utilize the Last Branch Record (LBR) feature to build CFI defense. CFIMon [48] leverages Branch Trace Store (BTS) to record control transfers and implement CFI checks. However, these approaches only implement coarse-grained security policies. To enforce fine-grained CFI, CFIGuard [49] proposes to combine the LBR with the Performance Monitoring Unit (PMU). By program the PMU to trigger an interrupt when the LBR stack is full, CFIGuard could check all executed indirect branches. However, CFIGuard incurs much runtime overhead because of the frequently generated interrupts. FlowGuard [50], GRIFFIN [51] and PT-CFI [52] leverage the Intel Processor Trace (PT) to record the execution trace of a monitored program and perform online control-flow checks based on the offline CFI policies. One advantage of these work is that they are capable of enforcing a variety of stateful CFI policies over unmodified binaries. Comparing to the above hardware-assisted mechanisms, our mechanism reuses the MPX to enforce stateful forward-edge CFI, which do not need to construct the CFG and offline trains.

7.2 Code Pointer Integrity

Memory errors are the root of control-flow hijack attacks. Though many of memory safety mechanisms have been designed, they have not been widely adapted by industry for their high runtime overhead. Kuznetsov et al. [41] propose the Code Pointer Integrity (CPI) mechanism based on the observe that integrity guarantee of code pointers is sufficient to prevent control-flow hijack attacks. They implement CPI by storing sensitive pointers in an isolated memory region, and further use the runtime information (such as bounds of pointers) to check the validation of pointer dereference. There are a large body of research leveraging cryptography to provide security for code pointers. Tuck *et al.* [53] protect the pointer by encrypting the stored value. Their work is designed to protection from buffer overflow and cannot prevent code-reuse attacks. To prevent code-reuse attacks, Cryptographic CFI (CCFI) [16] uses MACs to check the integrity of indirect control-transfer targets. As the MACs contain more runtime information than the encrypted pointers, CCFI provides CFI protection efficiently. Recently, ARM announced the ARMv8.3-A architecture added a pointer integrity mechanism, called Pointer Authentication (PA) [54]. Similar to CCFI, PA use short cryptographic MACs to verify the integrity of pointers. Essentially, we enforce forward-edge CFI by guarantee the integrity of control-transfer related pointers. Different form these CPI mechanisms, we compares one control-transfer related pointer with its copy to verify its integrity. This method is similar to shadow stack. Furthermore, we leverage Intel MPX to facility the integrity checking.

8 Conclusions

This paper presents an efficient stateful forward-edge mechanism based on Intel MPX. We guarantee the integrity of control-transfer related pointers by storing these pointers into shadow memory, which is managed by OS and accessed by the MPX `bndstx` and `bndldx` instructions. To implement our method based on MPX, we design a LLVM pass to identify the control-transfer related pointers and instrument the source code. We also develop a runtime library to facility the instrumentation and initialize the MPX hardware. Our evaluation results show that our method is effective in enforcing forward-edge CFI, while incurring acceptable performance overhead.

References

1. Shacham, H.:The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: ACM 14th Conference on Computer and Communications Security (CCS 2007), p.552-561 (2007)
2. Hund, R., Holz, T., Freiling, F. C.: Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In: USENIX 18th Security Symposium (SEC2009), p.383-398 (2009)

3. Bletsch, T., Jiang, X., Freeh, V. W., Liang, Z.: Jump-oriented programming: A new class of code-reuse attack. In: ACM 6th Symposium on Information, Computer and Communications Security (ASIACCS), p.30-40 (2011)
4. Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A. R., Holz, T.: Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In: IEEE 36th Symposium on Security and Privacy (S&P 2015), p.745-762 (2015)
5. Carlini, N., Wagner, D.: Rop is still dangerous: Breaking modern defenses. In: USENIX 23rd Security Symposium (SEC 2014), p.385-399 (2014)
6. Szekeres, L., Payer, M., Wei, T., Song, D.: SoK: Eternal War in Memory. In: IEEE 34th Symposium on Security and Privacy (S&P 2013), p.48-62 (2013)
7. Victor, V., Nitish, D., Lorenzo, C., Herbert, B.: Memory Errors: The Past, the Present, and the Future. In: ACM 15th International Conference on Research in Attacks, Intrusions, and Defenses (RAID 2012), p.86-106 (2012)
8. Cowan, C., Pu, C., Maier, D., Hintony, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: USENIX 7th Security Symposium (SEC 1998), P.63-78 (1998)
9. LNCS Microsoft Corporation: Data Execution Prevention. `https://msdn.microsoft.com/en-us/library/windows/desktop/aa366553(v=vs.85)`.
10. Xu, J., Kalbarczyk, Z., Iyer, R. K.: Transparent runtime randomization for security. In: IEEE 22nd Symposium on Reliable Distributed Systems (SRDS 2003), p.260-269 (2003)
11. Abadi, M., Budiu, M., Erlingsson, ¨²lfar, Ligatti, J.: Control-flow integrity. In: ACM 12th Computer and Communications Security (CCS 2005), p.340-353 (2005)
12. Burow, N., Carr, S. A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., Payer, M.: Control-flow integrity: Precision, security, and performance. In: ACM Comput. Surv., vol. 50, pp.16:1-16:33 (2017)
13. Carlini, N., Barresi, A., Payer, M., Wagner, D., Gross, T. R.: Control-flow bending: On the effectiveness of control-flow integrity. In: USENIX 24th Conference on Security Symposium (SEC 2015), p.161-176 (2015)
14. 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: ACM 22nd Conference on Computer and Communications Security (CCS 2015), p.901-913 (2015)
15. 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: ACM 22nd Conference on Computer and Communications Security (CCS 2015), p.952-963 (2015)
16. Mashtizadeh, A. J., Bittau, A., Boneh, D., Mazi¨¨res, D.: Ccfi: Cryptographically enforced control flow integrity. In: ACM 22nd Conference on Computer and Communications Security (CCS 2015), p.941-951 (2015)
17. Zhang, J., Hou, R., Fan, J., Liu, K., Zhang, L., A.McKee, S.: Raguard: A hardware based mechanism for backward-edge control-flow integrity. In: ACM Computing Frontiers Conference (CF 2017), p.27-34 (2017)
18. Zhang, M., Sekar, R.: Control flow integrity for cots binaries. In: USENIX 22th Conference on Security (SEC 2013), p.337-352 (2013)
19. 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 34th Symposium on Security and Privacy (S&P 2013), p.559-573 (2013)

20. Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, ¨², Lozano, L., Pike, G.: Enforcing forward-edge control-flow integrity GCC & LLVM. In: USENIX 23rd Security Symposium (SEC 2014), p.941-954 (2014)
21. Ge, X., Talele, N., Payer, M., Jaeger, T.: Fine-grained control-flow integrity for kernel software. In: IEEE 1st European Symposium on Security and Privacy (EuroS&P), p.179-194 (2016)
22. Devietti, J., Blundell, C., Martin, M. M. K., Zdancewic, S.: Hardbound: Architectural support for spatial safety of the c programming language. In: ACM 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008), p.103-114 (2008)
23. Nagarakatte, s., Zhao, J., Martin, M. M., Zdancewic.: Softbound: Highly compatible and complete spatial memory safety for c. In: ACM 30th SIGPLAN Conference on Programming Language Design and Implementation on proceedings (2009 PLDI), pp. 245–258. ACM, Dulin, Ireland (2010)
24. Nagarakatte, S., Martin, M. M. K., Zdancewic, S.: Watchdoglite: Hardware-accelerated compiler-based pointer checking. In: Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2014), p.175-184 (2014)
25. Intel Corporation: Intel Memory Protection Extensions Enabling Guide. https://software.intel.com/sites/default/files/managed/9d/f6/Intel_MPX_EnablingGuide.pdf.
26. Intel Corporation: Intel memory ptrotection extensions. Intel 64 and IA-32 Architectures Software Developer¡¯s Manual, vol. 1, chap. 17 (2017)
27. Oleksenko, O., Kuvaiskii, D., Bhatotia, P., Felber, P., Fetzer, C.: Intel MPX explained: An empirical study of intel MPX and software-based bounds checking approaches. In: Arxiv CoRR, vol. abs/1702.00719 (2017)
28. GCC Wiki: Intel Memory Protection Extensions (Intel MPX) support in the GCC compiler. https://gcc.gnu.org/wiki/Intel%20MPX%20support%20in%20the%20GCC%20compiler
29. gcc-mirror. https://github.com/gcc-mirror/gcc/tree/master/libmpx
30. The LLVM Compiler Infrastructure. http://llvm.org/.
31. , Wilander, J., Nikiforakis, N., Younan, Y., Kamkar, M., Joosen, W.: RIPE: Runtime Intrusion Prevention Evaluator. In: Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC 2011). p.41–50 (2011)
32. SPEC CPU2006 Benchmark. http://www.spec.org/cpu2006/.
33. Linux kernel profiling with perf. https://perf.wiki.kernel.org/index.php/Tutorial.
34. Clang 7 documentation:Control Flow Integrity. https://clang.llvm.org/docs/ControlFlowIntegrity.html.
35. , Wang, Z., Jiang, X.: HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy (S&P 2010). p.380–395 (2010)
36. , Niu, B., Tan, G.: Modular Control-flow Integrity. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (OSDI 2014). p.577–587 (2014)
37. , Niu, B., Tan, G.: Per-Input Control-Flow Integrity. In: Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS 2015). p.914–926 (2015)
38. , Payer, M., Barresi, A., Gross, T. R.: Fine-grained control-?ow integrity through binary hardening. In: Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability (DIMVA 2015). p.144–164 (2015)

39. , Mohan, V., Larsen, P., Brunthaler, S., Hamlen, K. W., Franz, M.: Opaque Control-Flow Integrity. In: Proceedings of The 2015 Network and Distributed System Security Symposium (NDSS 2015).
40. , Elsabagh, M., Fleck, D., Stavrou, A.: Strict Virtual Call Integrity Checking for C++ Binaries. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS 2015).
41. Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., Song, D.: Code-pointer integrity. In: USENIX 11th Conference on Operating Systems Design and Implementation (OSDI 2014), p.147-163 (2014)
42. 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 ACM/EDAC/IEEE Design Automation Conference (DAC 2015), p.1-6 (2015)
43. Sullivan, D., Arias, O., Davi, L., Larsen, P., Sadeghi, A.-R., Jin, Y.: Strategy Without Tactics: Policy-agnostic Hardware-enhanced Control-flow Integrity. In: Proceedings of the 53rd Annual Design Automation Conference (DAC 2016), p.163:1-163:6 (2016)
44. Christoulakis, N., Christou, G., Athanasopoulos, E., Ioannidis, S.: HCFI: Hardware-enforced Control-Flow Integrity. In: Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY 2016), p.38-49 (2016)
45. Intel Corporation: Control-flow enforcement technology preview. `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf`.
46. Pappas V., Polychronakis M., Keromytis A. D.: Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In: Proceedings of the 22nd USENIX Security Symposium (USENIX Security 2013).
47. , 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 (CCS 2015). p.927–940 (2015)
48. , Xia Y., Liu Y., Chen H., Zang, B.: CFIMon: Detecting violation of control flow integrity using performance counters. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS 2015). p.1–12 (2012)
49. , Yuan, P., Zeng, Q., Ding, X.: Hardware-assisted ?negrained code-reuse attack detection. In: Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID 2015). p.66–85 (2015)
50. , Liu, Y., Shi, P., Wang, X., Chen, H., Zang, B., Guan, H.: Transparent and Efficient CFI Enforcement with Intel Processor Trace. In: 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA 2017). p.529–540 (2017)
51. , Ge, X., Cui, W., Jaeger, T.: GRIFFIN: Guarding Control Flows Using Intel Processor Trace. In: Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2017). p.585–598 (2017)
52. , Gu, Y., Zhao, Q., Zhang, Y., Lin, Z.: PT-CFI: Transparent Backward-Edge Control Flow Violation Detection Using Intel Processor Trace. In: Proceedings of the 7th ACM on Conference on Data and Application Security and Privacy (CODASPY 2017). p.173–184 (2017)

53. , Tuck, N., Calder, B., Varghese, G.: Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow. In: Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2004). p.209–220 (2004)
54. Qualcomm Technologies, Inc: Pointer Authentication on ARMv8.3. `file:///E:/beifeng/code%20reuse%20attack/PointerAuthentication/whitepaper-pointer-authentication-on-armv8-3.pdf`.