TSINGHUA SCIENCE AND TECHNOLOGY
ISSNll1007-0214 04/10 pp546–556
DOI: 10.26599/TST.2018.9010128
Volume 24, Number 5, October 2019

Leakage Is Prohibited: Memory Protection Extensions Protected Address Space Randomization

Fei Yan* and Kai Wang

Abstract: Code reuse attacks pose a severe threat to modern applications. These attacks reuse existing code segments of vulnerable applications as attack payloads and hijack the control flow of a victim application. With high code entropy and a relatively low performance overhead, Address Space Layout Randomization (ASLR) has become the most widely explored defense against code reuse attacks. However, a single memory disclosure vulnerability is able to compromise this defense. In this paper, we present Memory Protection Extensions (MPX)-assisted Address Space Layout Randomization (M-ASLR), a novel code-space randomization scheme. M-ASLR uses several characteristics of Intel MPX to restrict code pointers in memory. We have developed a fully functioning prototype of M-ALSR, and our evaluation results show that M-ASLR: (1) offers no interference with normal operation; (2) protects against buffer overflow attacks, code reuse attacks, and other sophisticated modern attacks; and (3) adds a very low performance overhead (3.3%) to C/C++ applications.

Key words: Address Space Layout Randomization (ASLR); Intel Memory Protection Extensions (MPX); code reuse attack

1 Introduction

Code injection has always presented a major threat to software, but with the wide deployment of DEP[1] and $W \oplus X$, classic code injection attacks are no longer feasible. In response, attackers are now forced to seek out and utilize code segments of the target application to construct the attack payload. From the reuse of sensitive system functions in related libraries of the target application (e.g., return-into-libc[2]) to chaining code segments into attack payloads (e.g., Return-Oriented Programming (ROP)[3, 4]), code reuse has become a state-of-the-art attack pattern in modern applications. Such attacks grant to the adversary arbitrary execution privileges without requiring the injection of any malicious code. Consequently, code reuse attacks have made preventing the hijacking of control flow a vital pursuit for software security practitioners[5].

Because it is easy to deploy and efficient to run, Address Space Layout Randomization (ASLR)[6] has become the most widely explored mechanism for defending against code reuse in modern operating systems[7, 8]. However, the entropy brought by ASLR can be compromised through a single memory disclosure[9–11]. With the help of leaked information, the locations of sensitive components after randomization can be exposed to attackers. Even in cases of finer-grained randomization, the offset within a module remains unchanged. The adversary is able to exploit the same vulnerability repeatedly until finally locating all of the components necessary to launch a sophisticated attack[9, 12].

One way of enhancing the security of ASLR is to prevent code pointers from leakage. Lu et al.[13] promoted ASLR-Guard, which provides a safe memory region named the AG-Stack for the storage of code

• Fei Yan and Kai Wang are with the Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China. Email: yanfei@whu.edu.cn; blankaiwang@whu.edu.cn.

∗ To whom correspondence should be addressed.

Manuscript received: 2018-10-12; accepted: 2018-11-10

pointers, and encrypts those code pointers before performing operations on data elements. However, the complexity of program architectures and the challenges presented by low-level programming languages make it difficult to separate code. Furthermore, modern exploits such as Just-In-Time ROP (JIT-ROP) code reuse[12] can engage in malicious behaviors by probing the memory area and redirecting the code pointer, in which case code pointers are encrypted in vain. Consequently, protecting code pointers in applications is a necessary but also challenging task.

Starting with the Skylake CPU, Intel has provided secure Memory Protection Extensions (MPX) to eliminate memory disclosure vulnerabilities[14, 15]. In contrast to other vulnerability mitigation approaches, MPX guarantees the legality of code pointers through a new CPU architecture and instruction set. The introduced CPU architecture grants to the hardware the ability to check the validity of code pointers. With the help of MPX instructions, C/C++ compilers can check the lower and upper bounds of a code pointer before it is referenced; the elimination of buffer overflows is now a real possibility.

Bound Registers (BR): Intel MPX provides four new specialized 128-bit bound registers for temporarily holding the bounds of a code pointer and performing bounds operations. Those bound registers can only be modified via privileged instructions. As shown in Fig. 1, the lower 64 bits of the register hold the lower bound of the code pointer, and the upper 64 bits hold the upper bound. The specialized design and strict CPU-maintained access control mechanisms give assurances of integrity and confidentiality when processing bounds. In contrast to the segment registers used by Code-Pointer Integrity (CPI) for the same functionality, the bound registers in MPX prevent the unintended modification of the value of the registers on the part of the operating system and other applications.

Bound Table: To achieve the confidentiality and integrity of computed pointer bounds, Intel MPX provides a specialized data structure called a Bound Table in an isolated memory space. As shown in Fig. 2, the MPX Bound Table stores the computed pointer bounds with the pointer value and metadata. The privileged instruction pair BNDSTX and BNDLDX is designed to store and read bounds from the Bound Table.

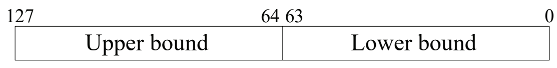

Fig. 1 Layout of bound register.

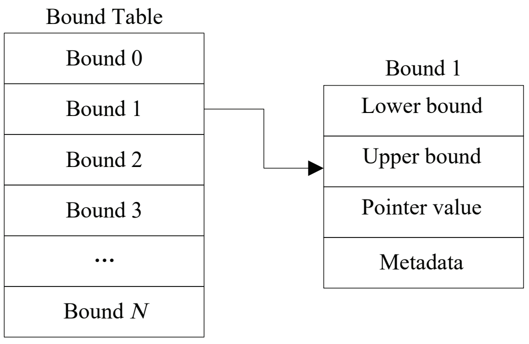

Fig. 2 Bound Table provided by Intel MPX.

Exception Handling: In default scenarios, code pointers stay within their ranges. However, code pointers can point out of range under some unexpected situations, such as an execution error or buffer overflow attack. Intel MPX provides the exception handler #BR for such conditions. The #BR exception is hardware generated and can be caught and processed by exception handling scripts provided by software developers. This exception can be raised by a bounds comparison fault or an unexpected read or write to the Bound Table. The exception handling mechanism ensures the completeness of bounds checking and maintenance.

Although MPX is an effective extension to the legality of code pointers, discovering all of the code pointers in the source code is not an easy task[16]. In this paper, with the aim to efficiently protect the legality and integrity of code pointers, we propose MPX-assisted Address Space Layout Randomization (M-ASLR), a novel code-space randomization scheme. M-ASLR leverages several characteristics of MPX, such as bound checking and the Bound Table, to help prevent memory leakages from the beginning. The main idea of M-ASLR is to bound code pointers while looking up the mapping table. M-ASLR is conceptually simple, but it does require considerable effort to distinguish code pointers from data segments and set their bounds appropriately. M-ASLR is practical and light-weight. In contrast to other work on enhancing the security of ASLR[13, 17–21], M-ASLR introduces no extra data structures in memory, and support for bound checking of a code pointer is provided by the CPU. This feature increases M-ASLR's versatility and ease of deployment. We have developed a fully functioning prototype of M-ASLR, and used it to evaluate the performance characteristics and security features of our scheme. The results show that M-ASLR: (1) offers no interference with normal operations; (2) protects against buffer overflow attacks, code reuse attacks, and other sophisticated modern attacks; and (3) adds a very low performance overhead (3.3%) to C/C++ applications.

In summary, our main contributions in this paper are:

• We propose M-ASLR, a novel code-space randomization scheme for enhancing the ASLR security feature.

• We introduce a new code pointer discovery strategy for MPX protection.

• We perform a systematic analysis of sophisticated code reuse attack patterns on M-ASLR, and demonstrate that M-ASLR is able to eliminate buffer overflow vulnerabilities and further resist such code reuse attacks.

• We implement a prototype of M-ASLR, with our evaluation showing that M-ASLR is a practical solution with a low performance overhead.

We organize the rest of this paper as follows. In Section 2, we detail our threat model and assumptions. The basic ideas behind M-ASLR and its implementation are illustrated in Section 3. We then evaluate the effectiveness and performance overhead of the proposed scheme in Section 4. In Section 5, we discuss the security features of M-ASLR. Section 6 reviews existing ASLR-related literature. Finally, we conclude our paper and provide a brief outlook for our work in Section 7.

2 Threat Model and Assumptions

To ensure that our scheme is practical, we define our threat model based on strong yet realistic assumptions. We assume the adversary has ready access to the technologies needed to launch a code reuse attack. Drawing on attack models used for related schemes[7, 13, 17–20], and the attack patterns illustrated in related studies[2–4, 9, 12], we generate our threat model and make assumptions as follows.

The adversary aims to hijack the control flow of a target application by launching a code reuse attack. We assume the target application has one or more vulnerabilities that can grant an adversary the privilege to read from and write to an arbitrary memory address; and that these vulnerabilities can be exploited repeatedly without causing a crash, such that the adversary is allowed to exploit the vulnerability at will. We further assume that the adversary has open access to the target application and is able to perform reverse computation on the executable file to locate useful code segments and function entries.

We assume the target platform is running a widely available operating system with standard defense mechanisms, such as $W \oplus X$ and ASLR, enabled by default. In M-ASLR, the Trusted Computing Base (TCB) is the CPU, and there are a few known side-channel exploits on CPU caches, such as Meltdown[22] and Spectre attacks[23]. By exploiting these vulnerabilities, CPU cache information can be dumped, and such information leakage through the platform itself would corrupt the TCB of our scheme. We therefore assume that these CPU information leakage vulnerabilities are well patched.

3 Proposed M-ASLR Scheme

M-ASLR enhances the security programs by bounding the validity range of code pointers. By performing source code instrumentation, MPX source code insertion and bounds narrowing, M-ASLR modifies the source program into an MPX program. Figure 3 shows an overview of our M-ASLR scheme. The compiler is responsible for code pointer discovery, after which the compiler inserts MPX code to the binary and narrows the validity bounds of each discovered code pointer. The modified code is then compiled into an executable file. The bounds of each pointer are stored in the Bound Table in an isolated memory space, meaning that the table is transparent to applications in userspace. At runtime, M-ASLR uses the features of Intel MPX to check the validity of each code pointer. When a code pointer attempts to access an address beyond its legal

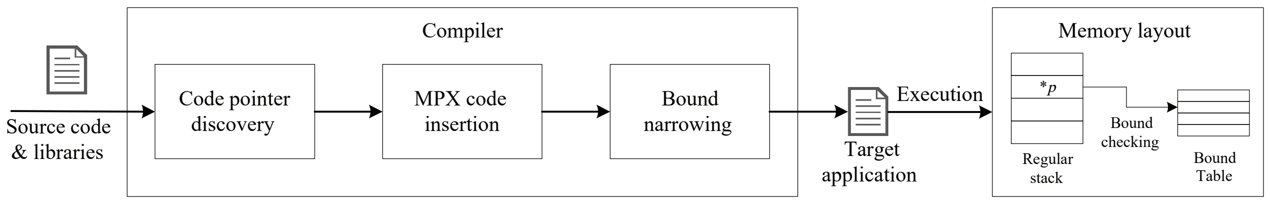

Fig. 3 Overview of M-ASLR.

limits, M-ASLR throws an exception and redirects the request to a valid memory location.

To implement our M-ASLR scheme, we present three key phases: code pointer discovery, MPX code insertion, and bound narrowing. The code pointer discovery phase is responsible for locating and extracting all code pointers. In the MPX code insertion phase, the compiler inserts MPX code into the binary. The inserted MPX code is responsible for computing the lower and upper bounds of each code pointer and providing means for checking the legality of code pointers. The bound narrowing phase analyzes the data structure and narrows the bounds to the minimum.

3.1 Code pointer discovery

As we mentioned, the key idea of M-ASLR is to bound the legal range of code pointers. The discovery of code pointers stands as the highest priority requirement of our scheme. However, previous research on enhancing the security feature of ASLR has paid less attention to the discovery of code pointers. The work of Lu et al.[13] presented the first study on the discovery of pointers or data that can leak the memory layout of code space. To counter the threats described in Section 2, and to support the realization of M-ASLR, we propose an analysis to discover such code pointers in a program. Compared to Ref. [13], our discovery method differs in application scenario and implementation.

Application scenario: The work of Lu et al.[13] on protecting the integrity of code space relies on encryption. The mapping of code space is crucial under these circumstances, so Lu et al.[13] extracted code pointers and related elements that can be used to infer the mapping of code space. In contrast, in M-ASLR, the range of a code pointer is strictly bounded, and the layout of code space is no longer crucial. In M-ASLR, we focus only on code pointers, not on related code or data elements.

Implementation: The work of Lu et al.[13] extracts code pointers from a binary, and places those pointers in an isolated memory space. In contrast, M-ASLR uses the Bound Table provided by MPX, the integrity of which is guaranteed by the CPU. This provides strong integrity assurance and avoids the need to design an additional data structure in memory.

Similar to the work of Lu et al.[13], we categorize the code pointers based on the execution status of the target application. Following this schema, code pointers can be divided into two main types: load-time code pointers and dynamic code pointers. Load-time code pointers are those that are generated as the program is loaded into the memory, such as Procedure Linkage Table (PLT) entry and virtual function pointers. Dynamic code pointers are those that are generated during runtime, such as return pointers; the values of this latter category of pointer change dynamically. According to different characteristics in the lifecycle of these two types of code pointers, M-ASLR extracts them separately.

Load-time code pointers: When the operating system loads the target application, the kernel allocates memory for it and loads the Executable and Linkable Format (ELF) file together with the dynamic linker. During this procedure, all code pointers are loaded into memory. Based on this observation, we modify the relocation functions in the dynamic linker to make it so all code pointers are processed by the relocation function. This ensures that all load-time pointers are extracted.

Runtime code pointers: Unlike load-time code pointers, runtime code pointers are generated dynamically and the value of the pointer changes during execution. Runtime code pointers are generated by control flow transfer instructions, such as call and ret. In order to extract all runtime code pointers, we leverage LLVM to instrument the source code. The instrumentation procedure helps us to identify control flow transfers in the target application and ensures that we discover all runtime code pointers.

3.2 MPX code insertion

After discovering all code pointers in source code, M-ASLR inserts MPX code into the source code. The inserted MPX code is responsible for computing the bounds of each code pointer and adding related bounds checking instructions.

3.2.1 Bound computing

The key idea of M-ASLR is bounding the legal range of each code pointer and thus eliminating potential buffer overflow vulnerabilities. This makes bounds computing an important node in the whole scheme. During a bounds computing procedure, the compiler iterates through the source code and performs an analysis of the data flow. For each discovered code pointer, M-ASLR adopts one of five different bounds computing strategies according to the source of the code pointer. We divide the pointer source into five main types, as listed in Table 1.

Table 1 Types of pointer source.

Pointer source type	Description	Example
Function call pointer	Pointer used to invoke a function call	*char *p = getchar*()
Load pointer	Pointer which is loaded from the memory	*int *p = int* [*m*]
Argument pointer	Pointer used as a function argument	*int func* (*int *p*)
Object address pointer	Pointer points to the address of an object	*struct A *p = & sample*
Field address pointer	Pointer points to the address of a certain field of an object	*char *p = & sample → fld*

M-ASLR applies different bounds computing strategies for each of these five types of pointer source. For a function call pointer, the bounds are consistent with the pre-allocated memory buffer of the invoked function. Similarly, the bounds of the corresponding data structure are read from the Bound Table and passed to a load pointer. An argument pointer serves as an argument to a function. Under this circumstance, M-ASLR assumes that the arguments to the function are in effect for the range of the whole function. Consequently, the bounds of an argument pointer are determined by the caller function, and the lower and upper bounds of the function are passed to the argument pointer. For an object pointer, the compiler leverages the address of the object as the lower bound of the pointer; the upper bound is then calculated according the size of the object. M-ASLR adopts a size-computation policy equal to C/C++ standards. For example, the size of a specific data structure is calculated according to its first variable. Note that the bounds of a field address pointer should be carefully processed. In this case, the pointer only points to a specific element in a field. In the bound computing phase, M-ASLR treats a field address pointer in the same way as an object pointer, and hands the bounds narrowing procedure to the bound narrowing phase. The bounds that have thereby been determined are saved in the Bound Table.

3.2.2 Bound checking

Once the valid bounds of each code pointer have been determined, they are saved into the Bound Table. In the bound checking procedure, M-ASLR inserts bounds checking and exception handling instructions into the source code. The specific algorithm for inserting bounds checking instructions is given as Algorithm 1. M-ASLR iterates the source code and inserts the bounds checking instructions (BNDCL and BNDCU) to check the lower and upper bounds before the pointer value is used. After the bounds checking instructions are inserted, M-ASLR adds an exception handling function to the end

Algorithm 1 Bound checking instructions insertion

```
Input:
    (1) List of code pointers, PList;
    (2) Exception handling function, Efunc.
Output:
    Code with bound checking instructions inserted, BCode.
1: for $p \in PList$ do
2:     insertBefore ($p$, BNDCL);
3:     insertBefore ($p$, BNDCU);
4:     insertBefore ($p$, throwBRexception);
5: end for
6: i ← the end instruction of code;
7: insertAfter($i$, catchBRexception);
8: insertAfter($i$, Efunc);
9: return BCode.
```

of the source code. The exception handling function is designed to catch any #BR exception raised by a bound violation and redirect the violating pointer to a random valid address.

3.2.3 Bound narrowing

The bound computing phase determines the bounds of most code pointers. However, as we have already noted, a field address pointer only points to a specific component in a data structure. This means that it has its own circumscribed bounds different from the bounds of the field data structure. In the bound computing phase, the compiler leaves the bounds of field address pointers unchanged, and hands the issue over to the bound narrowing phase.

The bound narrowing phase is responsible for narrowing the bounds of field address pointers to the minimum. To constrain the bounds of a field pointer, M-ASLR applies the following principles:

- For a static data structure, the bounds of the field address pointer are consistent with the outermost data structure;
- For a dynamic data structure, or for a data structure with uncertain properties, the bounds of the field address pointer are consistent with the innermost data structure.

The bound narrowing process for field code pointers follows Algorithm 2.

For example, as shown in Fig. 4, the sizes of structures Data1, Data2, and Data3 are 8, 48, and 296, respectively. Before bounds narrowing, the bound computing phase leaves the bounds of the three code pointers as [&data, &data+295]. During the bounds narrowing phase, M-ASLR makes an elaborate bounds computation according to the above two principles. For static data structures, such as the first two data pointers in our example, M-ASLR treats the outermost data structure Data1 as the bounds of the field code pointer. Whereas for a dynamic data structure, such as the final code pointer in Fig. 4, M-ASLR treats the innermost data structure Data2 d32[5] as the bounds of the field code pointer. After bounds narrowing, M-ASLR invokes BNDMK to update the corresponding bounds in the Bound Table.

Algorithm 2 Bound narrowing for field code pointer

```
Input:
    (1) List of discovered field code pointers, FList;
    (2) List of static data structure, SList;
    (3) List of dynamic data structure, DList;
    (4) Bound Table, BT.
Output:
    Updated bound table, UBT.
1:  for field code pointer $f \in FList$ do
2:      if $f \in SList$ then
3:          f.bounds = f.Outermostbounds;
4:          BNDMK(f, f.bounds, BT);
5:      else if $f \in DList$ then
6:          f.bounds = f.Innermostbounds;
7:          BNDMK(f, f.bounds, BT);
8:      else
9:          f.bounds = f.Innermostbounds;
10:         BNDMK(f, f.bounds, BT);
11:     end if
12: end for
13: return UBT.
```

```
Struct Data1                      Data3 data;
{                                 //before bound narrowing, all the following
  int d11; //size is 4            pointer's bounds are [&data, &data+295]
  int d12;
}//size is 8                      data.d31.d11;
                                  //principle 1
Struct Data2                      //bounds are [&data.d31, &data.d31+7]
{
  Data1 d21;                      data.d33.d21.d12;
  Data1 d22[5]; //size is 40      //principle 1
}//size is 48                     //bounds are [&data.d33.d21, &data.d33.d21+7]

Struct Data3                      data.d32[3].d22[2].d12;
{                                 //principle 2
  Data1 d31;                      //bounds are [&data.d32, &data.d32+239]
  Data2 d32[5]; //size is 240
  Data2 d33;
}//size is 296
```

Fig. 4 Examples of bound narrowing.

4 Evaluation

We perform a performance evaluation on a 64-bit 16.04 Ubuntu machine. The machine is equipped with an Intel i5-6500 CPU and 16 GB memory. The LLVM and Clang used for our evaluation are both at version 3.5.2.

4.1 Memory cost evaluation

M-ASLR inserts MPX instructions to the input source code in order to perform bounds checking and to add exception handling functions to catch bound comparison violations. Those inserted instructions are loaded into memory together with the application so, inevitably, the memory requirements for an M-ASLR application increase. In our experiment, we use M-ASLR to compile some frequently-used libraries so as to evaluate the memory space cost of M-ASLR, with the evaluation results shown in Fig. 5.

Compared to overall size of the source code, the inserted MPX bounds checking instructions and exception handling functions only occupy a small proportion of space. Our evaluation confirms that these inserted instructions offer no interference to the normal execution of applications, and that the average memory space overhead is only 1.2%.

4.2 Effectiveness

In order to evaluate the effectiveness of our scheme, we implement M-ASLR on a real-world application with one or more known buffer overflow vulnerabilities. The results of these tests indicate that M-ASLR can effectively prevent buffer overflow exploits and can prevent others, such as ROP attacks.

Proof of concept: In the first test, we use M-ASLR to protect a small program containing a single buffer overflow vulnerability. As shown in Fig. 6a, the target application calls *strcpy*() to make operations on strings without checking the validity of the relative arguments. With M-ASLR applied, the assembly code

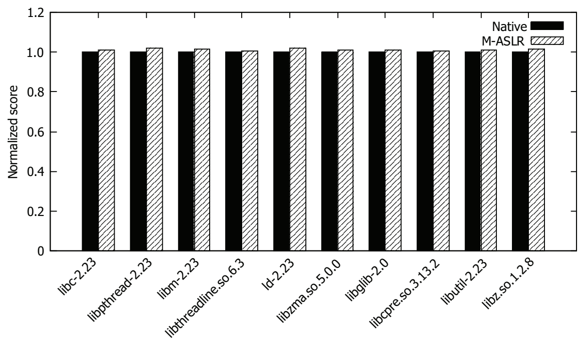

Fig. 5 Memory space overhead of M-ASLR.

```
mov rdx, rsi;
mov rax, rdi;
call 400470 <strcpy@plt>;
```

(a) Orignal assembly code

```
BNDCL BND0, [rdx];
BNDCU BND0, [rdx];
mov rdx, rsi;
mov rax, rdi;
call 400470 <strcpy@plt>;
```

(b) M-ASLR assembly code

Fig. 6 Functionality verification of M-ASLR: (a) shows the original assembly code, and (b) shows the assembly code compiled by M-ASLR.

after compilation is shown in Fig. 6b. M-ASLR discovers the code pointer and adds the bounds checking instructions BNDCL and BNDCU before the pointer is referenced. We then run a script to trigger the vulnerability 50 times; all of the 50 attempts trigger a #BR exception which is handled by our inserted exception handling function.

No-IP DUC: To further our testing, we use M-ASLR to compile the No-IP Dynamic Update Client (DUC) (version 2.1.9). This application fails to perform a boundary check while invoking the function *strcpy*(), which gives rise to a buffer overflow vulnerability. An adversary could exploit this vulnerability and then modify the control flow. M-ASLR fills in the vacant check and ensures that the corresponding code pointer remains in the range of the allocated memory buffer, thereby preventing the application from being vulnerable.

M-ASLR bounds the range of code pointers, mitigates buffer overflow vulnerabilities and prevents advanced attack patterns, such as the ROP exploit. Advancing on other ROP-mitigation approaches, CFI-based defense mechanisms, such as CCFIR[24] and bin-CFI[25], ensure the legality of control flow. However, researchers have responded with a new method of launching an ROP attack, thus compromising CFI[26]. On the other hand, M-ASLR adds bounds checking before each code pointer is referenced, and this effectively patches buffer overflow vulnerabilities. Consequently, the code pointer controlled by an attacker must remain within a legal range, which mitigates ROP attacks as well as other control flow hijacking attacks, such as return-to-libc[2], from the beginning.

4.3 Performance overhead

The realization of M-ASLR relies on instrumenting the source code of a target application together with its related libraries. This procedure helps M-ASLR to analyze the data flow and discover all of the code pointers. M-ASLR then performs bounds computation and bounds narrowing on code pointers. We evaluate the performance overhead of this whole procedure by adding the time-passes argument. The evaluation results for some frequently used libraries are shown in Fig. 7. Generally, the processing time is positively correlated with the size of the library; the larger the library is, the longer it takes to process. For example, for libraries with a size of several hundreds KB, such as libm-2.23 and libreadline.so.6.3, processing takes about 48 seconds; in the case of some huge libraries, such as libc-2.23, it takes about 140 seconds. However, the number of code pointers is also a factor in the processing time. For example, the size of library ld-2.23 is roughly the same as libpthread-2.23 and liblzma.so.5.0.0, but the former takes more time to process because of a greater number of code pointers.

As we have already described, M-ASLR adds bounds checking to the application. The runtime overhead of M-ASLR mainly comes from reading bounds from the Bound Table and making code pointer bounds comparisons. In order to evaluate the overhead of M-ASLR, we run the benchmark Phoronix Test Suite[27] with an optimization level of *-O2*. The evaluation results are shown in Table 2, where we see that M-ASLR introduces an average overhead of 3.3%. For test items with the primary function of processing data flows, such as encode-ape, encode-flac, and encode-mp3, M-ASLR incurs almost no overhead. The highest level of overhead is 16.55% for the test item apache, which calls for intense checking on code pointers. For most test items with a balanced proportion of code pointer operations, the runtime overhead of M-ASLR is around 2%. Comparing to other ROP mitigation approaches, we find that bin-CFI[25] imposes 4.29%, CCFIR[24] 3.6%, KBouncer[28] 4%, and ASLR-

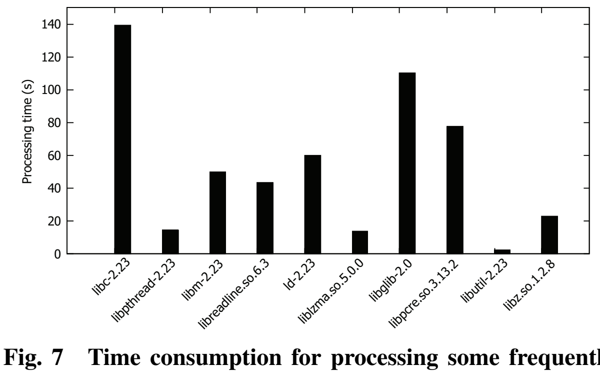

Fig. 7 Time consumption for processing some frequently-used libraries.

Table 2 Runtime overhead of M-ASLR.

Test item	Benchmark	M-ASLR	Overhead (%)
sqlite	78.98	78.65	0.42
fs-mark	44.87	43.99	1.96
iozone	78.80	78.41	0.49
openarena	30.05	29.04	3.36
tremulous	65.73	61.87	5.87
urbanterror	34.23	33.53	2.04
hmmer	48.39	47.15	2.56
gmpbench	539.26	524.19	2.79
byte	7 456 897	7 456 501	0.01
gcypt	6362	6300	0.97
tscp	441 759	433 940	1.77
john-the-ripper	1 478 751	1 477 934	0.06
himeno	370.23	346.64	6.37
c-ray	115.30	113.17	1.85
compress-pbzip2	47.89	47.07	1.71
smallpt	621	580	6.60
compress-lzma	790.23	790.08	1.75
dcraw	182.44	165.64	9.21
encode-ape	36.69	36.68	0.01
encode-flac	28.86	28.84	0.01
encode-mp3	41.19	41.18	0.01
encode-waypack	17.98	17.95	0.02
gnupg	24.07	23.58	6.19
openssl	36.80	33.01	10.30
pgbench	234.63	229.08	2.37
pybench	5751	5677	1.29
apache	9925.96	8283.21	16.55
Average			3.30

Guard[13] less than 1%. These comparison results show that the performance overhead of M-ASLR is within an acceptable range.

5 Discussion

Sophisticated attacks: The key idea of M-ASLR is to use MPX to bound code pointers and ensure they remain in the pre-assigned location. Traditional control flow hijacking attacks, such as return-into-libc[2] and ROP[3, 4], as well as more sophisticated attacks[26], require full control of a single code pointer. The attacker pre-assembles an attack payload and modifies the controlled pointer value to the start of the payload. However, the restriction on the ranges of code pointer locations provided by M-ASLR obstructs this prerequisite for an attack by preventing an attacker from manipulating the value of the code pointer to a memory address outside of the valid range. As a result, the attacker loses the capability to transfer the control flow to the specially chained attack payload. Consequently, M-ASLR is able to mitigate sophisticated control flow hijacking attacks.

Dynamic code generation: Attacks based on dynamic code generation and Just-in-Time compilation, such as JIT-ROP[12], generate an attack payload by probing the memory during runtime. This feature grants the adversary the privilege to dump memory and then bypass ASLR to hijack the control flow of the target application. The wide usage of the dynamic compilation technique in web browsers[29] and flash engines further expands the attack surface. However, the main idea of M-ASLR is to pre-assign legal bounds for each code pointer and add runtime validity checks. Due to the poor runtime boundary checking support provided by Intel MPX, M-ASLR applies a coarse-grained bounds computation method, by which runtime code pointers are treated as dynamic data structures while performing the bounds narrowing procedure. In accordance with Algorithm 2, the bounds are consistent with the outermost data structure. This strategy delivers relatively loose bounds for each runtime code pointer. Consequently, an adversary can probe the memory and joint gadgets under MPX restrictions, so attacks based on exploiting vulnerabilities during dynamic code generation are still able to seek a vulnerable code pointer to bypass M-ASLR.

Use-after-free vulnerability: Use-After-Free (UAF) attacks manipulate code pointers that are not freed on time (so-called “dangling pointers”) to alter the control flow of a target application. The checking mechanism of M-ASLR only guarantees the legal range of a code pointer before the pointer is referenced. In contrast, as their name indicates, attacks on the UAF vulnerability alter the code pointer after it is used, by which point the validity of the code pointer is no longer checked by M-ASLR. As a result, our M-ASLR mitigation approach is not a solution to UAF attacks; these need to be prevented by the programmer.

Remote control flow hijacking: Remote control flow hijacking attacks hijack the control flow of a remote web server via vulnerable web applications or services. BROP[9] stands as representative of this type of attack. BROP leverages the fact that the address space layout of a web server does not re-randomize after a crash, and exploits the same vulnerability to acquire enough memory layout information to launch an attack. The bound checking brought by M-ASLR significantly increases the robustness of target applications and renders a crash much less likely. Besides, M-ASLR would still bound the code pointer to protect the integrity of the control flow even if an attacker did manage to crash a target application. As a result, M-ASLR is an effective mitigation approach against remote control flow hijacking attacks.

6 Related Work

6.1 Defense against control-flow hijacking

Since code reuse attacks are a severe threat to modern software and applications, approaches to defending against such attacks are proposed regularly. ASLR[6] is a high-profile representative defense method. Based on the assumption that it is almost impossible for an attacker to locate the exact location of sensitive code information on the first attempt, ASLR reallocates the address space of the target application when the program is loaded into memory space. Potential attacks will then visit an incorrect place in memory and cause a crash. Being easy to deploy and featuring a relatively acceptable level of security, ASLR has been widely explored for use in modern operating systems. However, the entropy of code space brought by ASLR can be compromised through a single memory disclosure vulnerability[9–11].

In order to enhance security and robustness, enhancements to ASLR work from two main directions: finer granularity and runtime re-randomization. ASLR grants code space at module level, such that the offset within a module remains unchanged. Aiming to address this weakness, researchers have put forward finer-grained randomization approaches. For example, ASLP[20] randomizes the code space at function level, Remix[17] at basic block level, and Hiser et al.[19] at instruction level. However, as granularity increases, the runtime overhead caused by frequent addressing also rises. Besides, these enhancements to code entropy are still fragile in the face of modern attack techniques[12]. In contrast to these approaches, M-ASLR bounds the range of code pointers under the restriction of the CPU and its related architecture. This restriction overcomes the problem of buffer overflow. The Bound Table applies strict access control mechanisms, and there are no known attacks able to compromise it. Another enhancement to ASLR focuses on runtime re-randomization. Isomeron[18] keeps a copy of each function in code space, and randomly chooses the target for indirection control flow transfers during execution. However, the adversary is not prevented from launching a clone-probing attack, and a noticeable extra memory allocation is required by this method. RUNTIMEASLR introduced by Lu et al.[21] preserves the semantic state of parent processes while forking each child processes during execution. RUNTIMEASLR shows its effectiveness against clone-probing attacks such as BROP, but the scheme requires instrumentation on source code, which limits its ease of deployment.

In contrast to other ASLR-based defense schemes, M-ASLR stops memory leakage by leveraging Intel MPX to protect the integrity and validity of code pointers. CPI, proposed by Kuznetsov et al.[10], also defines a safe region away from the original application, and records the range of the code pointers, relying on two special segment registers. Although Kuznetsov et al.[10] discussed the feasibility of using the two segment registers, it cannot be guaranteed that the registers would never be modified by any real-world operating system or application. On the other hand, the registers leveraged by M-ASLR are designed by Intel to serve specifically as bounds registers. From this perspective, M-ASLR is more straightforward and offers a lower risk of incompatibility.

6.2 Research on Intel MPX

In 2013, Intel Corporation introduced to developers a new hardware-assisted memory protection approach named Intel MPX[30]. However, due to the far more complex instrumentation on source code required in order to discover the code pointers[15, 16], only limited research has focused on the use of Intel MPX. Oleksenko et al.[16] gives a systematic analysis on the features of MPX, pointing out that the specially designed hardware architecture of Intel MPX provides a security guarantee of the value of code pointers. However, discovering the code pointers remains a tricky problem for developers. In M-ASLR, we use the dynamic linker to solve this problem, upon noticing that all code pointers are loaded by the dynamic linker to give a randomized address allocation while the application is loaded. With the help of instrumentation, we are able to discover all code pointers at this stage.

7 Conclusion and Future Work

Intel's MPX provides a hardware-assisted code pointer validity check for software and applications, and features of MPX enhance control flow integrity for users. However, research on MPX-assisted security approaches remains limited. In this paper, we present an MPX-assisted address space layout randomization scheme, M-ASLR. Compared to other approaches, M-ASLR restricts the range of code pointers and thus eliminates buffer overflow, which is the prerequisite for a control flow hijacking attack.

We suggest the following extension to our research. The discussion and evaluation of M-ASLR in this paper focuses on analyzing the validity of code pointers, but leaves aside the security of code pointers used to perform dynamic code generation. However, with Just-in-Time engines being widely deployed in web browsers and flash plugins, Just-in-Time compilation now presents as an important threat vector in modern systems. We can therefore further improve M-ASLR by working to support JIT compilation.

Acknowledgment

This work was supported in part by the National Natural Science Foundation of China (No. 61272452), the National Key Basic Research and Development (973) Program of China (No. 2014CB340601), and the Natural Science Foundation of Hubei Province (No. 2017CFB663).

References

[1] S. Andersen and V. Abella, Data execution prevention, changes to functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory protection technologies, https://docs.microsoft.com/en-us/windows/desktop/memory/data-execution-prevention, 2004.

[2] M. Tran, M. Etheridge, T. Bletsch, X. X. Jiang, V. Freeh, and P. Ning, On the expressiveness of return-into-libc attacks, in *International Workshop on Recent Advances in Intrusion Detection*, Berlin, Germany, 2011, pp. 121–141.

[3] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, When good instructions go bad: Generalizing return-oriented programming to RISC, in *Proceedings of 15th ACM Conference on Computer and Communications Security (CCS'08)*, Alexandria, VA, USA, 2008, pp. 27–38.

[4] H. Shacham, The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86), in *Proceedings of 14th ACM Conference on Computer and Communications Security (CCS'07)*, Alexandria, VA, USA, 2007, pp. 552–561.

[5] J. C. Tang, M. Xu, S. J. Fu, and K. Huang, A scheduling optimization technique based on reuse in spark to defend against APT attack, *Tsinghua Science and Technology*, vol. 23, no. 9, pp. 550–560, 2018.

[6] PaX Team, PaX Address Space Layout Randomization (ASLR), https://pax.grsecurity.net/docs/aslr.txt, 2003.

[7] L. Davi, A. R. Sadeghi, and M. Winandy, ROP defender: A detection tool to defend against return-oriented programming attacks, in *Proceedings of 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS'11)*, Hong Kong, China, 2011, pp. 40–51.

[8] V. Pappas, M. Polychronakis, and A. D. Keromytis, Transparent ROP exploit mitigation using indirect branch tracing, in *Proceedings of 22nd USENIX Security Symposium (USENIX'13)*, Washington, DC, USA, 2013, pp. 447–462.

[9] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, Hacking blind, in *Proceedings of 2014 IEEE Symposium on Security and Privacy (S&P 14)*, San Jose, CA, USA, 2014, pp. 227–242.

[10] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, Code-pointer integrity, in *Proceedings of 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, Broomfield, CO, USA, 2014, pp. 147–163.

[11] H. Shacham, M. Page, B. Pfaff, E. J. Goh, N. Modadugu, and D. Boneh, On the effectiveness of address-space randomization, in *Proceedings of 11th ACM Conference on Computer and Communications Security (CCS'04)*, Washington, DC, USA, 2004, pp. 298–307.

[12] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. R. Sadeghi, Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization, in *Proceedings of 2013 IEEE Symposium on Security and Privacy (S&P'13)*, San Francisco, CA, USA, 2013, pp. 574–588.

[13] K. J. Lu, C. Y. Song, B. Lee, S. P. Chung, T. Kim, and W. K. Lee, ASLR-Guard: Stopping address space leakage for code reuse attacks, in *Proceedings of 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, Denver, CO, USA, 2015, pp. 280–291.

[14] Intel, Intel® 64 and IA-32 architectures software developer's manuals, https://software.intel.com/en-us/articles/intel-sdm, 2016.

[15] S. Ramakesavan and J. Rodriguez, Intel® memory protection extensions enabling guide, https://software.intel.com/en-us/articles/intel-memory-protection-extensions-enabling-guide, 2016.

[16] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, Intel MPX explained: An empirical study of intel MPX and software-based bounds checking approaches, arXiv preprint arXiv:1702.00719, 2017.

[17] Y. Chen, Z. Wang, D. Whalley, and L. Lu, Remix: On-demand live randomization, in *Proceedings of 6th ACM Conference on Data and Application Security and Privacy (CODASPY'16)*, New Orleans, LA, USA, 2016, pp. 50–61.

[18] L. Davi, C. Liebchen, A. R. Sadeghi, K. Z. Snow, and F. Monrose, Isomeron: Code randomization resilient to (just-in-time) return-oriented programming, presented at 2015 Network and Distributed System Security Symposium (NDSS'15), San Diego, CA, USA, 2015.

[19] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, ILR: Where'd my gadgets go? in *Proceedings of 2012 IEEE Symposium on Security and Privacy (S&P'12)*, San Francisco, CA, USA, 2012, pp. 571–585.

[20] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, Address Space Layout Permutation (ASLP): Towards fine-grained

randomization of commodity software, in *Proceedings of 22nd Annual Computer Security Applications Conference (ACSAC'06)*, Miami, FL, USA, 2006, pp. 339–348.

[21] K. J. Lu, W. K. Lee, S. Nürnberger, and M. Backes, How to make ASLR win the clone wars: Runtime re-randomization, Presented at *the 2016 Network and Distributed System Security Symposium (NDSS'16)*, San Diego, CA, USA, 2016.

[22] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, Meltdown, arXiv preprint arXiv:1801.01207, 2018.

[23] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, Spectre attacks: Exploiting speculative execution, arXiv preprint arXiv:1801.01203, 2018.

[24] C. Zhang, T. Wei, Z. F. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, Practical control flow integrity and randomization for binary executables, in *Proceedings of 2013 IEEE Symposium on Security and Privacy (S&P'13)*, San Francisco, CA, USA, 2013, pp. 559–573.

[25] M. W. Zhang and R. Sekar, Control flow integrity for COTS binaries, in *Proceedings of 22nd USENIX Security Symposium (USENIX'13)*, Washington, DC, USA, 2013, pp. 337–352.

[26] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, Out of control: Overcoming control-flow integrity, in *Proceedings of 2014 IEEE Symposium on Security and Privacy (S&P'14)*, San Jose, CA, USA, 2014, pp. 575–589.

[27] M. Larabel and M. Tippett, Phoronix test suite, http://www.phoronix-test-suite.com, 2011.

[28] V. Pappas, kBouncer: Efficient and transparent ROP mitigation, http://www.cs.columbia.edu/∼vpappas/papers/kbouncer.pdf, 2012.

[29] S. Liang, Y. Zhang, B. Li, X. J. Guo, C. F. Jia, and Z. L. Liu, SecureWeb: Protecting sensitive information through the web browser extension with a security token, *Tsinghua Science and Technology*, vol. 23, no. 5, pp. 526–538, 2018.

[30] Intel, Introduction to Intel® memory protection extensions, https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions, 2013.

Fei Yan received the PhD degree from Wuhan University, China, in 2007. He is currently an associate professor at the School of Cyber Science and Engineering, Wuhan University, China. His research interests include system security and formal security.

Kai Wang received the BS degree from Northwestern Polytechnical University, China, in 2016. He is currently a master student at the School of Cyber Science and Engineering, Wuhan University, China. His research interests include system security and software security.