Preliminary Study of the Effectiveness and Efficiency of Intel Memory Protection Extension[*]

Rob J. Dallara
Department of Computer Science,
University of North Carolina at Chapel Hill
`dallara@live.unc.edu`

Jun Jiang
Department of Computer Science,
University of North Carolina at Chapel Hill
`jiangcj@cs.unc.edu`

Abstract

The buffer overflow attack has been a long-lasting issue in computer security, and numerous defense mechanisms against it have been proposed and studied. In this paper, we study a novel hardware mechanism proposed by Intel known as the Memory Protection Extension (MPX) which prevents buffer overflow attacks at their sources. The idea of MPX stems from a research project named HardBound, but it bears some important differences. This mechanism has not yet been shipped to real hardware, so we performed our evaluation on the effectiveness and efficiency of the Intel MPX system based on emulation. In this paper, we report the design and implementation of HyperBound, a hypervisor-based emulation of MPX for guest operating systems, as well as our evaluation of MPX on Intel Software Development Emulator.

1 Introduction

The buffer overflow vulnerability is probably the most notorious security problem in the software industry. This vulnerability has a really long history and is still a common security issue in today's computing world despite decades of efforts in investigation and defense. Buffer overflows have arisen in a variety of situations including some of the most famous attacks in the history of computing. For example, in 1988, the Morris Worm, which effectively shut down the Internet at that time, was able to spread into some victim systems via a buffer overflow vulnerability in TCP finger service [26]. More recently, the OpenSSL heartbleed vulnerability, which allows an attacker to retrieve chunks of memory from victim systems, is due to a missing bound check in its heartbeat extension [5].

Software products that are performance critical or need to access low level system resources (such as operating systems or database management systems) are often written in the C/C++ programming languages, which do not perform native bound checking. Thus, such applications have become some of the most common sources of or buffer overflow attacks in the modern computing world. These software systems are usually very large, and subtle buffer overflow vulnerabilities may exist even when secure memory access functions are widely used. Even if there is only one such bug in a victim program, an adversary can craft malicious input (usually containing a piece of shellcode) into a vulnerable program in order to hijack its control flow to wherever he or she desires.

Buffer overflows can happen in many programs under various contexts and in different ways. Decades of work have shown that it is very difficult to eradicate this type of attack. As a result, a large number of mechanisms have been proposed to mitigate buffer overflow attacks over the years. They have ranged from stack-based protection mechanisms to randomization approaches as well as solutions which operate under the principle of least privilege. Despite all these efforts, the buffer overflow is still one of the most common attacks on today's computer systems.

In order to work towards a world without buffer overflow attacks, Intel has recently introduced a new security feature known as the Memory Protection Extension (MPX). This extension provides hardware-based pointer bound checking to defeat buffer overflow attacks. We investigated this extension under a variety of circumstances to figure out both how effective it is against buffer overflow attacks and how much overhead it introduces. The results in this paper will helpfully suggest whether this security extension should be deployed in the future, and if yes, how.

The remainder of the paper is organized as follows. Section 2 gives a brief introduction of the Intel Memory Protection Extension for readers who are not familiar

[*]To appear in the Proceedings of the 1st UNC Computer Science Department Workshop on Software Security (UNCSS '14)

with it. The design and implementation of HyperBound, a hypervisor-based MPX emulator, is given in Section 3, while Section 4 will provide our evalutation result of the effectiveness and efficiency of Intel MPX. We give discussion in Section 5 and related work in Section 6. Section 7 concludes the paper.

2 Preliminaries of Intel MPX

We introduce Intel MPX by synthesizing the content described in the developer's manual [13]. Readers familiar with this extension may want to skip this section.

2.1 Architectural Changes

Usually, a processor extension introduces new instructions and new registers, may modify the semantics of existing instructions, and even introduce new privilege levels. MPX is no exception, and the architectural changes of MPX include:

1. Four new bound registers to store bound information. A bound register contains two parts, lower bound and upper bound. Note that the upper bound is represent in the form of 1's complement, so a bound register with value 0 means that it allows the whole memory address space.
2. Eight new bound instructions to operate on bound registers. The opcodes, mnemonics, and descriptions are given in Tab. 1.
3. Three registers for configuration and status. `BNDCFGU` stores configuration of user mode (ring 3), while `BNDCFGS` stores configuration for supervisor mode (ring 0 to ring 2). The configuration registers store the bound directory starting address, whether to preserve bound registers through control flow change, and whether to enable Intel MPX. `BNDSTATUS` store the status including the error code and the error bound directory entry.
4. Changes to the semantics of `CPUID` to report the support for MPX. This includes setting bit 14 of `ebx` (indicating support of MPX), the setting of bit 4 of `eax` (indicating support of four bound registers) and bit 3 of `eax` (indicating support for status and configuration registers) when `CPUID` is called with `eax` = 13 and `ecx` = 0.
5. Changes to the semantics of all *call*, *return* and *jump* instructions. When these instructions are not prefixed by `F2` and the preserve bound register bit in `BNDCFGU`/`BNDCFGS` is not set, these instructions reset all bound registers.
6. Changes to the semantics of `XSAVE` and `XRSTOR` instructions. They can be used to enable access to configuration and status registers

There are also other minor changes in Intel MPX such as control register bits. We do not put all changes here but important ones.

2.2 How to Use

Intel MPX is used in a way that collaborates with the compiler, runtime and operating systems. Currently, to add MPX to an application, the source code must be compiled with an MPX-enabled compiler using the "-fmpx" parameter, and then it must be linked against the MPX runtime library. The compiler analyzes the code and adds bound instructions accordingly. Essentially, when the compiler sees a stack-based array definition, it generates a `BNDMK` instruction that creates a bound for that array. After that, if the array is accessed with an index, the compiler generates `BNDCL` and `BNDCU` instructions to compare it with the array's bounds. Note that the compiler does not generate `BNDMK` instructions for dynamically allocated pointers, so programmers need to add compiler intrinsics `__bnd_set_ptr_bounds` to manually cause the compiler to generate `BNDMK`. Since there are only four bound registers, MPX provides a bound table to save temporarily unused bounds, which can be accessed through `BNDSTX` and `BNDLDX`. This is similar to a "page file" for main memory. The compiler keeps track of the assignment of pointers and thus is able to use correct bounds. The compiled object code needs to be linked against a runtime library which turns on MPX and sets up the handler for the bound exception. Also, a program in this system can be linked against any other libraries such as libc. Each linked library may or may not be MPX-enabled since MPX is not an all-or-nothing mechanism. In practice, we found that no current libraries enable MPX support.

2.3 Design Goals

We conclude that there should be at least four design goals for Intel MPX:

1. *Effectiveness*: MPX should be able to effectively defeat all buffer overflow attacks.
2. *Efficiency*: Programs using MPX should incur minimal performance overhead.
3. *Binary compatibility*: The MPX-enabled binaries should be compatible with conventional hardware
4. *Source compatibility*: The developer needs to make few, if any changes to their programs to use MPX.

Opcode	Mnemonics	Dscription
`0F 1A`	`BNDLDX`	Load bound register from a memory location within the bound directory
`0F 1B`	`BNDSTX`	Store bound register into a memory location within the bound directory
`66 0F 1A`	`BNDMOV`	Move bounds from a bound register to a bound register or a memory location
`66 0F 1B`	`BNDMOV`	Move bounds from a bound register or a memory location to a bound register
`F2 0F 1A`	`BNDCU`	Check the upper bound for a pointer (bound stored in the form of 1's complement)
`F2 0F 1B`	`BNDCN`	Check the upper bound for a pointer (bound stored not in the form of 1's complement)
`F3 0F 1A`	`BNDCL`	Check the lower bound for a pointer
`F3 0F 1B`	`BNDMK`	Make bound for a pointer in a bound register

Table 1: Intel MPX instructions

2.4 Comparison with HardBound

The idea of Intel MPX stems from a research project in University of Pennsylvania named HardBound [6], but it has some substantive differences.

1. HardBound uses (shadow) memory to save bounds, while MPX uses bound registers backed by a bound directory.
2. HardBound introduces a new `setbound` instruction to create bounds, but uses normal move instructions to move bounds, and check bounds during dereferences, both implicitly. MPX introduces more instructions to perform these tasks explicitly.
3. HardBound tracks pointer assignment and arithmetic through move and arithmetic instructions, while in MPX the compiler is responsible for keeping track of pointer assignment and arithmetic and using correct bounds for checks.

3 HyperBound: Hypervisor-based Bound Checking via Emulation of Intel MPX

We tried to implement the unshipped Intel MPX feature in a hypervisor so that guest operating systems with supporting hardware could use this feature. However, due to a few unforeseen challenges and limited time, we were only able to implement part of its functionality and introduce a few workarounds.

3.1 Hypervisor Background

We introduce the virtualization environment that we use in this project. Our system is built on a Xen hypervisor residing on a processor which already supports Intel VT, a hardware assisted virtualization solution. Readers familiar with Intel VT and Xen may want to skip this section.

A hypervisor is a piece of software which manages multiple virtual machines running on the same hardware. It can be vividly explained as the "operating system of operating systems". Intel VT introduces a new privilege level known as VMX root mode (unofficially called ring -1), and previous ring 0 to ring 3 belong to VMX non-root mode. When the virtualization extension is enabled, the hypervisor runs under VMX root mode, and some sensitive instructions change their semantics and trap into the hypervisor so that it can handle them for correct virtualization. For example, if a guest OS wants to switch the context, it will write the page table base address to the control register `cr3`. Normally this instruction can be performed by the guest OS kernel, but when virtualization is enabled, this instruction (usually `mov cr3, eax`) will trap into the hypervisor through a mechanism called *VMexit* so that the hypervisor can prepare correct "shadow page tables" for the guest OS to use. When the hypervisor finishes its task, *VMentry* will give processor time back to the guest OS. Thus, the guest OS and the hypervisor alternately occupy the processor, and the hypervisor is only activated when the guest OS needs it to handle something. This trap/emulation paradigm has been used in a number of applications for security and other purposes.

3.2 Threat Model and Assumption

A key assumption is that the hypervisor itself is trusted. This is a common assumption in a number of research efforts that take advantage of hypervisors to perform security or other tasks [10, 18, 22, 40]. This assumption may not always hold, since vulnerabilities of hypervisors such as Xen do appear from time to time. However, it's reasonable to exclude attacks on the hypervisor from the scope of this work. Also, a number of research efforts have been proposed to protect the security of hypervisor itself [2, 31, 32].

3.3 Challenges and Design

The main challenge here is that the newly introduced bound instructions are `NOP`s on conventional hardware, which means that they do not cause any type of *VMexit* and cannot be directly intercepted by the hypervisor.

Thus, it's necessary to deliberately make a few events that could trap into the hypervisor. A somewhat "canonical" way to do this is through debug exceptions. The x86 architecture provides four hardware debug registers that can cause debug exceptions which will be intercepted by the hypervisor when enabled. This mechanism is useful here since a *VMexit* is generated when specific instructions are to be executed. Thus, the following major challenge is identification of the locations of the new bound instructions in MPX-enabled programs. One possible solution is to disassemble the instructions from a current execution location (`eip`). We actually do not need to get the disassembled instructions, but we are instead only concerned about the lengths of the instructions and whether we can continue to disassemble (which means that we do not meet any control flow instructions that could take the program to another basic block).

So, to start, we disassemble from the current `eip` and find an instruction of interest, which is either a control flow instruction that changes the program's control flow or an MPX instruction that need emulation. We then put that instruction address in the debug register and wait for a corresponding debug exception to happen and be intercepted by the hypervisor. We then emulate that instruction if it is an MPX instruction. After that, we allow the program to continue its execution. If it is an MPX instruction, it will be interpreted as a `NOP`. If it a control flow instruction, it may change `eip` to a new location in the program. To retain control, we set the trap flag in `EFLAGS` to 1, so execution of that instruction also causes a debug exception and traps into the hypervisor and we can start another loop.

While the above scheme seems good, there are also other challenges that must be considered. For example, an asynchronous control flow such as an interrupt can break this paradigm. Specifically, a clock interrupt can cause a context switch, which means that the same linear address may refer to a different physical address. Also, some instructions where breakpoints are set may cause a *VMexit* to occur for another reason. For example, if a breakpoint was set at an instruction that writes to `cr3`, then two reasons can cause a *VMexit* to arise. In this circumstance, we do not yet know about the priorities of the two reasons and this may cause a few problems. As a result, our implementation includes quite a few workarounds which are described and explained in the next section.

3.4 Implementation Details and Adopted Workarounds

Approximately 1100 lines of code were added to the Xen source. Approximately 650 lines of this code are basically disassembler, and 450 lines are used for VMexit handling. The real implementation is different from the design in a number of ways due to several difficulties. Here we present some implementation details and workarounds:

First, we use undefined instructions to signal the start and end of emulation. Our previously conceived approach was to intercept context switches and identify whether the upcoming process was the target program by comparing the process name and the target program's name. However, we were unable to make this approach work correctly. As a result, we take an alternative approach and use undefined instructions `UD2` (opcode `0F0B`) to signal the start of emulation and `UD1` (opcode `0FB9`) to signal the end of emulation. These undefined instructions will cause the invalid opcode exception (`#UD`) which can be intercepted by the hypervisor. In the handler, the opcode is checked and if it indicates that the emulation should start, we set the debug exception bit in the exception bitmap so that debug exceptions can be intercepted by the hypervisor and the debug register bit in `dr7` so that the debug register is enabled (we use `dr0`). Otherwise, the debug exception bit is cleared from the exception bitmap, and dr0 is disabled.

Second, when the emulation starts, we disassemble from the current `eip` and find an MPX instruction, a control flow instruction, or a "too-lazy-to-handle" instruction. A "too-lazy-to-handle" instruction is a really uncommon instruction that we do not want to disassemble but rather put a breakpoint there. This does not break the debug exception handler since we emulate only MPX instructions and single step all other instructions.

Last, we do not handle the case of interrupts happening during emulation, but we handle context switches which cause the address space to change in the guest OS. When a context switch occurs, we examine whether the emulation is on. If yes, that means the current current process is the target program, so we record the value of control register `cr3` and turn off emulation. Later on, when a context switch happens with a new page table base address equal to what was recorded, our system recognizes that our target program is upcoming and turn on emulation. This could be problematic in rare cases which did not happen in our experiment.

3.5 Discussion of Limitations and Future Work

The first major limitation of our approach is that it uses too many workarounds, such as the use of illegal instructions to signal start and end, making it not compatible with an MPX-enabled binary. We plan to get rid of this in the future, if possible.

Another limitation is that we do not support configuration and status registers. The manual does not provide enough information about how to use these registers but only mentions that BNDCFGU and BNDSTATUS can only be accessed through XSAVE/XRSTOR instructions and BNDCFGS can only be accessed through RDMSR/WRMSR instructions. Detailed information about how to use these registers resides in the MPX runtime library which has no available source code. This means that we cannot know the bound directory starting address and cannot emulate BNDLDX/BNDSTX with fidelity. Instead, we store the bounds in the hypervisor's memory space instead of the target program's memory space (within the guest OS). Another problem in our system is that we cannot access the BNDSTATUS register so when we inject a bound exception into the guest OS, we cannot specify the error code. This confuses the handler in the MPX runtime and causes it to behave in an abnormal manner. A solution to this problem is to reverse-engineer the runtime library. This is doable but requires a large amount of time.

We also do not support control flow instructions with F2 prefix. The original semantics for control flow instructions indicates that if an instruction is prefixed by F2 or the bound configuration register indicates preserving bounds in control flow transfer by default, the execution of that instruction does not clear the bound registers. Otherwise, bound registers are reset. Our implementation does not support this feature, so the bounds are always preserved during such instruction. This feature can be easily added to our current framework by just adding these instructions to our list of interesting instructions that need a breakpoint to be set.

In addition, our implementation does not support the BNDCN instruction. BNDCN checks the upper bound which is not in the form of 1's complement. Since the BNDMK instruction creates the upper bound in 1's complement form. So, to use BNDCN, one needs to move the bound register to a memory location, change the representation of the upper bound, and move that back to the bound register. Since the BNDCU instruction exist which checks the upper bound in 1's complement form, there is no reason to use this nonsensical BNDCN instruction. Actually, the MPX-enabled compiler never generates this instruction, so we do not handle it. For fidelity consideration, we may handle it in the future, and the handler should be very similar to one of BNDCU.

Our current system only supports 32-bit guest operating systems, but the effort to migrate it to 64-bit OSes should be pretty minimal. The 64-bit machine code can be a little bit different, such as REX prefixes replace INC instructions, and changes to default operand size. We may want to update the disassembler a little bit and use larger space to store bound information.

At last, though experiments show that this system is effective, it may incur pretty high performance overhead. The main source of overhead is due to the disassembling-breakpoint loop. Since new bound instructions are NOPs in conventional hardware, this is the only approach to intercept and emulate them. We do not do the performance test but we expect the overhead to be between 100% and 1000%. One possible solution is that we pre-process the program and replace the interesting instructions with ones that could cause *VMexit* such as `int 3` (opcode CC). This however again sacrifices compatibility.

4 Evaluation of Intel MPX on Software Development Emulator

We perform evaluation of MPX based on the software development emulator provided by Intel. We mainly focus on effectiveness and efficiency in this study.

4.1 Threat Model

The threat model in our investigation is canonical among most defenses against buffer overflows. A program is assumed to have some type of buffer overflow vulnerability. An attacker knows this vulnerability and is able to access the input interface of the program, either locally or remotely. Thus, the attacker can use any input that is beneficial to him or her, which has arbitrary length and bears arbitrary functionality. It is assumed that the attacker can access powerful computing resources, will spend a large amount of time on his or her attack, is highly experienced in writing shellcode, and has a strong will and determination to cause as much damage as possible to the victim system. However, for our purposes, it is assumed that the attacker only targets a specific program and nothing else. Also, no memory protection mechanisms other than the Intel Memory Protection Extension are used on the victim system.

4.2 Experiment Setup

Simulation of Intel MPX requires support from a series of software packages including compilers, runtime libraries, and operating systems. We obtain the MPX-enabled GCC compiler and the software development emulator (SDE) from Intel's website [24, 28]. Since Intel MPX works differently between kernel mode and user mode (and current OS distributions do not support the MPX kernel feature), only user-level experiments were performed.

4.3 Effectiveness Evaluation

Intel MPX is very effective in preventing buffer overflow attacks. In order to actually test its security effectiveness, a variety of C programs vulnerable to buffer overflow attacks were obtained and written, and then they were compiled to run on Intel SDE using the MPX-supported GCC compiler. In addition, two pieces of shellcode were written, and two others were obtained from external sources in preparation for the buffer overflow attacks against the emulated Intel MPX system.

The first attack tried against Intel MPX was a simple buffer overflow attack against a program called `simple_example.c` during which Address Space Layout Randomization (ASLR) was turned off. For this attack to succeed, the function's return address was overwritten to point somewhere within a `NOP` sled leading up to the shellcode. Since the attacked buffer was fixed in memory from iteration to iteration, return addresses were sequentially guessed from higher values to lower values. Since the program did not make any deep sequences of function calls, the guessed return address was decremented to a correct value within a few thousand iterations. This attack was tested on four different pieces of shellcode: one which shuts down a Linux system, one which prints "`Hello, World!`" [1], one which kills all processes [16], and one which spawns a shell [7].

After this attack was completed, a simple return-to-libc attack was carried out against a different program. In order to carry out this attack, the attack input was padded right up to the return address with random bytes, and the return address itself was overwritten by the address of the `system` function for that particular program. The next piece of memory (four addresses higher) was overwritten by the address of the `exit` function, and the following piece of memory was overwritten by a pointer to a "`/bin/sh`" string. All of this was done to cause the victim program to make a series of returns to create a shell. A shell was successfully spawned through this attack (although the addresses of the functions and "`/bin/sh`" string were printed out since they varied slightly from program to program and GDB).

Finally, a second buffer overflow attack was carried out against `simple_example.c` with ASLR enabled on the native Ubuntu machine. As a result, all programs running inside the emulator (with and without MPX) effectively ran with the protection of ASLR. For this particular attack, the piece of shellcode to spawn a root shell was inserted into the attack input during each trial. After 13,692 attempts of guessing the same return address over and over again through an automated command line loop, a shell was spawned.

After all of these attacks were carried out with MPX mode disabled, each of these attacks was attempted with MPX turned on. Fortunately, all of the previous three attacks on `simple_example.c` failed due to a bound exception when MPX mode was turned on for each of the previous cases. In addition, simple buffer overflow attacks without shellcode were carried out on the following test programs: `notetaker.c` (from [7], Chapter 2), `notesearch.c` (from [7], Chapter 2), `auth_overflow.c` (from [7], Chapter 3), `bounded_auth_overflow.c` (`auth_overflow.c` with an MPX compiler intrinsic added to protect the buffer), `exploit_example.c` (adapted from [21]), `guessing_game.c`, `memcpy_example.c`, `mpx.c` (from [29]), `secure_auth_overflow.c` (`auth_overflow.c` with its own MPX-compatible string copy function), `secure_memcpy.c`, `set_index.c`, and `test.c`, both with and without memory protection. When MPX mode was turned off, all of the buffer overflows succeeded. When MPX was turned on, the attacks failed on `mpx.c`, `secure_auth_overflow.c`, `secure_memcpy.c`, and `test.c`. The buffer overflow attacks still succeeded on the other programs due to the lack of MPX support in vulnerable C library functions used by these programs.

[1]The first two pieces of shellcode were written by us.

4.4 Efficiency Evaluation

In order to evaluate the efficiency of the Intel MPX system, two programs (`1st_perf_test.c` and `2nd_perf_test.c`) were written which repeatedly accessed different indices of an array, but only valid ones. Each program used `rdtsc` to measure the number of clock cycles between the beginning and the end of each execution. In addition, a small, general-purpose program called `run_perf_test.c` was written to automatically test each of them (with and without MPX) 500 times. The time results were printed out during each program execution, converted into milliseconds based on the 2.4 GHz clock speed of the processor on which the emulator was running, and averaged out for each test case in each mode in Tab. 2.

	`1st_perf_test.c`	`2nd_perf_test.c`
MPX off	34.97	28.31
MPX on	100.37	77.45
Overhead	187%	174%

Table 2: Intel MPX Performance

5 Discussion

Overall, the Intel MPX system is fairly effective in preventing buffer overflow attacks whenever its memory protection feature is turned on. Although all of the buffer overflow attacks eventually succeeded whenever MPX was turned off, all of those same attacks failed whenever they targeted memory in ways which MPX can currently detect. This makes the Intel MPX system very promising in terms of future deployment on real computer systems, especially once MPX support is also added to common vulnerable functions in the libraries of C/C++.

The performance overhead of the MPX feature in this system ranged from approximately 175% to 187% depending on the performance test program on which it ran (the different performance programs accessed array indices different numbers of times). This may be due to the fact that the new bound checking instructions take a particularly long time to execute, but a more reasonable explanation is that the overhead of these bound checking instructions may be higher in the emulator than what they would be on an actual Intel MPX hardware platform. We actually expect the real performance of MPX to be really high.

Another major strength of the Intel MPX is that it requires few, if any changes to the high level C/C++ source code, making it extremely compatible with many current and legacy C/C++ applications. This is due to the fact that its memory protection is implemented through bound registers at the hardware level, so only the assembly code must actually be changed in most cases, which is taken care of by the compiler. Whenever extra instructions are needed at a high programming level, it is usually because of the current lack of Intel MPX implementation in vulnerable C program library functions. In addition, since only memory accesses need protection, the number of instructions in a program compiled in the Intel MPX system doesn't seem to go up dramatically when the bound instructions are added. In fact, our executable programs designed to run in the Intel SDE were actually a little bit smaller than the natively compiled versions for the Ubuntu system (perhaps due to the fact that more libraries were linked into the Ubuntu-based executables). For example, the native compiled version of `simple_example.c` is 8729 bytes in size, while the MPX-enabled version is 5928 bytes.

6 Related Work

If shipped to the real hardware, Intel MPX will be the only available hardware-based memory safety mechanism designed to actually prevent buffer overflow attacks. Many software solutions to prevent or mitigate buffer overflow attacks, however, do exist, and can be categorized into the following:

6.1 Stack Protection

The stack is a dangerous place which buffer overflows frequently target. A number of buffer overflow attacks stem from stack smashing. There continue to be multiple research efforts which focus on protection of the stack. StackGuard [4] puts a "canary" value next to the return pointer for each function call. This canary value is randomly generated and is usually difficult for attackers to guess. If a buffer overflow overwrites the return address of a function, then it must have also overwritten the canary value, allowing the attack to be detected. StackGhost [8] simply encrypts the return address of a function when it is called and decrypts it when that same function returns. The encryption is carried out through a simple exclusive or. Thus, whenever a return address is overwritten by an attacker, the control flow will not follow the attackers intention when the return address is decrypted upon return. On the other hand, Stack Shield [30] copies the return address to a safe place and checks whether the stack version and the safe version are the same when the function returns. All of these ideas rely on the protection of return addresses so that an attacker cannot take over a program's control flow, whereas Intel MPX attempts to prevent out of bounds memory accesses in the first place.

6.2 Randomization Method

A number of approaches based on randomization can also be used to defeat buffer overflow attacks. Instruction Set Randomization [17] creates a runtime environment for a process and uses a key to encrypt all instructions. The Randomized Instruction Set Emulator [3] proposes a similar approach, but these two efforts use different methods and work for different platforms and systems. These two methods are effective against code injection attacks since any injected code will be treated as cipher code and execute only after decryption. Unfortunately, these randomization approaches introduce such high performance overheads that they cannot be used at all in practice. On the other hand, Address Space Layout Randomization [36] randomizes the memory spaces of different programs in their various executions, rendering it impossible for attackers to hard-code many addresses. There are also a number of extended works based on randomization such as RandSys [15], STIR [33] and practical ISR using AES [12].

6.3 Application Sandboxing

Application sandbox-based solutions confine the behavior of an application to a safe subset. Most sandboxes do this by targeting system calls that substantially influence their underlying computer system. Janus [11] is one of the earliest works in this direction. While it is relatively simple and works fine for many situations, it does possess a few deficiencies such as Time-of-check-to-time-of-use (TOCTTOU) race conditions. Systrace [23] uses a user-kernel hybrid approach to resolve these deficiencies. Ostia [9] leverages a new "delegating" architecture to improve flexibility and compatibility. A more recent sandbox-based solution is TxBox [14], which employs the concept of "transactions" from databases into sandboxes, thus allowing rollback from a corrupted state. In summary, sandboxes limit the damage buffer overflows can cause but do not actually prevent buffer overflow attacks from occurring in the first place.

6.4 Miscellany

The mechanism of control flow integrity [1] enforces a policy which prevents a program's control flow from deviating from a model created (in most cases) by program analysis. A number of extensions employing this technique exist such as CFIMon [39], BinCFI [42] and CCFIR [41].

W$\oplus$X [38] is a mechanism that successfully defeats code injection attacks carried out through traditional buffer overflows. The essential idea of this mechanism is to make segments of memory either writable or executable, but never both. This mechanism has been implemented in both operating systems and processors, and is widely deployed in today's computing world. However, the W$\oplus$X approach is ineffective against return-into-libc attacks [27, 35] which can circumvent this type of policy.

The Harvard computer architecture [37] is another solution against code injection by buffer overflow. A Harvard architecture-based computer actually has separate memories for data and program code. The stack region is always stored in the data memory and is thus never executed. On the other hand, most modern computers use von Neumann architectures on which data and code memory are not separated. Interestingly, some von Neumann architectures such as the Intel x86 system have separate data and code caches, allowing for software implementations of Harvard architectures on these machines [25].

Another key to defeating multiple types of buffer overflow attacks is the principle of least privilege. Given appropriate privileges, attacked software will usually not cause much harm to its underlying computer system. A number of efforts follow this direction, such as capability systems [34] and privilege separation [23].

We consider pointer bound checking to be the best possible defense against buffer overflow attacks. While type-safe languages such as Java and Python already do this, C/C++ do not. As a result, additional mechanisms must be added to vulnerable software or underlying hardware. A number of such efforts exist, such as SoftBound [20], HardBound [6] and WatchdogLite [19].

7 Conclusion

Intel MPX is very effective in preventing buffer overflow attacks at their sources. Using bounded pointers at the hardware level make it more efficient than many other bound checking based solutions available to date [6, 19, 20]. While many solutions exist to prevent or mitigate buffer overflow attacks, this solution prevents buffer overflow attacks from their source through bound registers in hardware, requiring few if any changes to already existing C/C++ code. The performance overhead in this system was somewhat higher than expected, but the real hardware implementation should be much more efficient. To sum up, this mechanism is promising and should be ported to hardware and used widely to meet the security demands in the future computing world.

References

[1] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2005), CCS '05, ACM, pp. 340–353.

[2] AZAB, A. M., NING, P., WANG, Z., JIANG, X., ZHANG, X., AND SKALSKY, N. C. HyperSentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 38–49.

[3] BARRANTES, E. G., ACKLEY, D. H., PALMER, T. S., STEFANOVIC, D., AND ZOVI, D. D. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2003), CCS '03, ACM, pp. 281–289.

[4] COWAN, C., PU, C., MAIER, D., HINTONY, H., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium* (Berkeley, CA, USA, 1998), Sec '98, USENIX Association.

[5] CYBEROAM NETWORK SECURITY APPLIANCES. OpenSSL heartbleed vulnerability fix. http://kb.cyberoam.com/..., 2014. Online. Apr 14, 2014.

[6] DEVIETTI, J., BLUNDELL, C., MARTIN, M. M. K., AND ZDANCEWIC, S. HardBound: architectural support for spatial safety of the C programming language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2008), ASPLOS XIII, ACM, pp. 103–114.

[7] ERICKSON, J. *Hacking: the art of exploitation*. No Starch Press, 2008.

[8] FRANTZEN, M., AND SHUEY, M. StackGhost: hardware facilitated stack protection. In *Proceedings of the 10th USENIX Security Symposium* (Berkeley, CA, USA, 2001), Sec '01, USENIX Association.

[9] GARFINKEL, T., PFAFF, B., AND ROSENBLUM, M. Ostia: a delegating architecture for secure system call interposition. In *NDSS* (2004).

[10] GARFINKEL, T., AND ROSENBLUM, M. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Annual Network and Distributed Systems Security Symposium* (2003), NDSS '03.

[11] GOLDBERG, I., WAGNER, D., THOMAS, R., AND BREWER, E. A. A secure environment for untrusted helper applications: confining the wily hacker. In *Proceedings of the 6th USENIX Security Symposium* (Berkeley, CA, USA, 1996), Sec '96, USENIX Association.

[12] HU, W., HISER, J., WILLIAMS, D., FILIPI, A., DAVIDSON, J. W., EVANS, D., KNIGHT, J. C., NGUYEN-TUONG, A., AND ROWANHILL, J. Secure and practical defense against code-injection attacks using software dynamic translation. In *Proceedings of the 2nd International Conference on Virtual Execution Environments* (New York, NY, USA, 2006), VEE '06, ACM, pp. 2–12.

[13] INTEL. Architecture instruction set extensions programming reference. https://software.intel.com/..., 2014. Online. Mar, 2014.

[14] JANA, S., PORTER, D. E., AND SHMATIKOV, V. TxBox: building secure, efficient sandboxes with system transactions. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2011), S&P '11, IEEE Computer Society, pp. 329–344.

[15] JIANG, X., WANGZ, H. J., XU, D., AND WANG, Y.-M. RandSys: thwarting code injection attacks with system service interface randomization. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems* (Washington, DC, USA, 2007), SRDS '07, IEEE Computer Society, pp. 209–218.

[16] KATTERJOHN, K. linux/x86 kill all processes 11 bytes. http://www.exploit-db.com/exploits/13345/, 2006. Online. Nov 13, 2006.

[17] KC, G. S., KEROMYTIS, A. D., AND PREVELAKIS, V. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2003), CCS '03, ACM, pp. 272–280.

[18] LITTY, L., LAGAR-CAVILLA, H. A., AND LIE, D. Hypervisor support for identifying covertly executing binaries. In *Proceedings of the 17th USENIX Security symposium* (Berkeley, CA, USA, 2008), Sec '08, USENIX Association, pp. 243–258.

[19] NAGARAKATTE, S., MARTIN, M. M. K., AND ZDANCEWIC, S. WatchdogLite: hardware-accelerated compiler-based pointer checking. In *Proceedings of 12th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (New York, NY, USA, 2014), CGO '14, ACM, pp. 175:175–175:184.

[20] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., AND ZDANCEWIC, S. SoftBound: highly compatible and complete spatial memory safety for C. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2009), PLDI '09, ACM, pp. 245–258.

[21] PALMER, T. Course lecture 04c. http://people.cs.umass.edu/..., 2008. Online. 2008.

[22] PAYNE, B. D., CARBONE, M., SHARIF, M., AND LEE, W. Lares: an architecture for secure active monitoring using virtualization. In *Proceedings of the 29th IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2008), S&P '08, IEEE Computer Society, pp. 233–247.

[23] PROVOS, N., FRIEDL, M., AND HONEYMAN, P. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium* (Berkeley, CA, USA, 2003), Sec '03, USENIX Association.

[24] RB. Introduction to Intel memory protection extensions. http://software.intel.com/..., 2013. Online. Jul 16, 2013.

[25] RILEY, R., JIANG, X., AND XU, D. An architectural approach to preventing code injection attacks. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (Washington, DC, USA, 2007), DSN '07, IEEE Computer Society, pp. 30–40.

[26] SEELEY, D. A tour of worm. http://www.thehackademy.net/..., 1988. Online. Nov, 1988.

[27] SHACHAM, H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2007), CCS '07, ACM, pp. 552–561.

[28] TAL, A. Introducing the Intel software development emulator. http://software.intel.com/..., 2012. Online. Jun 15, 2012.

[29] TAL, A. Using Intel MPX with the Intel software development emulator. https://software.intel.com/..., 2013. Online. July 23, 2013.

[30] VENDICATOR. Stack shield. http://www.angelfire.com/..., 2000. Online. Jan 8, 2000.

[31] WANG, Z., AND JIANG, X. HyperSafe: a lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of 31rd IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2010), S&P '10, IEEE Computer Society, pp. 380–395.

[32] WANG, Z., WU, C., GRACE, M., AND JIANG, X. Isolating commodity hosted hypervisors with HyperLock. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 127–140.

[33] WARTELL, R., MOHAN, V., HAMLEN, K. W., AND LIN, Z. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 157–168.

[34] WATSON, R. N. M., ANDERSON, J., LAURIE, B., AND KENNAWAY, K. Capsicum: practical capabilities for UNIX. In *Proceedings of the 19th USENIX Security Symposium* (Berkeley, CA, USA, 2010), Sec '10, USENIX Association.

[35] WIKIPEDIA. Return-to-libc attack. http://en.wikipedia.org/..., 2013. Online. Oct 17, 2013.

[36] WIKIPEDIA. Address space layout randomization. http://en.wikipedia.org/..., 2014. Online. Jan 28, 2014.

[37] WIKIPEDIA. Harvard architecture. http://en.wikipedia.org/..., 2014. Online. Jan 29, 2014.

[38] WIKIPEDIA. W^X. http://en.wikipedia.org/wiki/W%5EX, 2014. Online. Jan 29, 2014.

[39] XIA, Y., LIU, Y., CHEN, H., AND ZANG, B. CFIMon: detecting violation of control flow integrity using performance counters. In *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (Washington, DC, USA, 2012), DSN '12, IEEE Computer Society, pp. 1–12.

[40] XIONG, X., TIAN, D., AND LIU, P. Practical protection of kernel integrity for commodity OS from untrusted extensions. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium* (2011), NDSS '11.

[41] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2013), S&P '13, IEEE Computer Society, pp. 559–573.

[42] ZHANG, M., AND SEKAR, R. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Security Symposium* (Berkeley, CA, USA, 2013), Sec '13, USENIX Association, pp. 337–352.

A Respective Contribution of Each Author

A.1 Contribution of Rob J. Dallara

Rob did research on evaluating Intel MPX on software development emulator. Rob co-wrote or independently wrote Section 1, Section 4, Section 5, Section 6.4 and Section 7. Rob did the final proofreading.

A.2 Contribution of Jun Jiang

Jun did research on designing and implementing HyperBound. Jun co-wrote or independently wrote Abstract, Section 1, Section 2, Section 3, Section 4.1, Section 4.2 and Section 6. Jun created the structure of the paper and typeset the paper in LaTeX.