

Received February 23, 2018, accepted April 12, 2018, date of publication May 14, 2018, date of current version July 19, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2835838

BoundShield: Comprehensive Mitigation for Memory Disclosure Attacks via Secret Region Isolation

HAI JIN¹, (Senior Member, IEEE), BENXI LIU¹, YAJUAN DU², AND DEQING ZOU^{1,3}

¹Services Computing Technology and System Lab, Big Data Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China

²School of Computer Science, Wuhan University of Technology, Wuhan 430070, China

³Shenzhen Huazhong University of Science and Technology Research Institute, Shenzhen 518057, China

Corresponding author: Yajuan Du (dyj@whut.edu.cn)

This work was supported in part by the National 973 Fundamental Basic Research Program under Grant 2014CB340600 and in part by the Shenzhen Fundamental Research Program under Grant JCYJ20170413114215614.

ABSTRACT Address space layout randomization (ASLR) is now widely adopted in modern operating systems to thwart code reuse attacks. However, an adversary can still bypass fine-grained ASLR by exploiting memory corruption vulnerabilities and performing memory disclosure attacks. Although *Execute-no-Read* schemes have been proven to be an efficient solution against read-based memory disclosures, existing solutions need modifications to kernel or hypervisor. Besides, the defense of execution-based memory disclosures has been ignored. In this paper, we propose BoundShield, a self-protection scheme that provides comprehensive protection against memory disclosure attacks, especially against those based on executing arbitrary code by leveraging Intel *Memory Protection Extension*. BoundShield protects code memory by defending not only read-based memory disclosure attacks but also execution-based memory disclosure attacks. On one hand, read-based memory disclosures can be eliminated by hiding all code sections and code pointers in a secret region separated from the user address space. On the other hand, BoundShield prevents return addresses from being corrupted and ensures that all function pointers point to the legitimate entries whenever they are dereferenced, which significantly reduces the attack surface for execution-based memory disclosures. We have implemented a prototype of BoundShield based on a set of modifications to compiler toolchain and the standard C library. Our evaluation results show that the BoundShield can provide strong defenses against memory disclosure attacks while incurring a small performance overhead.

INDEX TERMS Memory disclosure attacks, execute-only memory, software security.

I. INTRODUCTION

Code reuse attacks have been a paramount threat to software security in the last decade. Due to the widespread adoption of $W\oplus X$ [1], the attackers turned to reusing original code residing in memory rather than injecting malicious code into the address space. By injecting code pointers of existing code sequences (gadgets), an adversary can hijack the control flow of an application and force it to perform malicious behaviors [2]. To successfully launch code reuse attacks, the adversary has first to know the memory layout of the application to locate the gadgets needed to construct the attack payload.

Address Space Layout Randomization (ASLR) [3] has been proposed to counter code reuse attacks. By randomizing the base addresses of program modules, attackers are

forced to guess the locations of gadgets. Unfortunately, ASLR is vulnerable to memory disclosure attacks, in which the attacker exploits memory errors to locate gadgets on-the-fly. An adversary may either collect gadgets by reading code pages and pointers in the memory [4], or by diverting control flow to arbitrary memory addresses to figure out which codes have been executed [5], [6].

Many protection schemes have been proposed to enhance ASLR [7]–[11], raising the bar of memory disclosure attacks. One solution is information hiding [7], [8], which isolates code pointers into a safe region and only allows legitimate code to access them. Information hiding mechanisms assume that the adversary cannot infer the locations of safe regions due to the high entropy of 64-bit virtual address space.

However, it has been shown that this assumption is easily broken down by advanced attacks, which can probe these safe regions with ingenious methods [12], [13]. A more rigorous approach to mitigate memory disclosure attacks is to apply *Execute-no-Read* (XnR) system [9]–[11]. The main idea of XnR solutions is to prevent attackers from reading code pages. Typically, such solutions rely on modifications to operating system kernel or virtual machine hypervisors and aim to mark code pages execute-only. Recent solutions also protect code pointers to stop indirect memory disclosures [9], [10], in which attackers infer the locations of gadgets from leaked code pointers in memory. However, static data pointers (i.e., pointers to data sections) can also leak out information of code sections, since code sections are placed with fixed offsets from data sections. Additionally, XnR solutions mainly focus on defending read-based memory disclosure attacks [4], leaving programs still vulnerable to execution-based memory disclosure attacks [5], [6].

In this paper, we present BoundShield, a practical system providing comprehensive protection against memory disclosure attacks, especially against those based on executing arbitrary code. The key design of BoundShield is to divide the user address space into two regions with a *Software Fault Isolation* (SFI)-based mechanism, and use the secret region to hide code sections and code pointers in the memory. BoundShield prevents illegal memory accesses into the secret region by leveraging Intel *Memory Protection Extension* (MPX) to provide strong access isolation for the two regions. In details, BoundShield stops read-based memory disclosures by hiding all code sections and code pointers in the secret region. Meanwhile, BoundShield stops execution-based memory disclosures by preventing return addresses from being corrupted and checking whether function pointers point to legitimate entries when they are dereferenced. In this way, not only attacks based on reading code pages or pointers but also those based on executing arbitrary code can be prevented.

Our prototype implementation of BoundShield consists of two components: a compile-time component as a custom compiler toolchain, and a runtime component modified from the standard C library. The compile-time component is responsible for instrumenting and transforming the input program. The runtime component is in charge of setting up the memory layout and initializing runtime environment.

Our evaluation results demonstrate that BoundShield can provide a strong defense against memory disclosure attacks, and can be adopted to real-world server programs such as nginx and lighttpd. BoundShield only incurs minor overhead, in details, 5.8% slowdown for SPEC CPU2006 benchmarks on average. Moreover, BoundShield does not require any OS-level or hypervisor-level modifications and can be readily adopted to defend more complex attacks.

In summary, our contributions are listed as follows:

- 1) We have studied existing solutions on preventing memory disclosure attacks and figure out that they either provide incomplete protection or require modifications to the original operating system.

- 2) We present BoundShield, a practical approach providing comprehensive protection against both read-based and execution-based memory disclosure attacks without requiring any modifications to the original operating system.
- 3) We have implemented a prototype system of BoundShield based on a custom compiler toolchain and the standard C library with a set of modifications. Our evaluation results demonstrate that BoundShield provides a reliable defense against memory disclosure attacks, and incurs a minor performance overhead on the SPEC CPU2006 benchmarks and server programs.

II. BACKGROUND AND RELATED WORKS

This section first introduces preliminary knowledge about memory disclosure attacks and the Intel MPX. Then, related works about prevention methods against memory disclosure attacks are presented.

A. MEMORY DISCLOSURE ATTACKS

The ultimate goal of code reuse attacks is to hijack the control flow of an application and perform malicious behaviors with existing code [14]. One most common technique of code reuse attacks is *Return-oriented programming* (ROP) [2], which chains instruction sequences with a return instruction (gadgets) together to perform arbitrary operations. To successfully launch such attacks, the adversary must identify the useful gadgets in the application and shared libraries in advance. ASLR is currently deployed in modern operating systems to prevent attackers from probing memory layout information. It randomizes the base addresses of modules, forcing attackers to guess the locations of gadgets. However, an adversary can still bypass ASLR by exploiting memory corruption vulnerabilities to locate valuable gadgets on-the-fly. We discuss the methods that an attacker may use to find gadgets as follows:

1) READ-BASED MEMORY DISCLOSURES

Armed with an arbitrary memory read primitive, an adversary can either read from code pages to collect gadgets [13], or harvest enough code pointers in data regions to infer the locations of gadgets without directly reading code pages [15]. Additionally, the adversary can also use static data pointers to infer the code layout, as code sections have fixed offsets with data sections. The prerequisite for these attacks is the ability to read from any memory regions, which allows the attacker to retrieve enough information about code layout.

2) EXECUTION-BASED MEMORY DISCLOSURES

These methods identify gadgets by diverting a control transfer to an arbitrary address and measuring program behaviors to infer which codes have been executed. Such attacks can be applied to crash-tolerant applications such as server programs. Seibert et al. [6] overwrite a function pointer to different addresses within a function, and measure the execution time to identify different functions. Blind ROP [5]

overwrites return addresses to arbitrary addresses and probes the executed codes by observing application behaviors. The prerequisite for such attacks is the ability to overwrite a code pointer to arbitrary addresses, which allows attackers to probe any code gadgets.

B. INTEL MPX

Intel *Memory Protection Extension* (MPX) is proposed to provide efficient protection against memory errors and is available on Intel processors starting from Skylake microarchitecture. MPX provides four bound registers from `%bnd0` to `%bnd3` and a set of new instructions. The bound registers are 128-bit and used to store a 64-bit lower bound address and a 64-bit upper bound address. In an MPX protected program, once a pointer is made, its associated bounds are generated and stored in the memory. Whenever a pointer is dereferenced, the associated bounds are loaded into a bound register, and the pointer is checked by bound check instructions. Bound check instructions are used to check whether the address of a memory reference is in the legitimate range.

Among them, `bndcu` is used to check the upper bound, while `bndcl` is used to check the lower bound. If the pointer exceeds the bounds stored in the bound register, a `#BR` fault will be triggered and caught by the exception handler. While providing most robust security guarantees against memory errors, the standard use of MPX imposes a significant runtime overhead (over $2\times$ on SPEC benchmarks) [16]. However, previous works have shown that the vast impact on performance is mainly introduced by repeatedly loading and storing bounds rather than performing bound check instructions, which often takes little performance overhead [17], [18]. Thus, it will be efficient to apply MPX for bound checking in our proposed BoundShield technique.

C. RELATED WORKS AGAINST MEMORY DISCLOSURE ATTACKS

Many defense schemes have been proposed to thwart memory disclosure attacks. One solution is to store sensitive data (e.g., code pointers) in a safe region, which is hidden in the large 64-bit virtual address space. These techniques are based on information hiding, which hides sensitive data rather than isolates them to reduce the performance overhead. For example, *Code Pointer Integrity* (CPI) [8] separates all code pointers from other data, and stores them in the safe region. To restrict memory accesses to the safe region, CPI ensures that addresses pointing to the safe region are never stored in memory. Hence, the adversary can neither locate code pointers nor corrupt them to divert control flows.

ASLR-Guard [7] shares a similar design but exploits two safe regions to encrypt function pointers and hide return addresses respectively. Since all code pointers are encoded or separated from regular memory space, the adversary cannot locate gadgets through code pointers. Although information hiding based solutions seem to be promising, recent attacks have shown that information hiding is, actually, a weak isolation model, since sensitive data

are not truly isolated. An adversary can probe the safe regions with ingenious methods and therefore bypass the defense [12], [13], [19].

A more rigorous solution is to employ *Execute-no-Read* (XnR) system. Typically, such solutions rely on a modified OS kernel or hypervisor to mark code pages as execute-only. Backes *et al.* [20] proposed to manipulate the page fault handler to mark only a small window of code pages as readable during execution. Crane *et al.* [9], [10] proposed Readactor and Readactor++, both of which leverage a hardware-assisted virtualization support, named as *Extended Page Tables* (EPT) to mark code pages as execute-only. To further prevent indirect memory disclosures, Readactor replaces all function pointers and return addresses with substitute pointers to trampolines. Thus the code pointers can only leak out the addresses of trampolines also marked as execute-only. However, Readactor does not provide a comprehensive defense. Since there are fixed offsets between code sections and data sections, an adversary can still infer the code layout with a static data pointer. Moreover, XnR systems are typically vulnerable to execution-based memory disclosures. Even if the code pointers and return addresses are replaced with substitutes, an adversary can still overwrite them to any addresses in code pages, perform an execution-based memory disclosure to reveal the code pages, and identify gadgets.

Other solutions have been proposed to mitigate memory disclosure attacks or code reuse attacks. Re-randomization solutions [21]–[23] repeatedly randomize the code layout during program execution, leaving little probability for attackers to reuse a gadget. *Control-Flow Integrity* (CFI) solutions [24]–[26] mitigate code reuse attacks by analyzing the targets of indirect branches and checking whether code pointers are in the legitimate set during execution.

III. THE DESIGN OF BoundShield

In this section, we first define the threat model of our defense. Then we give an overview of BoundShield, and describe it in detail.

A. THREAT MODEL

Our defense against memory disclosure attacks is based on a practical threat model. The assumptions are as follows:

- 1) The target system is configured with $W \oplus X$, which is supported by modern operating systems.
- 2) We assume that the hardware and the operating system are in the trusted computing base. In this paper, we focus on preventing adversaries from learning about the memory layout and searching gadgets while kernel attacks, hardware attacks, and data-only attacks are out of scope.
- 3) We assume that the attackers cannot tamper with our implementation of BoundShield.
- 4) The adversary can read from or write to arbitrary memory addresses repeatedly by exploiting memory

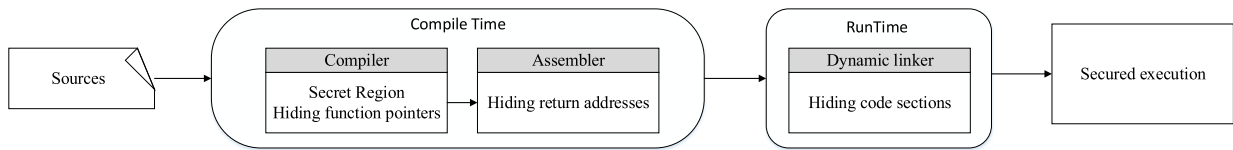


FIGURE 1. Overview of BoundShield.

corruption vulnerabilities. Thus, the attacks can happen within a read-based memory disclosure by reading from memory explicitly or within an execution-based memory disclosure by overwriting code pointers and diverting control transfers to arbitrary addresses.

B. OVERVIEW OF BoundShield

To provide comprehensive protection against memory disclosure attacks, we need to: 1) prevent attackers from reading code and code pointers and 2) prevent attackers from executing arbitrary code by overwriting code pointers. Figure 1 illustrates the overview of BoundShield. BoundShield takes program's source codes as input and makes a series of transformations on the program at compile-time and runtime. At compile time, our custom compiler transforms the code to support the secret region and hide function pointers, while the assembler transforms the code to hide return addresses. At runtime, we use a modified dynamic linker to hide code sections and set up the environment. During execution, the protected programs will be shielded from memory disclosure attacks. The brief introduction of each transformation in BoundShield is listed as follows:

- 1) **Secret region.** Since information hiding has been proven to be unreliable, we apply a strong SFI scheme to protect critical parts of the memory. Specifically, BoundShield divides the user address space into two regions: one secret region to store code and code pointers, and one regular region for the rest. We provide substantial isolation by checking load and store instructions with MPX, which forbids an adversary reading from or writing to the secret region.
- 2) **Hiding code sections.** As attackers may read directly from the code pages and locate gadgets, BoundShield separates code sections from data sections, and relocates all code sections into the secret region. Hence, all code pages are execute-only. Besides, as we break the fixed offsets between code sections and data sections, an adversary can be prevented from inferring code layout with static data pointers.
- 3) **Hiding function pointers.** Code pointers are critical parts in the memory, as attackers can exploit them to perform both read-based and execution-based memory disclosures. To protect function pointers, BoundShield first replaces all function pointers with pointers to trampolines hidden in the secret region so that they can only leak the addresses of trampolines, but not the addresses of original code. To further thwart execution-based

memory disclosures, we ensure all function pointers point to trampoline entries when dereferenced, which significantly reduces the attack surface.

- 4) **Hiding return addresses.** Return address is another kind of code pointers. To protect return addresses, BoundShield maintains a hidden stack placed in the secret region. All return addresses are stored on the hidden stack so that attackers cannot read, or overwrite them. This strategy eliminates both read-based memory disclosures and execution-based memory disclosures which exploit return addresses.

C. SECRET REGION

Rather than relying on information hiding, BoundShield creates a secret region to hide sensitive parts (e.g., code and code pointers) in the program. We divide the large user address space into two regions: the secret region which occupies the lower part of the address space and the regular region which occupies the higher part. To provide effective isolation, BoundShield prevents any memory accesses into the secret region. In practice, we preserve the boundary of the secret region in the lower bound of MPX bound register `%bnd0`, and insert `bndcl` instructions before load and store instructions in the application. Figure 2 shows an example of BoundShield's instrumentation for memory accesses. Note that all load and store instructions are checked against the same boundary, so that there is no need to change `%bnd0` during execution. We also propose some optimizations to reduce the number of inserted instructions, which will be discussed in Section III-G.

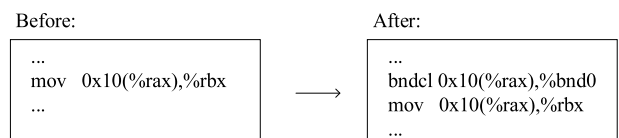


FIGURE 2. An example of instrumentation for memory accesses.

BoundShield further leverages the secret region to hide critical parts of the program. All regular data in the stack, heap, and data sections are placed in the regular region. Code sections, trampolines, and the hidden stack are hidden in the secret region, generating immunity to the malicious read or write operations. Figure 3 shows the runtime memory layout for BoundShield applications. The metadata records the base address and the size of each memory area in the secret region, and is only accessed by BoundShield's

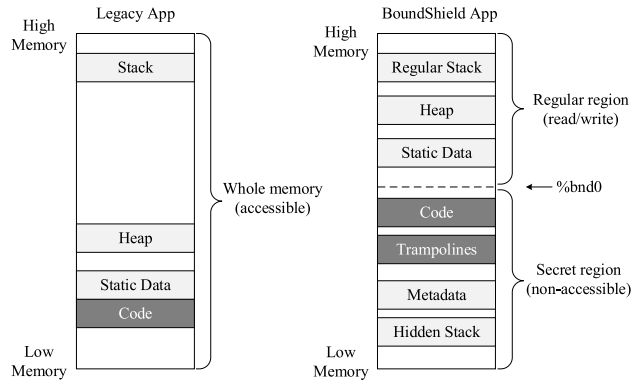


FIGURE 3. Memory layout of legacy applications and BoundShield applications. In legacy applications, attackers can access the whole memory layout. In BoundShield applications, attackers can only read from or write to the regular region. Code, trampolines, the hidden stack, and metadata are stored in the secret region.

runtime code. We use the metadata to manage the memory allocation in the secret region and help BoundShield hide code and code pointers. We set `%bnd0` during the program initialization. At execution time, if the address of a memory reference is below the lower bound of `%bnd0`, a `#BR` fault will be triggered. Unlike previous SFI schemes [27], [28], BoundShield has no limitation on boundary locations. By simply changing the value in `%bnd0`, the size of the secret region can be freely set on demand.

D. HIDING CODE SECTIONS

Armed with an arbitrary read primitive, an adversary can directly read code pages and gather gadgets. Execute-only memory indeed prevents code pages from being read. However, current operating systems do not natively support this feature. Rather than relying on modifications to kernel or hypervisor similar to previous work [9]–[11], BoundShield enforces an execute-only memory by hiding all code sections in the secret region.

In x86-64 architecture, programs and libraries are compiled as position-independent executables so that they can be loaded at any addresses in memory and can benefit from ASLR. Accesses to static data in data sections are typically using PC-relative instructions, which are hardcoded with fixed offsets. To hide code sections, BoundShield separates code sections from data sections while ensuring all accesses to static data behave as usual. During link time, we record the positions of all PC-relative instructions that should be fixed and append this patching information to the executables. At load time, BoundShield initially relocates all code sections to random addresses in the secret region by calling `mmap()`, and then fix the offsets in PC-relative instructions according to the patching information. As shown in Figure 3, all code sections are hidden in the secret region during execution and protected by the isolation mechanism. Note that another benefit of hiding code sections is that the fixed offsets between code sections and data sections can be broken so that adversaries cannot infer the code layout with a static data pointer.

E. HIDING FUNCTION POINTERS

Hiding code sections only stops attackers from directly reading the code. If code pointers are left unprotected in the regular memory region, an attacker can 1) read from memory and harvest code pointers to infer gadget locations with prior knowledge of code layout; 2) overwrite code pointers to execute arbitrary code and probe codes that have been executed.

One way to protect function pointers is to replace them with substitutes (e.g., pointers to trampolines) [9], [10]. Each trampoline entry contains a small piece of code, which directly jumps to the entry of the corresponding function. This strategy successfully stops read-based memory disclosures, since function pointers can only leak out trampoline addresses. However, the solution is vulnerable to execution-based memory disclosures, as attackers are still able to overwrite trampoline pointers to arbitrary addresses and execute any code in code sections.

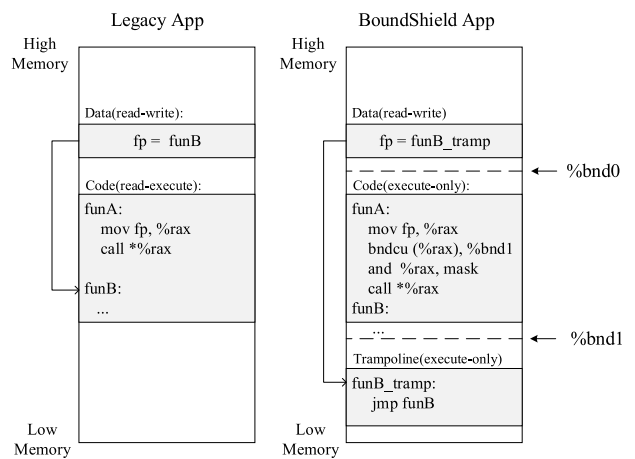


FIGURE 4. Hiding function pointers. The mask is set to `0xffffffffffff8`.

Our design shares a similar idea with previous trampoline-based solutions, as shown in Figure 4. We generate trampoline entries for functions the addresses of which have been taken and replace all function pointers in the regular region with corresponding trampoline pointers. We put all trampoline entries into a trampoline section, which is hidden in the secret region during runtime, so that real function pointers can be hidden within the trampolines and leaked function pointers can only reveal trampoline addresses. Different from previous solutions, we further thwart execution-based memory disclosures by only allowing function pointers to point to trampoline entries, which can significantly reduce the attack surface. At load time, we perform a further relocation as mentioned in III-D for trampoline sections and place them at lower addresses than other code sections in the memory. The boundary between code sections and trampoline sections is stored in the upper bound of an MPX register, i.e., the value of `%bnd1`. For each indirect call to functions, we insert a `bndcu` instruction to check against `%bnd1`. If the address in a function pointer is higher than the boundary, a fault will be triggered. We further insert an `and` instruction to ensure an

8-byte aligned trampoline entry is pointed but not the middle of a trampoline instruction.

Our strategies can reduce the attack surface for execution-based memory disclosure attacks significantly because of two reasons. First, attackers can only probe trampoline entries, but not the gadgets in the whole code sections. Second, blindly guessing trampoline entries will probably trigger a trap, which further limits the targets to leaked trampoline pointers in the regular region.

Jump tables are used to optimize the `switch/case` in the program. Each entry in a jump table contains an offset from the jump table to a basic block. Normally, jump tables are stored in `.rodata`. To prevent attackers from inferring code layout with jump table entries, we modify the compiler to emit all jump tables into code sections, and never save the addresses of jump tables or the addresses obtained from jump tables to the memory. During compilation, we identify load instructions of jump tables and do not insert checks before them. Such instructions typically use the base address of a jump table and a scaled index to access the jump table entries. As the base address is derived with a PC-relative instruction, and the scaled index is already checked in the code sequence, such instructions can only access the corresponding jump tables. Besides, the contents that these instructions read are never stored into memory. Thus, it will not bring any risk to remove checks for these instructions.

Nowadays, most programs are dynamically linked against shared libraries that are compiled as position-independent and can be loaded at any addresses in the memory. Programs use two data structures to support dynamic linking: *procedure linkage table* (PLT) and *global offset table* (GOT). Each entry in PLT is a small piece of code, which will read the address in the related GOT entry and indirectly jump to it. Calls to external functions are presented as calls to related PLT entries. During execution, the dynamic linker will resolve the addresses of external functions, and fill them into GOT so that calls to external functions work properly. As GOT contains code pointers and may leak out code layout, we configure the dynamic linker to resolve all external function addresses at load time and replace each entry in PLT with a direct jump to the external function rather than store the address of the external function in GOT. We also consider a special case in which libraries are loaded during execution by using `dlopen()` so that external function addresses can be obtained by using `dlsym()`. To cover this case, BoundShield generates a new trampoline entry for the resolved function, and return the trampoline's address rather than the original address in `dlsym()`.

F. HIDING RETURN ADDRESSES

Attackers can also exploit another source of code pointers, return addresses, to perform memory disclosure attacks. Comparing with function pointers, return addresses are used more frequently during execution. If the trampoline-based mechanism is applied to protect return addresses, we need to generate a trampoline for each call site in the program,

which would impose non-trivial performance overhead and substantially increase the attack surface for execution-based memory disclosures. For these reasons, we choose an alternative method: to store all return addresses on a hidden stack in the secret region.

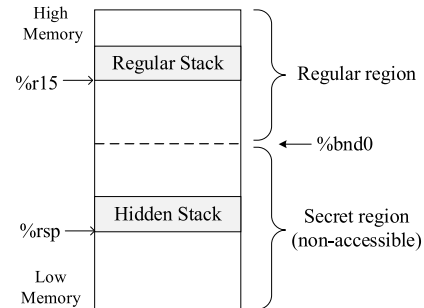


FIGURE 5. Hiding return addresses.

Commonly, the return addresses are only used with paired `call/ret` instructions and are not read or written by other instructions. Inspired by ASLR-Guard [7] and CPI [13], we separate the original stack into two stacks during execution: one hidden stack that stores all return addresses and one regular stack that stores regular data. Figure 5 illustrates our method for hiding return addresses. The hidden stack uses `%rsp` as the stack pointer, so that `call` and `ret` work as usual. We use a dedicated register, `%r15`, to serve as the stack pointer of the regular stack. All other data, such as local variables, function parameters, and spilled registers, are stored on the regular stack and accessed through `%r15`. Special instructions such as `push`, `pop`, `enter`, and `leave` are replaced by using equivalent `mov` instructions.

One significant difference between BoundShield and ASLR-Guard is that we provide substantial isolation for the hidden stack rather than rely on information hiding. In practice, we use the original stack as the regular stack and allocate a memory region which is as large as the default size of the stack for the hidden stack at a random address in the secret region. As all load/store instructions are forbidden to access the secret region, it is impossible for attackers to read or overwrite any return addresses. Furthermore, to support multi-threaded processes, we modify the thread library to allocate a hidden stack at thread create and destroy it at thread exit.

We also notice a special use of return addresses that `setjmp()` takes the return address of current call site from the stack, and passes it to the paired `longjmp()`. This will return to the call site of `setjmp()` rather than its own call site. For this rare case, we allow `setjmp()` and `longjmp()` to access the return address with `%rsp`. To further protect the return address, we encrypt it with a XOR instruction in `setjmp()`, and decrypt it in `longjmp()`. The key is determined at runtime and is never leaked to the regular region.

G. OPTIMIZATIONS

An initial implementation of BoundShield would check all load and store instructions in the program. Noticing that checks for load/store instructions are not always necessary [17], we perform several optimizations to further improve the efficiency of BoundShield.

1) MERGING DUPLICATE CHECKS

Load/store instructions in code sequences may use the same registers with different displacements [29]. Checks of such instructions can be merged by checking against the minimal displacement under two conditions. One condition is that two load/store instructions use the same base register and the same scaled index register if used. The other condition is that the value in the register(s) is not changed or spilled to memory in all paths between these two instructions. Figure 6 presents an example of merging duplicate checks. We first merge the duplicate checks in each basic block and then analyze the control flow graph to further eliminate duplicate checks among basic blocks in each function.

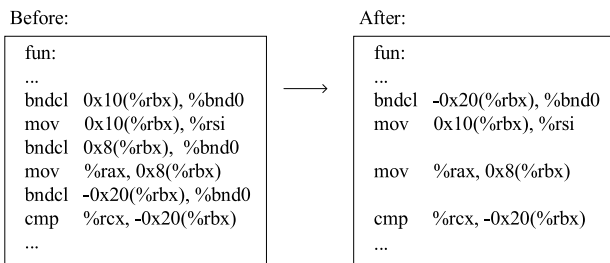


FIGURE 6. An example of merging duplicate checks.

2) STACK CHECKS

In BoundShield, normal stack data are stored on the regular stack, which uses `%r15` as its stack pointer. Load and store instructions will use `%r15` as the base register to access the regular stack. In x86-64 architecture, the displacement of instructions is limited to 32-bit. By setting the regular stack at an appropriate address in the regular region, we can make sure that an instruction which only uses `%r15` and the displacement can never reach the secret region. However, it is possible for attackers to pivot `%r15` near or into the secret region and exploit the unchecked stack load/store instructions. We eliminate this minor possibility by 1) allocating guard pages above the secret region so that repeatedly minus `%r15` will eventually trigger a fault; 2) checking `%r15` after instructions that write an absolute value or register into the `%r15` so that exploiting such instructions will trigger a fault. Therefore, we just need to instrument stack load/store instructions which use both `%r15` and a scaled index register, thus eliminating unnecessary checks on stack operations.

3) PC-RELATIVE CHECKS

As we mentioned in III-D, accesses to static data in position-independent executables typically use PC-relative instructions. These instructions can have two patterns, both of which

use PC register (`%rip`) explicitly or implicitly. The first pattern directly uses `%rip` as the base register and add an offset to decide the effective address of load/store instructions, such as `mov 0x20b808(%rip), %rax`. Another pattern is usually used to access fields in a data structure; it is a small code sequence which first loads the effective address of the static variable into a register and then loads from or stores to memory by using the register as the base register, such as `lea 0x20d36a(%rip), %rbx; mov (%rbx), %rdx`. As the offsets are hardcoded, attackers cannot modify the effective address of load/store instructions in both patterns. For this reason, we can avoid checking these instructions for better performance.

IV. IMPLEMENTATION

In order to verify the effectiveness of BoundShield on defending memory disclosure attacks, we have built a prototype on the Linux 64-bit 4.4.0 kernel. BoundShield has two main components: compile-time component and runtime component. The first component is made up of a set of modifications to GNU toolchain and is responsible for code transformations in applications. The second component is built on the standard C library, `glibc`, which is in charge of transforming the memory layout and initializing environment at runtime. To further clarify the working process of our implementation and show detailed information of the transformed program memory layout, an example using a simple demo program has been demonstrated in Appendix.

A. COMPILE-TIME COMPONENT

Our compile-time component has four main tasks: 1) instrumenting load/store instructions to provide substantial isolation for the secret region; 2) adopting trampoline mechanism and instrumenting indirect calls to hide function pointers; 3) replacing the usage of `%rsp` with `%r15` to hide return addresses; 4) generating patching information for instructions that need to be fixed at runtime.

We first modify GCC-4.8.4 to instrument load/store instructions and adopt trampoline mechanism. We add two extra passes in register transform language optimization phase. The first pass initially generates trampolines for functions the addresses of which have been taken, and inserts check instructions before indirect calls. Then, function symbols in address taking instructions and data sections are replaced to ensure that all function pointers can be replaced with trampoline pointers during runtime. The second pass follows after the first one. Load and store instructions in programs are scanned, and instrumentation related to the proposed optimizations are performed.

The transformation of the hidden stack is performed by the cooperation between compiler and assembler. We first configure GCC to reserve `%r15`, reduce the usage of push/pop and emit code as if there is no return address on the regular stack. Then we use a modified assembler which is implemented based on open source code of ASLR-guard [7] to replace all usage of `%rsp` with `%r15`.

For special stack instructions such as `push`, `pop`, `enter`, and `leave`, we replace them with equivalent `mov` instructions.

At last, we rely on a static linker to generate patching information for instructions that need to be fixed at runtime. Particularly, we consider three kinds of instructions: 1) PC-relative instructions to access static data; 2) direct jump instructions in trampolines; 3) instructions to take the address of a trampoline entry. We append the patching information to the binary to help the dynamic linker fix instructions at runtime.

B. RUNTIME COMPONENT

The runtime component of BoundShield is implemented on `glibc-2.19`. For dynamically linked programs, Linux kernel first loads the main executable and the dynamic linker, then controls the dynamic linker to load all shared libraries. We use custom linker scripts to decide the initial base addresses of the main executable and the dynamic linker so that the kernel can load them at appropriate addresses. The initial base addresses of shared libraries are decided by the dynamic linker.

One task of the dynamic linker is to set up the hidden stack at the beginning of the execution. In `_dl_start()`, we allocate a new memory area in the secret region to serve as the hidden stack. Then, we copy the current return address (the address of `_dl_start_user()`, which will be adjusted to the relocated dynamic linker's code later) to the hidden stack, and point `%rsp` to the start of the hidden stack. In this step, BoundShield also allocates memory for the metadata, which records the address and the size of each memory area in the secret region.

Another task of the dynamic linker is relocating code sections and trampolines to random addresses in the secret region. Four steps are involved to achieve this objective. First, all code sections in the main executable and shared libraries are separated from data sections by relocating them to random addresses in the secret region. Second, trampolines are further relocated to lower addresses, and the layout of them is randomized. Third, the instructions in code sections and trampolines are fixed according to the patching information generated by the static linker. Finally, the function pointers in data sections are adjusted to point to the relocated trampoline entries. There may be some corner cases that a function pointer holds the address of an original function. In these cases, BoundShield will generate a new trampoline entry and replace the function pointer with the new trampoline pointer. Function pointers in GOT are handled specially. For GOT entries, we configure the dynamic linker to resolve them at load time, and modify the corresponding PLT entries rather than GOT entries. In this process, the dynamic linker is the first module to be relocated. After that, the control is delivered to the relocated code of the dynamic linker. The code sections of the main executable and shared libraries are relocated afterward. Once relocation is finished, the original code pages are unmapped.

By default, the kernel allocated memory (e.g., stack and heap) are placed above the initial base address of the main executable in the user memory space, which resides in the regular region. If they are located in the secret region, BoundShield can reallocate a corresponding memory in the regular region. To further ensure that the regular data always reside in the regular region, we wrap `mmap()` to only allow BoundShield allocate memory in the secret region.

The runtime component can be launched before `main()` by first setting the values of `%bnd0` and `%bnd1` according to boundaries of the secret region and trampolines, and then enabling MPX. From now on, the application is under the protection of BoundShield.

C. HANDWRITTEN ASSEMBLY

As most parts of BoundShield are implemented on compiler toolchain, handwritten assembly code may bring some compatibility problems during deployment. We manually fix some handwritten assembly code in `libgcc` and `glibc`: 1) We provide trampoline support for handwritten assembly code by modifying instructions that take addresses of functions, and adding new trampolines for functions if needed (e.g., program entry). 2) We fix execute-only conflicts in handwritten assembly code to ensure that static data and jump tables are stored in the right sections. 3) We fix the compatibility problems between handwritten assembly code and the hidden stack to make sure that stack parameters can be accessed correctly.

V. EVALUATION

In this section, we first analyze the effectiveness of BoundShield in defending against memory disclosure attacks and then evaluate the performance impact of BoundShield with standard benchmarks and server programs. We perform the experiments on Ubuntu 14.04 with an Intel Core CPU i7-6700HQ @ 2.60GHz and 16GB RAM.

A. SECURITY EVALUATION

1) READ-BASED MEMORY DISCLOSURES

In read-based memory disclosures, attackers can either directly read from code pages and collect gadgets, or infer code layout with leaked code pointers and static data pointers. BoundShield relocates all code sections to random addresses in the secret region and prevents any illegal accesses to the secret region. Hence, the code pages are execute-only, and directly reading from them will trigger a `#BR` fault. As the fixed offsets between code sections and data sections are broken, it is unreliable to infer code layout with static data pointers. As a result, attackers can only infer code layout with leaked code pointers in memory.

To stop such attacks, BoundShield replaces function pointers with trampoline pointers and hides all return addresses in the secret region, which means that only trampoline pointers are leaked to the regular region. Hence, attackers can only reveal addresses of trampolines, but not the addresses of

the original code. To measure the effectiveness to hide code pointers, we write a memory analysis tool to dump the whole memory of programs at runtime and scan the memory to find code pointers. Particularly, each 8-byte aligned word in the memory is scanned and considered as a code pointer when it points to a valid instruction. Then we apply BoundShield to two real-world server programs, nginx 1.4.0 and lighttpd 1.4.35. To simulate the attack scenario, we first use Apache benchmark [30] to issue 100,000 requests to the server programs, and then use the memory analysis tool to check whether there are any unprotected code pointers in the regular region. During the analysis, we also calculate the number of gadgets (i.e., the total number of leaked trampoline pointers without repetition) in the regular region. Table 1 shows the results of our experiment. There is no single code pointer which points to the code in the regular region, illustrating BoundShield successfully hides code pointers.

TABLE 1. Code pointers in the regular region.

	pointers to code	pointers to trampoline	gadgets
lighttpd	0	6,826	355
nginx	0	2,528	1,196

Information hiding schemes [7], [8] are typically vulnerable to memory probing attacks [12], [13], [19]. These schemes store code pages and code pointers in safe regions, and relies on the large 64-bit virtual address space to hide them. However, attackers can repeatedly read from the user address space and probe the locations of safe regions. Once discover the safe regions, attackers can directly read code pages and code pointers to collect gadgets. Comparing with information hiding schemes, BoundShield provides substantial isolation for code pages and code pointers. As any read attempts from the secret region are prohibited, attackers cannot bypass BoundShield with memory probing attacks.

Previous XnR schemes [9]–[11] can effectively stop attackers from collecting gadgets by reading code pages and code pointers. However, attackers can still infer gadgets from static data pointers, as code sections have fixed offsets with data sections. BoundShield breaks the fixed offsets by relocating all code sections into the secret region, thus makes inferring gadgets from static data pointers unreliable. Comparing with previous XnR schemes, BoundShield provides stronger protection against read-based memory disclosure attacks.

2) EXECUTION-BASED MEMORY DISCLOSURES

Tough we have eliminated read-based memory disclosures, attackers can still conduct an execution-based memory disclosure. Normally, attackers can overwrite both function pointers and return addresses to perform execution-based memory disclosure attacks.

As for protecting return addresses, BoundShield stores all return addresses on the hidden stack placed in the secret region. As a result, attackers cannot overwrite any return addresses to probe gadgets. Meanwhile, the legitimate

targets of indirect calls are limited to trampoline entries, which means even if an attacker can overwrite a function pointer, only trampoline entries can be probed, but not any gadgets in the original code sections.

TABLE 2. AIT metric for SPEC CPU2006 benchmarks in XnR schemes and BoundShield.

	Original(XNR)	BoundShield	
	call / ret	call	ret
perlbench	2,605,232	1,198	1
bzip2	1,278,928	455	1
gcc	4,272,032	1,718	1
mcf	1,683,320	459	1
gobmk	2,300,520	2,565	1
hmmer	1,902,752	481	1
sjeng	1,332,128	460	1
libquantum	1,702,464	460	1
h264ref	2,091,624	512	1
milc	1,769,544	461	1
lbm	1,684,456	459	1
sphinx3	1,803,880	466	1
average	2,035,573	808	1

To measure the effectiveness of our strategy, we first calculate *Average Indirect Target* (AIT) metric [7] for SPEC CPU2006 benchmarks. AIT measures the average number of targets that an indirect control transfer can have, which indicates the average number of targets that an attacker can probe by corrupting a code pointer. Table 2 shows the AIT metric for SPEC CPU2006 benchmarks in XnR schemes and BoundShield. XnR schemes such as Readactor [9] mainly focus on stopping read-based memory disclosures and lift no restriction on code pointers. As a result, they have the same AIT as the original programs, which means attackers can overwrite code pointers to any addresses and probe gadgets. In BoundShield, the AIT of indirect calls is limited to the number of legitimate trampoline entries. For return sites, BoundShield allows no modifications to return addresses, thus reducing the AIT to 1. On average, the targets of execution-based memory disclosures are reduced to 0.04%. Moreover, as BoundShield inserts trap entries during compilation and randomizes the trampoline layout at runtime, blindly guessing trampoline entries will probably trigger a trap, which further limits the attack surface to leaked trampoline pointers in the regular region.

We adopt BoundShield to a server program with a stack buffer overflow vulnerability, nginx 1.4.0, and use Blind ROP [5] as an example to verify the effectiveness of BoundShield in defending execution-based memory disclosure attack. The attack has three steps: 1) it first guesses the return address and the canary on the stack; 2) then it overwrites return addresses to arbitrary addresses and locates potential gadgets in the code layout; 3) finally, it builds the exploit with collected gadgets. In theory, BoundShield can stop Blind ROP at both Step 1) and Step 2), thus avoiding Step 3) to happen. Since all return addresses are stored on the hidden stack instead of the regular one, attackers can neither read or overwrite them. In our experiment, we first use the Blind ROP tool [31] to attack an unprotected nginx server,

TABLE 3. Number of available gadgets to attackers in CFI and BoundShield.

	bin-CFI/CCFIR		BoundShield	
	Forward-edge gadgets	Backward-edge gadgets	Forward-edge gadgets	Backward-edge gadgets
lighttpd	4,079	11,134	355	0
nginx	7,460	35,298	1,196	0
average	5,770	23,216	776	0

and the exploit succeeds in 3 minutes. We then use the tool to attack a nginx server which is protected by BoundShield. The exploit cannot find a single return address for more than 12 hours, and we consider the attack is unrealistic.

Information hiding solutions [7], [8] can stop execution-based memory disclosure attacks on condition that attackers cannot locate code pointers. However, as discussed in V-A.1, such solutions can be bypassed with memory probing attacks [12], [13], [19]. After revealing the locations of safe regions, attackers can still probe gadgets by overwriting code pointers. Existing XnR solutions [9]–[11] mainly focus on read-based memory disclosure attacks, leaving programs vulnerable to execution-based memory disclosure attacks. One of the most comprehensive XnR solutions, Readactor, protects code pointers by replacing them with trampoline pointers. However, attackers can still overwrite trampoline pointers and probe any gadgets in the original code pages. Moreover, trampoline-based mechanisms may introduce new gadgets, as attackers can divert control flow to the middle of a trampoline. BoundShield fixes these issues by hiding return addresses and ensuring that function pointers only point to legitimate trampoline entries. As a result, attackers can only probe and reuse leaked trampoline pointers in the regular region, which greatly reduces the attack surface for execution-based memory disclosure attacks.

3) TRAMPOLINE REUSE ATTACKS

As described in previous sections, BoundShield provides a strong defense against memory disclosure attacks. However, attackers may still try to use the leaked trampoline pointers as gadgets and perform a trampoline reuse attack. We check the residual attack surface on real-world server programs by measuring the number of gadgets in the regular region. As shown in Table 3, the number of available gadgets to attackers in BoundShield is much smaller than the ones in CFI solutions [32], [33]. In CFI solutions, forward-edge gadgets include all legitimate targets of indirect calls and indirect jumps, while backward-edge gadgets include all call sites. In BoundShield, targets of indirect jumps (i.e., jump table entries and GOT entries) are never leaked to the regular region. Hence, attackers can only divert indirect calls. Since the targets of indirect calls are limited to only legitimate trampoline entries, attackers can only reuse a small portion of functions with leaked trampoline pointers as forward-edge gadgets. Due to the adoption of the hidden stack, backward-edge gadgets are eliminated in BoundShield. Since attackers can only reuse trampoline pointers, we believe that such attacks can be stopped by enforcing a more strict check (e.g., type-based check) at call sites [34] in BoundShield.

B. PERFORMANCE EVALUATION

We evaluate the impact of BoundShield on performance with all C programs in the SPEC CPU2006 benchmarks and two widely-used server programs, nginx 1.4.0 and lighttpd 1.4.35. We compile all programs and libraries with our custom toolchain and take the average number of 10 runs as the result. We compare boundshield with baseline which has the same basic setting, but does not perform any transformations during compilation. Table 4 shows the performance overhead of BoundShield on SPEC CPU2006 benchmarks. The *sjeng* has the highest overhead of 11.6%, while *mcf* induces the lowest overhead of 0.3%. The average performance overhead on SPEC CPU2006 benchmarks is 5.8%.

TABLE 4. Performance overhead of BoundShield on SPEC CPU2006.

Programs	Baseline(s)	BoundShield(s)	Overhead
perlbench	3.52	3.61	2.6%
bzip2	9.20	9.68	5.2%
gcc	1.31	1.35	3.1%
mcf	2.86	2.87	0.3%
gobmk	21.7	23.1	6.5%
hammer	6.92	7.52	8.7%
sjeng	4.64	5.18	11.6%
libquantum	0.0860	0.0902	4.9%
h264ref	25.0	27.0	8.0%
milc	10.6	11.5	8.5%
lbm	2.23	2.35	5.4%
sphinx3	1.95	2.05	5.1%
average			5.8%

To benchmark server programs, we use Apache benchmark [30] to send requests from a client machine which is connected to the host machine through a network cable. We configure the Apache benchmark to send 100,000 requests with a concurrency of 10. Table 5 illustrates the results. In general, BoundShield incurs a small performance overhead on server programs: 4.9% on nginx and 5.9% on lighttpd.

TABLE 5. Performance overhead of BoundShield on server programs (time per request).

Programs	Baseline(ms)	BoundShield(ms)	Overhead
nginx	0.364	0.382	4.9%
lighttpd	0.373	0.395	5.9%

By analysis, the performance impact is mainly caused by checks of load/store instructions, as they appear more frequently than indirect calls in programs. In general, we consider this performance loss acceptable, in case that BoundShield provides comprehensive protection against memory disclosure attacks.

VI. CONCLUSION AND FUTURE WORK

We have presented the design, implementation, and evaluation of BoundShield, a practical system to provide comprehensive protection against memory disclosure attacks. BoundShield separates a secret region from virtual memory space, and hide code sections, function pointers, and return addresses in the secret region. It makes novel uses of Intel MPX to protect both load/store instructions and indirect calls. Comparing with previous solutions, BoundShield requires no OS-level and hypervisor-level modifications and can provide a more comprehensive defense. We have implemented a prototype of BoundShield, and our evaluation results demonstrate that BoundShield can defend both read-based and execution-based memory disclosure attacks while only incurring a small performance overhead.

Currently, BoundShield focuses on protecting code and code pointers in the secret region to stop memory disclosure attacks. In future work, we plan to extend BoundShield to also protect critical non-control data in the secret region. In this way, BoundShield can stop other attack vectors, such as data-oriented attacks [35].

APPENDIX

EXAMPLE OF HOW BOUNDSHIELD WORKS

In this section, we take a simple demo program as an example to show the working process of BoundShield. The source code of the demo program is shown in Figure 7. Normally, it will output the address of `demo()`, a literal string, and the code content at the address of `demo()`.

```
#include <stdio.h>

void demo() {
    printf("this is a BoundShield demo.\n");
}

int main(void) {
    void (*fn)() = demo;
    printf("function pointer of demo() is: %p\n", fn);
    fn();
    unsigned long *a = (unsigned long *)fn;
    printf("code of demo() is %lx\n", *a);
    return 0;
}
```

FIGURE 7. Source code of the demo program.

We compile and run the program with BoundShield. As shown in Figure 8, at compile time, BoundShield makes a series of transformations on the program with the custom compiler, assembler, and static linker. The initial base address of the program is decided by a linker script. During compilation, BoundShield also specifies the dynamic linker for the program. When executed, the program outputs the address of the corresponding trampoline of `demo()` and a literal string, then it crashes. With BoundShield adopted, all function pointers are replaced with trampoline pointers,

```
lxb@lxc:~$ ./leakmap.py --path=/home/lxb/.local/share/lxc/containers/leakmap --demo=5
lxb@lxc:~$ ./leakmap.py --path=/home/lxb/.local/share/lxc/containers/leakmap --demo=5
PATH=/home/lxb:/home/lxb/shell/binutils:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/home/lxb/.vimproj/bin:/home/lxb/.rvm/bin:/home/lxb/BoundsShield/gcc/bin/gcc -Wl,-dynamic-linker=/home/lxb/BoundsShield/glibc/lib64/ld-2.19.so -pie -fPIC -maccumulate-outgoing-args -mno-push-args -ffixed-r15 -fplugin=/home/lxb/backpack/bndchx/bndchx.so -fplugin=/home/lxb/backpack/tramp/tramp.so -Wl,-T,/home/lxb/BoundsShield/linkerscripts/exe.ld -s /home/lxb/BoundsShield/glibc/lib64/leakmap.deno.c
lxb@lxc:~$ ./leakmap.py --path=/home/lxb/.local/share/lxc/containers/leakmap --demo=5
lxb@lxc:~$ ./leakmap.py --path=/home/lxb/.local/share/lxc/containers/leakmap --demo=5 .demo
function pointer of demo() is: 0x5554ac0f009d
this is a BoundsShield demo
Segmentation fault (core dumped)
```

FIGURE 8. Compile and run the demo program with BoundShield.

```

nbd->pages info registers bnd0
      [bound = 0x5554f0000000, ubound = 0x5554f0000000] : size 1 [bound = 0x554f0000000, ubound = 0x5554f0000000] : size 1
nbd->pages info registers bnd1
      [bound = 0x5554c0000000, ubound = 0x5554c0000000] : size 9382248683927 [bound = 0x0, ubound = 0x5554c0000000] : size 9382248683927
nbd->pages vmmmap
Start      End      Perm      Name
0x00000020e0000000 0x00000020e0800000 rw-p mapped
0x0000300000000000 0x0000300001000000 rw-p mapped
0x00005554c0f70000 0x00005554c0f70000 r-xp /home/lbx/boundsShield/glibc/lib/libc-2.19.so
0x00005554c9f72000 0x00005554c9f72000 r-xp /home/lbx/boundsShield/glibc/lib/libd-2.19.so
0x00005554cafc0000 0x00005554cafc01000 r-xp /home/lbx/boundsShield/demo/demo
0x00005554db000000 0x00005554db001000 r-xp mapped
0x00005554db100000 0x00005554db101000 rw-p mapped
0x00005554dc0f7000 0x00005554dc0f7000 r-xp /home/lbx/boundsShield/glibc/lib/libc-2.19.so
0x00005554dcdd0000 0x00005554dcdd2000 r-xp /home/lbx/boundsShield/glibc/lib/libd-2.19.so
0x00005554ddddd000 0x00005554ddddd01000 r-xp /home/lbx/boundsShield/demo/demo
0x00005554ffcc0000 0x00005554ffcf5000 r-xp /home/lbx/boundsShield/glibc/lib/libc-2.19.so
0x00005554ffdf0000 0x00005554ffff5000 -p-p /home/lbx/boundsShield/glibc/lib/libc-2.19.so
0x00005554ffff5000 0x00005554ffff9000 r-xp /home/lbx/boundsShield/glibc/lib/libc-2.19.so
0x00005554ffff9000 0x00005554ffffc000 r-xp /home/lbx/boundsShield/glibc/lib/libc-2.19.so
0x00005554fffffb00 0x00005554fffffb00 r-p mapped
0x0000555500000000 0x000055550002b000 r-xp /home/lbx/boundsShield/glibc/lib/libd-2.19.so
0x00005555002b0000 0x00005555002c2000 r-xp /home/lbx/boundsShield/glibc/lib/libd-2.19.so
0x00005555002c2000 0x00005555002d0000 r-xp /home/lbx/boundsShield/glibc/lib/libd-2.19.so
0x00005555002d0000 0x0000555500500000 rw-p mapped
0x0000555500500000 0x0000555500501000 rw-p mapped
0x0000555510200000 0x0000555510201000 rw-p /home/lbx/boundsShield/demo/demo
0x0000555510201000 0x0000555510201000 rw-p [heap]
0x00007ffff7fa8000 0x00007ffff7ff7b000 rw-p mapped
0x00007ffff7ff7b000 0x00007ffff7ffdf000 r-xp [lvarg]
0x00007ffff7ffdf000 0x00007ffff7ffdf000 r-xp [lvarg]
0x00007ffff7ffdf000 0x00007ffff7ffdf000 r-xp [stack]

```

FIGURE 9. User address space layout of the demo program.

```

RAX: 0x5554ac009040 --> 0xf83493d0df6e3e9
RBX: 0x0
RCX: 0xffffffff
RDX: 0x5554ffffb7a0 --> 0x0
RSI: 0x7ffff7fa8000 ("this is a BoundsShield demo.\n: 0x5554ac009040\n")
RDI: 0x0
RBP: 0xfffffffffaaf0 --> 0x0
RSP: 0x2007fdb0c --> 0x5554d99239e5 (mov    ebx,eax)
RIP: 0x5554ddd0082b --> 0x48008b48001a0ff3
R8 : 0x2e6f6d656426646c ('ld demo.')
R9 : 0x5554fffff86e0 --> 0x0
R10: 0x0
R11: 0x246
R12: 0x5554ac009000 --> 0xf10f30dfdf7be9
R13: 0xfffffffffeabc8 --> 0x1c
R14: 0x0
R15: 0x7fffffff8ae0 --> 0x5554ac009040 --> 0xf83493d0df6e3e9
EFLAGS: 0x10212 (carry parity ADJUST zero sign trap INTERRUPT direction overflow)
-----code-----
0x5554ddd0081f: mov     rax,QWORD PTR [rbp-0x10]
0x5554ddd00823: mov     QWORD PTR [rbp-0x18],rax
0x5554ddd00827: mov     rax,QWORD PTR [rbp-0x18]
=> 0x5554ddd0082b: bndcl   bnd0,[rax]
0x5554ddd0082f: mov     rax,QWORD PTR [rax]
0x5554ddd00832: mov     rsi,rax
0x5554ddd00835: lea     rdi,[rdi+0x3230019f]          # 0x5555100009db
0x5554ddd0083c: mov     eax,0x0
-----stack-----
0000 0x2007fdb0c --> 0x5554cd992385 (mov    ebx,eax)
0000 0x2007fdb0c --> 0x5554ddd008bb --> 0x558d48004d10ff64
0016 0x2007fdb0c --> 0x0
0024 0x2007fdb0c --> 0x0
0032 0x2007fdb0c --> 0x0
0040 0x2007fdb0c --> 0x0
0048 0x2007fdb0c --> 0x0
0056 0x2007fdb0c --> 0x0
-----
Legend: code, data, rodata, value
Stopped reason: rtxSEGV
0x00005554ddd0082b in ?? ()

```

FIGURE 10. BoundShield stops the demo program from reading the code content.

so the demo program outputs the address of the corresponding trampoline rather than the address of `demo()`. Since all code sections are relocated into the secret region at runtime, reading from code pages will indeed trigger a `#BR` fault. By default, the `#BR` fault handler will terminate the program, which makes the demo program crash.

To give a more detailed explanation, we use `gdb` to trace the demo program. At runtime, BoundShield will transform the memory layout and initialize the runtime environment. Figure 9 illustrates the user address space layout of the demo program. The lower bound of `%bnd0` is set to

0x5554f0000000, and the upper bound of %bnd1 is set to 0x5554c0000000. The initial base address of each module is above the boundary of the secret region. All code sections are relocated into the secret region, while trampoline sections are relocated to lower addresses. The hidden stack starts from 0x200000000 and uses %rsp as its stack pointer. The regular stack is located at the higher end of user address space and uses %r15 as its stack pointer.

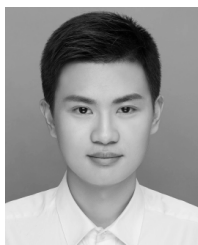
Figure 10 shows how BoundShield stops the demo program from reading the code content. The load instruction `mov(%rax), %rax` is used to read an 8-byte word from the code page, while %rax preserves the address of the corresponding trampoline of `demo()`, namely 0x5554acf00940. BoundShield inserts a `bndcl` instruction right before the load instruction during compilation. Since the address in %rax is lower than the lower bound of %bnd0, the `bndcl` instruction triggers a #BR fault, and the program crashes before it can read the code content.

REFERENCES

- [1] A. van de Ven. (2004). *Exec Shield*. [Online]. Available: https://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf
- [2] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. 14th ACM Conf. Comput. Commun. Secur.*, 2007, pp. 552–561.
- [3] PaX Team. (2003). *PaX Address Space Layout Randomization (ASLR)*. [Online]. Available: <https://pax.grsecurity.net/docs/aslr.txt>
- [4] 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 *Proc. IEEE Symp. Secur. Privacy*, May 2013, pp. 574–588.
- [5] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *Proc. IEEE Symp. Secur. Privacy*, May 2014, pp. 227–242.
- [6] J. Seibert, H. Okhravi, and E. Söderström, "Information leaks without memory disclosures: Remote side channel attacks on diversified code," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 54–65.
- [7] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "ASLR-Guard: Stopping address space leakage for code reuse attacks," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 280–291.
- [8] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *Proc. 11th USENIX Conf. Oper. Syst. Design Implement.*, 2014, pp. 147–163.
- [9] S. Crane et al., "Readactor: Practical code randomization resilient to memory disclosure," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 763–780.
- [10] S. J. Crane et al., "It's a TRaP: Table randomization and protection against function-reuse attacks," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 243–255.
- [11] J. Gionta, W. Enck, and P. Ning, "HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities," in *Proc. 5th ACM Conf. Data Appl. Secur. Privacy*, 2015, pp. 325–336.
- [12] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida, "Poking holes in information hiding," in *Proc. 25th USENIX Secur. Symp.*, 2016, pp. 121–138.
- [13] I. Evans et al., "Missing the point(er): On the effectiveness of code pointer integrity," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 781–796.
- [14] Y. Ding, T. Wei, H. Xue, Y. Zhang, C. Zhang, and X. Han, "Accurate and efficient exploit capture and classification," *Sci. China Inf. Sci.*, vol. 60, p. 052110, May 2017.
- [15] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming," in *Proc. 22nd Annu. Netw. Distrib. Syst. Secur. Symp.*, 2015, doi: 10.14722/ndss.2015.23262.
- [16] O. Oleksenko, D. Kuvaikii, P. Bhatotia, P. Felber, and C. Fetzer. (2017). "Intel MPX explained: An empirical study of intel MPX and software-based bounds checking approaches." [Online]. Available: <https://arxiv.org/abs/1702.00719>
- [17] M. Pomonis, T. Petsios, A. D. Keromytis, M. Polychronakis, and V. P. Kemerlis, "kR²X: Comprehensive kernel protection against just-in-time code reuse," in *Proc. 12th Eur. Conf. Comput. Syst.*, 2017, pp. 420–436.
- [18] W. Huang, Z. Huang, D. Miyani, and D. Lie, "LMP: Light-weighted memory protection with hardware assistance," in *Proc. 32nd Annu. Conf. Comput. Secur. Appl.*, 2016, pp. 460–470.
- [19] E. Göktas, R. Gawlik, B. Kollenda, G. Portokalidis, C. Giuffrida, and H. Bos, "Undermining information hiding (and what to do about it)," in *Proc. 25th USENIX Secur. Symp.*, 2016, pp. 105–119.
- [20] M. Backes, T. Holz, B. Kollenda, P. Koppe, and S. Nürnberger, and J. Pewny, "You can run but you can't read: Preventing disclosure exploits in executable code," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 1342–1353.
- [21] X. Chen, H. Bos, and C. Giuffrida, "CodeArmor: Virtualizing the code space to counter disclosure attacks," in *Proc. IEEE Eur. Symp. Secur. Privacy*, Apr. 2017, pp. 514–529.
- [22] Z. Wang et al., "ReRanz: A light-weight virtual machine to mitigate memory disclosure attacks," in *Proc. 13th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.*, 2017, pp. 143–156.
- [23] D. Williams-King et al., "Shuffler: Fast and deployable continuous code re-randomization," in *Proc. 12th USENIX Conf. Oper. Syst. Des. Implement.*, 2016, pp. 367–382.
- [24] V. van der Veen et al., "Practical context-sensitive CFI," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 927–940.
- [25] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI: Cryptographically enforced control flow integrity," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 941–951.
- [26] J. Yin, G. Tan, X. Bai, and S. Hu, "WebC: Toward a portable framework for deploying legacy code in Web browsers," *Sci. China Inf. Sci.*, vol. 58, no. 7, pp. 1–15, 2015.
- [27] K. Braden et al., "Leakage-resilient layout randomization for mobile devices," in *Proc. 23rd Annu. Netw. Distrib. Syst. Secur. Symp.*, 2016, doi: 10.14722/ndss.2016.23364.
- [28] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proc. 14th ACM Symp. Oper. Syst. Princ.*, 1993, pp. 203–216.
- [29] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel Corp., Santa Clara, CA, USA, 2017.
- [30] *Apache Benchmark*. Accessed: Jan. 10, 2018. [Online]. Available: <https://httpd.apache.org/docs/2.4/programs/ab.html>
- [31] *Blind ROP Tool*. Accessed: Jan. 10, 2018. [Online]. Available: <http://www.scs.stanford.edu/brop/>
- [32] C. Zhang et al., "Practical control flow integrity and randomization for binary executables," in *Proc. IEEE Symp. Secur. Privacy*, May 2013, pp. 559–573.
- [33] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *Proc. 22nd USENIX Secur. Symp.*, USENIX Association, 2013, pp. 337–352.
- [34] V. van der Veen et al., "A tough call: Mitigating advanced code-reuse attacks at the binary level," in *Proc. IEEE Symp. Secur. Privacy*, May 2016, pp. 934–953.
- [35] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *Proc. IEEE Symp. Secur. Privacy*, May 2016, pp. 969–986.



HAI JIN (SM'06) received the Ph.D. degree in computer engineering from the Huazhong University of Science and Technology in 1994. From 1998 to 2000, he was with The University of Hong Kong. From 1999 to 2000, he was a Visiting Scholar with the University of Southern California. He is currently the Chief Scientist of the National 973 Basic Research Program Project of Virtualization Technology of Computing System. He is also currently the Cheung Kung Scholars Chair Professor of computer science and engineering with the Huazhong University of Science and Technology. He has co-authored 15 books and published over 400 research papers. He is a member of the ACM. He received the German Academic Exchange Service Fellowship to visit the Chemnitz University of Technology, Germany, in 1996, and the Excellent Youth Award from the National Science Foundation of China in 2001.



BENXI LIU received the B.S. degree in software engineering from Huazhong University of Science and Technology, Wuhan, China, in 2015, where he is currently pursuing the master's degree in computer system organization. His research interests include system security.



DEQING ZOU received the Ph.D. degree from the Huazhong University of Science and Technology (HUST), China, in 2004. He is currently a Professor of computer science with HUST. He has applied almost 20 patents. He has published two books *Xen virtualization Technologies* and *Trusted Computing Technologies and Principles* and over 50 high-quality papers, including papers published by the IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, the IEEE TRANSACTIONS

ON PARALLEL AND DISTRIBUTED SYSTEMS, the IEEE TRANSACTIONS ON SERVICES COMPUTING, the IEEE TRANSACTIONS ON CLOUD COMPUTING, and so on. His main research interests include system security, trusted computing, virtualization, and cloud security. He has served as a PC Chair/PC Member of over 40 international conferences.

...



YAJUAN DU received the Ph.D. degree from the Huazhong University of Science and Technology in 2017 and the Ph.D. degree from the City University of Hong Kong in 2018. He is currently an Assistant Professor with the School of Computer Science, Wuhan University of Technology. Her current research interests include computer storage systems, such as flash-based solid-state drives, error correction coding theory and algorithms, such as the low-density parity-check codes,

and security problems in software and clouds, such as the vulnerability detection and attack defense.