LMP: Light-Weighted Memory Protection with Hardware Assistance

Wei Huang Zhen Huang Dhaval Miyani David Lie
Department of Electrical and Computer Engineering
University of Toronto

Abstract

Despite a long history and numerous proposed defenses, memory corruption attacks are still viable. A secure and low-overhead defense against return-oriented programming (ROP) continues to elude the security community. Currently proposed solutions still must choose between either not fully protecting critical data and relying instead on information hiding, or using incomplete, coarse-grain checking that can be circumvented by a suitably skilled attacker. In this paper, we present a light-weighted memory protection approach (LMP) that uses Intel's MPX hardware extensions to provide complete, fast ROP protection without having to rely in information hiding. We demonstrate a prototype that defeats ROP attacks while incurring an average runtime overhead of 3.9%.

CCS Concepts

•**Security and privacy** → **Malware and its mitigation;**

Keywords

Stack Protection, CFI, ROP, MPX

1. INTRODUCTION

In languages such as C/C++, the programmers are ultimately responsible for enforcing the memory safety of their programs. However, inevitably, programmers produce code with flaws that violate memory safety, and some of these flaws result in memory corruption vulnerabilities that allow attackers to maliciously alter the control flow of programs [29], corrupt critical data [19], or cause sensitive information leakage [13].

There have been numerous proposed or deployed defenses to mitigate memory corruption vulnerabilities. Despite this, memory corruption vulnerabilities continue to be exploitable. For example, ASLR (Address Space Layout Randomization) [27] randomizes memory locations of code and data segments, but can be circumvented via vulnerabilities such as address space leakage, timing side-channels [20] or attacks such as just-in-time code reuse [32]. DEP (Data Execution Prevention) [2] prevents injecting and executing new code in vulnerable programs. However, it cannot prevent reusing existing code in an application via a return-to-libc or ROP (Return-Oriented Programming) attack [29].

To address ROP attacks, Abadi et al. propose Control-Flow Integrity (CFI) [1]. CFI protection enforces both forward-edge protection (i.e. indirect function calls) and backwards-edge protection (i.e. function returns) to ensure that a memory corruption vulnerability does not allow an attacker to corrupt a code pointer and redirect execution along an edge not specified by the original program. While the target of a forward-edge function call can be resolved to a single or small number of targets statically, the target of a backwards-edge function return cannot generally be determined with much precision using only static analysis. As a result, backwards-edge protection generally requires a runtime component. To determine and enforce backward-edges precisely, shadow stacks are proposed in [1] and software-based fault isolation (SFI) [37] is further used to protect the contents of the shadow stacks from corruption by an attacker. Unfortunately, the runtime overhead of the memory checking required to properly implement this runtime component can be as high as 2× [9].

To reduce this overhead, various proposals weaken the properties of the backwards-edge protection in return for better runtime performance. For example, some propose coarse-grain protections, which do not use a shadow stack to precisely track backwards-edge targets. Since shadow stacks are not used, there is no need for SFI, which avoids the expensive checks required to implement memory protection for the shadow stacks. This coarse-grain approach is taken by proposals such as kBouncer [26], ROPGuard [16], ROPecker [6], which have significantly lower overheads ranging from 1.59% to 2.60%. These coarse-grain methods are imprecise in that they do not actually validate that the return address on a backwards-edge actually points to the original caller; instead, they either only check that the return address points to an instruction that follows some call instruction, or they heuristically check the number of returns to detect gadgets executions. They have all been shown to be circumventable [12, 18] and ineffective against a knowledgeable attacker.

Information hiding is another way to mitigate the overhead of complete CFI backwards-edge protection. In this approach, rather than protecting the data in the shadow stacks with memory access checks, the shadow stacks are

ACSAC '16, December 05-09, 2016, Los Angeles, CA, USA
© 2016 ACM. ISBN 978-1-4503-4771-6/16/12...$15.00
DOI: http://dx.doi.org/10.1145/2991079.2991089

placed at a random location in a 64-bit address space. Because the size of the address space is large, it is assumed infeasible for the attacker to guess the location of the shadow stacks. One method called code-pointer integrity (CPI) [23] is able to provide CFI protection with 2.9% overhead (on C applications). However, information hiding techniques can be broken by memory safety vulnerabilities that leak the location of the shadow stacks [15]. Other work has also shown that various side-channel attacks can be used to leak information that can be used to find the hidden shadow stacks [30, 33]. The lesson here is that ultimately information hiding is not equivalent to memory protection, as they are vulnerable to address information leakage, while memory protection is not.

In this paper, we propose Light-Weighted Memory Protection (LMP), a new method that leverages Intel's Memory Protection Extensions (MPX) to make backwards-edge CFI both secure and efficient. LMP tackles two essential problems that stand in the way of memory safety in system software: critical memory region protection in backwards-edge CFI approaches and non-trivial overheads in checking memory access violations.

While hardware-supported memory checks are naturally more efficient than software memory checking, which is also proven in recent work on using customized hardware for CFI enforcement [8, 11], we find that the hardware extensions like Intel MPX have to be applied carefully in order to truly reap the performance benefits of specialized hardware. In particular, not all of the operations supported by Intel MPX have low overhead. Therefore, we design LMP to minimize the use of the high-overhead components of MPX and still enable it to effectively protect shadow stacks from unauthorized modification.

We build a proof-of-concept prototype implementation of LMP and measure the performance overhead with SPEC 2006 benchmarks. The LMP system introduces an average overhead of 3.90%, which is much less than the $2\times$ overhead from the reference implementation of the original CFI [9]. In fact, LMP achieves roughly same overhead as information hiding techniques [10, 23], which have generally about 3% overhead. LMP is also comparable with recent coarse-grained CFI approaches, which have overheads between 1.59% (ROPGuard [16]) and 2.60% (kBouncer [26]). However, LMP provides stronger security guarantees than both information hiding and coarse-grain approaches, as it is both not vulnerable to either side-channel leakage and enforces a much stricter policy.

We summarize three main contributions this paper makes:

1. We propose an alternative use of hardware assisted pointer checker with Intel MPX that is different from the standard proposed use of MPX.

2. We provide the first stack protection solution that is assisted by the available CPU feature of Intel MPX.

3. We achieve a low overhead among existing equivalent solutions, while provide stronger protection than coarse-grain backward-edge CFI approaches.

The rest of this paper is organized as follows: We present background information about hardware assistance of Intel MPX we depend on and threat model we assume in Sec. 2, describe the method we use in Sec. 3 and details of implementation in Sec. 4, evaluate our results in Sec. 5, discuss related work in Sec. 6 and conclude in Sec. 7.

2. BACKGROUND

Before describing our approach to protection, we first describe the base MPX hardware that LMP leverages. Intel's Memory Protection Extensions (MPX) are a set of extensions to the x86-64 instruction set architecture in the Intel Skylake processors. To check pointer references at runtime and prevent illegal memory accesses, the idea was implemented previously as the feature of Pointer Checker [17] in the Intel compiler for debugging: a pair of bounds is created whenever a pointer is made, then the compiler will also generate code to check the bounds when the pointer is used. Pointer Checker is fully software-based, while MPX provides hardware acceleration for the bound checks that Pointer Checker would have done in software. MPX has software and hardware components.

MPX introduces several new registers and instructions to the instruction set architecture:

- 4 bound registers: `BND0-BND3`. Each of the registers is 128-bit, and they store the lower bound memory address with 64 bits and the upper bound memory address with 64 bits. Bound registers hold the upper and lower bounds that memory accesses are checked against.

- 2 configuration registers: `BNDCFGU` for user mode and `IA32_BNDCFGS` for supervisor mode.

- 1 status register: `BNDSTATUS` which stores error code when exception occurs.

- Bound management instructions: `BNDLDX` and `BNDSTX` load `BND` registers from a table of object-specific address bounds in memory. `BNDMK` and `BNDMOV` allow a programmer to manually manage the `BND` registers.

- Bound check instructions: `BNDCU` and `BNDCL` are used to check that a pointer meets the respective upper and lower bound limits of a specific `BND` register. If the pointer falls outside of the bounds, then the instruction throws an exception, saving the need for an instruction to explicitly check the result of the comparison.

For the software part, the MPX requires the following system software support:

- MPX-enabled Compiler: The compiler is responsible for inserting bound checks before pointer dereferences. Because bound information must be loaded in a limited number of `BND` registers before it can be used to check a pointer, the compiler must also load and spill bounds information between the `BND` registers and memory. For now, Intel has added MPX support to GCC main branch since version 5.0 for C/C++ and x86 targets only.

- MPX Runtime: The MPX runtime library is linked against program at compile-time. The library provides an API that the application developer can use to configure MPX hardware features, as well as functions to help compiler generated code manage MPX registers.

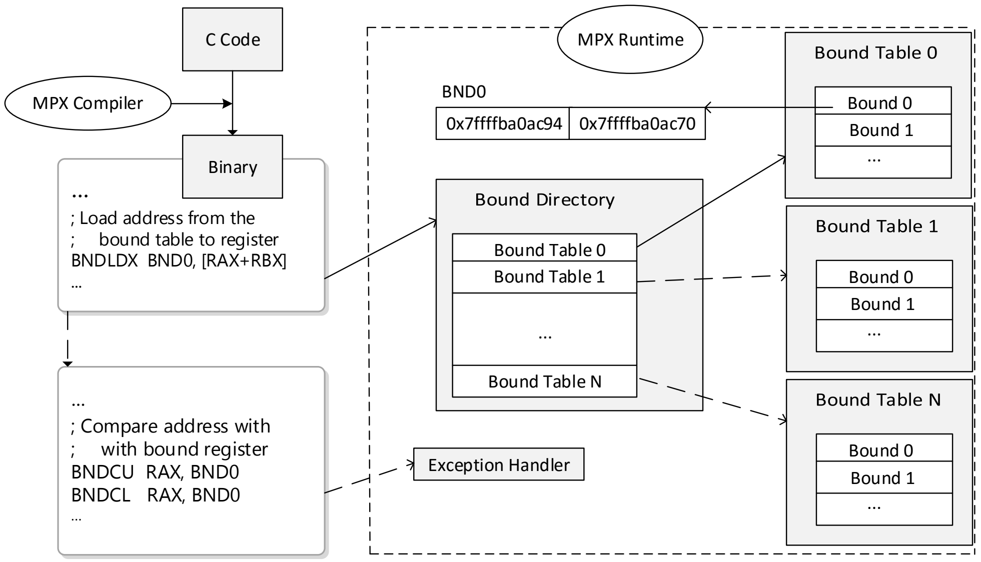

Figure 1: *An example of how MPX works.*

- Operating system: The OS, together with the compiler, needs to support the new MPX instructions. If a bound check instruction fails, the OS must catch the generated exception and signal the application.

We now give an example of how these MPX components can be used to bound-check a small program. Consider a program that declares and manipulates data in 5 arrays:

$$int \quad A[10],\ B[20],\ C[30],\ D[40],\ E[50];$$

Anytime a pointer pointing into one of these arrays is dereferenced, the MPX compiler needs to insert bound-checks to ensure that the pointer falls within one of these arrays. To do this, the MPX compiler needs to determine which array the pointer should be pointing into, load the upper and lower bounds of the array into a `BND` register and then insert the appropriate `BNDCU` and `BNDCL` checks before the pointer dereference to check it against the upper and lower bounds of the array. For example as showed in Figure 1, if array A is stored at addresses `0x7ffffba0ac70`-`0x7ffffba0ac94`, the MPX compiler must first load the upper and lower bound addresses `0x7ffffba0ac70` and `0x7ffffba0ac94` into one of the bound registers (say `BND0`). This is done using the `BNDLDX` instruction, which loads the bound information from the bound directory in memory to into the appropriate register. Then the MPX compiler instruments bound checking instructions to compare the pointer dereference with bound values in `BND0`. If the dereference falls out of the bound, a #BR exception will be generated by hardware and caught by the exception handler in MPX runtime.

For a pointer into an array to be bound-checked, the bounds for that array must be loaded into a `BND` register. Since the arrays `A`, `B`, `C`, `D` and `E` are all located in different regions in memory, the MPX compiler must load the appropriate array bounds into a `BND` register whenever a pointer is used to dereference a location in a different array. Because there are 5 arrays but only 4 `BND` registers, it is impossible for the compiler to keep the bounds for all the arrays in a `BND` register all the time. This results in many `BNDLDX` and `BNDSTX` instructions being generated by the compiler to load and spill the bounds information to and from memory.

The bound checking instructions (`BNDCU` and `BNDCL`) have very low execution cost. However, the `BNDSTX` and `BNDLDX` instructions have to access to the 2-layer structured bound tables stored in the main memory, they are very slow compared to bound checking instructions. To measure this cost, we did an experiment comparing `BNDCU` with `BNDSTX`/`BNDLDX` instructions. We randomly generate 1000 memory addresses, and use an address lower than them all to perform 1000 times `BNDCU` instructions, and made sure there are not bound violations. Then we use `BNDSTX` to store the first 500 instructions into bound tables, and load them all back one by one to a bound register `BND0`. The results of this experiment show that the bound checking instruction, `BNDCU`, has almost same execution time as a `NOP` instruction (1000 instructions in 0.45ms), while the bound store+load instructions `BNDSTX`/`BNDLDX` cost almost 1000× more than `NOP` (1000 instructions in 432ms).

With real applications, the number of objects in the bound table can become quite large. However, as the number of `BND` registers is fixed at 4 in the hardware architecture, this causes heavy use of the `BNDSTX` and `BNDLDX` instructions, resulting in high overhead. To see this in practice, we used a recent MPX-enabled version of GCC (version 6.1) to compile the SPEC 2006 benchmarks and found that this imposed 2× to 4× runtime overhead. Thus, to ensure low overheads, these result indicate that the number of `BNDSTX` and `BNDLDX` instructions must be minimized. Ensuring this is one of the main reasons LMP is able to provide low overhead.

3. METHODOLOGY

3.1 Threat Model

We assume a realistic attacker that can exploit memory corruption vulnerabilities to change arbitrary memory locations (so long as they are permitted by the hardware) to values of their choosing. We also assume that the attacker is aware of the address locations of key data structures such as pointers, stacks and meta-data and can arbitrarily target them with the memory corruption vulnerability. We assume the goal of the attacker is to corrupt a code pointer to compromise the control-flow integrity of a program.

Despite this powerful attacker, we do assume that the attacker is limited in some realistic ways. For example, the attacker cannot directly modify registers in CPUs or change any memory that is marked read-only, such as the code pages, as both would allow the attacker to remove or bypass the compiler-inserted instrumentation that LMP uses. The attackers also cannot compromise the integrity of the target program before it is loaded into the memory, which means that attacks on the program loader and operating system are out of scope for LMP. LMP is intended to mitigate the exploitation of memory corruption vulnerabilities by remote or unprivileged attackers for the purposes of privilege escalation.

In general, there are two types of code pointers that need to be protected: function-pointers (i.e. forward-edge) and return addresses (i.e. backwards-edge). LMP focuses on protecting against attacks on return addresses and assumes use of an existing forward-edge CFI protection scheme to protect functions pointers from being corrupted. There is a rich body of literature addressing the problem of forward-edge protection. For example, the virtual calls in C++ indirect-control transfers through VTables can be hijacked by attackers [5] to redirect execution to malicious code. These type of protections can be attained with low overhead by previous work, such as VTV [34], VTable Interleaving [3] and VTrust [40]. Our LMP system can work together with current forward-edge CFI defenses to provide full CFI protection.

3.2 Memory Protection with MPX

LMP uses two components to protect return addresses: the shadow stacks and the protected memory region allocator. First, standard shadow stacks are used to maintain a second copy of return addresses. The shadow stack is updated on a function call and checked when functions return. An attacker would have to corrupt both the program stack at function call site and the shadow stack to successfully corrupt a return address. Thus, to prevent the attacker from corrupting the shadow stack, MPX instructions are inserted by LMP to ensure that only the instructions inserted by LMP at function calls to update the shadow stack can write to the shadow stack.

Based on the threat model described in Sec. 2, only store operations could modify the shadow stack area, and the code pages are read-only so an attacker could not remove bound checks to store operations. An attacker could try to jump directly to a store instruction and avoid executing the bound-checks, but to do this, the attacker would have to corrupt a code pointer, which the CFI provided by LMP a complementary forward-edge CFI scheme prevents. Thus, the backwards-edge protection LMP provides hinges on the ability to protect the shadow stacks from corruption by a memory safety vulnerability.

To protect the shadow stack, we instrument each store instruction in the program to make sure that it cannot access the memory region of shadow stacks even if the attacker has modified the effective address that the instruction targets. Despite, there being many store instructions in the program, they are all checked against the same bounds, as LMP need only check that they do not target the shadow stack. This is efficient since this avoids the need to use the expensive `BNDLDX` and `BNDSTX` to modify the bounds that LMP must check – LMP simply sets the upper and lower bounds of a `BND` register to the lower and upper regions of the shadow stack and proceeds to instrument each store in the program to ensure that it does not fall within that region. However, in multi-threaded programs, there will be one shadow stack for each thread. A naïve solution would use a different `BND` register to store the upper and lower addresses for each stack, but this would require the expensive `BNDLDX` and `BNDSTX` instructions to load and store the stack bounds into the `BND` registers, hurting performance. Instead, we observe that all shadow stacks are in the same protection class – i.e. regardless of which thread a store is executing in, it should not be able to access *any* of the shadow stacks. This means that all shadow stacks can be placed in a contiguous region of memory and protected with a single `BND` register. Thus, the other component of LMP is a scheme that allocates standard shadow stacks so that they are in a single contiguous region of memory. In the same way, all other auxiliary data structures that LMP employs are also be protected from modification, by being allocated in the protected region that is restricted by MPX instructions.

3.3 Using the Shadow Stack

In order to restrict return instructions, LMP records the return address in the shadow stack upon each function call, where it will be protected from corruption by an attacker. We illustrate the idea of shadow stack layout of the LMP system in Figure 2.

Another difference from the other shadow stack approaches is that LMP compares function return address with the one stored in the shadow stack using MPX bound checking instructions. It optimizes the overhead from compare/branch instructions in standard shadow stack implementation and details will be presented later in this section.

As mentioned earlier, the shadow stacks are all located in a contiguous region of memory. Moreover, this region is statically defined at program startup and since it is inaccessible to any memory instruction other than shadow stack operations inserted by LMP, the region cannot be used to store any other type of data other than shadow stacks. The main difference between our shadow stack implementation and other shadow stack or safe stack implementations [10] is that LMP is not free to place shadow stacks any location or offset-based region for convenience, but must instead place them in the predefined shadow stack region. Since each thread must have its own shadow stack, we must define a mapping function that allows the shadow stack code to find the shadow stack for any given thread, but also maps each shadow stack into the predefined region.

One option is to make the predefined region as large as the region where regular stacks can be allocated. This would be efficient as each shadow stack could then be located at a fixed offset from the thread's regular stack. However, the pthread interface permits stacks to be created anywhere in a process' virtual address space. As a result, we would have to reserve one half of the virtual address space for the predefined region. While this is likely acceptable in most cases

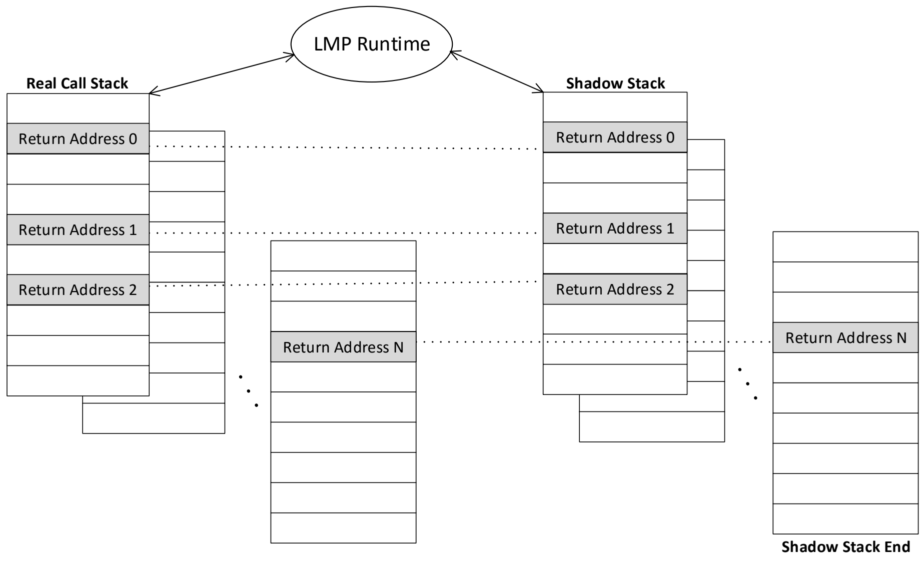

Figure 2: *The illustration of LMP shadow stacks*

for 64-bit code, it can present problems if processes need to allocate memory at a particular virtual address space.

Instead, a more costly, but flexible alternative is to dynamically allocate and map stack space from the predefined region as threads and their corresponding shadow stacks are created. While this might be slightly more expensive than the fixed-offset approach, we show that it is still practical, and can give a more conservative estimate of the overhead of different LMP implementation options. LMP uses a mapping table that stores the offset between a thread's regular stack and corresponding shadow stack. Both the function entry and function return instrumentation use the mapping table to find the corresponding shadow stack for the thread. The predefined region is then partitioned into several fixed-sized shadow stacks, and another table records which shadow stacks are in use and which are free. When a thread is created, LMP finds an unallocated shadow stack and updates the mapping table with the offset between the thread's regular stack and its newly allocated shadow stack. When a thread is destroyed, the thread is deallocated and the offset in the table is cleared. These allocation and deallocation operations only occur during thread creation and destruction.

LMP inserts instrumentation on function entry that stores the return address into the shadow stack. Because this memory operation is inserted by LMP, it needs not be bound-checked. At function return, LMP inserts instrumentation that will find the corresponding return address in the shadow stack and compare it against the address that control flow is going to. In this way, the shadow stack can ensure that when execution returns, the integrity of the return address is not tampered with. A thread's regular and shadow stack have the same layout so a return address on the regular stack will have the same offset from the base of the stack as the corresponding return address' offset from the base of the shadow stack. Thus, only the offset between the regular stack base and the shadow stack base needs to be stored in the mapping table. This design is different from RAD [7] which uses a custom stack layout. Because of this, they must search through the shadow stack to find a match, while LMP does not.

```
...
PUSH   %rsp
CALL   _map_table                 # find shadow stack via mapping table
                                  # return shadow stack address in %rax
...
MOV    (%rsp), %rdx
MOV    %rdx, (%rax)               # copy ret addr to shadow stack address in %rax
...
(FUNCTION CALL BODY)
...
MOV     (%rsp), %rdx              # put function return address in %rdx
BNDMK %bnd0, [(%rax), 0]          # put the address in shadow stack in a bnd
                                  # register %bnd0
BNDCU  %rdx, %bnd0
BNDCL  %rdx, %bnd0                # check return address with the one in shadow
                                  # stack
...
```

Figure 3: *Assembly code example for instrumented function entry/exit.*

We give an example of execution sequence in steps after code instrumentation for shadow stack operations, and an assembly code snippet in Figure. 3:

1. On function entry:
   (1) prepare shadow stack address in register %rax
   (2) copy return address in %rsp to shadow stack
2. Execute function call and body
3. On function return:
   (1) copy return address in shadow stack to bound register %bnd0
   (2) use bound checking instruction to check return address in %rsp and %bnd0

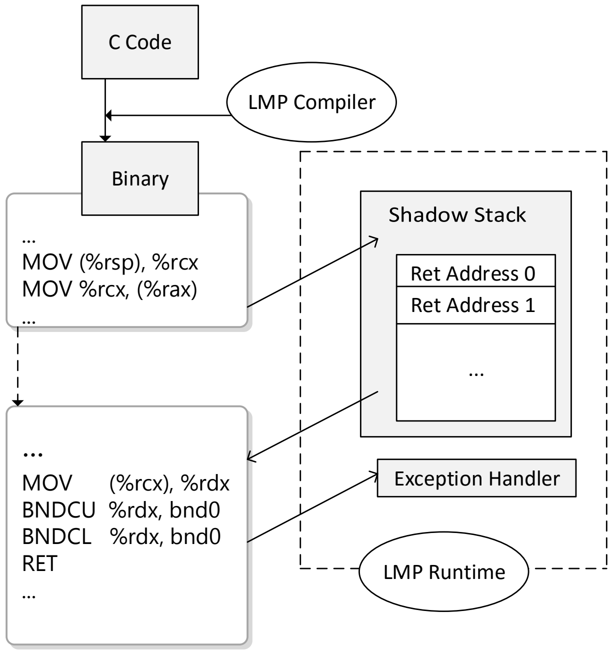

Figure 4: *A flow chart of how LMP system works.*

We use MPX bound checking instructions `BNDCL` and `BNDCU` instead of a series of compare and jump instructions to do the equality comparison. We set the return address in the shadow stack as the upper and lower bound in the bound register (`BND0`), then bound-check it against the function return address. Using MPX instructions to check the return address improves performance the same way the MPX instructions improve memory bound-checks – the MPX instructions avoid extra branch and check instructions that would normally be needed to check the result of the comparison. Instead, MPX instructions will throw an exception if the check fails.

3.4 Execute a Program with LMP

We give an illustration of our LMP system conceptual design by providing a simple example of how the LMP system works with a user program, as shown in Figure. 4.

The LMP-enabled compiler instruments the application source code at compile-time. When the program starts, the LMP runtime prepares the shadow stack memory region and stores its lower boundary and upper boundary to the bound register `BND1`. This is for the protection of the shadow stack from any illegal modification. When the program is running, it stores return addresses to the shadow stack when a function call happens and the return address is pushed to the normal call stack. When the function returns, two addresses stored in the normal stack and in the shadow stack is compared. Throughout the program, whenever there is a memory operation that stores values to a memory address, we instrument the code to verify that the address is not in the range of the shadow stack using bound checking instructions.

Under certain special cases, such as C++ exception handling, the call stack will unwind due to `setjmp`/`longjmp` instructions causing function call and return mismatching. In the method we propose with LMP, as long as the compiler does not change the original call stack with exception information (e.g., GCC stores it in another side-table), the return addresses in original call stacks and in shadow stacks correspond to the same offset to the stack top addresses, thus the stack unwinding by exception handling operations is not affected.

While we have not implemented it, we believe LMP can be extended to provide backward-edge protection for binary-only CFI. With a control-flow graph (CFG) generated through disassembly analysis of a binary and some changes to pthread library functions, the LMP system can also work with binary-only CFI approaches that employ binary-rewriting to add CFI instrumentation.

4. IMPLEMENTATION

The LMP system has two main parts: The LMP-enabled compiler and the LMP runtime library. For the compiler part we modify the register transfer language (RTL) passes for instrumenting boundary checking to ensure that there can be no unauthorized writes to the memory region where the shadow stacks is stored. The LMP runtime is responsible for managing the allocation of shadow stack and store of the return addresses from function call stacks.

4.1 LMP-enabled Compiler

The implementation of LMP-enabled compiler is based on GCC 5.2.0 with approximately 600 lines of code modified/added to the RTL passes. The main reason for modifying the compiler and adding new RTL passes is to do code instrumentation at the assembly level. Both shadow stack operations and code to protect the shadow stack memory region from being modified are instrumented by the LMP compiler.

In the GCC RTL passes, we modify the source code in `final.c` and `insn-output.c` that take care of assembler code output for functions. Among them, `final_end_function()` helps emit assembly code in function exit, we add our code here to do instrumentation for shadow stack operations.

To implement shadow stacks, at each function call stack operation when the function pushes return address, the compiler instruments the code to get the address and save a copy to the thread's shadow stack. The location of the shadow stack is found by indexing into the stack region using the calling thread's Thread ID, which is retrieved via the system call `gettid()`. At first, it might seem like a system call would be overly expensive, but such operations are highly optimized and our measurements show that the cost of `gettid()` on modern Linux kernels is negligible. At each return instruction, the compiler instruments the code to get the Thread ID and ask the LMP runtime for the return address stored in the shadow stack. If the address in the return instruction does not match the one in the shadow stack, it sends a bound violation message to LMP runtime. In the GCC passes, we identify the function calls by looking for the RTL expression code `call_insn`, with the format:

$$(call \quad (mem : fm \quad addr) \quad nbytes)$$

where the `addr` is the address of that subroutine.

For bound checking of memory operations, we change the RTL passes of GCC to find RTL expressions containing memory operations that store values to main memory

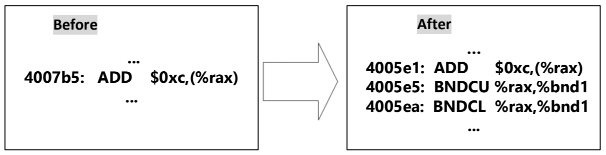
Before
```
      ...
4007b5:  ADD   $0xc,(%rax)
      ...
```
After
```
      ...
4005e1:  ADD     $0xc,(%rax)
4005e5:  BNDCU  %rax,%bnd1
4005ea:  BNDCL  %rax,%bnd1
      ...
```

Figure 5: *An example of LMP instrumentation for store instruction.*

address. The address is taken to compare with the upper and lower boundary addresses of the shadow stack, which is stored in the bound register BND1, where the bounds of the memory region where the shadow stacks reside is stored. A bound violation will be triggered if the address falls into the memory range of the shadow stack which means the pointer that the memory store uses as its target may have been corrupted by an attacker.

We give an example of the code instrumentation results in Figure. 5 to show the assembly code before and after instrumentation. The `add` instruction writes to main memory, and the instrumented assembly code `bndcu` and `bndcl` checks if the memory address to be changed is within the protected shadow stack region.

4.2 LMP Runtime

The LMP runtime is implemented with approximately 700 lines of C source code. As this is a proof-of-concept prototype design, we allocate a virtual memory region of 2GB for the shadow stacks. The reason behind the number of memory size is that in our test environment the OS has maximum number of 62057 threads (from `/proc/sys/kernel/threads-max`), and for each possible thread we give 32KB to the shadow stack, which we believe is more than enough as the benchmarks we used never exceed 8KB per thread in call stack. In our implementation, both the numbers of maximum threads and the space for each shadow stack are tunable. Since the shadow stacks are allocated in the 64-bit virtual address space, they only take a tiny fraction of it. Also, because most of the shadow stacks may never be written to, they only consume virtual address space and the operating system never needs to actually allocate physical memory to back them.

We could have also dynamically allocated shadow stacks in memory, which would allow the shadow stack region to be dynamically extended and reduced in size to accommodate growth and reduction in shadow stack usage. This would likely add some overhead in exchange for better virtual address space utilization. However, given that virtual address space is generally not a limiting factor on 64-bit architectures, we do not believe that this extra overhead is justified.

When the instrumented program needs the LMP runtime to store a function return address to the shadow stack, the runtime takes the offset between the base of the call stack and the address that stores the return address, and a Thread ID to process them in function `LMP_push_ss(return_addr, offset, threadID)`, then finds the shadow stack prepared for that thread and stores the function return address in the shadow stack. When the program function returns and the address needs to be compared with the one stored in the shadow stack, it calculates the offset between the base

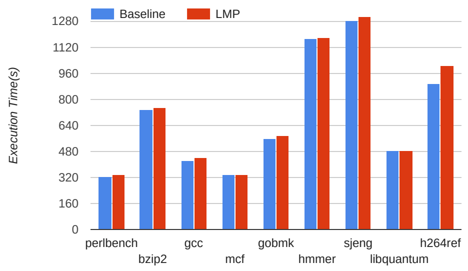

Figure 6: *LMP overhead by comparison of execution time between baseline and LMP.*

of the call stack and the address that stores the function return address and uses `return_addr=LMP_pop_ss(offset, threadID)`, then LMP runtime will get the return address stored in the shadow stack.

5. EVALUATION

In this section we evaluate the effectiveness and different aspects of overheads of our LMP system. We run our experiments on an Intel i5-6600K with 4 cores @3.5GHz in 64-bit mode with 8G RAM. The benchmarks are run on Fedora 22 with Linux kernel 4.1.7.

5.1 Performance Overhead

We evaluate the overheads of the LMP system using CINT 2006 benchmarks. All results are 5-time average numbers that gathered from the non-reportable mode of SPEC benchmark. We compare the results with the baseline without applying LMP. As shown in Figure. 6, the average performance overhead of LMP in comparison to the baseline performance is 3.90%. The `h264ref` benchmark has the highest overhead of 12.55%, mainly because it has many more function calls and RET instructions than others. Without the `h264ref` benchmark the average overhead is only 2.12%.

To justify the main sources of overheads introduced by the LMP system, we further separate them into three parts of the system: context settings, bound-checking and shadow stack operations. Context settings includes the runtime library initialization, retrieving ThreadID via system calls etc. Bound-checking involves the time that spent by MPX bound instructions. Shadow stack operations consist of all operations dealing with the shadows stacks.

We measure how much each component contributes to the overall overhead by removing the other 2 components and measuring the overhead with only one component added to each benchmark. Over all the CINT 2006 benchmark results, the average overhead of context settings is 0.1%, bound checking is 0.52% and shadow stack operations is 3.27%. From Figure 7 we can find that context setting and bound-checking almost contribute negligible amount of overheads. Shadow stack operations are the main contributor, which on average accounts for 84% of all the overheads. The performance penalty of the memory protection is only 15% of the overall overhead and the remaining 1% can be attributed to infrequent setup and stack allocation/deallocation operations. The results here are inline with other heavily optimized shadow stack implemen-

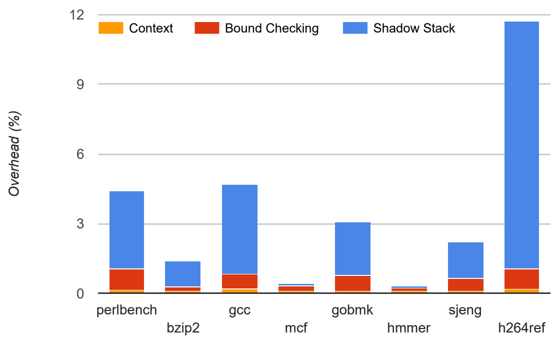

Figure 7: *Overhead components of LMP.*

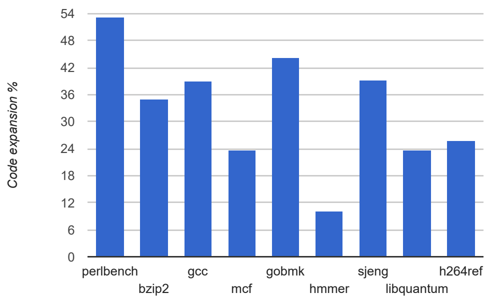

Figure 8: *Code Expansion of LMP.*

tations [10] that claim a few variants of shadow stacks performance overheads around between 2% and 10% for the same benchmark set. As a result, we believe this overhead is representative of the costs of LMP on current processors.

5.2 Code Expansion

LMP-enabled GCC emits assembly code to instrument the target program in the RTL passes, so there is an increase in code size. We directly compare the sizes of the binaries of each benchmark and calculate the percentage of code expansions that LMP introduces.

From Figure. 8, we can see that across the 9 benchmarks we have run, the code at assembly level expands by 39.27% in average. There is some variance among the code expansion numbers of the benchmarks, while the majority of which is contributed by the bound checking instructions, when there are more function calls/returns and memory store instructions of the benchmark, the more bound checking instructions are instrumented. We note that our prototype includes some extra debugging code which could be removed to further reduce code expansion.

5.3 Memory Overhead

The memory overhead introduced to the benchmarks on average is 19.3MB per program, and the average percentage of the maximum resident memory overhead is 9.73%. The memory overhead is mainly from the runtime library part of LMP system which manages the shadow stacks. As mentioned in Sec. 4 the memory allocation is not optimized in this research prototype implementation and we belive that there is likely space for improvement. We expect the memory overhead could be decreased significantly by adding dynamically allocating the mapping table as needed instead of pre-emptively allocating it for the maximum number of threads.

6. RELATED WORK

We review literature in the area of defense technologies to protect programs from control flow hijacking attacks.

Traditional attack methods using stack-smashing and code injection [28] can be protected by applying recent adoption of data execution prevention (DEP) [2]. Hardware support for DEP is present in virtually all x86 processors as a non-execute bit (NX bit, or called XD/XN bit depending on processor architecture), such that code in the data segment cannot be executed.

To counter the protection above, attackers have developed more sophisticated methods that do not rely on injecting new code, and that instead, rely on using existing code in the program. One of the early examples is return-into-libc attack [35], which can redirect program execution flow through libc functions. Similar exploitations such as return-oriented programming (ROP) attack [29] can also execute arbitrary computations by using a chain of existing code after changing return address at the function call stack. The latter has been shown to be Turing-complete.

Randomization is practical in hiding information about the memory layout of a program from attackers. Address Space Layout Randomization (ASLR) [27] has been proposed to defend against ROP attacks by mapping program processes and dynamic libraries into random virtual address space every time. Address Space Layout Permutation (ASLP) further re-orders sub-routines at the code segments on the basis of the randomization provided by ASLR [22]. However, the implementations of ASLR were soon to be found ineffective against de-randomization attack [31], costing only an few hundred seconds of additional time to compromise the target program. Similarly, ASLP is also vulnerable to de-randomization attacks [24].

CFI (Control Flow Integrity) [1] is introduced to guarantee that indirect control-flow transfers point to legitimate locations. To ensure that the return addresses in function call stacks are not tampered with, shadow stacks to store copies of return addresses are suggested. However, the performance overhead of original CFI is reported as high as 2× if the exact policy is enforced, so there are variants of coarse-grained CFI proposed with changes to the original policy. kBouncer [26] uses the Last Branch Record (LBR) x86 register that stores recent branches that CPU executed. It validates if the return address points to an instruction follows a call instruction, so the procedure is actually a heuristic mitigation of ROP attack. Using the same LBR register and similar policy as kBouncer, the work of ROPecker [6] adds additional static analysis to speculate future execution of a program to defend against ROP gadgets running, unfortunately however, is by-passible too [12]. The ROPGuard [16] proposes to check if the stack pointer points to a memory address outside of the stack, so the system would not allow ROP attackers execute payloads on the heap, however, before the target function is called the adversaries could still modify the stack pointer. The above defenses are also vulnerable to attacks that leverage hooks and hide malicious code within non-control data [36], if critical memory region is not protected at runtime. O-CFI [25] explores randomization approach to conceal program control-flow graph and applies MPX in bound-checking for guarding the branch instructions. However, it is still a coarse-grained CFI method and only provides probabilistic security guarantees since it does not fully protect function return addresses. Our LMP approach sticks to the original CFI policy in backward-edge protection, i.e., checking every function return address and ensuring the return address points to the function caller.

For forward-edge CFI protection, the paper that proposes VTV [34] finds out more than 90% indirect calls are virtual calls. Their method aims at protecting VTables from being hijacked, validates at runtime that the target VTables in a legit set, before a virtual method call is made. Performance of VTV depends on the size of legit VTable set, so the complexity of C++ class hierarchy would affect the overhead. On the basis of the idea, VTrust [40] and VTable Interleaving [3] improve the performance of VTV without needing global class hierarchy, and prevent VTable hijacking attacks. Our LMP system does not provide protection with forward-edge CFI, because with above mentioned approaches, the LMP can be easily combined with them by applying patches to the LMP-enabled compiler, thus a full-CFI protection is possible.

There are CFI variants proposed with different security targets. The techniques of original CFI have been used for the purpose of enforcing software-based fault isolation (SFI) [39]. XFI [14] also employs CFI policies with the help of debugging information in Windows PDB files to defend against ROP attacks. Data-flow Integrity (DFI) [4] follows CFI approach to prevent non-control data attacks. Hypersafe [38] is similar to fine-grained CFI protection. It has a target table for indirect branches and aims at protecting control-flow integrity of hypervisor.

Code-Pointer Integrity (CPI) [23] explores a security mechanism that divides process memory into two parts: safe memory region and regular memory region. Through static analysis, memory objects that have pointers including code and data pointers are put into a safe memory region for protection against illegal tampering. However, flaws of CPI approach have been pointed out [15] because its safe memory region is not well-protected. The essential idea of LMP is also guarding the memory region where shadow stacks located. We use new hardware feature of fast memory boundary checking to ensure the allocated shadow stack region is protected effectively and efficiently.

Other hardware-based CFI approaches have recently been proposed, e.g., HCFI [8] and HAFIX [11] have their system implemented running on customized FPGA board or SPARC embedded system. In comparison, LMP is the first system with hardware-assisted memory protection compatible with commercially available CPU and other hardware. Control-Flow Enforcement Technology (CET) [21] was announced in a technology preview as of June 2016. CET introduces a new exception class (#CP) with interrupt vector 21 and a new `ENDBRANCH` ISA instruction to help mark legal targets for an indirect branch or jump. It also uses hardware protections to limit access to the shadow stack to only function call and return instructions so that regular memory stores are prohibited from modifying the shadow stack. Since CET was only recently announced, and no hardware is available, we are unable to evaluate the overhead of CET against LMP at this time.

7. CONCLUSION

Memory protection is a keystone of all defense techniques against memory corruption attacks. Without properly protecting the shadow stack, CFI approaches cannot effectively prevent ROP attackers and have been proven to be insecure in general. Our work proposes a light-weighted memory protection system to prevent critical memory region storing return addresses of function call stacks, namely the shadow stacks. Leveraging recent available MPX hardware features, our approach achieves low overhead in enforcing only legal accesses to the protected region is allowed, so that return addresses cannot be tampered with by an attacker. For future work, we will complete the LMP protection on forwarding-edge and explore the possibility of applying LMP without the limitation of recompilation of the program, for example, use the help of binary re-writing to perform the shadow stack functions for protection.

Acknowledgement

We would like to thank Professor Ding Yuan, Mariana D'Angelo, Michelle Wong, Beom Heyn Kim, Afshar Ganjali, Sukwon Oh, Diego Bravo Velasquez and Peter Sun for their valuable feedback. We also thank the anonymous reviewers for their comments and help in improving the quality of this paper. The research in this paper was supported by an NSERC Discovery Grant.

8. 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* (Alexandria, Virginia, 2005).

[2] Andersen, S., and Abella, V. Data execution prevention. https://technet.microsoft.com/en-us/library/bb457155.aspx, 2004. Last accessed: 2016-09-01.

[3] Bounov, D., Kici, R. G., and Lerner, S. Protecting C++ dynamic dispatch through VTable interleaving. In *Proceedings of the 23rd Annual Networked & Distributed System Security Symposium (NDSS)* (San Diego, California, 2016).

[4] Castro, M., Costa, M., and Harris, T. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington, 2006).

[5] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (Chicago, Illinois, 2010).

[6] Cheng, Y., Zhou, Z., Yu, M., Ding, X., and Deng, R. ROPecker: A generic and practical approach for defending against rop attacks. In *Proceedings of the*

21st Annual Networked & Distributed System Security Symposium (NDSS) (San Diego, California, 2014).

[7] Chiueh, T.-C., and Hsu, F.-H. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings of the The 21st International Conference on Distributed Computing Systems* (Washington, DC, 2001).

[8] Christoulakis, N., Christou, G., Athanasopoulos, E., and Ioannidis, S. HCFI: Hardware-enforced control-flow integrity. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy* (2016).

[9] Criswell, J., Dautenhahn, N., and Adve, V. Virtual Ghost: Protecting applications from hostile operating systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Salt Lake City, Utah, 2014).

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

[11] Davi, L., Hanreich, M., Paul, D., Sadeghi, A.-R., Koeberl, P., Sullivan, D., Arias, O., and Jin, Y. HAFIX: Hardware-assisted flow integrity extension. In *Proceedings of the 52nd Annual Design Automation Conference* (2015).

[12] Davi, L., Sadeghi, A.-R., Lehmann, D., and Monrose, F. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 35th IEEE Symposium on Security and Privacy* (San Jose, California, 2014).

[13] Durumeric, Z., Kasten, J., Adrian, D., Halderman, J. A., Bailey, M., Li, F., Weaver, N., Amann, J., Beekman, J., Payer, M., and Paxson, V. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (Vancouver, BC, Canada, 2014).

[14] Erlingsson, U., Abadi, M., Vrable, M., Budiu, M., and Necula, G. C. XFI: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington, 2006).

[15] Evans, I., Fingeret, S., Gonzalez, J., Otgonbaatar, U., Tang, T., Shrobe, H., Sidiroglou-Douskos, S., Rinard, M., and Okhravi, H. Missing the point(er): On the effectiveness of code pointer integrity. In *Proceedings of the 36th IEEE Symposium on Security and Privacy* (San Jose, California, 2015).

[16] Fratrić, I. ROPGuard: Runtime prevention of return-oriented programming attacks. http://www.ieee.hr/_download/repository/Ivan_Fratric.pdf, 2012. Last accessed: 2016-09-01.

[17] Ganesh, K. Pointer checker: Easily catch out-of-bounds memory accesses. https://software.intel.com/sites/products/parallelmag/singlearticles/issue11/7080_2_IN_ParallelMag_Issue11_Pointer_Checker.pdf. Last accessed: 2016-09-01.

[18] Göktas, E., Athanasopoulos, E., Bos, H., and Portokalidis, G. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy* (San Jose, California, 2014).

[19] Hu, H., Shinde, S., Sendroiu, A., Chua, Z. L., Saxena, P., and Liang, Z. Data-oriented programming: On the expressiveness of non-control data attacks. In *Proceedings of the 37th IEEE Symposium on Security and Privacy* (San Jose, California, 2016).

[20] Hund, R., Willems, C., and Holz, T. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of the 34th IEEE Symposium on Security and Privacy* (Washington, D.C., 2013).

[21] Intel. Control-flow enforcement technology preview, Document Number: 334525-001, Revision 1.0. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf, June 2016. Last Last accessed: 2016-09-01.

[22] Kil, C., Jim, J., Bookholt, C., Xu, J., and Ning, P. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Proceedings of Computer Security Applications Conference (ASAC)* (Miami Beach, Florida, 2006).

[23] Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., and Song, D. Code-pointer integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation* (Broomfield, Colorado, 2014).

[24] Liu, L., Han, J., Gao, D., Jing, J., and Zha, D. Launching return-oriented programming attacks against randomized relocatable executables. In *Proceedings of the 10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)* (Changsha, China, 2011).

[25] Mohan, V., Larsen, P., Brunthaler, S., Hamlen, K. W., and Franz, M. Opaque control-flow integrity. In *Proceedings of the 22nd Annual Networked & Distributed System Security Symposium (NDSS)* (San Diego, California, 2015).

[26] Pappas, V., Polychronakis, M., and Keromytis, A. D. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Security Symposium* (Washington, D.C., 2013).

[27] PaX-Team. PaX ASLR (address space layout randomization). http://pax.grsecurity.net/docs/aslr.txt, 2003. Last Last accessed: 2016-09-01.

[28] Pincus, J., and Baker, B. Beyond stack smashing: recent advances in exploiting buffer overruns. *IEEE Journal of Security and Privacy 2*, 4 (July 2004), 20–27.

[29] Roemer, R., Buchanan, E., Shacham, H., and Savage, S. Return-oriented programming: Systems, languages, and applications. *ACM Transaction on Information and System Security 15*, 1 (March 2012), 2:1–2:34.

[30] Seibert, J., Okhravi, H., and Söderström, E. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014).

[31] Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security* (Washington, D.C., 2004).

[32] Snow, K. Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., and Sadeghi, A.-R. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy* (Washington, D.C., 2013).

[33] Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., and Walter, T. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security* (2009).

[34] Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, U., Lozano, L., and Pike, G. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium* (San Diego, California, 2014).

[35] Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V., and Ning, P. On the expressiveness of return-into-libc attacks. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection* (Menlo Park, California, 2011).

[36] Vogl, S., Gawlik, R., Garmany, B., Kittel, T., Pfoh, J., Eckert, C., and Holz, T. Dynamic hooks: Hiding control flow changes within non-control data. In *Proceedings of the 23rd USENIX Security Symposium* (San Diego, California, 2014).

[37] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient software-based fault isolation. *SIGOPS Operating System Review 27*, 5 (Dec. 1993), 203–216.

[38] Wang, Z., and Jiang, X. HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 31st IEEE Symposium on Security and Privacy* (San Jose, California, 2010).

[39] Zeng, B., Tan, G., and Morrisett, G. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (Chicago, Illinois, 2011).

[40] Zhang, C., Carr, S. A., Li, T., Ding, Y., Song, C., Payer, M., and Song, D. VTrust: Regaining trust on virtual calls. In *Proceedings of the 23rd Annual Networked & Distributed System Security Symposium (NDSS)* (San Diego, California, 2016).