Secure and Efficient Control Data Isolation with Register-Based Data Cloaking

Xiayang Wang, Fuqian Huang, and Haibo Chen, *Senior Member, IEEE*

Abstract—Attackers often exploit memory corruption vulnerabilities to overwrite control data and further gain control over victim applications. Despite progress in advanced defensive techniques, such attacks still remain a major security threat. In this article, we present Niffler, a new technique that provides lightweight and practical defense against such attacks. Niffler eliminates the threat of memory corruption over control data by cloaking all control data in registers along its execution and only spilling them into a dedicated read-only area in memory upon a shortage of registers. As an attacker cannot directly overwrite any register or read-only memory pages, no direct memory corruption on control data is feasible. Niffler is made efficient by compactly encoding return address, balancing register allocation, dynamically determining register spilling and leveraging the recent Intel Memory Protection Extensions (MPX) for control data lookup during register restoring. We implement Niffler based on LLVM and conduct a set of evaluations on SPECCPU 2006 and real-world applications. Performance evaluation shows that Niffler introduces an average of only 6.3 percent overhead on SPECCPU 2006 C programs and an average of 28.2 percent overhead on C++ programs.

Index Terms—Control data isolation, memory error, processor registers

1 Introduction

Programs intensively use indirect control transfer like function returns and indirect function calls. Such instructions may use in-memory data as the destination addresses, which, however, can easily be corrupted and exploited through memory corruptions like buffer overflows. The exploited data is named *control data* or *code pointer*. The exploits, named *control data attacks*, mislead the indirect control transfer into arbitrary instructions and ultimately execute arbitrary code [1]. Though the classic shellcode insertion can be defeated in modern systems with the support of W$\oplus$X and Data Execution Prevention (DEP) [2], attackers can reuse existing code [3], [4] or even reinterpret the code [5], [6] for arbitrary execution.

Isolating the control data protects the data from direct memory corruption and completely defeats control data attacks like return-oriented programming (ROP) [4], [6], [7] and corrupting the function pointers. For example, the isolated return addresses form a shadow stack, which restrict any return instruction to only transfer the control flow back to the corresponding call site. Similarly, the destinations of indirect calls are restricted by the isolated function pointers, whose values are not affected by direct memory corruption.

Control data isolation provides stronger security guarantee than related control data attack defence techniques like code address randomization [8], [9], [10] and Control Flow Integrity (CFI) [11]. The security guarantee of randomization depends on information hiding. However, researchers have developed multiple advanced attacks to disclose memory pages and break the information hiding [3], [12], [13], [14]. Furthermore, the recent Meltdown [15] and Spectre [16] attacks have made the community question the reliability of information hiding given the successful exploits of leaking data without crashing the victim program. CFI restricts the program's runtime execution to follow only a pre-defined set of paths. Due to the inaccuracy of the legal paths, CFI enforces a relaxed security guarantee, such as permitting a function returns to multiple call sites invoking the same function.

Code Pointer Separation (CPS) [17] and Cryptographic Control Flow Integrity (CCFI) [18] are two examples of control data isolation. The security guarantee provided and performance overhead incurred depend on the technique used to enforce the isolation. CPS [17] places return addresses and function pointer in a separate memory region. On 32-bit x86 processors, it protects the isolated region with segmentation. However, when an x86 processor runs in 64-bit mode where segmentation is limitedly supported, the randomization-based CPS implementation can be broken by information leak [13]. CCFI [18] encrypts control data to defeat direct memory corruption. However, placing the encrypted control data into memory is potentially vulnerable to replay attacks. Furthermore, CCFI incurs an average of more than 20 percent performance overhead, even though it leverages AES-NI [19] and processor speculative execution.

Other software-based shadow stacks are vulnerable under the threat of arbitrary memory corruption [20] or information leak [9], depend on an obsolete hardware

• *The authors are with the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai 200240, China.*
E-mail: {xywang.sjtu, sjtu.ticholas.huang, haibochen}@sjtu.edu.cn.

Manuscript received 25 Dec. 2018; revised 10 Sept. 2019; accepted 30 Sept. 2019. Date of publication 11 Oct. 2019; date of current version 13 Jan. 2020.
(Corresponding author: Xiayang Wang.)
Recommended for acceptance by T. Li.
Digital Object Identifier no. 10.1109/TC.2019.2946770

segmentation feature [11], or suffer from high performance overhead [21], [22].

In this paper, we show that control data isolation can be enforced both efficiently and securely by cloaking the data into registers, which are neither vulnerable to information leaks nor broken by replay attacks. Reading and writing to a register have short latency. Memory corruption exploits can only be effective when the victim data can be pointed to by an attacker-controlled pointer variable, while processor registers are only addressable by their names which are hard-coded in the program instructions if no aliased register exists. Cache appears to be another case of in-processor data storage, which has been utilized to defend physical attacks like cold boot. However, cache is addressable by virtual addresses, indicating the data in cache can still be vulnerable to memory corruptions.

Related works like StackGhost [23] and PointGuard [24] also embrace the benefit of processor registers. StackGhost exploits the register window feature on SPARC architecture to save return addresses in registers. It is also restricted by the SPARC architecture that it traps to the kernel on every eight consecurive function calls. PointGuard ensures the pointers are encrypted unless they are in registers. They are limited by the frequent time-consuming operations that securely spill the register values.

We embody the idea of in-register control data cloaking in Niffler, which incorporates compiler instrumentation and works on commodity hardware. During compilation, Niffler instruments an application to place the control data into and retrieve it from a processor register. When registers become full, it is necessary to spill them securely without the risk of being corrupted. The instrumented program traps to the kernel and spills registers in a kerne writable memory space, which is immutable in user mode.

Though register spillings have been made secure, it will kill the performance if the application traps to the kernel too frequently. Niffler incorporates a series of techniques to reduce of frequency of kernel trapping, such as efficient data representation, balanced register allocation, dynamic spilling detection and fast register restoring.

We implement Niffler by extending LLVM on x86 platform. We utilize the MMX registers as they are rarely used. Therefore, Niffler would not conflict with the instrumented application in register use. We further utilize Intel Memory Protection Extensions (MPX) to optimize register restoring. We extend the Linux kernel to create secure memory regions for spilled register values, respond to register spilling, and support dynamic linking, multi-threaded and multi-processed applications. The kernel-side implementation is contained in a kernel module without modifying any kernel code.

We evaluate Niffler by instrumenting the SPECCPU 2006 benchmarks written in C/C++ and numerous real-world applications like servers and Linux utilities. The C programs in SPECCPU 2006 show only 6.3 percent overhead, which is much lower than the 23 percent overhead reported by CCFI [18]. The overhead on C++ programs in SPECCPU 2006 is 28.2 percent on average, which is also lower than CCFI.

TABLE 1
Number of CVEs on Memory Corruption Reported from January to April in 2018

Category	Count
Buffer Overflow	126
Buffer Underflow	4
Double Free	17
Use-After-Free	42
Memory Corruption	280
Memory Corruption $->$ Code Execution	106

Many of them are not marked with a specific type of memory corruption.

This paper makes the following contributions:

- A novel control data isolation technique that leverages the insight that registers defeat memory corruption naturally.
- The design of Niffler which embodies techniques to increase register utilization and supports dynamic linking, multi-threaded and multi-processed applications. We also show a novel utilization of Intel Memory Protection Extensions to optimize function pointer cloaking.
- Detailed analysis and evaluation of Niffler that shows the register-based control data isolation technique provides strong security guarantee and incurs low overhead on applications with moderate number of control transfers.

2 Background

2.1 Memory Corruption and Control Data Attack are Still Prevalent

Memory corruption is regarded as one of the most common types of program vulnerabilities [1], [9], [25], [26]. An infamous case of memory corruption is buffer overflow, by which the program writes to memory beyond the end of a buffer on the stack or the heap. Nowadays, memory corruptions are still routinely discovered and reported, though the history of buffer overflow attacks dates back to the 1980s [27]. In March 2018, a buffer overflow vulnerability was revealed in the widely used email server, Exim, which imperiled an estimated 400,000 email servers [28].

We have studied the CVEs on memory corruptions based on the description in the reports. As Table 1 shows, during the first four months in 2018 alone, 280 memory corruption vulnerabilities are reported. In addition, memory corruption comes in multiple variants, among which the buffer overflow is the most common one. Notably, 106 out of the 280 memory corruption CVEs in Table 1 are marked by the reporter to have a risk of being exploited as arbitrary code execution. The significant number of exploitable memory corruptions that still exist nowadays calls for an effective defense on control data.

2.2 Intel MPX

Memory Protection Extensions are a set of Intel x86 processor features, starting from the Skylake microarchitecture, and work specifically on bound checking. MPX introduces new instructions and registers to define, remember and compare pointer bounds. The compiler can instrument the program with the new instructions. On a bound violation, the processor triggers a `#BR` exception and traps to the kernel.

MPX consists of four 128-bit registers, named `%bnd0-3`, to record the bounds of a pointer, which are made up of a 64-bit upper bound and a 64-bit lower bound. The instruction `bndcn` compares the pointer value in a general purpose register against one `%bndX` register as the upper bound. Similarly, `bndcl` compares against a lower bound.

MPX utilizes a two-level radix tree to record the mapping from a pointer variable, specified by its memory address, to its bounds. When a program needs to introduce more than four pairs of bounds, the `%bnd0-3` registers are spilled into the radix tree. MPX introduces a new instruction `bndldx` to restore one MPX register from the radix tree.

MPX manages one 32-byte block for each pointer variable in the radix tree. The block contains the upper bound, the lower bound, the value of the pointer variable and a reserved field. Each field has 8 bytes. The first level of the radix tree is as large as 2 GB, consisting of 2^{28} different pointers to the second level. The base address of the bound directory is kept in a register `%bndcfgu`. The 4 MB-sized second level consists of 2^{17} different 32-byte blocks.

3 Overview

Niffler ensures a *shadow-stack* policy for return address protection that forces each function to return to the exact corresponding call site. For other types of control data like function pointers and virtual function pointers in C++ applications, Niffler provides a *write-protected* policy that only the instrumented instructions can modify the isolated control data. Additionally, it handles stack unwinding during C++ exception handling and ensures correct unwinding according to the shadow stack.

Listing 1. An Example of Return Address Cloaking. Modeled After a BROP Exploit on Nginx. The Instrumentations are Underlined

```
1   typedef struct {
2    off_t content_length_n;
3   } ngx_http_headers_in_t;
4
5   void func2(void) {
6    @asm volatile ("movq $Line8, %func1_reg");
7    func1();
8   }
9
10  void func1(void) {
11   char buffer[NGX_HTTP_DISCARD_BUFFER_SIZE];
12   size = (size_t) ngx_min(
13    NGX_HTTP_DISCARD_BUFFER_SIZE,
14    r->headers_in.content_length_n);
15   /* attack: corrupt return address */
16   n = r->connection->recv(r->connection,
17    buffer, size);
18
19   asm volatile ("movq %func1_reg, (%rsp)");
20   return;
21  }
```

We illustrate the idea of register-based control data cloaking with an example. Listing 1 shows a stack buffer overflow exploit on Nginx from an advanced ROP attack named BROP [3]. The attacker controls `content_length_n` with a crafted HTTP request. A negative `content_length_n` is selected in Lines 12-14 and is casted into a large unsigned `size`. Thus Lines 16-17 write beyond the bound of `buffer` and corrupt the return address on the stack with the address of an ROP gadget. Note that BROP is able to bypass the canary on the stack [3].

Niffler instruments the application to defeat BROP. The instrumentation in Line 6 saves the return address in a register named `%func1_reg`, which is specified in a static analysis during compilation. The cloaked return address in register survives the buffer overflow in Lines 16-17. It is saved to the stack in Line 19 which reverts the effect of the corruption.

3.1 Challenges and Solutions

The amount of control data keeps increasing along with the program execution. When the registers fall short to hold a new piece of control data, Niffler makes the application to spill all registers to memory. In order to protect the in-memory data from memory corruptions in user space, the application traps to the kernel and spills all registers into a memory region, named *the secure region*, which is immutable in user mode.

Constant switches between kernel and application during register spilling are time-consuming. Niffler instruments the application in a way of high register utilization to reduce the frequency of spilling. It encodes the return addresses into short forms, named *return address IDs*. Multiple return address IDs can be placed into one register at the same time. Each function is assigned one register to hold its return address. Niffler balances the register allocation by assigning different registers to functions along a function call graph path.

Determining when to spill the registers at compile time can only make conservative decisions, which leads to a large number of spilling events even if there is no shortage of registers at runtime. Niffler lets the program detect register shortage dynamically. Before appending a return address ID to a register, the instrumented program checks whether that register has enough room for the appended data.

To reduce the cost of register restoring, Niffler maps the kernel-writable secure region as readable in user mode to make registers get restored without trapping to the kernel.

In addition to return addresses, Niffler instruments the application to cloak updated function pointers and virtual function table pointers in registers. Precisely determining a pointer's address during compilation is challenging. Instead, Niffler makes the application dynamically select one register to use.

3.2 Threat Model

We assume that the attacker is able to read and write to arbitrary locations in the application's memory address space. We trust the operating system kernel to load the instrumented program binary securely and set page permissions properly. As we focus on control data attacks, corrupting non-control data is outside our consideration.

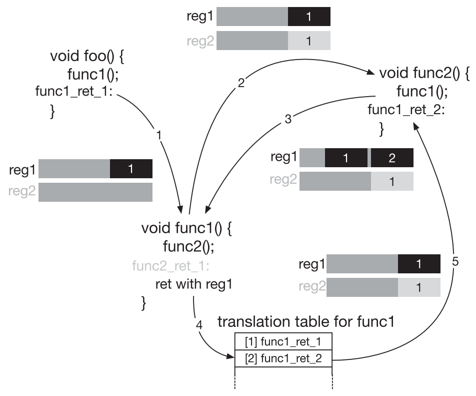


Fig. 1. Direct calls and returns in instrumented programs. Labels after the callsites to the same function are displayed in the same color in this figure. We assume that `func1()` returns after Step 3 rather than stepping into `func2()`.

4 Return Address Cloaking

Niffler assigns one register for each function to hold *return address IDs*. Fig. 1 illustrates how the return address cloaking works. All of the return address IDs that `func1()` uses when it returns to its caller function can be found in register `reg1`. All mappings from return address IDs to the corresponding full-length return addresses are recorded in *translation tables*.

In Steps 1-3 of Fig. 1, return address IDs are pushed into assigned registers on function invocations. Steps 4-5 pop the return address ID on returning from `func1()` and translate it back with the help of the *translation table*.

When the register does not have enough room to hold another return address ID, the application traps to the kernel and spills the registers into memory which is immutable in user mode. Registers are restored when the application tries to pop a return address ID from a register but finds that register is empty.

4.1 Return Address Encoding

The encoding of return address ID is based on the observation that all return addresses can be enumerated finitely by iterating over all function invocation instructions. Niffler maps each return address to an ID, a positive integer, during compilation. The reverse mappings from IDs to the corresponding virtual return addresses are generated at compile and link time, and recorded in translation tables. Translation tables are read-only data and their contents are loaded into memory during program initialization.

Niffler scans a linked program to set up the mappings. All direct call sites with the same callee function are aggregated into one set. Niffler assigns a *set-unique* ID to each call site in the set, starting from one, as the return address ID. Each callee function is associated with one translation table. Note that ranges of return address IDs in different sets can overlap. For example in Fig. 1, `func1_ret_1` and `func2_ret_1` are both encoded as 1. Thanks to different translation tables used by `func1()` and `func2()`, Niffler ensures the two IDs are translated back differently as long as they come from different direct call site sets.

For an indirect call site like `callq *%rax`, Niffler does not infer its possible destinations. Instead, Niffler treats all indirect call sites in the same linkage unit into a single set and assigns a return address ID to each of them in this range. A function is *address-taken* when its symbol or address is referred in statements like assignments. The callee of an indirect call site must be an address-taken function. All address-taken functions share the single translation table. Due to the uniqueness of return address IDs, they can be translated back differently even though they share the same table.

Supporting Dynamic Linking. The application and its dependent libraries are compiled and instrumented separately. However, the set of indirect call sites are collected locally in one linkage unit, which makes the application and shared libraries create separate translation tables for address-taken functions. Return address IDs for indirect call sites assigned within different linkage units may well collide after linking and loading.

Niffler solves the issue by renaming all indirect call sites in shared libraries after loading. All translation tables for indirect call sites from all linkage units are merged before program execution. The instrumented shared library remembers the offset to the beginning of the merged translation table, where its part of translation table resides. When pushing a return address ID on an indirect call site in a shared library, Niffler adds the offset to the locally assigned ID, which results in a globally unique ID. The per-shared-library offset value is kept read-only after merging.

Niffler treats all direct invocations to shared library functions as indirect calls, as these call sites are invisible to the callee function during the process of assigning return address IDs.

Length of Return Address IDs. Niffler determines the lengths of all return address IDs at compile time. The return address IDs for direct call sites are encoded with just enough bits, as the total number of direct call sites to the same function within a linkage unit is known at compile time. For example, if `callq bar` is assigned ID 5 in a total of 7 peer call sites, Niffler encodes all IDs in this range with only three bits, and this call site is represented as 101 in binary.

For all indirect call sites and external function call sites among all linkage units, return address IDs are encoded with the same number of bits, as the total amount of call sites cannot be determined at compile time.

4.2 Register Allocation

Each function is assigned one register to hold all return address IDs used by this function to return to the caller. Multiple functions may be assigned the same register. With the help of the compressed encoding scheme, a single register is able to hold multiple return address IDs at the same time. Each ID can be distinguished as its length is known at compile time.

Function invocations can be nested and recursive, which may form a deep call stack of dozens of functions during execution. The worst register allocation assigns all functions with the same register. As the result, each call site puts the return address ID into that register, which makes it quickly become full and trigger a spilling, a time-consuming operation in Niffler.

A better register allocation balances the chance that each register is pushed with a return address ID during execution, which increases the chance that the assigned register has enough room and needs no spilling. Thus, along each program execution path, the different return address IDs should be equally distributed to different registers.

To balance register allocation, Niffler utilizes the function call graph extracted from the source program to predict all possible execution paths. Niffler colors the call graph in a topological order which colors the two consecutive nodes in an execution path differently. Thus it maps color numbers to registers with modulo operations. All cycles in the call graph are eliminated before the topological ordering.

An indirect call site can reach multiple callee functions depending on the value of the function pointer. It is challenging to determine the exact register to place the return address ID at compile time if the possible callees are assigned different registers. Instead, Niffler assigns the same register to all address-taken functions.

As mentioned in Section 4.1, Niffler treats direct calls to externally defined functions as indirect calls. Similarly, Niffler assigns all functions that can be the callees of an external call site the same register. Thus the instrumentations are the same around an indirect call cite and an external call site.

By pinning all return addresses IDs for indirect and external call sites into the same register, Niffler can correctly handle application callbacks from shared libraries. The instrumentations around these two types of call sites are the same, as they are indirect call sites from the view of the shared library. Furthermore, a callback function is assigned the same register as it must be address-taken and passed to the shared library as a function pointer.

4.3 Register Spilling and Restoring

Before pushing another return address ID on a function invocation, regardless of being direct or indirect, Niffler instruments the program to check whether the register has room for it. If not, the program traps to kernel and saves registers to the kernel writable secure region. The kernel clears all registers and the application continues pushing return address IDs then. The secure region is reclaimed after the process exits.

Before a function returns, the instrumentation restores the registers if the register assigned to that function is empty. To reduce kernel trappings, Niffler maps the secure region read-only in user mode so that the registers can be restored by directly reading into the secure region.

On forking a process, the kernel unmaps all pages of the secure region inherited from the parent process in the child process and maps new ones. The secure region in the child process is initialized with the same content as its parent process. Similarly, Niffler creates a new secure region of spilled return address IDs for a new thread.

5 Function Pointer Cloaking

To prevent function pointers from being corrupted, Niffler allocates registers to cloak function pointers and forces the application to read from and write to the set of registers.

A static register allocation requires an accurate point-to analysis to determine which variable an instruction accesses. Considering a function pointer variable `fptr` assigned with register `%R`. On instrumenting an instruction that accesses `fptr`, we must hard-code the register name as `%R`. Instrumenting with a wrong register may crash the program.

Instead, unlike return address cloaking, Niffler dynamically allocates registers for function pointers. Niffler packs the address of the function pointer variable and its value into one register. For a function pointer variable `fptr`, located at `&fptr` and holding the address of function `func1`, the register will be filled with the pair of (`&fptr`, `func1`).

Niffler identifies all function pointer read and write operations and instruments them. For a function write operation `*fptr = func1`, the instrumented program scans all registers for a vacancy. If there is one, it saves the pair of (`&fptr`, `func1`) in that register. When there is not, the instrumented program traps to the kernel and spills the registers into the kernel writable secure region. It is remapped as read-only in user mode and shared between all threads in the same process.

Reading a function pointer requires a scan over all dirty registers. Intuitively, function pointer read operations are more common than write operations. Niffler instruments the application with a fast path of directly reading from the secure region. To determine whether the function pointer variable is being cloaked in some register at this moment, the instrumented application also reads the function pointer variable and compares the two read results. Note that Niffler does not eliminate the function pointer write operation in the application after instrumentation. The function pointer variable is updated anyway. If the two values are different, the instrumented program will fall back into a slow path and scan over all dirty registers. If no dirty register holds the function pointer, the instrumented program will treat it as a case of function pointer corruption.

In this design, if a function pointer is cloaked in a register, but the in-memory variable is unfortunately corrupted back to the previous value, the instrumented program will not detect this case, as the value read from the secure region matches the corruption result. To mitigate this issue, the kernel checks the dirty registers with the corresponding function pointer variables in memory on spilling. A mismatch will indicate that the function pointer has been corrupted.

Synchronization in Multi-Threaded Program. Registers are private to a thread. If a function pointer is updated by one thread, it is temporarily invisible to its peer threads until the dirty register is spilled. Niffler makes the program spill all of the registers that belong to the thread and cloak function pointers when the program releases a lock by calling the thread synchronization library functions. We expect the developers to explicitly invoke the synchronization functions if multiple threads race on function pointer variables.

6 Implementation

Niffler is implemented on x86 platform as extensions and modifications to LLVM 3.9.0. Niffler analyzes and instruments each application and its dependencies at intermediate representation (IR) level after each compilation unit is linked. Different compilation units can be analyzed and instrumented separately as Niffler supports dynamic linking.

Niffler also provides a character device driver, a type of kernel module, which works with unmodified Linux 4.8.11. All kernel-side implementations of Niffler are contained in the module without modifying any code of Linux kernel.

In the prototype, we pick MMX registers `%mm0-7`, a set of 64-bit registers provided by x86 single instruction multiple data (SIMD) instruction set, for return address cloaking. MMX registers are aliased to the x87 register stack which is arguably obsolete [29]. We disable the usage of the x87 register stack. We also reserve `%xmm15` to save spilled `%mm6`. Function pointer cloaking is supported by 128-bit Memory Protection Extensions registers.

6.1 Extracting Function Call Graph

When extracting the call graph, Niffler infers the possible callees for an indirect call site by matching the function signature. Though such inference overly approximates the set of possible callee functions, it does not impact the correctness of the instrumentation. Whether a pair of caller and callee functions are assigned the same register only affects the frequency of register spilling.

A function may act as the callee of both a direct call and an indirect call. However, the instrumentations are different between these two types of calls. For example, they push the return address IDs to different registers. Niffler duplicates any function that is possibly an indirect call target and modifies all address-taking sites to use this duplicate.

To handle the functions that can be the callees of external call sites, Niffler duplicates all functions exported by a shared library and replaces the internal direct invocations inside the shared library with a call to the duplicate.

6.2 Cloaking Return Address ID

Niffler reserves six MMX registers, `%mm0-5`, to hold return address IDs for direct calls. Another register `%mm6` is reserved for indirect and external function calls. The kernel maintains a stack in the secure memory region for the spilled MMX registers. The top of the stack is kept in `%mm7`.

Listing 2. Instrumentations before a Direct Call

```
1   movq %mm0, %rsi
2   shrq $60, %rsi
3   je  no_need_to_spill
4   ... # trap to kernel
5  no_need_to_spill:
6   movq %mm0, %rsi
7   salq $4, %rsi
8   orq %rdi, %rsi
9   movq %rsi, %mm0
```

Listing 2 shows an example of how Niffler instruments the program to dynamically determine when to spill. The number of bits of a return address ID is statically determined during compiling. In this example, the return address ID is encoded in 4 bits. Lines 2-3 decide whether there is enough room in `%mm0` for another 4-bit return address ID by checking whether the 4 leading bits of `%mm0` are zero. Niffler reserves the return address ID *zero* so that the 4 leading bits of zero cannot be any return address ID in use.

In the prototype, we fix the length of all return address IDs for indirect calls and external calls to 16 bits, which support at most $2^{16} - 1 = 65535$ different call sites. We use the 64-bit register `%mm6` to hold these fixed-length IDs. To reduce the frequency of `%mm6` spilling, Niffler instruments the application to copy `%mm6` to `%xmm15` once `%mm6` needs to spill. The application traps to the kernel if `%xmm15` is not empty. For applications with more than 65,535 indirect call sites, we can set the ID length to 32 bits and use a 128-bit XMM register. A second option is to reserve both `%mm5` and `%mm6` for non-direct calls and leave `%mm0-mm4` for direct calls.

Listing 3. Instrumentations Before a Return Instruction

```
1    movq %mm0, %rdi
2    testq %rdi, %rdi
3    jne no_need_to_restore
4    ... # restore MMXs in user mode
5   no_need_to_restore:
6    movq %mm0, %rdi
7    andq 0xf, %rdi
8    movq func1_tbl(,%rdi,8), %rdi
9    movq %rdi, (%rsp)
10   movq %mm0, %rdi
11   shrq 4, %rdi
12   movq %rdi, %mm0
```

Listing 3 shows the instrumentation before a return instruction. Similar to Listing 2, the return address ID is also encoded in 4 bits. Lines 1-3 check whether `%mm0` is zero. An empty `%mm0` indicates all registers must be restored from the secure region. The following instructions translate the return address ID, overwrite the return address on the stack and update `%mm0`.

Handling setjmp()/longjmp(). Niffler instruments the application to trap to the kernel on `setjmp()` and save the current state of all MMX registers and `%xmm15` in kernel writable memory. The kernel keeps track of all `setjmp()` records with a map structure, which is indexed by the record pointer accepted by `setjmp()`. On `longjmp()`, the registers are restored with the record pointer accepted by `longjmp()`.

6.3 Optimization With MPX

Niffler's function pointer protection is optimized with Intel Memory Protection Extensions. We use the radix tree of MPX as the secure region for spilled registers. Niffler maps the whole radix tree as read-only in user mode and remaps it as writable in kernel mode.

We list the instrumented instructions after a function pointer read operation in Listing 4. `%rdi` holds the address of function pointer variable and `%rsi` holds the content of function pointer variable. Line 1 clears `%bnd0`. `bndldx` in Line 2 loads `%bnd0` with two 8-byte values stored in the radix tree, which are set by the kernel on MPX register spilling. During spilling, the kernel sets the first and second 8 bytes of the 32-byte block with the same value `func1`, if the spilled register contains the pair (`&fptr`, `func1`). These two values are treated as the upper bound and the lower bound in MPX. If `%rsi` matches the pointer variable's content loaded from the radix tree, the bound checks in Lines 3-4 will succeed.

If there is another MPX register cloaking the pointer variable, either Line 3 or Line 4 will detect a mismatch. A `#BR` exception is triggered by the processor. The character device driver extends the #BR exception handler, spills all MPX registers to the radix tree and clears all MPX registers if it finds the cloaked function pointer variable in any MPX register. Otherwise, it kills the process.

Listing 4. Instrumentations to Verify a Function Pointer

```
bndmov zero, %bnd0
bndldx (%rdi), %bnd0
bndcn  (%rsi), %bnd0
bndcl  (%rsi), %bnd0
```

Listing 4 shows the fast path of function pointer verification. Note that though bndcn and bndcl compare the two operands, they bring no branch to the control flow. In comparison, accesses to the MPX registers form the slow path as x86 requires saving the 128-bit MPX registers to memory and reading the in-memory data. Furthermore, matching the MPX register value brings conditional branches.

Niffler reserves three MPX registers %bnd1-%bnd3 to cloak an updated function pointer variable. A free MPX register is found by scanning over all MPX registers. The application traps to the kernel when no MPX register is empty.

The radix tree does not require to be fully backed up by physical pages. The pages of the radix tree are not physically allocated until a first access. Additionally, function pointers variables are allocated only in a few memory regions like the stack and heap, which limits the range of memory containing function pointers. Even if some pages are unintentionally touched, the Linux kernel maps all empty read-only pages to the same physical page.

6.4 Identifying Function Pointer Access

Niffler identifies function pointer accesses in the LLVM intermediate representation by looking for load and store instructions with function type. In-register values like function arguments are not specially considered in our prototype as the attacker cannot directly corrupt any general purpose register value through memory corruption. In order to capture type information from source code without being optimized away, we identify function pointer accesses before running -O2 mode optimizations. We transform some indirect calls into direct ones if the destinations can be determined with an intra-procedural analysis, to eliminate unnecessary function pointer variable accesses.

Niffler instruments the libc function sigaction() to explicitly save the registered signal handler in a processor register. Programs may statically initialize function pointers in global variables, without any store instruction. Niffler collects all global variables and generates the corresponding store operations. For other implicit function pointer update like memcpy(), Niffler depends on the programmer to explicitly mark them and generates the store instructions. We expect pointer analysis [30], [31] would help to infer the type of copied memory data.

6.5 Instrumenting C++ Program

Virtual Function Table Pointers. C++ virtual function table (VTable) pointers are identified by their types. As Niffler keeps the 48-bit address and 64-bit value in MPX registers, the VTable pointers are treated similarly in the way of handling function pointers.

We make an optimization based on an observation that a C++ program may store a VTable pointer and read it just after a few instructions, which reflects that the program initializes an object and invokes one virtual function immediately. Niffler instruments the VTable pointer read operations by directly reading and checking the latest updated MPX register. Then it turns to other registers. This optimization avoids a #BR exception if there is a matched register.

Stack Unwinding During Exception Handling. Niffler instruments each function with stack unwinding operations to pop the return address IDs from the assigned register which holds the return address used by this function. Furthermore, Niffler also ensures the integrity of the return addresses on the stack which are used during exception handling by replacing the return address on the stack with the one translated from the in-register return address ID.

7 Evaluation

In this section, we analyze the design and implementation of Niffler to discuss its security guarantees and evaluate the performance overhead it incurs.

7.1 Security Analysis

We demonstrate how an application instrumented by Niffler defeats control data attacks. First of all, Niffler ensures the registers that cloak the control data can only be modified by the instrumented instructions. No memory corruption can forge a virtual address that targets a register. Niffler reserves the use of those registers so that the compiler will neither spill them into memory nor use them intentionally. Furthermore, as Niffler instruments the application to get control data like return addresses and function pointers from registers, the attacker has no chance to directly manipulate the control data and mislead the program to jump into the middle of an instruction to reinterpret the instructions. In addition, we assume the integrity of code pages so that it is impossible to create new instructions that modify the reserved registers.

Second, the return address IDs are hard-coded in instrumented instructions, which are determined during compilation. For indirect calls in shared libraries and direct calls to shared library functions, the added offset values are protected with read-only page permission. Then the return address IDs pushed to registers are immutable to memory corruptions. The function addresses used to update MPX registers are either initialized by taking a function's address or copied from a function pointer variable. The initial value is an operand of the instruction, which is hard-coded in code pages. Niffler forces a load from a function pointer variable to read from MPX registers or the secure region, which restricts the result to values that reside in the secure region.

Third, the spilled register values are either cached in XMM registers or write-protected by page permission. The attacker cannot corrupt the content of the secure region in user mode.

Lastly, the instrumented application distinguishes all return address IDs by interpreting them with different translation tables. Though all IDs for indirect call sites are translated with the same table, each of them is globally

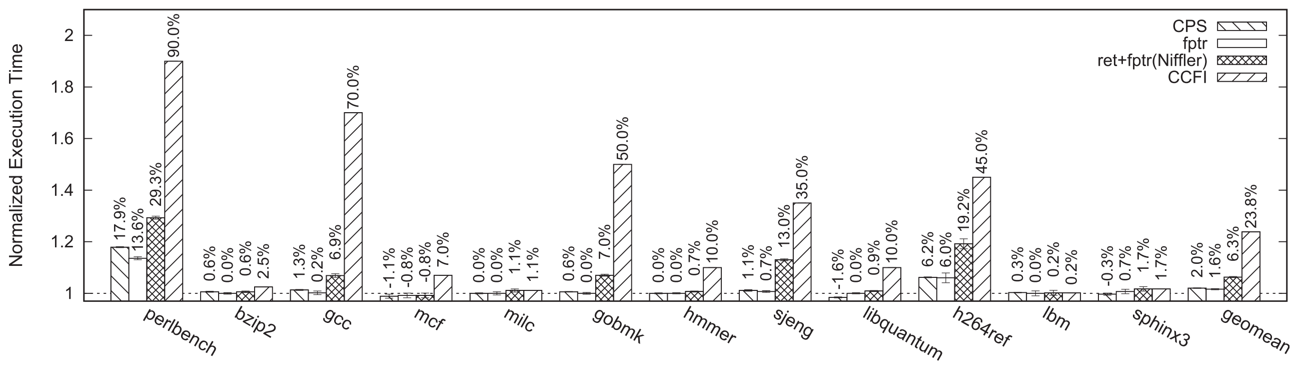


Fig. 2. SPECCPU 2006 benchmarks in C execution time overhead. The dashed horizonal line indicates the normalized execution time of baseline. The normalized execution time overhead relative to baseline is labelled above each bar. *CPS*: the overhead incurred by CPS as we measure. *fptr*: the overhead incurred by Niffler function pointer protection. *ret+fptr*: the total overhead incurred by Niffler return address and function pointer protection. *CCFI*: the overhead of CCFI as reported in the paper.

unique with the help of the different offset values among shared libraries.

Though the memory region holding spilled register is accessible in user and kernel mode, there is no risk of leaking the kernel information as no kernel internal data will be saved in this region. The address of the read-only memory region is in the user space as the region is allocated by `mmap()`. Niffler does not prevent any unintentional read to the read-only region as it does not rely on information hiding to ensure data isolation.

Case Study: Defending Memory Corruptions in Nginx. We use *Nginx* as an example to show how Niffler helps applications defeat memory corruptions. We use *gdb* to inject memory corruptions, which are tougher than real-world exploits as we are free to inject memory corruptions at any time and address. We randomly pick 15 places of code pointer read operations that are surely executed during the test. They include 5 direct function call returns, 5 indirect function call returns, and 5 indirect function calls. We set breakpoints before the *Nginx* worker process reaching the code pointer read operation. We then use `set` command to emulate a memory corruption. We change the 10 return address and the 5 function pointers to an invalid address.

After the execution continues, we find as expected that the worker process is not affected if the return address is corrupted. This is because the corrupted return addresses have been overridden by the correct ones in register. For the corrupted function pointers, we find Niffler always kills the victim worker process at the address of `bndcn` or `bndcl`, which indicates Niffler has identified the corrupted function pointer and prevented the corrupted value to be used.

Limitations. In the prototype, Niffler defends control data attacks. The attacker may corrupt the parameters used to update the radix tree to trick the instrumented code to write to arbitrary entry [32], [33]. Protecting parameters is an example of non-control data attacks [34], [35], [36], [37]. Neither CPS nor CCFI considers such data pointer corruption attacks while the enhanced CPI isolates the data pointers into a randomize-address memory region. Additionally, the prototype does not consider the spilled general purpose registers on the stack. It might be corrupted if the attacker reveals its spilled location.

As a possible mitigation, we can cloak these data in a way similar to function pointers and VTable pointers. It requires additional typed data to be marked and the function type information to be passed to the backend compiler. The amount of data and the frequency of access events have effects in terms of performance, as the pressure on registers may increase and more spilling events are needed.

Niffler compares the function pointer variable and the read-only copy from the radix-tree on reading the function pointer. It detects a mismatch if the function pointer variable is corrupted into a different value. The only case that bypasses this check is that the updated function pointer value cloaked in a register has not been spilled but the function pointer variable is changed to the previous value, which is the same as the in-radix-tree copy. As discussed in Section 5, Niffler mitigates this case by checking the in-register value and the in-memory variable during register spilling. And the corrupted value is no longer valid after register spilling.

Niffler requires all compilation units to be recompiled and instrumented. An uninstrumented binary file may contain unprotected return address usage and accesses to the reserved registers.

7.2 Performance

We evaluate the instrumented SPECCPU 2006 benchmarks and real-world applications. We use a machine with a quad-core Intel Core i5-6600 CPU running at 3.80 GHz, 8 GB memory and Samsung SSD 850. The kernel is Linux 4.8.11. All evaluations in this section are based on an unmodified Linux kernel where all the kernel-side implementations of Niffler are enabled by dynamically loading a kernel module.

We have compiled and instrumented musl libc [38] as Clang, the LLVM frontend, is not able to parse glibc [39]. We use the uninstrumented application as the baseline. The instrumented programs are dynamically linked to the instrumented musl libc, while the baseline programs are linked to the uninstrumented musl libc.

For all programs, the instrumented one and the baseline are compiled with the same optimization level, as defined in the Makefile provided in the application source code.

In the prototype, we disable the tail call optimization in the compiler backend as it makes a nested callee function use the caller's return address, conflicting with the instrumentations at intermediate representation level. We find it has insignificant performance impact.

SPECCPU 2006 Benchmarks in C. We recompile the 12 C programs in SPECCPU 2006 benchmark set. The results for

TABLE 2
The Frequency of Executing the Instrumented Operations

Benchmarks	Drt. Call	Id./Ex. Call	Read FP	Write FP	MMX Sp.	MPX Sp.	#BR	MMX Sp.%	MPX Sp.%
perlbench	65.8m	21.7m	18.4m	483.2k	2.3k	36.3k	88.5	0.002%	7.512%
h264ref	46.1m	51.4m	51.4m	834.0k	0	208.5k	41.9k	0	30.0%
sjeng	51.2m	16.0m	16.0m	23.1	45.9k	5.8	0	0.068%	–
gobmk	37.6m	616.5k	186.6k	182.3	112.5k	45.6	0	0.294%	–
gcc	40.3m	5.0m	2.0m	18.9k	23.5k	4.7k	139.3	0.052%	24.868%
bzip2	17.3m	4.6	0.73	0.70	0	0.175	0.12	0	17.1%
libquantum	3.6m	3.004	0	0.07	0	0.016	0	0	0

The unit of the numbers is number of events per second. *Suffix*-m*: million. Suffix*-k*: thousand.* Drt. Call*: direct calls.* Id./Ex. Call*: indirect/external calls.* Read FP*: reading function pointers.* Write FP*: writing function pointers.* MMX Sp.*: MMX register spilling.* MPX Sp.*: MPX register spilling.* #BR*: #BR exceptions triggered.* MMX Sp.% *= (MMX Sp.)/(Drt. Call + Id./Ex. Call): the percentage of function call events that trigger MMX spilling.* MPX Sp.% *= (#BR +MPX Sp.)/(Write Fp): the percentage of function pointer writes that eventually trap to kernel.*

workload *ref* are displayed in Fig. 2. The y-axis indicates the normalized execution time of the instrumented application relative to baseline. The geometric average overhead of the 12 benchmark programs is 6.3 percent.

We also recompile the benchmarks with CPS under the same optimization level. The evaluation results for CPS are shown in Fig. 2 as the left-most bar in each cluster. The geometric average overhead of CPS is 2.0 percent. We find CPS brings a smaller performance overhead than Niffler while its simple design sacrifices security guarantee, as CPS can be broken by information leak [13].

We fail to port CCFI to our evaluation environment as it is implemented on FreeBSD. Fig. 2 shows the CCFI overhead reported in the CCFI paper. We find Niffler's overhead is much smaller than CCFI [18] which reports an average overhead of 23 percent for C programs.

We further break down our implementation and evaluate the performance overhead for SPECCPU 2006 benchmarks with only function pointer cloaking enabled. The results are presented in Fig. 2 as the second bar in each cluster.

We try to understand the performance overhead of the benchmarks by studying the frequency of executing the instrumented operations. The results are shown in Table 2. The unit for all event counts is the number of events *per second*. We find that the benchmarks with the larger overhead (Row 2-6) have a higher frequency of function calls and function pointer accesses (Column 2-5) than the other two (Row 7-8). We calculate the frequency of register spilling against the register update events. The results in Column *MMX Sp.%* of Table 2 indicate an efficient utilization of MMX registers as the frequency of spilling registers is very low. For example, the 87.5 million invocations *per second* in perlbench only trap to the kernel for register spilling less than 3,000 times *per second*.

We further measure the benefit of copying `%mm6` to `%xmm15` in *perlbench*, the recursion-intensive benchmark in SPECCPU 2006. If the instrumented program spills all register once `%mm6` is full, the MMX spilling frequency significantly increases from 2.3 thousand to 1.6 million *per second*. The overall execution time overhead of *perlbench* increases from 29.3 to 34.6 percent.

To study the performance benefit of dynamically detecting register shortage rather than statically inserting spilling into the program, we have implemented the latter and evaluated on SPECCPU 2006 benchmarks. We find that most benchmarks perform much worse with static detection. For example, *perlbench* has 362 percent overhead and *gcc* has 208 percent overhead. 27.4 percent overhead for *gobmk* and *sjeng* has 16.6 percent overhead. The outlier *h264ref* performs better with about 10 percent overhead. We think the improvement comes from the fact that h264ref does not spill MMX registers in both implementations, while the static detection implementation takes fewer instructions, which appends to a register without any check.

To study the performance benefit of ID encoding, we evaluate gobmk, which is a SPECCPU 2006 benchmark whose overhead mainly comes from return address cloaking, to bring some insights. We implement a naive encoding that assigns ID for all direct jumps as a single set. The performance overhead increases to 20.5 percent, comparing to 7.0 percent in Niffler, as it needs 13 bits for each encoded ID and the registers are easier to get full.

To see the benefit of the balanced register allocation, we compare Niffler's algorithm with a simple round-robin allocation algorithm, which assigns IDs in the order of iterating over all functions. The result of the simple round-robin is close to Niffler's result. However, it will be different if LLVM keeps a different order of all functions. We made another evaluation where the probability that three consecutive nodes in a callgraph share the same color is less than 50 percent. The performance overhead increases to 8.86 percent, which is higher than Niffler's result (7.0 percent).

SPECCPU 2006 Benchmarks in C++. To compare the performance impact in C++ programs, we recompile all C++ programs in SPECCPU 2006 benchmark set. We also recompile and instrument all shared libraries that the C++ programs depend, namely *libc++*, *libunwind* and *libc++abi*, which are LLVM-implemented C++ standard and runtime libraries. All programs and shared libraries are dynamically linked to instrumented musl libc. We fail to build *dealII* due to a missing class implementation. The evaluation results of *ref* workload are shown in Fig. 3 as the cluster of bars on the left-hand side. Niffler incurs a geometric average overhead of 28.2 percent on the six benchmarks. We break down the overhead incurred by different parts of the instrumentation. Shadow stack unwinding incurs negligible overhead among the six benchmarks that we evaluate.

We also list the two benchmarks that CCFI evaluated, namely *astar* and *xalancbmk* in Fig. 3, as the cluster of two bars on the right-hand side of the figure. These results are

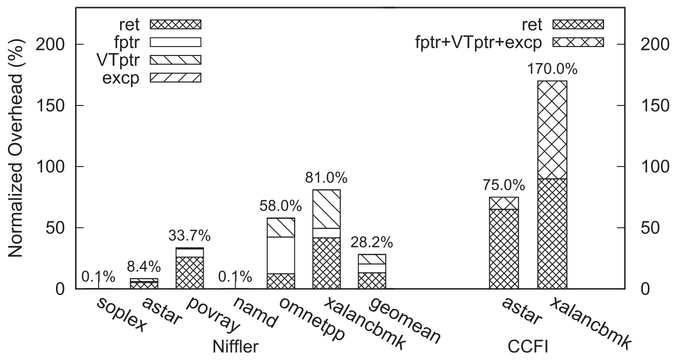


Fig. 3. SPECCPU 2006 benchmarks in C++ execution time overhead. *ret*, *fptr*, *VTptr*, *excp*: the overhead of Niffler in return address, function pointer, VTable pointer cloaking and shadow stack unwinding during exception handling. *fptr+VTptr+excp*: the total overhead of these parts reported in CCFI paper. CCFI doesn't further breakdown them.

TABLE 4
Cycles and Instructions Reduced with MPX

Criterion	No-MPX	MPX
Total Cycles	12.4	7.5
Number of Instructions	17	4

TABLE 5
Performance of Servers in Requests/Second

	Nginx		Redis	
	4KB	80MB	GET	MSET
Baseline	34655 ± 14.5	74.99 ± 0.6	296399 ± 3438	148309 ± 5003
Niffler	33909 ± 37.5	75.00 ± 0.05	299601 ± 6027	136671 ± 4631
Overhead	2.2%	−0.01%	−1.1%	7.9%

reported in the CCFI paper. For these two benchmarks, Niffler incurs much lower overhead than CCFI does.

Microbenchmarks. We have evaluated the number of cycles used for executing each piece of the instrumented code. We put the inserted assembly into a tight loop and capture the average CPU cycles among 1 million iterations.

From the results in Table 3, we find Niffler instruments the applications with lightweight operations, most of which are around 10 cycles and more efficient than the encryption and decryption operations in CCFI [18]. Though spilling MMX registers requires a context switch between the user mode application and the kernel, we find the evaluation platform that we use, i.e., Intel Core i5-6600 CPU, performs well on executing the system call, with merely 155 cycles to trap into and exit from the kernel. The most time-consuming operation is handling a `#BR` exception. However, as the analysis results in Table 2 show, the `#BR` exceptions are infrequent events.

We study the performance benefit we gain from Intel MPX. We implement the function table lookup without using MPX's `bndldx` and compare it with MPX-based implementation. Both of the implementations are executed in a tight loop 10 million times. The evaluation results listed in Table 4 show that MPX-based implementation uses fewer CPU cycles and is much simpler.

Real-world Applications. We recompile two commonly used server programs, the web server *Nginx* and the in-memory database *Redis*. *Nginx* is evaluated with the ApacheBench benchmarking tool, which sends 100 thousand requests to download small files in 4 KB and large files in 80 MB. To evaluate *Redis*, we use its own benchmarking tool [40]. The two servers are on the same machine as the benchmark tool. We start up the two servers with their default configuration files. The evaluation results are displayed in Table 5. For transferring small web pages in 4 KB, the instrumented *Nginx* serves 2.2 percent fewer requests in one second on average than the uninstrumented one. For larger file transfer requests, the instrumented *Nginx* performs essentially the same with the uninstrumented one. The *Redis* benchmark sends 17 different types of request to the server from 50 clients. Each type includes 100,000 requests. The reported throughput fluctuates among different evaluations. The performance difference is between −1.4 and 7.9 percent. We list the medium throughput for two types of requests in Table 5. *GET* retrieves the value of one key and *MSET* stores 10 keys in each request. We find the instrumented *Redis* server has a comparable throughput to the uninstrumented one.

We evaluate three commonly used Linux utility programs *dd*, *tar* and *make*. We use *dd* to copy two files of size 90 MB and 630 MB respectively, *tar* to pack 6,082 files from three directories in total of 785 MB, and *make* to instruct the compiling toolchain to build two non-trivial projects, *Nginx* and musl libc. *dd* and *tar* are single-threaded programs, while *make* executes in 4 jobs and spawns child processes from time to time. The performance difference between the instrumented and baseline presented in Table 6 shows Niffler incurs insignificant overhead, though *dd*'s performance fluctuates.

8 Related Work

Data Protection with Registers. StackGhost [23] exploits the register window feature on SPARC architecture and is

TABLE 3
Time Cost of Instrument Operations

Operation	Num. of Instr.	Cycles
Direct Call	7	6.3
Indirect Call	15	11.5
Return	8	8.8
Read Function Pointer	4	7.5
Write Function Pointer	10	13.3
Spill+Restore MMX Registers	8*	155
Handle #BR Exception	4*	723

*Values marked with * only consider instructions in user mode.*

TABLE 6
Performance of Instrumented Linux Utilities

	make		dd		tar
	Nginx	musl	90 MB	630 MB	Files in 785 MB
Baseline	3.07s	11.87s	0.18s	2.23s	1.806s
Niffler	3.10s	11.98s	0.17s	2.13s	1.812s
Overhead	1.2%	0.95%	−5.6%	−4.3%	0.3%

restricted that it traps to the kernel on every eight consecurive function calls. In comparison, Niffler works on a platform without the register window feature. As registers can easily become full of code pointers, Niffler instruments the applicatiom to explicitly trap to kernel to save registers and provides several mechanisms to reduce the frequency of kernel trapping.

PointGuard [24] encrypts the spilled registers in memory. In order to reduce overhead, it uses XOR to encrypt memory data where the secret value can be inferred by comparing the raw code pointer value and encrypted value. In comparison, Niffler sets read-only memory permission for the spilled register data to exclude any chance of memory corruption in user space.

TRESOR [41] builds a cryptographic key storage to achieve out-of-RAM encryption by pinning the encryption keys in x86 debug registers. In comparison, Niffler addresses a different problem and utilizes a different set of processor registers.

Though all them of them have leveraged processor registers to protect security data to some degrees, they use registers in straightforward ways which are either incomplete or insecure on a commercial platform like x86. Niffler provides a life-time protection to control data and makes a novel reuse of MPX for efficient lookup.

Control Data Isolation. Like Niffler, CPS and CCFI force the application to take indirect control transfers based on isolated control data. In x86-64 that has no segmentation support, CPS hides the control data in a region with a randomized base address. It does not prevent the attacker from reading the in-memory pointers to the region, which breaks the information hiding. Though the enhanced CPI additionally isolates these pointers, recent studies have provided multiple exploits to leak the region's address anyway [12], [13]. In contrast, Niffler is secure under the threat of information leak. Neither leaking which registers are used nor the location of the secure region holding spilled registers makes the isolated data corruptable.

CPS is possible to be implement in other ways like Software Fault Isolation (SFI) [42] which is enhanced in security guarantee but loses its simplicity in design. SFI instruments all memory writes with a bound check to ensure only certain writes can access the secure region. Additionally, though there is an optimization suggesting to use simple instrumentation of erasing the most-significant bit and to isolate code pointers in the upper half of virtual memory, it significantly changes the process virtual memory layout which requires complex kernel modification. Furthermore, as SFI is broken when any memory write misses a bound check, the SFI-based CPS is hard to be implemented flawlessly as LLVM cannot instrument memory writes in assembly.

In comparison, the complexity brought by SFI can be eliminated by Niffler with the help of architectural features available on commodity hardware. In addition, Niffler's kernel support can be simply implemented into a kernel module without modifying any kernel code.

CCFI's encrypted control data suffers from replay attacks, which replace encrypted data with another piece of encrypted data. Though CCFI mitigates this vulnerability by encrypting the control data address, it does not reject the encrypted data located at the same address. In contrast, Niffler simply rejects the direct writes. Even the optimization on function pointer restoring allows one recent value of the variable content, it accepts no other corrupted value. Furthermore, Niffler incurs much lower performance overhead than CCFI.

Intel has released a new technology named Control-flow Integrity Technology (CET) [43] to prevent ROP by keeping the return addresses on a shadow stack. The shadow stack is protected by a new set of page permissions, indicated by some newly added bits in page table entries. CET also introduces a mark instruction to specify valid indirect control transfer target. As the hardware has not been released to the market, it cannot be adopted at the moment.

Shadow Stack. Shadow stacks vary on the mechanisms to enforce the memory isolation, such as placing the shadow stack into a separate segment [11], memory area with randomized base address [9], [17], [44], and memory area protected by guard pages [20]. However, segmentation support is limited in x86_64. Randomization-based techniques are vulnerable to information leak [12], [13].

Software Enforced Control Flow Integrity. Control Flow Integrity can be implemented either at binary level or at source code level. Binary instrumentation ensures CFI in legacy software in case that source code is unavailable [45], [46], [47]. Program analysis at binary level is hard as the compiler has removed most of the type information and control flow structure.

Compiling the source code with CFI enforcement has the advantage of the abundant information collected from the source code. IFCC [48] provides only forward-edge restriction. πCFI [49] incrementally expand CFI checks and valid indirect jump targets by patching the program on the fly. Its protection falls back to a relaxed CFI policy if all edges from the statically defined CFG graph are activated.

Hardware Enforced Control Flow Integrity. Researchers have proposed hardware extensions to restrict program execution paths [50] and detect code reuse attacks [51]. In comparison, Niffler is designed to work on commodity hardware. Researchers have also explored the processor features in a special way, such as Branch Tracing Store (BTS) [52], Indirect Branch Tracing (IBT) [53], [54], [55] and Intel Processor Trace (IPT) [56], [57], [58], to dynamically check the execution trace with pre-defined control flow policies. Hardware support does not come at no cost. BTS and IBT have limited storage space for trace while IPT-based systems rely on additional core to parse the IPT packets parallelly. Most systems can only enforce a relaxed CFI policy, restricted by the offline static analysis.

9 Conclusion

Niffler is a novel approach to efficiently enforce control data isolation in applications and defeat memory corruptions on control data. Niffler cloaks control data into registers and spills the registers securely. With the help of techniques to increase register utilization, the performance overhead is low on applications with moderate number of control transfers.

Acknowledgments

This work is supported in part by the National Key Research & Development Program (No. 2016YFB1000104).

REFERENCES

[1] L. Szekeres, M. Payer, T. Wei, and D. Song, “SoK: Eternal war in memory,” in *Proc. IEEE Symp. Security Privacy*, 2013, pp. 48–62. [Online]. Available: http://dx.doi.org/10.1109/SP.2013.13

[2] Data execution prevention, 2018. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/aa366553(v=vs.85).aspx

[3] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, “Hacking blind,” in *Proc. IEEE Symp. Security Privacy*, 2014, pp. 227–242. [Online]. Available: http://dx.doi.org/10.1109/SP.2014.22

[4] R. Hund, T. Holz, and F. C. Freiling, “Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms,” in *Proc. 18th Conf. USENIX Security Symp.*, 2009, pp. 383–398. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855768.1855792

[5] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, “Jump-oriented programming: A new class of code-reuse attack,” in *Proc. 6th ACM Symp. Inf. Comput. Commun. Security*, 2011, pp. 30–40.

[6] 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. Security*, 2007, pp. 552–561. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315313

[7] J. Lee, *et al.*, “Hacking in darkness: Return-oriented programming against secure enclaves,” in *Proc. 26th USENIX Security Symp.*, Aug. 2017, pp. 523–539. [Online]. Available: https://www.microsoft.com/en-us/research/publication/hacking-darkness-return-oriented-programming-secure-enclaves/

[8] PaX address space layout randomization (ASLR), 2004. [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt

[9] S. Bhatkar, R. Sekar, and D. C. DuVarney, “Efficient techniques for comprehensive protection from memory error exploits,” in *Proc. 14th Conf. USENIX Security Symp.*, 2005, pp. 17–17. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251398.1251415

[10] D. Williams-King, *et al.*, “Shuffler: Fast and deployable continuous code re-randomization,” in *Proc. 12th USENIX Symp. Operating Syst. Design Implementation*, 2016, pp. 367–382. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/williams-king

[11] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Control-flow integrity,” in *Proc. 12th ACM Conf. Comput. Commun. Security*, 2005, pp. 340–353.

[12] Y. Jang, S. Lee, and T. Kim, “Breaking kernel address space layout randomization with intel TSX,” in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2016, pp. 380–392. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978321

[13] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, “Missing the point(er): On the effectiveness of code pointer integrity,” in *Proc. IEEE Symp. Security Privacy*, 2015, pp. 781–796. [Online]. Available: http://dx.doi.org/10.1109/SP.2015.53

[14] R. Hund, C. Willems, and T. Holz, “Practical timing side channel attacks against kernel space ASLR,” in *Proc. IEEE Symp. Security Privacy*, 2013, pp. 191–205. [Online]. Available: http://dx.doi.org/10.1109/SP.2013.23

[15] M. Lipp, *et al.*, “Meltdown: Reading kernel memory from user space,” *27th USENIX Secur. Symp.*, 2018.

[16] P. Kocher, *et al.*, “Spectre attacks: Exploiting speculative execution,” *40th IEEE Symp. Secur. Privacy*, 2019.

[17] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, “Code-pointer integrity,” in *Proc. 11th Symp. Operating Syst. Design Implementation*, 2014, vol. 14, pp. 147–163.

[18] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, “CCFI: Cryptographically enforced control flow integrity,” in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Security*, 2015, pp. 941–951. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813676

[19] K. Akdemir, *et al.*, “Breakthrough AES performance with intel AES new instructions,” *White Paper, June*, p. 11, 2010.

[20] T. H. Dang, P. Maniatis, and D. Wagner, “The performance cost of shadow stacks and stack canaries,” in *Proc. 10th ACM Symp. Inf. Comput. Commun. Security*, 2015, pp. 555–566. [Online]. Available: http://doi.acm.org/10.1145/2714576.2714635

[21] Z. Wang and X. Jiang, “HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity,” in *Proc. IEEE Symp. Security Privacy*, 2010, pp. 380–395. [Online]. Available: http://dx.doi.org/10.1109/SP.2010.30

[22] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, “XFI: Software guards for system address spaces,” in *Proc. 7th Symp. Operating Syst. Design Implementation*, 2006, pp. 75–88. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298463

[23] M. Frantzen and M. Shuey, “StackGhost: Hardware facilitated stack protection,” in *Proc. 10th Conf. USENIX Security Symp.*, 2001, Art. no. 5. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251327.1251332

[24] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, “PointGuardtm: Protecting pointers from buffer overflow vulnerabilities,” in *Proc. 12th Conf. USENIX Security Symp.*, 2003, pp. 7–7. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251353.1251360

[25] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos, “Memory errors: The past, the present, and the future,” in *Proc. 15th Int. Conf. Res. Attacks Intrusions Defenses*, 2012, pp. 86–106.

[26] M. Castro, M. Costa, and T. Harris, “Securing software by enforcing data-flow integrity,” in *Proc. 7th USENIX Symp. Operating Syst. Design Implementation*, 2006, pp. 11–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267308.1267319

[27] Morris worm, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Morris_worm

[28] 400k servers may be at risk of serious code-execution attacks. patch now, 2018. [Online]. Available: https://arstechnica.com/information-technology/2018/03/code-execution-flaw-in-exim-imperils-400k-machines-have-you-patched/

[29] A. Fog, “Stop the instruction set war,” 2009. [Online]. Available: http://www.agner.org/optimize/blog/read.php?i=25

[30] B. Hardekopf and C. Lin, “Flow-sensitive pointer analysis for millions of lines of code,” in *Proc. 9th Annu. IEEE/ACM Int. Symp. Code Generation Optim.*, 2011, pp. 289–298. [Online]. Available: http://dl.acm.org/citation.cfm?id=2190025.2190075

[31] B. Hardekopf and C. Lin, “The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code,” in *Proc. 28th ACM SIGPLAN Conf. Program. Lang. Design Implementation*, 2007, pp. 290–299. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250767

[32] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, “Enforcing kernel security invariants with data flow integrity,” in *Proc. Netw. Distrib. Syst. Security Symp.*, San Diego, CA, Feb. 2016.

[33] I. Evans, *et al.*, “Control jujutsu: On the weaknesses of fine-grained control flow integrity,” in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Security*, 2015, pp. 901–913. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813646

[34] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, “Non-control-data attacks are realistic threats,” in *Proc. 14th Conf. USENIX Security Symp.*, 2005, pp. 12–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251398.1251410

[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. Security Privacy*, May 2016, pp. 969–986.

[36] S. Vogl, *et al.*, “Dynamic hooks: Hiding control flow changes within non-control data,” in *Proc. 23rd Conf. USENIX Security Symp.*, 2014, pp. 813–328. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/vogl

[37] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, “Automatic generation of data-oriented exploits,” in *Proc. 24th USENIX Security Symp.*, 2015, pp. 177–192. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/hu

[38] musl libc, 2016. [Online]. Available: https://www.musl-libc.org/

[39] W. Arthur, B. Mehne, R. Das, and T. Austin, “Getting in control of your control flow with control-data isolation,” in *Proc. 13th Annu. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2015, pp. 79–90. [Online]. Available: http://dl.acm.org/citation.cfm?id=2738600.2738611

[40] “How fast is redis?” 2018. [Online]. Available: https://redis.io/topics/benchmarks

[41] T. Müller, F. C. Freiling, and A. Dewald, “TRESOR runs encryption securely outside RAM,” in *Proc. 20th USENIX Conf. Security*, 2011, pp. 17–17. [Online]. Available: http://dl.acm.org/citation.cfm?id=2028067.2028084

[42] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, “Poster: Getting the point(er): On the feasibility of attacks on code-pointer integrity,” 2015. [Online]. Available: http://dslab.epfl.ch/pubs/cpi-getting-the-pointer.pdf

[43] "Intel releases new technology specifications to protect against ROP attacks," 2016. [Online]. Available: https://software.intel.com/en-us/blogs/2016/06/09/intel-release-new-technology-specifications-protect-rop-attacks

[44] "Clang 7 documentation," 2018. [Online]. Available: http://clang.llvm.org/docs/SafeStack.html

[45] C. Zhang, *et al.*, "Practical control flow integrity and randomization for binary executables," in *Proc. IEEE Symp. Security Privacy*, 2013, pp. 559–573. [Online]. Available: http://dx.doi.org/10.1109/SP.2013.44

[46] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *Proc. 22nd USENIX Conf. Security*, 2013, pp. 337–352. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534796

[47] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-free: Defeating return-oriented programming through gadget-less binaries," in *Proc. 26th Annu. Comput. Security Appl. Conf.*, 2010, pp. 49–58. [Online]. Available: http://doi.acm.org/10.1145/1920261.1920269

[48] C. Tice, *et al.*, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *Proc. 23rd USENIX Conf. Security Symp.*, 2014, vol. 26, pp. 27–40.

[49] B. Niu and G. Tan, "Per-input control-flow integrity," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Security*, 2015, pp. 914–926. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813644

[50] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Branch regulation: Low-overhead protection from code reuse attacks," in *Proc. 39th Annu. Int. Symp. Comput. Architecture*, 2012, pp. 94–105. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337159.2337171

[51] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu-Ghazaleh, "SCRAP: Architecture for signature-based protection from code reuse attacks," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Archit.*, Feb. 2013, pp. 258–269.

[52] Y. Xia, Y. Liu, H. Chen, and B. Zang, "CFIMon: Detecting violation of control flow integrity using performance counters," in *Proc. 42nd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2012, pp. 1–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=2354410.2355130

[53] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP exploit mitigation using indirect branch tracing," in *Proc. 22nd USENIX Conf. Security*, 2013, pp. 447–462. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534805

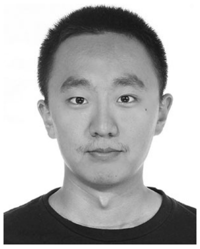
[54] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, H. DENG, *et al.*, "ROPecker: A generic and practical approach for defending against ROP attack," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2014, pp. 1–14.

[55] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive CFI," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Security*, 2015, pp. 927–940.

[56] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan, "Transparent and efficient CFI enforcement with intel processor trace," in *Proc. 23rd IEEE Symp. High Perform. Comput. Archit.*, 2017, pp. 529–540.

[57] X. Ge, W. Cui, and T. Jaeger, "GRIFFIN: Guarding control flows using intel processor trace," in *Proc. 22nd Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2017, pp. 585–598.

[58] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient protection of path-sensitive control security," in *Proc. 26th USENIX Security Symp.*, 2017, pp. 131–148. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ding



Xiayang Wang received the BS degree in software engineering from Fudan University, China, in 2014. He is now working toward the PhD degree with the School of Software, Shanghai Jiao Tong University. His research interests include program analysis and software security.

Fuqian Huang received the BS degree in software engineering from Shanghai Jiao Tong University, China, in 2018. He is now working toward the master's degree with the School of Software, Shanghai Jiao Tong University. His research interests include program analysis and software security.

Haibo Chen received the PhD degree in computer science from Fudan University, in 2009. He is currently a tenured full professor at the School of Software, Shanghai Jiao Tong University. His research interests include operating systems and parallel and distributed systems. He is a senior member of the IEEE.