Address-Oblivious Code Reuse: On the Effectiveness of Leakage-Resilient Diversity

Robert Rudd,* Richard Skowyra,* David Bigelow,* Veer Dedhia,* Thomas Hobson,*
Stephen Crane,† Christopher Liebchen,‡ Per Larsen,§ Lucas Davi,¶
Michael Franz,§ Ahmad-Reza Sadeghi,‡ Hamed Okhravi*
*MIT Lincoln Laboratory. {firstname.lastname,dbigelow}@ll.mit.edu
†Immunant, Inc. sjc@immunant.com
‡CYSEC/TU Darmstadt.{christopher.liebchen,ahmad.sadeghi}@trust.tu-darmstadt.de
§University of California, Irvine. {perl,franz}@uci.edu
¶University of Duisburg-Essen. lucas.davi@uni-due.de

Abstract*—Memory corruption vulnerabilities not only allow modification of control data and injection of malicious payloads; they also allow adversaries to reconnoiter a diversified program, customize a payload, and ultimately bypass code randomization defenses. In response, researchers have proposed and built various leakage-resilient defenses against code reuse. Leakage-resilient defenses use memory protection techniques to prevent adversaries from directly reading code as well as pointer indirection or encryption techniques to decouple code pointers from the randomized code layout, avoiding indirect leakage. In this paper, we show that although current code pointer protections do prevent leakage per se, they are fundamentally unable to stop code reuse. Specifically, we demonstrate a new class of attacks we call *address-oblivious code reuse* that bypasses state-of-the-art leakage-resilience techniques by profiling and reusing protected code pointers, without leaking the code layout. We show that an attacker can accurately identify protected code pointers of interest and mount code-reuse attacks at the abstraction level of pointers without requiring any knowledge of code addresses. We analyze the prevalence of opportunities for such attacks in popular code bases and build three real-world exploits against Nginx and Apache to demonstrate their practicality. We analyze recently proposed leakage resilient defenses and show that they are vulnerable to address oblivious code reuse. Our findings indicate that because of the prevalence of code pointers in realistic programs and the fundamental need to expose them to "read" operations (even indirectly), diversity defenses face a fundamental design challenge in mitigating such attacks.

DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.

This material is based upon work supported by the Department of Defense under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of Defense.

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA
Copyright 2017 Internet Society, ISBN 1-1891562-46-0
http://dx.doi.org/10.14722/ndss.2017.23477

I. Introduction

Memory corruption has been a primary attack vector against computer systems for decades [2]. Memory corruption attacks range from conventional stack smashing techniques [44] to more sophisticated code-reuse attacks (CRAs) such as return-oriented programming (ROP) [50], which emerged in response to the widespread adoption of defenses such as W⊕X (Write⊕eXecute). Part of the appeal of memory corruption to attackers is the ability to execute arbitrary code on a remote target system after hijacking the control flow. Despite numerous advances, comprehensively protecting native code written in C/C++ from ROP and other CRAs remains an open challenge.

Code-reuse defenses are either based on enforcement [1, 33, 41] or randomization [7, 23, 34, 37]. In this paper, we focus on state-of-the-art code randomization techniques that provide resilience against information leakage attacks and have shown to be both efficient and scalable to large codebases. We call such techniques *leakage-resilient defenses*.

Preventing all types of information leakage is extremely challenging. Direct leakage of memory content (a.k.a., memory disclosure) [53, 56], indirect leakage of addresses from the stack or heap [17], and side-channels [5, 9, 42, 48] are different forms of information leakage that have been used successfully to bypass recent code randomization defenses [17, 18]. Due to the prevalence and threat of such information leakage attacks, recent defenses have been based on a threat model that assumes the attacker can read and write arbitrary memory if allowed by the page permissions [7, 11, 15, 16].

Execute-only code memory (a.k.a. execute-no-read, XnR, or X-only) is used by many leakage-resilient defenses [3, 11, 15, 22] to shield randomized code layouts from direct leakage. Some defenses [3, 58, 61] relax ideal X-only code permissions to handle legacy binaries that may embed readable data in code, but these defenses have been shown to be lacking [54, 61]. The most powerful leakage-resilient defenses also prevent indirect leakage by hiding code pointer destinations, e.g., using pointer encryption [38] or an indirection layer [15].

Goals and Contributions.

In this paper, we will consider a *state-of-the-art leakage-resilient defense* that combines the strengths of all concrete, leakage-resilient defenses [3, 11, 15, 16, 22, 38] proposed to date. We will not consider *leakage-tolerant* defenses which seek to obsolete leaked information via runtime re-randomization [7, 23, 37]. We use Readactor–one of the most comprehensive implementations of a leakage-resilient code randomization defense–to demonstrate a new class of CRAs, which we call Address-Oblivious Code Reuse (AOCR), that can generically bypass leakage-resilient defenses without knowledge of the code layout. The intuition behind AOCR is that execute-only permissions apply just to code, not code pointers (*e.g.*, function pointers and return addresses). Code pointers/identifiers must be readable for programs to function correctly. Various execute-only defenses use indirection or encryption to protect these code pointers, but these alternative code pointer representations remain exploitable by adversaries.

Specifically, we demonstrate that an attacker can *profile* this layer of indirection in code pointers by observing the state of the protected program remotely, and extract these indirect code pointers. We then show that by only reusing these indirect code pointers, an attacker can achieve malicious behavior without explicitly requiring read access to the diversified code. We call our attack Address-Oblivious Code Reuse because its strength lies in the fact that it does not need to leak or otherwise learn the address of code snippets in order to successfully exploit them. Rather, by stitching together their indirect code pointers, the attacker can successfully execute code snippets while remaining oblivious to their randomized (hidden) addresses. Unlike COOP attacks [47], our AOCR attack does not rely on the layout of vtables, the allocations, or the use of registers. In a sense, AOCR can be thought of as position-independent form of CRA.

To accurately profile indirect code pointers in a running process remotely, we devise a new attack technique that we call Malicious Thread Blocking (MTB). To chain indirect code pointers, we show a new exploitation technique against imperative programming languages we call Malicious Loop Redirection (MLR).

Using these techniques, we build two real-world AOCR exploits against Nginx and one against Apache, that each hijack the control flow and execute arbitrary code. Unlike recent attacks on some leakage-resilient techniques that focus on particular technique-specific or implementation weaknesses [39], these attacks generically bypass leakage-resilient techniques, and are not limited to Readactor, which we solely use for demonstration and evaluation purposes. We discuss the generality of these attacks against other recent defenses and show that many of them are also vulnerable to AOCR in Section VII.

In contrast to previous attacks on the implementation of leakage resilience [39], our AOCR attack can bypass ideal implementations and comprehensive applications of leakage-resilience techniques. However, we also show that enforcing ideal and comprehensive execute-only defenses is surprisingly difficult. We call these caveats *implementation challenges* and discuss two practical attack vectors in Linux systems. The first vector maliciously redirects Direct Memory Access (DMA) operations that do not abide by page permissions. Unlike related work in this domain that focuses on abusing DMA through malicious hardware devices, we show that an attacker can trick the system to issue malicious DMA requests on its behalf using software-only attacks. The second vector uses Linux's `proc` filesystem to directly leak memory content even in the presence of defenses such as GRSecurity [55]. Both these vectors can be used to maliciously leak actual non-readable code pages, after which conventional ROP attacks become straightforward. In other words, if these vectors are not blocked by the defense, the attacker can use conventional ROP attacks instead of having to resort to AOCR.

In summary, our contributions are as follows:

- We present AOCR, a new class of CRAs that generically bypass *state-of-the-art leakage-resilient* defenses by reusing indirect code pointers. Unlike existing attacks, AOCR does not rely on the layout of vtables or the allocation and use of registers which renders COOP-centric defenses ineffective [16, 59].
- We demonstrate that code-reuse attacks can be constructed out of protected code pointers without direct knowledge of the code layout. We do so by building three AOCR exploits targeting Nginx and Apache.
- We present two techniques to accurately profile the indirection layer (Malicious Thread Blocking) and chain (Malicious Loop Redirection) AOCR gadgets that make our attacks highly practical.
- We discuss two main implementation challenges to achieve ideal leakage resilience in modern operating systems that further demonstrate the difficulty of effectively and universally enforcing memory permissions.

II. Threat Model

Our threat model assumes that a remote attacker uses a memory corruption vulnerability to access arbitrary memory and achieve remote code execution on the victim machine. We assume W$\oplus$X is deployed to prevent code injection and modification. Moreover, we assume that the software executing on the target system is protected by a state-of-the-art leakage-resilient randomization-based defense capable of stopping conventional [50] and just-in-time [53] CRAs. In particular, we assume that the target system:

1) maps code pages with execute-only permissions to prevent direct leakage [3, 15, 22];
2) hides, encrypts, or obfuscates code pointers to prevent indirect leakage [11, 15, 38];
3) randomizes the code layout at any granularity up to (and including) individual instructions [27, 45];
4) randomizes the entries of function tables [16] rendering COOP [47] and return-into-`libc` attacks cumbersome.

We assume that dynamically generated code is protected in the same way as code compiled ahead of time since the alternative is insecure as mentioned by Crane, *et al.* [15]. We do not consider side-channel attacks arising from memory sharing and deduplication between processes and virtual machines or attacks exploiting weaknesses in the underlying hardware. Our threat model is consistent with related work on leakage-resilient randomization-based defenses against code reuse.

While strong enforcement-based and randomization-based defenses in the literature have assumed that the adversary can read and write arbitrary memory, we demonstrate that practical attacks can in fact be mounted by a less powerful adversary.

III. Address-Oblivious Code Reuse (AOCR) Attack

Current state-of-the-art randomization-based defenses [3, 4, 11, 15, 16, 22, 38] aim to prevent CRAs by limiting an attacker's ability to disclose the code layout, either by leaking the code itself or by leaking code pointers. As noted in Section II, the adversary is assumed to have arbitrary read and write capabilities. Two primary techniques are employed to stop these adversaries:

- *Execute-only permissions* prevent read accesses to code pages (existing W⊕X policies already prevent writes to code pages). Thus, any attempts by an attacker to directly disclose the locations and contents of code pages will lead to a segmentation violation. Execute-only permissions are either implemented in software using page fault handlers [3] or with hardware-assisted paging (*e.g.*, Extended Page Tables) [15, 16, 29].
- *Code pointer indirection and encryption* seeks to prevent indirect memory disclosure by decoupling code pointers from the code layout. Indirect memory disclosure happens when an attacker learns about code locations from the code pointers temporarily stored on stack or heap. Some approaches alter the pointer representation using fast XOR encryption [11, 14, 38] to prevent indirect leakage. Others use indirection mechanisms [4, 15]. For instance, Readactor [15] replaces all observable code pointers with pointers to trampolines. A forward trampoline is simply a direct jump to a function stored in execute-only memory. Because the location of the forward trampoline and the function it jumps to are randomized independently, attackers cannot infer the function layout by observing the trampoline layout.

In this section, we describe a code-reuse attack that generically circumvents leakage-resilience techniques described above, even under the strong assumption that these techniques are universally and ideally enforced. We show how an attacker can indeed use indirect code pointers to launch meaningful exploits, without requiring the knowledge of code addresses. This is achieved by profiling indirect code pointers to determine the underlying code to which they point. We demonstrate how multiple profiled indirect code pointers can be used together to launch a chained AOCR attack akin to traditional ROP, but at the granularity of code blocks identified by indirect code pointers.

A. *Profiling and Malicious Thread Blocking*

The goal of profiling is to determine the original function `F()` that is invoked by an indirect code pointer `icptr`. Various X-only defenses use different names for indirect code pointers. For example, Readactor [15] calls them trampoline pointers, while ASLR-Guard [38] calls them encrypted code locators. We use the generic name "indirect code pointers", but the discussions apply to these and similar defenses.

An attacker who can identify the mapping of `icptr`→`F` can redirect control flow to `F` indirectly via `icptr`. "→" denotes that `icprt` is the pointer to the indirection layer (trampoline or encrypted pointer) that corresponds to `F`.

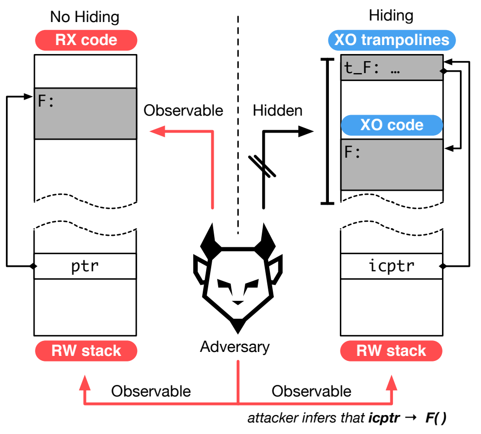

Fig. 1. Profiling of indirect code pointers by comparing Protected and Unprotected Execution States

To infer this mapping, we exploit the fact that programs execute in a manner that inherently leaks information about the state of execution. Knowledge about the execution state of a program at the time of a memory disclosure enables us to infer the `icptr`→`F` mapping from a leaked `icptr`.

An attacker can use her knowledge about function addresses in the unprotected version of the program (*i.e.*, attacker's local copy) to infer the locations of indirect code pointers in the protected version. We illustrate this idea in Fig 1. By observing what function pointers are placed in observable memory (*i.e.*, stack or heap) in the unprotected version, an attacker can infer that the pointers observed in the protected version must be the corresponding indirect code pointers of the same functions.

At a high-level, to perform the profiling, the attacker collects a list of function pointers from an unprotected version of the application *offline*, then she collects some indirect code pointers from the protected application in an *online* manner by sending the victim a few queries and observing parts of its data memory (e.g., stack). This allows the attacker to create a mapping between the discovered indirect code pointers and their underlying functions. The attacker can then chain these indirect code pointers to achieve the desired malicious behavior. Since the code snippets pointed to by these indirect code pointers behave like traditional ROP gadgets we call them AOCR gadgets. Note that this attack can be completed successfully without knowing the actual location (addresses) of the underlying functions. Although these steps look straightforward, in practice the attacker faces a number of technical challenges. Here, we describe the techniques we devised to overcome these challenges.

Repeatedly disclosing memory with precise timing to read indirect code pointers is a naïve first approach to create an accurate mapping of pointers to underlying functions. However, since the state of the system changes rapidly, this can result in inaccuracies in the mappings that may eventually cause a crash at exploitation time. To enhance the precision of the mapping, we devised a technique that we call Malicious Thread Blocking (MTB).

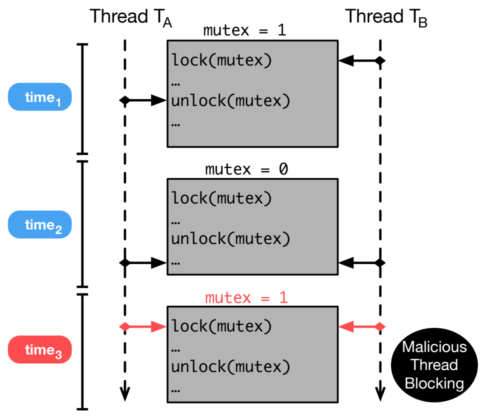

Fig. 2. Normal vs. Malicious Thread Blocking

In the case of programs that utilize threading, we can employ MTB to enable us to profile a broader range of indirect code pointers and avoid dependence upon strict timing requirements for triggering the disclosure vulnerability.

The approach of MTB is to use one thread, T_A, to cause another thread, T_B, to hang at an opportunistic moment by manipulating variables that cause T_B's execution to block, *e.g.*, by maliciously locking a mutex. By opportunistically blocking a thread, we can more easily locate and map desired indirect code pointers without worrying about rapid changes in memory. A memory disclosure vulnerability may be triggered in T_A that enables memory inspection at a known point in execution in T_B. Note that this technique avoids any timing unpredictability that the attacker may face when trying to trigger a disclosure in thread T_A at the appropriate time in execution for thread T_B. The idea of this approach is illustrated in Figure 2.

As one example of this technique in practice, we show in Section IV how an attacker can lock a mutex in Nginx to cause a thread to block upon returning from a system call. Triggering a memory disclosure vulnerability in another thread at any point after the system call enables the attacker to inspect a memory state that she knows contains indirect code pointers relevant to the system call. To more easily distinguish one system call from another, the attacker can supply a unique input and scan disclosed memory for that input. For instance, if the attacker wishes to profile the `open()` call, she may supply a unique file name as normal input to the program. Upon inspecting the stack of a blocked thread, the attacker would expect to find this unique value as an argument to the `open()` call. An attacker can continually block and unblock a thread by manipulating the mutex until this value is discovered in disclosed memory, which indicates that the attacker has located the relevant frame for `open()`. We illustrate this technique in more depth in our real-world exploits.

B. Passing Proper Arguments

After the attacker has mapped relevant indirect code pointers to their underlying functions, it is straightforward to redirect control flow to one of the functions. For the purpose of control-flow hijacking, knowing an indirect code pointer address is just as good as knowing the address of a function.

Consider the following code fragment:

```
call(int arg1, int arg2) {
    fptr(arg1, arg2); }

call_with_defaults() {
    fptr(default_arg1, default_arg2);}
```

If the attacker modifies the region of memory containing `fptr`, the next invocation of `call` or `call_with_defaults` will be redirected to an indirect code pointer chosen by the attacker. Unlike ROP and similar attacks, this redirection is consistent with the high-level semantics of C, and is thus unaffected by any underlying randomization (at the instruction-level, basic block-level, function-level, or library-level). In a valid C program `fptr` can potentially point to any function in the program.

Hijacking control flow in this manner does have limitations. If the attacker ends up hijacking a call like the one in `call`, the attacker will have very limited ability to control the arguments. The x86_64 ABI mandates a calling convention in which the first few arguments must be passed via registers. It is much more difficult to control a register value than it is to control a memory value. Some diversity techniques further complicate this by randomizing how registers are allocated to variables and how registers are saved to and restored from the stack [15, 45].

An attacker can overcome these defenses by concentrating on hijacking calls like the call in `call_with_defaults`. `call_with_defaults` invokes `fptr` on global variables. As global variables are stored in memory, they are trivial to modify. If an attacker is able to locate a function like `call_with_defaults`, she will be able to redirect control to a function of her choosing, with up to two arguments of her choosing. We found many such cases in our experiments with Nginx and Apache, as discussed in Section IV.

C. Chaining via Malicious Loop Redirection

An attacker wishing to chain multiple AOCR gadgets together faces another challenge: after calling an indirect code pointer, the execution returns to the original call site. This makes it difficult for the attacker to take the execution control back after a single function call. For example, in Readactor, trampolines consist of a call immediately followed by a jump to the original call site; any redirected call will end with a return to normal program execution. Theoretically, there is a window, potentially very narrow, between the invocation of a redirected call and the return of the redirected call in which an attacker may modify the return address to maintain control. This approach requires a very precise level of timing which may be difficult to achieve in practice.

Another option can be to use COOP-style attacks [47] to chain AOCR gadgets together using virtual functions. However, COOP attacks have various limitations that make them undesirable in an address oblivious attack. First, COOP relies on the dynamic dispatch implementation based on vtables; as such, it requires the leakage of vtable addresses. Second, COOP is hindered by vtable-randomization or register-randomization defenses [16, 59]. Third, COOP only applies to object-oriented languages, and is thus unavailable in applications developed in C (*e.g.*, Nginx).

```
while (task) {
    task->fptr(task->arg);
    task = task->next;
}
```

Fig. 3. A loop with a corruptible call site appropriate for MLR

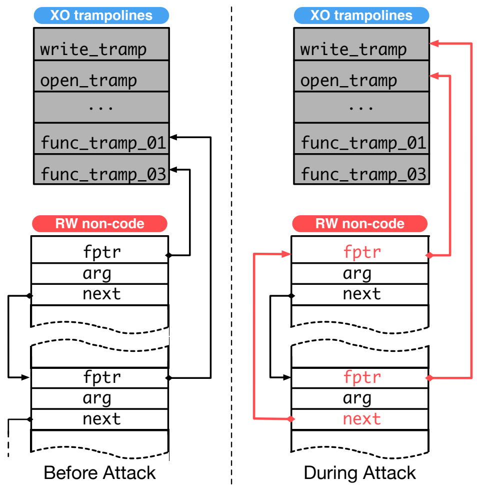

Fig. 4. Malicious Loop Redirection (MLR)

To overcome this, we introduce a new technique we call Malicious Loop Redirection (MLR), which only depends on language-level semantics widely found in C applications (loops over function pointers). In an MLR attack, an attacker chains a set of indirect code pointers (i.e., AOCR gadgets) using loops that contain indirect call sites in their body.

An appropriate redirectable loop gadget in MLR is a loop that:

1) has a loop condition that can be subverted by an attacker (e.g., the loop condition is in RW memory), and
2) it must call functions through code pointers

A simple example is presented in Fig 3. If `task` points to attacker-controllable memory, the attacker can cause the program to perform calls to multiple functions of her choosing by creating several fake `task` structures and setting up their `task->next` pointers to point to the next AOCR gadget. When the loop runs, AOCR gadgets are executed one by one without loss of control on attacker's side. We depict this attack graphically in Fig 4.

While some defenses implement register randomization to prevent chaining computations together, it does not prove to be an effective deterrent in this situation. The high-level semantics of the call dictate that the first argument will be taken from `task->arg` and moved to `rdi`, so our method of chaining AOCR gadgets using MLR succeeds against any randomization technique that preserves the x86_64 ABI.

IV. Real-World Exploits

In this section we present three real-world exploits combining various techniques described earlier. The first two exploits target Nginx and the third targets the Apache HTTP Server. The attacks are tested on Readactor as a proof-of-concept, but they are generally applicable to other leakage-resilient defenses as we discuss in Section VII. Nginx Attack 1 uses profiling to locate and call indirect code pointers (trampolines) for `open` and `_IO_new_file_overflow` and uses these to hijack control. Nginx Attack 2 and the Apache Attack use profiling to locate call trampolines for functions that eventually reach `exec`. Our exploits only assume the existence of memory corruption vulnerabilities akin to CVE 2013-2028 (Nginx) and CVE-2014-0226 (Apache).

A. *Nginx Attack 1*

We ran our experiments on stock Nginx 1.9.4 configured with support for asynchronous I/O. While this configuration is not default, it is very commonly used. The default configuration is generally not used in practice, and Nginx strives to be as portable as possible in its default configuration. By default, Nginx does not support HTTP SSL (https), ipv6, asynchronous I/O, or threading and runs a single worker process. Nginx specifically recommends the use of thread pools for performance benefits [6].

The aim of our attack is to cause Nginx to perform a malicious write to a file from a buffer located in execute-only memory. This requires locating addresses of functions that open and write files. We must also locate an indirect call site with enough corruptible arguments to call our target functions.

We began by inspecting the Nginx source code for suitable, corruptible call sites. We were able to find an indirect call site that retrieved both of its arguments from memory in Nginx's main loop for worker threads. On line 335 of `core/ngx_thread_pool.c`, the following call is made:

```
task->handler(task->ctx, tp->log);
```

This call site is ideal for our purposes: both the function pointer itself and the arguments are obtained by referencing fields of structs retrieved from memory, which are thus corruptible.

While this call site is suitable for calling `open`, which only requires two arguments, it does not allow us to call `write`, which requires three. As it seemed unlikely that a better callsite could be found, we began searching for ways to perform a write via a function that only takes two arguments. We eventually found `_IO_new_file_overflow`, an internal function in the GNU C Library (glibc) used when a write to a file is about to overflow its internal buffer. The signature for this function is included below:

```
_IO_new_file_overflow(_IO_FILE *f,int
ch)
```

`f` is a pointer to an `_IO_FILE`, glibc's internal version of the C standard library type `FILE`. `ch` is the character that was being written when the overflow occurred. If a pointer to an attacker controlled `_IO_FILE` were to be passed to this function, they would be able to reliably perform a write from an arbitrary buffer of arbitrary size to an arbitrary file descriptor.

To locate the indirect code pointers of these functions during an attack, we perform profiling as described in Section III.

`_IO_new_file_overflow` can be located using only analysis of in-memory values. Locating `open`, however, requires the use of the MTB technique. At a high level this attack proceeds in four phases:

1) Locate a mutex for MTB.
2) Profile an indirect code pointer for `open` (our first AOCR gadget).
3) Profile an indirect code pointer for `_IO_new_file_overflow` (our second AOCR gadget).
4) Corrupt Nginx's task queue so that a worker thread makes calls to our profiled trampolines using the MLR technique.

1) Locating a mutex: During execution, Nginx makes several `open` system calls. During these calls, the address of the trampoline (*i.e.*, indirect code pointer) for `open` is vulnerable to being read by an attacker. However, in practice, determining the exact address of this trampoline is difficult. Furthermore, the attacker would have to perform a read within the very narrow window of opportunity in which the address is on the stack. We overcome this difficulty by employing MTB. glibc's threading implementation supports a feature known as *thread cancellation*. There are two forms of cancellation: *asynchronous*, which means a thread's execution can be cancelled at any point in its execution, and *deferred*, which means any cancellation requests are deferred until a special predetermined point known as a cancellation point.

Every thread in a program contains a Thread Control Block (TCB). This structure contains thread-specific information and is used by glibc for maintaining metadata such as *thread local storage*, the current extent of the thread's stack, and thread cancellation. Inside the TCB is a field named `cancelhandling`. This field contains flags representing various aspects of a thread's cancellation state. We are concerned with the following flags:

- TCB_CANCELTYPE: indicates that the thread can be cancelled asynchronously via a signal.
- TCB_CANCELING: indicates that the thread's cancellation state is being mutated.
- TCB_CANCELED: indicates that the thread was successfully canceled.

Before entering a cancellation point, glibc executes `__pthread_enable_asynccancel`, a function that enables asynchronous cancellation by setting `TCB_CANCELTYPE` to true. After exiting a cancellation point, glibc executes `__pthread_disable_asynccancel`, a function that disables asynchronous cancellation by setting `TCB_CANCELTYPE` to false. A thread's cancellation can be requested by calling `pthread_cancel`, which will set `TCB_CANCELED` to true if asynchronous cancellation is disabled. If asynchronous cancellation is enabled the requesting thread will send a signal to the target thread and a signal handler will mark the thread for cancellation. This use of signals creates the possibility of a data race: if a thread is in the process of requesting a cancellation and the target thread disables asynchronous cancellation before the requesting thread sends its signal, the target thread will be forced to execute its cancellation handler while in an unexpected state. To prevent this, before sending a signal, the requesting thread uses a *Compare and Exchange* instruction that can ensure `TCB_CANCELING` is false, `TCB_CANCELTYPE` is true and set `TCB_CANCELING` to true atomically. `pthread_cancel` performs this instruction in a loop until it succeeds.

Analogously, upon exiting a cancellation point, a thread uses a *Compare and Exchange* instruction to both ensure `TCB_CANCELING` is false and to set `TCB_CANCELTYPE` to false. This instruction is also executed in a loop until it succeeds. Therefore `TCB_CANCELING` is a Mutual Exclusion Device (mutex) that prevents concurrently disabling asynchronous cancellation and sending an asynchronous cancellation signal. By setting `TCB_CANCELING` to true, an attacker can force a thread to loop in `__pthread_disable_asynccancel`, forever waiting for a signal that will never come.

Many cancellation points map directly to system calls and these system calls are surrounded by `__pthread_enable_asynccancel` and `__pthread_disable_asynccancel`. A simplified example of glibc's implementation of open is presented below:

```
__pthread_enable_asynccancel();
open syscall;
__pthread_disable_asynccancel();
```

Since glibc's `open` function uses a cancellable system call we can profile a trampoline for `open` by setting `TCB_CANCELING` to true and reading it off the stack when it hangs in the `__pthread_disable_asynccancel` after `open`.

Having identified a suitable mutex for MTB, we then determine a way to locate it at runtime. As `cancelhandling` is a field of a thread's TCB, given the base address of a TCB, it is trivial to locate `cancelhandling`. In glibc every TCB contains a header with the type `tcbhead_t`. The first field of this structure is defined as `void *tcb;` which is, actually, just a pointer to itself. The fact that the TCB begins with a pointer to itself makes it easily distinguishable in memory. Given an 8-byte value aligned at a known 8-byte aligned address, if the address is equal to its contents the address might represent the beginning of a TCB. In practice, for Nginx, the TCB is the only value on the stack that satisfies this property. Thus, starting from a known stack address, we can locate a thread's TCB by scanning backward for self-referential pointers. Once we locate a thread's TCB we can leverage `cancelhandling` to execute MTB against the thread. Additionally, since all TCBs are connected via linked list pointers, locating a single TCB allows us to locate the TCBs of all other threads.

2) Profiling `open`*:* Using the mutex found in the previous section, we can cause a thread of our choosing to hang at a non-determinate system call. By modifying `TCB_CANCELING` to false then true in quick succession, we can permit that thread's execution to continue and then stop again at a non-determinate system call. As Nginx makes many system calls that involve cancellation points, locating `open` requires the ability to distinguish when a thread is blocked at `open` versus when it is blocked at some other cancellable system call. We distinguish these situations by exploiting knowledge of how Nginx responds to requests for static files.

When Nginx receives an `HTTP GET` request for static content, it transforms the requested path into a path on the local filesystem. It calls `open` on this path and, if successful, responds with the file's contents. If `open` fails, it responds with `HTTP 404 Not Found`. During this process, a pointer to a string containing the path will be present on the stack. To determine whether or not Nginx is blocked at an `open` call we craft an HTTP request with a unique string and examine all strings pointed to from Nginx's stack.

```
void next_syscall(struct pthread *tcb)
{
        /* Rapidly mutate the cancelhandling field
         * to allow thread to proceed to next system
         * call
         */
        tcb->cancelhandling |= 4;
        tcb->cancelhandling &= ~4;
}

void *profile_open()
{
        const char *fprint = "4a7ed3b71413902422846"
        struct pthread *main_tcb = find_main_tcb();
        while (next_syscall()) {
               if (string_in_stack(rsp, fprint)) {
                      return *rsp
               }
        }
}
```

Fig. 5. Pseudocode for profiling `open`

If our string is sufficiently distinct (an example request is provided below), we can easily determine whether or not the current system call was made while processing our request. In practice, if Nginx does not find a requested file, the string holding the path is discarded and no pointers will appear to it on the stack in subsequent system calls. Thus, if we know that both (1) Nginx is blocked at a system call and (2) Nginx's stack contains a pointer to our constructed string, we can be sure Nginx is blocked at `open`. On average, we have been able to locate `open` by inspecting under 50 blocked system calls. The pseudocode for profiling `open` is shown in Fig 5.

```
GET /4a7ed3b71413902422846 HTTP/1.1
```

3) Profiling `_IO_new_file_overflow`*:* We profile `_IO_new_file_overflow` by taking advantage of glibc's implementation of the `stdio FILE` type. Every `FILE` contains a file descriptor, pointers to the file's buffers, and a table of function pointers to various file operations. `_IO_new_file_overflow` is included among these function pointers. By locating a valid `FILE`, we can easily locate `_IO_new_file_overflow` as the ordering of functions within the table is fixed. Finding a valid `FILE` pointer in Nginx proved to be a challenge as Nginx uses file descriptors instead of `FILE` pointers. In this situation, scanning the stack will not yield a pointer to a valid `FILE` object. To overcome this, we locate glibc's `FILE` for the standard output stream `stdout`. `stdout` is a global variable and is always automatically initialized on startup. Since `stdout` is a global variable defined by glibc, it is located in glibc's data segment. Due to ASLR, the location of glibc's data segment cannot be known *a priori*; nor can it be directly inferred from the address of the stack. Additionally, Nginx does not keep many pointers to glibc structures in local variables, meaning few pointers to glibc's data segment are on the stack. The attack is further complicated by the fact that we cannot dereference random stack values due to the risk of causing a segmentation fault.

Instead, we find a pointer into the heap, which occur more frequently in the stack. While pointers into the heap are common, they are not easily distinguishable from non-pointer values. To distinguish heap pointers we perform a simple statistical analysis on the values of the stack, the details of which we will present in a technical report for the sake of brevity. Here we briefly describe this analysis.

We collect 8-byte values from a 2-page range starting at the bottom of the stack. We bin these values based on their top 48 bits; *i.e.*, all values in the range (0, 0x0FFFFF) are placed in the first bin, all values in the range (0x100000, 0x1FFFFF) are placed in the second bin, and so on. We then sort the bins by their size. In our experiments, when Nginx's stack is partitioned in this fashion, the largest bin corresponds to non-pointers, the second largest bin corresponds to stack pointers, and the third large bin corresponds to heap pointers. This is due to the size of the address space available to a 64-bit program; any individual region of allocated memory will be several orders of magnitude smaller than the distance between the regions causing clustering of values. We found that, for Nginx, 2 pages of values collected at a single point in time is enough to reliably distinguish heap pointers. If, for some reason, we needed a higher degree of precision, this technique could be extended to either collect values at multiple points in time, or to collect values from more pages of the stack.

Now that we have pointers into the heap, it becomes possible for us to analyze the heap. We leverage this to find a pointer to `main_arena`, a glibc global variable. `main_arena` is a structure used by glibc to maintain information on allocated chunks of memory. To accelerate allocation operations, glibc partitions chunks into pre-sized bins and stores them in `main_arena`. Every heap chunk allocated via `malloc`, `calloc`, or `realloc` is prefixed with metadata containing a pointer back to the `main_arena` bin it came from. We take advantage of this to locate a pointer into `main_arena`.

Starting from the smallest pointer in our bin of heap pointers, we collect 8-byte values from a 20 page range of the heap. We then filter out values unlikely to be pointers.

Our criteria for discarding non-pointers is described below.

1) Discard all values that are not multiples of 8
2) Discard all values greater than 0x7FFFFFFFFFFF
3) Discard all values less than 0x1000

Finally, we partition the remaining values into bins of size 0x100000. The most common pointer of the largest bin will be a pointer into `main_arena`. This is due to most chunks of the heap being allocated out of the same bin.

Now that we have a pointer into glibc's data section we can search for `stdout`. We identify `stdout` by scanning backwards from `main_arena`, and looking for a region that is both a valid `FILE` and has the value `1` for its underlying file descriptor. At this point, the location of `_IO_new_file_overflow` can be trivially read off of `stdout`.

4) Corrupting the Nginx Task Queue: The main loop for Nginx worker threads is located in `ngx_thread_pool_cycle`. All new worker threads spin in this loop, checking if new tasks have been added to their work queue. A simplified version of this loop is presented below:

```
for (;;) {
    task = queue_get(tp->task_queue);
    task->handler(task->ctx, tp->log);
}
```

To carry out the attack, we leverage our MLR technique. We craft a fake task structure in the region of the stack that originally contained Nginx's environment variables. At startup Nginx copies these to a new location and the original location goes unused.

We initialize our fake task such that `task->handler` points to `open` and `task->ctx` points to `html/index.html`. We also modify `tp->log` to be equal to `(O_DIRECT | O_SYNC | O_WRONLY | O_TRUNC)`. While this invalidates the `tp->log` pointer, in practice, threads do not log unless Nginx is compiled in debug mode. When the worker thread executes this task, it will open the file in `O_DIRECT` mode, allowing us to perform an FDMA attack.

Once we have our fake task structure, we can append it to the task queue and wait for Nginx to execute the task. In most cases, this happens instantaneously, so after a few seconds we can be confident our call has occurred. We repeat this process 100 times so that there will be at least 100 file descriptors in `O_DIRECT` mode opened by the Nginx process.

For the call to `_IO_new_file_overflow`, we begin by creating a fake `FILE` that matches `stdout` except for the following fields:

1) `file->file__fileno = 75`
2) `file->fileIO_write_base = file->vtable->__overflow & 0xFFF`
3) `file->fileIO_write_ptr = file->fileIO_write_base + 0x1000`
4) `file->fileIO_read_end = file->fileIO_write_base`

Next, we modify our fake task such that `task->handler` points to `_IO_new_file_overflow` and `task->ctx` points to our fake `FILE`. We also modify `tp->log` to be -1 `EOF`. This will cause `_IO_new_file_overflow` to think the write buffer overflowed just as the end of the file was reached, so it will immediately flush the buffer via a write. Once we have crafted our fake arguments we append the fake task to the task queue and wait for the task to be executed. Conceptually `_IO_new_file_overflow` will be executing the equivalent of the following code:

```
write(75,\_IO\_FILE\_Overflow & ~0xFFF, 0x1000);
```

Which results in a dump from execute only memory into the file `html/index.html`. We can then retrieve this page of code by sending `GET /index.html HTTP/1.1`. We now have the contents of a page of code at a known location and that can be reused in arbitrary ways. If necessary, we can perform this as many times as we want to leak more pages of memory. Note that this is an optional and additional step to the initial exploit detailed above. The initial exploit is completely address oblivious, but further steps built on top of it can take advantage of conventional ROP or even code injection techniques (after disabling W$\oplus$X) for ease of implementation.

B. Nginx Attack 2

We now illustrate the generality of our techniques by performing a second attack against Nginx that both (1) targets different functions and (2) corrupts a different call site.

This attack relies on invoking Nginx's master process loop from an attacker-controlled worker in order to trigger a specific signal handler and cause arbitrary process execution. There are three phases to this attack:

1) Use profiling to get the address of the master process loop.
2) Use MTB to corrupt a function pointer to point at the master process loop.
3) Set global variables via MTB to cause the master process loop to call `exec` under attacker-chosen parameters.

For the sake of brevity, we describe the details of this attack in Appendix A.

C. Apache Attack

Finally, we describe an attack using similar techniques against the Apache HTTP Server. While previous attacks have focused on Nginx, MTB and profiling are general and can be applied to other targets. Arbitrary process execution can be achieved on the Apache web server using a similar approach:

1) Use profiling to find the indirect code pointer of the `exec`-like function `ap_get_exec_line`.
2) Use MTB to corrupt a function pointer to point at `ap_get_exec_line` and cause an `exec` call under attacker control.

For the sake of brevity, we describe the details of this attack in Appendix B.

All exploits succeeded in control hijacking while Apache and Nginx were protected by full-featured Readactor. Note that Turing completeness is trivially provided if the inputs to `exec()` or `system()` can be compromised. An example of this is in Nginx Attack 2, where we leverage an `exec()` call made by Nginx to execute a reverse shell written in python.

V. Generality of AOCR Attacks

The sophisticated exploitation techniques discussed in this paper may provide the impression that opportunities for such exploits are rare. We argue that, in fact, the exploitable constructs are very common in real-world code bases. Code pointers are extremely common in any production-level application or server, so the opportunities for leaking indirect code pointers are almost certainly present too. It is also a common idiom in C to pass around structs filled with function pointers as a way to perform dynamic dispatch, which also provides additional code pointers.

We also argue that the MTB technique used to facilitate the exploits is both optional and surprisingly easy to find.

A. *Necessity of MTB*

In our exploits, we leveraged MTB to simplify the identification of the `open()` callstack. MTB allowed us to assume that at the time of our probe Nginx or Apache was blocked immediately before a system call, reducing the set of possible callstacks to less than 10 and providing unlimited time to read the stack. This allowed us to profile a pointer to the `open()` system call in seconds.

While MTB makes profiling substantially easier, it is not required to successfully perform such an attack. The attack can succeed even if the targeted thread was still running. However, instead of having to identify the `open()` callstack out of approximately 10 possibilities, the attacker would have to identify it out of potentially hundreds of possibilities. To quantify the difficulty of such an attack we captured a sample of stack traces, and analyzed whether an attacker is still able to identify a targeted function call.

Therefore, we ran Nginx under Linux `perf` tools in sampling mode, and captured 2,500 samples of the callstack (approximately 200 unique) over a period of 10 seconds. Of those 2,500 sampled callstacks, about 12 samples ($\sim$0.5%) were the targeted `open()` callstack from our attack. Without MTB, the application might modify the stack during a probe. An attacker can tolerate this by repeating the probes to confirm the results. Given this small adjustment, we found that our profiling attack (see Section III-A) is still effective in identifying the `open()` callstacks without generating false-positives.

B. *Applicability of MTB*

While MTB utilizes mutexes to exert control over a target, it is important to note that this does not mean a program needs to rely on mutexes to be vulnerable to MTB. In fact, in our exploits, we do not target a mutex used by Nginx. Instead we target a mutex used by glibc. Due to the use of mutexes by glibc to implement POSIX compliance, any application that both (1) is multithreaded and (2) performs I/O is potentially vulnerable. As threads are used to perform I/O without blocking an applications execution, this makes MTB applicable to a very large variety of server applications. In fact, on Linux, performing I/O on threads is essentially the only way to achieve non-blocking file I/O (Nginx claims performance improvements of up to 9x by simply enabling threaded I/O).

Furthermore, it is not necessary for an application to be explicitly multi-threaded. There are many situations in which application frameworks make use of threads internally, unbeknownst to the application. Examples of this include (1) libuv: the framework for asynchronous I/O. Used in projects such as node.js and the Julia language, libuv implements all file operations via a thread pool; (2) OpenJDK: the open source implementation of the Java Platform, Standard Edition. OpenJDK implements asynchronous I/O via a thread pool; (3) glibc: the POSIX asynchronous I/O functions (e.g. `aio_read()`) are implemented with glibc-internal threads and mutexes. Thus, even seemingly single-threaded applications may be vulnerable simply due to underlying frameworks creating threads.

VI. X-Only Implementation Challenges

Up until this point, we assumed an ideal and comprehensive implementation of execute-only memory that our AOCR attack can bypass. However, actually achieving such an ideal and comprehensive implementation is surprisingly difficult in practice. Leaky code pointer protection is not the only challenge facing code randomization defenses. Modern operating systems such as UNIX-based systems provide a myriad of facilities that can potentially leak protected memory to an attacker. In this section, we briefly discuss two such vectors that are hard to mitigate and are in fact unprotected in the execute-only defenses that we studied. The first, Direct Memory Access (DMA), offers attackers the potential to bypass execute-only protection by abusing memory access and leaking code directly. Unlike related work that focuses on abusing DMA via malicious hardware devices, we discuss an attack that is a form of the confused deputy attack through which an application can fool the system to make malicious DMA requests on its behalf using software-only attacks. The second vector is the `proc` filesystem in Linux that can potentially leak information about execute-only memory. Preventing this vector is hard because disabling it would break many benign applications.

If these vectors are available, an attacker can use them to leak code pages directly, and does not have to resort to AOCR techniques. We discuss them here to further illustrate the challenges of effectively preventing CRA attacks in complex, modern systems.

A. *Forged Direct Memory Access Attack*

Execute-only defenses protect code pages from direct read accesses by applying additional permissions to memory pages in software [3] or hardware [15, 22]. This enforcement, however, applies only to regular memory accesses (*i.e.*, TLB-mediated). Accesses performed by devices capable of Direct Memory Access (DMA), *e.g.*, GPUs, disk drives, and network cards, do not undergo translation by the MMU and are unaffected by page permission. We call these accesses "non-TLB-mediated."

The idea of exploiting systems via DMA is well studied, especially in the context of DMA-capable interfaces with external connectors, *e.g.*, IEEE 1394 "Firewire" and Thunderbolt.

As described in the threat model (Section II), we are mainly concerned about a remote attacker. For that, the attacker must be able to perform software-based DMA from a userspace application. Typically, user space applications cannot directly make requests to DMA-capable devices. However, some user space functionality is implemented via the kernel requesting a device to perform DMA against a userspace-controlled address. Examples of this include OpenCL's `CL_MEM_USE_HOST_PTR` flag and Linux's `O_DIRECT` flag.

An attacker can use Linux's `O_DIRECT` flag to maliciously request software-based DMA to bypass execute-only memory permissions, thus alleviating the need for compromised peripheral devices or hardware attacks. We call such an attack a Forged DMA (FDMA) attack which is a form of confused deputy attack, and briefly demonstrate its feasibility. The novelty of FDMA is its broad applicability remotely and from user space applications. Unlike well-studied DMA attacks such the one used in bypassing Xen [62], FDMA does not require a malicious device or kernel permissions.

Applications that use the O_DIRECT flag natively are vulnerable to our FDMA attack. More surprisingly though, even applications that never use the O_DIRECT flag, but pass the flags to file read or write operations through a flags variable residing in data memory are also vulnerable to this attack. An attacker can perform a simple data-only attack to maliciously change the flag variable to O_DIRECT in order to force a regular file operation to become a DMA access.

We investigated the prevalence of direct I/O and flags variables is in popular real-world software packages. Our analysis focused on Internet-facing web servers (AOLserver, Apache, Boa, lighttpd, Nginx, OpenSSH, Squid, and Firebird) due to their exposure and database managers (Hypertable, MariaDB, Memcached, MongoDB, MySQL, PostgreSQL, Redis, and SQLite) due to their focus on fast I/O. The results indicate that the majority of web servers and database managers (13 out of 16) do not natively use the O_DIRECT flag; however, 10 out of 16 of them (AOLserver, Nginx, OpenSSH, Squid, Firebird, Hypertable, MongoDB, MySQL, PostgreSQL, and SQLite) use variables to store flags that can be corrupted by an attacker to set the O_DIRECT flag. As such, an attacker can use an FDMA attack in these applications to read execute-only code pages to build a conventional ROP attack even in the presence of execute-only defenses. The FDMA attack would obviate the exploit, and does not require an AOCR attack to bypass execute-only memory permissions.

B. Procfs Attack

The proc filesystem is another implementation challenge that can obviate execute-only bypasses.

The proc filesystem is a file-like structure that contains information about each process. It is implemented for a variety of UNIX-like operating systems [20, 32]. In this paper, we focus on the Linux implementation of procfs [10].

The Linux kernel creates a directory for each process that can be accessed via /proc/<process id>/. Processes can access their own directory via /proc/self/. The files within the procfs directory are, for the most part, treated in the same way as any other file in a filesystem. They have ownership settings and assigned permissions, and are accessed via the same mechanisms as any other file. Through them, a wealth of information about the process is made available: details about program invocation, processing status, memory access, file descriptors, networking, and other internal details.

Several of the procfs files (e.g., auxv, maps, numa_maps, pagemaps, smaps, stat, syscall, exe, stack, and task) include memory addresses that reveal information about the randomized code layout. The mem file even allows direct disclosure of the process memory regardless of memory permissions.

To carry out a procfs attack, the attacker needs to (1) discover the location of a suitable piece of executable memory, and (2) leak executable memory directly by corrupting the filename argument to a file read operation. The maps and smaps files provide, among other things, the starting and ending addresses of each mapped memory region, along with that region's memory permissions and the file (if any) with which the region is associated. After that, reading the mem file directly leaks the executable regions. Note that even when the vulnerability does not allow arbitrary file reads, the procfs attack can be mounted by performing a data-only corruption on any file read operation.

The procfs attack also allows a leakage of the actual code pointers followed by a traditional ROP attack, without requiring the sophistication of an AOCR attack.

Because procfs is baked into the Linux ecosystem as the needed native interface for many system utilities and programs, removing or otherwise blocking access to it would disrupt a major kernel API and break a Linux distribution. Fundamental Linux command-line tools depend on access to procfs, most notably free, kill, pkill, pgrap, pmap, ps, pwdx, skill, slabtop, snice, sysctl, tload, top, uptime, vmstat, w, and watch. Similarly, additional programs in GNU coreutils and binutils, and the util-linux package make use of procfs. Debuggers like gdb and system monitoring tools like nmon are among the many other programs reliant upon the continued functionality of procfs.

The exposed nature of procfs has long been recognized and attacks proposed to exploit it especially with regard to differential privacy [30, 64]. Although it cannot be removed entirely due to the above-mentioned concerns, some defenses have attempted to restrict access to procfs. For example, GRSecurity's kernel patchset [55] has several configuration options to restrict access to procfs entries by user or group, with the intent that different critical processes can run as different users and be unable to compromise other processes. One recent defense [63] proposes falsifying information in procfs to mitigate other types of attacks.

However, these defenses focus on blocking *other* processes' access to the procfs of a given process; they do not prevent access by a process to its *own* procfs entry set, and any finer-grained procfs restriction by username would result in breaking benign applications. As such, effectively securing procfs without breaking benign applications remains an open research problem.

VII. Impact on Leakage-Resilient Defenses

Defenses that do not provide leakage resilience are trivially vulnerable to AOCR and weaker forms of information leakage. Therefore, we focus on those that offer (some) resilience.

Direct leakage refers to attacks that read code pages, while indirect leakage refers to attacks that leak code addresses from the stack or heap during execution. Since AOCR attacks leak hidden or indirection (*e.g.*, trampoline) pointers indirectly from the stack or heap, they are a form of indirect leakage attacks. Also, since non-TLB-mediated leakages directly read code pages (using mechanisms not protected by memory permissions), they are a form of direct information leakage. Accordingly, there are four sub-classes of information leakage: direct leakage via TLB-mediated code reads, direct leakage via non-TLB-mediated code reads, indirect leakage of code pointers, and indirect leakage of indirect code pointers.

Our attacks are applicable to randomization defenses regardless of the granularity or type of randomization. For example, various randomization defenses propose library-level, function-level, or instruction-level randomization approaches.

TABLE I. DEFENSES PROTECTING AGAINST DIFFERENT CLASSES OF INFORMATION LEAKAGE ATTACKS

	Direct Leaks		**Indirect Leaks**	
Defenses	**TLB-Mediated** (*e.g.*, buffer over-read [56])	**Non-TLB-Mediated** (*e.g.*, DMA§ VI)	**Code Pointer Leaks** (*e.g.*, Ret addr. leak [17])	**Indirect Code Pointer Leaks** (*e.g.*, AOCR § III)
PointGuard [14]			✓	
Oxymoron [4]			✓	
Isomeron [17]			✓	
XnR [3]	✓			
HideM [22]	✓			
Readactor [15, 16]	✓		✓	
Heisenbyte [58]	✓			
NEAR [61]	✓			
ASLR-Guard [38]			✓	
TASR [7]	✓	✓	✓	✓

In AOCR, we abuse and chain indirect code pointers to achieve control-flow hijacking. Regardless of how the underlying code has been randomized, as long as the semantics remain intact, our profiling attack remain applicable. In attacks that use implementation challenges (FDMA), the exact contents of code pages are read (via non-TLB-mediated accesses), so regardless of the how intrusive the randomization is, we can disclose the randomized code and perform a conventional ROP attack.

Table I summarizes leakage-resilient randomization defenses and their vulnerabilities to various types of attacks. We briefly discuss each defense and how our attacks apply in the following.

PointGuard [14] protects all pointers stored in memory by masking them with an XOR key. It therefore prevents leakage of code addresses via pointers. However, indirect leakage of encrypted pointers and direct leakage attacks remain possible.

Oxymoron [4] attempts to prevent JIT-ROP attacks by adding a layer of indirection to instructions such as branches that reference other code pages. While Oxymoron thwarts the recursive disassembly step of the original JIT-ROP attack, it does not protect all pointers to code. Davi *et al.* [17] show an attack against Oxymoron, exploiting indirect address leakage. They then propose Isomeron that combines execution-path randomization with code randomization to build indirect leakage resistance. Neither of these techniques prevent direct code reads.

XnR [3] and HideM [22] perform permission checks on memory accesses to implement execute-only, thus preventing TLB-mediated code reads. They, however, do not check non-TLB-mediated code reads. They are also vulnerable to indirect leakage attacks, since code pointers are not hidden or protected in any way during execution. Leakage of return addresses or function pointers from the stack or heap remains possible during execution.

Readactor [15] utilizes Extended Page Table permissions to enforce execute-only permission and adds a layer of indirection (trampolines) to prevent indirect leaks. Therefore, it prevents TLB-mediated direct code reads and indirect leaks of code pointers (*e.g.*, return addresses and function pointers). Its permissions, however, do not apply to non-TLB-mediated accesses as demonstrated in Section VI. Moreover, leakage of trampoline pointers (*i.e.*, indirect code pointers) are possible as demonstrated by our AOCR attack against Apache and Nginx.

Heisenbyte [58] and NEAR [61] prevents executable region leakages by making any code-area read destructive. Therefore, these techniques can only mitigate TLB-mediated direct leakage. Non-TLB-mediated memory accesses do not cause a byte destruction; thus, they are not mitigated. Indirect leakages also remain possible because code pointers are not protected in any way.

ASLR-Guard [38] provides leakage resistant ASLR by decoupling code and data, storing code locators in a secure region of memory, and encrypting code locators that are stored in observable memory. As a result, code locators themselves cannot leak because they are encrypted whenever placed in regular memory. However, the encrypted forward pointers can be profiled and reused by an attacker. This is hinted at in the paper itself: "... attackers may reused [*sic*] the leaked encrypted code locators to divert control flow." Direct code reads, whether they are through TLB (*e.g.*, buffer over-reads) or not, also remain possible in ASLR-Guard.

TASR [7] re-randomizes code regions at every I/O system call pair to mitigate any potential information leakage. It also fixes the code pointers on the stack and heap for every re-randomization. It can potentially mitigate all classes of remote leakage attacks, but it requires source code for compilation and it cannot mitigate leakages within the application boundary (*e.g.*, in JIT-ROP attacks).

VIII. MITIGATING ADDRESS-OBLIVIOUS CODE REUSE

Since AOCR attacks induce unintended control flows, enforcing control-flow integrity is one way to mitigate them. Isolating indirect code pointers using code-pointer integrity is another option. These mitigations, however, come with their own set of performance and security challenges, so we only consider ways to extend leakage-resilient diversity to counter AOCR in this section.

Although our variant of code reuse is oblivious to the code layout, it is *not* oblivious to the data layout. In particular, it makes assumptions on the layout of structures as well as the layout of global variables. Therefore, one might argue that these areas need randomization too. Techniques to do so are well known in the literature [13, 23, 36]. However, this would not prevent adversaries from reading and writing the data structures after diversification so our attacks could, at least in theory, be extended to leak the data layout.

Perhaps a better strategy is to extend the code pointer indirection layer with an authentication step to prevent misuse. One way to instantiate this idea is to have every calling function store a cookie in a register before using an indirect pointer and have the callee function check the register for the expected cookie value (after which the cookie register must be cleared to avoid spills to the stack [35]). Cookies would simply be random values stored as immediate operands in execute-only code to prevent leakage. This scheme would prevent abuse of trampolines for direct calls but would not prevent abuse of indirect calls or returns because we may not know their control-flow targets at compile time [19].

To protect indirect calls and returns from abuse, we can still verify that the function pointer used in an indirect call or return was correctly stored and not forged without having to compute the control-flow graph at compile time. To do so, we can leverage the recently proposed cryptographically-enforced control flow integrity, CCFI, technique by Mashtizadeh, *et al.* [40]. In Readactor, an indirect code pointer is simply the address of the forward trampoline; to prevent AOCR, we can associate each indirect pointer and return address with a hash-based message authentication code, HMAC. Note that if we use the storage location of the indirect code pointer (not the address it points to) as input to the HMAC function, copying the indirect pointer from one storage location to another (as our AOCR requires) will cause the HMAC check to fail. CCFI uses an 128-bit AES-based HMACs and stored the AES key in SIMD registers. This led to high performance overheads (52% on average on SPEC CPU2006), especially relative to conventional CFI approaches, but has the undeniable advantage of enforcing a very precise CFI policy without the need for complicated and brittle static analysis. Moreover, future hardware will likely include hardware support for protection of return addresses, which would simplify our task to protection of forward pointers.

Our work adds to the growing body of evidence showing that it is nigh impossible to avoid all types of information leakage. If given the choice between randomizing more implementation aspects or incorporating (or switching to) enforcement-based mitigations, the latter seems like a better choice w.r.t. attainable security. Enforcement-based mitigations also have important practical advantages in that they naturally do not interfere with code signing, distribution, memory de-duplication, or debugging; code diversity engines must be carefully designed to avoid interference in these areas.

IX. Related Work

Our work mainly relates to memory corruption vulnerabilities and mitigation thereof. The literature in these areas is vast. We refer the interested reader to the relevant surveys [12, 34, 43, 52, 57] and focus on closely related work.

Early work on the effectiveness of ASLR found that 32-bit address spaces do not allow sufficient entropy in the layout to prevent brute force guessing [51]. A decade later, it became clear that not even 64-bit ASLR implementations are impervious to brute-force attacks [8] and exploits now routinely bypass ASLR using a variety of techniques [42, 49, 56]. This motivated fine-grained diversity approaches [34] that randomize at the level of individual code pages [4], functions [31], basic blocks [60], or single instructions [25, 27, 45]. The emergence of JIT-ROP [53] and side-channel attacks [5, 28, 48] that directly or indirectly disclose the randomized code layout undermined the assumption that these finer-grained diversity techniques address the shortcomings of ASLR [26]. These findings led to work on leakage-resilient code randomization defenses. We already discussed these defenses and how their security is impacted by AOCR attacks in Section VII. For the sake of brevity, we do not repeat that discussion here.

Davi, *et al.* [17] demonstrated the first attack against leakage-resilient diversity approaches. In particular, they showed that execute-only memory (on its own) does not provide sufficient protection against all JIT-ROP attacks. This inspired subsequent work on code pointer hiding [15, 38]. Maisuradze, *et al.* [39] then demonstrated that the predictability of dynamically compiled code provides another way to bypass execute-only defenses without directly disclosing the code. However, our AOCR attacks are strictly more powerful as none of these earlier attacks are fully oblivious to the code layout. The attack by Davi, *et al.*, requires that the code is either readable or that the code is not randomized below the page level. The attack by Maisuradze, *et al.* assumes that i) pointers into JIT compiled code are not protected against indirect leakage, and ii) that JIT compiled code is not randomized below the function level. Since we only rely on the high-level semantics of the code, our bypass is not even stopped by instruction-level randomization.

In work closely related to ours, Snow, *et al.* [54] evaluated the effectiveness of leakage-resilience techniques relying on destructive reads such as Heisenbyte [58] and NEAR [61]. Their main finding was that destructive reads can be bypassed using so called constructive reloads. Such reloads exploit the fact that multiple copies of the same code are often loaded into the same process which means that adversaries can disclose one copy and reuse code from another, thereby avoiding any gadgets destroyed by adversarial reads. However, the constructive read techniques are limited to bypassing leakage-resilience defenses relying on destructive reads while our techniques generalize to all of the defenses listed in Table I.

Gawlik, *et al.* [21] reported that the security assumptions of leakage-resilient defenses can be weakened by using crash resistant exploitation primitives. These primitives allow adversaries to scan memory without crashing when trying to read execute-only memory or accessing unmapped memory. However, as the authors note, crash-resilience techniques cannot bypass the Readactor++ [16] system, unlike our AOCR attacks.

Göktaş, *et al.* [24] demonstrated that malicious thread spraying can, in certain instances, be used to find even very small hidden memory regions associated with a particular thread (the safe stack). However, malicious thread spraying does not disclose the code layout in our threat model since we assume perfect use of code pointer hiding and fine-grained randomization.

Memory deduplication between processes or between virtual machines in a public cloud poses another threat to information hiding. Early work demonstrated how to leak the base address of 64-bit ASLR in a cloud environment [5]. Subsequent work showed how to leak entire pages [9, 46]. These techniques rely on timing side channels induced by the copy-on-write semantics of deduplicated, writable pages and thus do not help leak the contents of read-only or execute-only memory pages.

X. Conclusion

In this paper, we evaluated the effectiveness of leakage-resilient code randomization. We presented a generic class of attacks, Address-Oblivious Code Reuse (AOCR), that can bypass ideal execute-only defenses including the state-of-the art system, Readactor, and showed two new attack techniques to facilitate AOCR. We demonstrated that AOCR is a realistic threat with three concrete attacks against Nginx and Apache. We also discussed two important implementation challenges that practitioners must address to correctly deploy leakage-resilient defenses.

Our findings add to the mounting body of evidence that preventing information leaks without addressing the root causes of memory corruption vulnerabilities is fiendishly hard if not downright impossible. As long as the adversaries can observe and swap code pointers (or their encrypted/indirect equivalents), code reuse attacks remain possible. Our main contribution is to show, for the first time, that such attacks can be constructed without any knowledge of the randomized code addresses. Thus, we conclude that i) the research community is running up against the limits of leakage-resilient diversity techniques and that ii) enforcement techniques seem like the most attractive way to further raise the bar against exploitation.

Acknowledgments

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-15-C-0124, FA8750-15-C-0085, and FA8750-10-C-0237, by the National Science Foundation under award number CNS-1513837 and CNS-1619211, as well as gifts from Oracle and Qualcomm.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government.

This work was supported in part by the German Science Foundation (project S2, CRC 1119 CROSSING), the European Unions Seventh Framework Programme (609611, PRACTICE), and the German Federal Ministry of Education and Research within CRISP.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security*, CCS, 2005.

[2] J. P. Anderson. Computer security technology planning study. volume 2. Technical report, DTIC Document, 1972.

[3] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *ACM Conference on Computer and Communications Security*, CCS, 2014.

[4] M. Backes and S. Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *23rd USENIX Security Symposium*, USENIX Sec, 2014.

[5] A. Barresi, K. Razavi, M. Payer, and T. R. Gross. CAIN: Silently Breaking ASLR in the Cloud. In *WOOT*, 2015.

[6] V. Bartenev. Thread pools in nginx boost performance 9x, 2015, https://www.nginx.com/blog/thread-pools-boost-performance-9x/.

[7] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi. Timely rerandomization for mitigating memory disclosures. In *ACM Conference on Computer and Communications Security*, CCS, 2015.

[8] A. Bittau, A. Belay, A. J. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *35th IEEE Symposium on Security and Privacy*, S&P, 2014.

[9] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *37th IEEE Symposium on Security and Privacy*, 2016.

[10] T. Bowden, B. Bauer, J. Nerin, S. Feng, and S. Seibold. The /proc filesystem. Linux Kernel Documentation, 2009.

[11] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi. Leakage-resilient layout randomization for mobile devices. In *23rd Annual Network and Distributed System Security Symposium*, NDSS, 2016.

[12] N. Burow, S. A. Carr, S. Brunthaler, M. Payer, J. Nash, P. Larsen, and M. Franz. Control-flow integrity: Precision, security, and performance. *CoRR*, abs/1602.04056, 2016.

[13] P. Chen, J. Xu, Z. Lin, D. Xu, B. Mao, and P. Liu. A practical approach for adaptive data structure layout randomization. In *20th European Symposium on Research in Computer Security*, ESORICS, 2015.

[14] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: protecting pointers from buffer overflow vulnerabilities. In *12th USENIX Security Symposium*, USENIX Sec, 2003.

[15] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *36th IEEE Symposium on Security and Privacy*, S&P, 2015.

[16] S. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. D. Sutter, and M. Franz. It's a TRaP: Table randomization and protection against function-reuse attacks. In *ACM Conference on Computer and Communications Security*, CCS, 2015.

[17] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (Just-In-Time) return-oriented programming. In *22nd Annual Network and Distributed System Security Symposium*, NDSS, 2015.

[18] 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 *36th IEEE Symposium on Security and Privacy*, S&P, 2015.

[19] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *ACM Conference on Computer and Communications Security*, CCS, 2015.

[20] R. Faulkner and R. Gomes. The process file system and process model in unix system v. In *USENIX Technical Conference*, ATC, 1991.

[21] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz. Enabling client-side crash-resistance to overcome diversification and information hiding. In *23rd Annual Network and Distributed System Security Symposium*, NDSS, 2016.

[22] J. Gionta, W. Enck, and P. Ning. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *5th ACM Conference on Data and Application Security and Privacy*, CODASPY, 2015.

[23] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *21st USENIX Security Symposium*,

USENIX Sec, 2012.

[24] E. Göktaş, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos. Undermining information hiding (and what to do about it). In *25th USENIX Security Symposium*, 2016.

[25] J. Hiser, A. Nguyen, M. Co, M. Hall, and J. Davidson. ILR: Where'd my gadgets go. In *33rd IEEE Symposium on Security and Privacy*, S&P, 2012.

[26] T. Hobson, H. Okhravi, D. Bigelow, R. Rudd, and W. Streilein. On the Challenges of Effective Movement. In *ACM CCS Moving Target Defense (MTD) Workshop*, Nov 2014.

[27] A. Homescu, T. Jackson, S. Crane, S. Brunthaler, P. Larsen, and M. Franz. Large-scale automated software diversity—program evolution redux. *IEEE Transactions on Dependable and Secure Computing*, PP(99):1, 1 2015. Pre-Print.

[28] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space ASLR. In *34th IEEE Symposium on Security and Privacy*, S&P, 2013.

[29] Intel. Intel 64 and IA-32 architectures software developer's manual. ch 28, 2015.

[30] S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints. In *33rd IEEE Symposium on Security and Privacy*, S&P, 2012.

[31] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (ASLP): towards fine-grained randomization of commodity software. In *22nd Annual Computer Security Applications Conference*, ACSAC, 2006.

[32] T. J. Killian. Processes as files. In *USENIX Association Software Tools Users Group Summer Conference*, STUG, 1984.

[33] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2014.

[34] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *35th IEEE Symposium on Security and Privacy*, S&P, 2014.

[35] C. Liebchen, M. Negro, P. Larsen, L. Davi, A.-R. Sadeghi, S. Crane, M. Qunaibit, M. Franz, and M. Conti. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *ACM Conference on Computer and Communications Security*, CCS, 2015.

[36] Z. Lin, R. D. Riley, and D. Xu. Polymorphing software by randomizing data structure layout. In *6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA, 2009.

[37] K. Lu, S. Nürnberger, M. Backes, and W. Lee. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. In *23rd Annual Network and Distributed System Security Symposium*, NDSS, 2016.

[38] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks. In *ACM Conference on Computer and Communications Security*, CCS, 2015.

[39] G. Maisuradze, M. Backes, and C. Rossow. What Cannot Be Read, Cannot Be Leveraged? Revisiting Assumptions of JIT-ROP Defenses. In *25th USENIX Security Symposium*, USENIX Sec, 2016.

[40] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. CCFI: cryptographically enforced control flow integrity. In *ACM Conference on Computer and Communications Security*, CCS, 2015.

[41] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *ACM Conference on Programming Language Design and Implementation*, PLDI, 2009.

[42] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida. Poking holes in information hiding. In *25th USENIX Security Symposium*, USENIX Sec, 2016.

[43] H. Okhravi, T. Hobson, D. Bigelow, and W. Streilein. Finding focus in the blur of moving-target techniques. *Security Privacy, IEEE*, 12(2):16–26, Mar 2014.

[44] A. One. Smashing the stack for fun and profit. *Phrack magazine*, 7, 1996.

[45] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *33rd IEEE Symposium on Security and Privacy*, S&P, 2012.

[46] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos. Flip feng shui: Hammering a needle in the software stack. In *25th USENIX Security Symposium*, 2016.

[47] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *36th IEEE Symposium on Security and Privacy*, S&P, 2015.

[48] J. Seibert, H. Okhravi, and E. Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *ACM Conference on Computer and Communications Security*, CCS, 2014.

[49] F. J. Serna. CVE-2012-0769, the case of the perfect info leak, 2012.

[50] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security*, CCS, 2007.

[51] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proc. of ACM CCS*, pages 298–307, 2004.

[52] R. Skowyra, K. Casteel, H. Okhravi, N. Zeldovich, and W. Streilein. Systematic Analysis of Defenses Against Return-Oriented Programming. In *16th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'13)*, LNCS, pages 82–102, Oct 2013.

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

[54] K. Z. Snow, R. Rogowski, J. Werner, H. Koo, F. Monrose, and M. Polychronakis. Return to the zombie gadgets: Undermining destructive code reads via code inference attacks. In *37th IEEE Symposium on Security and Privacy*, 2016.

[55] B. Spengler. Grsecurity. *Internet [Nov, 2015]. Available on: http://grsecurity.net*, 2015.

[56] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *2nd European Workshop on System Security*, EUROSEC, 2009.

[57] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *IEEE Symposium on Security and Privacy*, 2013.

[58] A. Tang, S. Sethumadhavan, and S. Stolfo. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *ACM Conference on Computer and Communications Security*, CCS, 2015.

[59] V. van der Veen, E. Goktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *37th IEEE Symposium on Security and Privacy*, 2016.

[60] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *ACM Conference on Computer and Communications Security*, CCS, 2012.

[61] J. Werner, G. Baltas, R. Dallara, N. Otternes, K. Snow, F. Monrose, and M. Polychronakis. No-execute-after-read: Preventing code disclosure in commodity software. In *11th ACM Symposium on Information, Computer and Communications Security*, ASIACCS, 2016.

[62] R. Wojtczuk. Subverting the Xen hypervisor. In *Blackhat USA*, BH US, 2008.

[63] Q. Xiao, M. K. Reiter, and Y. Zhang. Mitigating storage

side channels using statistical privacy mechanisms. In *ACM Conference on Computer and Communications Security*, CCS, 2015.

[64] K. Zhang and X. Wang. Peeping tom in the neighborhood: Keystroke eavesdropping on multi-user systems. In *18th USENIX Security Symposium*, USENIX Sec, 2009.

Appendix

A. Nginx Attack 2 Details

Nginx's design employs a master process, which provides signal handling and spawns worker processes to handle requests via `fork` calls. This processing loop is implemented by the `ngx_master_process_cycle` function, which is called from `main` after Nginx configures itself. The trampoline address of this function can be determined via profiling after causing a system call to hang. Since `ngx_master_process_cycle` forks, worker processes inherit the parent's current stack. This includes the return address trampoline of `ngx_master_process_cycle`. Recall that return addresses are replaced with a pointer to a trampoline whose code resembles the following:

```
call  ngx_master_process_cycle
jmp callsite_main
```

The return address points to the `jmp` instruction. From that address, we can easily derive where the `call` instruction is located.

Identifying the relevant return address on the stack is straightforward, as Nginx's initial execution is predictable. The `ngx_master_process_cycle` frame will be near the base of the stack, immediately after the `main` stack frame.

Once the address of `ngx_master_process_cycle` is found, we can take advantage of a function pointer in the Nginx worker's log handler. The `log_error_core` function contains a pointer to a log handler function taking three arguments: `p = log -> handler(log, p, last-p)`. There are multiple system calls in the function prior to the pointer being dereferenced during a logging event, which enables us to hang the program via MTB and corrupt the handler to point instead at `ngx_master_process_cycle`. In order to prevent a program crash, we must also modify the first argument (`log`) to resemble the `ngx_cycle_t` expected by `ngx_master_process_cycle`. The parameter is not used in our attack, so any non-crashing value suffices.

```
ngx_argv[0] = "/usr/bin/python3"
ngx_argv[1] = "-c"
ngx_argv[2] = "import os,socket,subprocess;
   s=socket.socket(socket.AF_INET,
     socket.SOCK_STREAM);
   s.connect((\\\'127.0.0.1\\\',1234));
   [os.dup2(s.fileno(),i) for i in range(3)];
   subprocess.call([\\\'/bin/sh\\\',\\\'-i
   \\\']);"
 ngx_argv[3] = 0
```

Fig. 6. Reverse Shell in Nginx with AOCR

Once we have pointed the log handler at `ngx_master_process_cycle`, we must ensure that the target function's execution causes an `exec` under our control. This can be achieved via the range of signals that Nginx can handle in `ngx_master_process_cycle`. In particular, Nginx provides a `new_binary` signal used to provide rolling updates to a new version of the server without compromising availability. This signal handler is invoked whenever a global integer variable named `ngx_change_binary` is non-zero. The path to the binary is stored in `ngx_argv`, another global variable. By corrupting the first global value we ensure that an `exec` call will eventually be made when the log handler pointer is dereferenced. By corrupting the latter, we ensure that a binary of our choice is executed. For example, setting `ngx_argv` to the values shown in Figure 6 will create a reverse shell bound to a chosen IP address (127.0.0.1 in this case).

B. Apache Attack Details

In order to maintain portability across operating systems, Apache uses its own portable runtime libraries (APR and APR-Util) instead of directly calling functions in libc. However, modules may call functions in this library that the base Apache process does not. The build process must ensure that all APR functions and related utility libraries are linked during compilation whether or not they are explicitly used in the base code. This is achieved via an `exports.c` file for each library. Each of these files contains function pointers to every function in that library. They are linked to the executable during program compilation, and loaded into the data section of memory on execution.

One of these exported functions is `ap_get_exec_line` in Apache's server utility library (`httpd.h`), which takes three arguments: a pointer to a valid memory pool, a command to run, and the arguments to supply that command. We recover the trampoline for this function by profiling while hanging execution via MTB. The region of memory containing pointers from `exports.c` is easily identified, as it contains nothing but function pointers (with common higher-order bits) pointing to functions in one library. The order in which function pointers are declared in `exports.c` is deterministic, so recovering the pointer for `ap_get_exec_line` is straightforward.

Next, we corrupt a function pointer to point to the revealed address. When choosing the pointer, we must ensure that the parameters passed to `ap_get_exec_line` are passed correctly, as this attack does not rely on global variables like the Nginx variant. Additionally, our ability to modify memory is limited to the periods surrounding system calls. Only functions that pass parameters via pointers to memory addresses are viable. Given these criteria we chose to corrupt the `errfn` pointer in `sed_reset_eval`, part of Apache's `mod_sed`. The `errfn` pointer is dereferenced in the `eval_errf` function, which pulls all of its parameters from pointers to memory. Similar functions are available in other modules, should `mod_sed` not be available.

Finally, we set `errfn` to point to `ap_get_exec_line`. The first argument pointer is corrupted to point at a valid `apr_pool_t` object, which the attacker-controller worker will likely already have. (APR pools are used to handle memory allocation in Apache.) The second pointer is made to point at a string containing the path to a binary of our choice. When the `errfn` pointer is dereferenced, the binary is executed.