No-Execute-After-Read: Preventing Code Disclosure in Commodity Software

Jan Werner
RENCI
jjwerner@cs.unc.edu

George Baltas
University of North Carolina
baltas@cs.unc.edu

Rob Dallara
University of North Carolina
dallara@cs.unc.edu

Nathan Otterness
University of North Carolina
otternes@cs.unc.edu

Kevin Z. Snow
University of North Carolina
kzsnow@cs.unc.edu

Fabian Monrose
University of North Carolina
fabian@cs.unc.edu

Michalis Polychronakis
Stony Brook University
mikepo@cs.stonybrook.edu

ABSTRACT

Memory disclosure vulnerabilities enable an adversary to successfully mount arbitrary code execution attacks against applications via so-called *just-in-time* code reuse attacks, even when those applications are fortified with fine-grained address space layout randomization. This attack paradigm requires the adversary to first read the contents of randomized application code, then construct a code reuse payload using that knowledge. In this paper, we show that the recently proposed *Execute-no-Read* (`XnR`) technique fails to prevent just-in-time code reuse attacks. Next, we introduce the design and implementation of a novel memory permission primitive, dubbed *No-Execute-After-Read* (`NEAR`), that foregoes the problems of `XnR` and provides strong security guarantees against just-in-time attacks in *commodity binaries*. Specifically, `NEAR` allows all code to be disclosed, but prevents any disclosed code from subsequently being executed, thus thwarting just-in-time code reuse. At the same time, commodity binaries with mixed code and data regions still operate correctly, as legitimate data is still readable. To demonstrate the practicality and portability of our approach we implemented prototypes for both Linux and Android on the ARMv8 architecture, as well as a prototype that protects unmodified Microsoft Windows executables and dynamically linked libraries. In addition, our evaluation on the SPEC2006 benchmark demonstrates that our prototype has negligible runtime overhead, making it suitable for practical deployment.

1. INTRODUCTION

Despite constant hopes to the contrary, research in computer security has yet to produce a foolproof method for preventing malicious users from exploiting one the most commonplace and necessary components of a computer: its memory. This is certainly not for lack of effort–after all, it was fewer than 20 years ago that non-executable (NX) memory, a staple of modern memory safety, was introduced in response to rampant but trivial attacks in which an attacker could leverage a single stack corruption vulnerability to trigger execution of arbitrary machine code. NX memory succeeded in preventing these *code injection* attacks, which allowed attackers to run arbitrary code of their choice, but failed to prevent code reuse attacks.

In the new wave of memory exploits, the attacker redirects execution to portions of existing benign code, but manipulates inputs or the order of execution in order to carry out malicious behavior. These so-called *code reuse attacks* leverage entire functions, as in *return-to-libc* attacks, or they may reuse small fragments of existing functions, as in the case with *return-oriented* or *jump-oriented* programming attacks [30, 10, 7]. Since early code-reuse attacks required foreknowledge of code addresses, a natural defense was Address Space Layout Randomization (ASLR), which shifts the addresses of blocks of memory by a randomized offset at runtime, theoretically removing attackers' abilities to create code reuse payloads consisting of known code addresses. Soon after, however, the rise of scripting environments in exploitable applications undermined the ability of ASLR to prevent code reuse attacks, due to the availability of memory disclosure vulnerabilities.

In response, further attempts were made to strengthen ASLR, collectively known as *fine-grained* ASLR, which limit the amount of information an attacker can infer from a single memory disclosure. We discuss these in more detail in §2, but for now it is sufficient to say that all the status quo was shown to be ineffective in the face of *just-in-time code reuse* attacks [34], which use multiple memory disclosures to read not only pointers to executable code, but also the code itself.

Thus, in today's feature-rich landscape, it is now abundantly clear that randomization alone is not sufficient to protect applications' memory. In contrast with randomization, over the past several years the security community has revisited the notion of *control-flow integrity* (CFI) originally proposed by Abadi et al. [1]. CFI aims to mitigate attacks at the time of the control-flow pivot, *e.g.,* when initially returning to or calling an arbitrary code pointer controlled by the adversary. Just recently, however, Carlini et al. [9] demonstrated that CFI can be defeated—even when considering a "perfect" implementation—by constructing a form of mimicry attack [39] wherein code reuse payloads make use of commonly called library functions. Since these functions are similarly called in normal program operation, control-flow integrity is preserved by

ASIA CCS '16, May 30-June 03, 2016, Xi'an, China
© 2016 ACM. ISBN 978-1-4503-4233-9/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2897845.2897891

these malicious payloads, and thus they fundamentally bypass the CFI protection [15].

Other recent works [3, 4, 17, 12] have proposed additional protections on top of fine-grained ASLR in order to cope with just-in-time code reuse attacks. Unfortunately, as we will elaborate on later, these protections either inherently suffer from prohibitively poor runtime performance, have weak or undefined security guarantees, or do not provide binary compatibility to support legacy applications. We instead propose a memory primitive that *allows* disclosure of code, but prevents the subsequent execution of that code. Redefining the problem in this way enables us to forgo all the aforementioned shortcomings while still mitigating just-in-time code reuse attacks. Specifically, we make the following contributions in this paper:

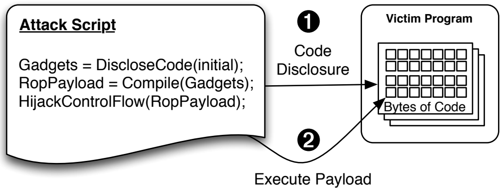
- We introduce the concept of NEAR, a new protection primitive that prevents the execution of previously disclosed code.
- We describe a practical design and implementation of the NEAR primitive using commodity hardware, operating systems and applications.
- We catalog instances of mixed code and data and describe algorithms for reliably identifying and separating a majority of these instances.
- We demonstrate how to leverage the *extended page tables* (EPT) feature on x86, the upcoming Intel Memory Protection Keys (MPK), and ARMv8 memory permissions—all of which support execute-only memory—to provide NEAR capabilities. We also show how to gracefully handle legitimate data reads in code regions.

The remainder of the paper is organized as follows. Background and pertinent related work is presented in §2. We then show that XnR is not effective in defeating just-in-time code reuse attacks in §3. Our goals and assumptions are presented in §4, followed by a description in §5 of the design and implementation of NEAR for various platforms. We provide an extensive evaluation in §6 to demonstrate the power and practicality of our approach. We follow-up with a discussion of limitations in §7 and conclude in §8.

2. BACKGROUND AND RELATED WORK

The key to understanding our work's motivation and design lies in the complex history of low-level defenses and the attacks warranting them. For brevity, we assume that the reader is already familiar with a typical *code injection* attack, where a snippet of malicious code is written directly into memory, and *non-executable memory* (NX), which prevents such attacks by rendering executable memory unwritable. Instead, we begin by discussing *Address Space Layout Randomization*–a defense that arose to counter *code reuse* attacks, which invoke existing benign code in order to carry out malicious activity.

Address-Space Layout Randomization: The concept of address-space layout randomization dates back to Forrest et al. [16], but most implementations in current use such as PaX [38] only randomize the starting address of large chunks of contiguous memory (often referred to as *regions*). Unfortunately, these implementations suffer from several serious problems. First, ASLR is susceptible to brute force attacks on 32-bit systems, as the entropy of possible region locations is small [25, 31]. As alluded to earlier, adversaries can also actively take advantage of in-built scripting of applications to defeat this *coarse-grained* randomization via memory disclosure attacks [29, 35]. In short, one leverages the disclosure vulnerability to reveal the value of a function pointer in a program data section, such as a function callback address on the heap or a return address on the stack. Additionally, one must precompute the offset from the leaked function pointer to necessary ROP gadgets in an offline analysis step. Combining this offset information with the leaked function pointer allows the adversary's script to adjust a predefined ROP payload by adding the offset to each gadget pointer in their payload. Fundamentally, leaking the value of a single code pointer in the data section provides enough information for one to *infer* the location of ROP gadgets.



```
Gadgets = DiscloseCode(initial);
RopPayload = Compile(Gadgets);
HijackControlFlow(RopPayload);
```

Figure 1: An adversary first requests that code is disclosed to generate a ROP payload on-the-fly (step ❶), then redirects program control-flow to execute the gadgets of the payload (step ❷).

Fine-grained randomization, on the other hand, seeks to limit or completely prevent the adversary's ability to infer the location of ROP gadgets given known code pointers. Many fine-grained ASLR approaches have been proposed [26, 21, 41, 18]. The idea underlying all of these works is to not only randomize the memory region locations, but also to shuffle functions, basic blocks, instructions, registers or even the overall structure of code and data. While each instantiation of fine-grained randomization may carry out different subsets of these actions, all implementations achieve their goal of enforcing a policy to prevent the adversary from simply calculating ROP gadget locations using offsets from a single leaked code pointer. Unfortunately, this policy was shortly thereafter shown to be insufficient in preventing all types of code reuse.

Just-in-Time Code Reuse Attacks: Snow et al. [34] demonstrated a new attack paradigm in which the attacker doesn't only leak the *location* of code, but also *the code itself*. Figure 1 gives a high-level overview of this type of attack. In step ❶, a memory disclosure vulnerability is first invoked to obtain a code pointer in a similar manner to attacks bypassing coarse-grained randomization. The major change in the behavior of just-in-time code reuse attacks comes in what follows–rather than attempting to infer locations of ROP gadgets, the attack recursively reads and disassembles code pages in order to find both potential ROP gadgets and references to further code pages. In step ❷ of Figure 1, the attack combines discovered gadgets into a new ROP payload and executes this payload in order to carry out the malicious behavior requested by the attacker. This attack is fundamentally enabled by the attacker's ability to read actual code in addition to code pointers, so a knee-jerk reaction is to design a method for making code regions executable but not readable, preventing the disclosure in step ❶.

Execute-only Memory: Along these lines, a number of defenses have been proposed to prevent the disclosure step of the just-in-time code reuse attack by inhibiting step ❶. Two general approaches have been explored: either inhibit the ability to recursively discover additional code pages or prevent the reading of code entirely. Backes and Nürnberger [3] presented an example of the former, which eliminates pointers in code to additional executable pages. This is accomplished by forcing all code-to-code references to go through an additional translation step, where offsets are stored in a table unavailable to the attacker. This approach has several drawbacks, however. First, it requires access to source code in order to achieve the necessary transformations. Second, it requires use of the x86 segment registers, which are not only being phased out in x86-64 hardware, but also are already in use in Microsoft Windows for several other essential operations. Finally, the greatest weakness of the work proposed by Backes and Nürnberger [3] is the fact that it fails to account for multiple code pointers in non-executable memory, such as return addresses on the stack or function pointers on the heap. Davi et al. [13] demonstrated that such code pointers on the stack and heap are sufficient for constructing a just-in-time code reuse payload, even if code pointers in executable memory are ignored. Davi et al. [13] proposed a strategy to account for this weakness which relies on code-path randomization, but this also requires the existence of secret data regions, which is not necessarily feasible in practice.

Backes et al. [4] attempt a different approach, called `XnR`, which directly blocks reading from executable memory, even assuming the adversary knows the locations of code pages. They note that x86 and ARM architectures do not provide hardware support for execute-only memory, so their solution attempts to emulate execute-only memory in software. They do so by blocking all accesses to executable regions of memory, so that any access will cause a page fault in the kernel. Within the page fault handler, a process is terminated if the access is due to a read, or allowed if the access is due to execution. In order to improve performance, `XnR` can be configured to allow a sliding window of several accessible pages at a time, where, as newer pages are accessed, older pages within the window are made inaccessible again.

While `XnR` [4] is interesting in theory, it suffers from a number of practical weaknesses that are completely overlooked. Intuitively, multi-threaded applications that make use of many libraries will necessarily jump between code pages (*e.g.,* when switching threads or calling APIs) with a higher frequency than the single-threaded, self-contained, programs used in their evaluation. A much bigger problem, however, is the failure to *directly* address applications that contain data mixed into the code section, which has significant performance and security implications. Mixed code and data occur for a number of reasons discussed in §5. Rather than attempting to eliminate this challenge, `XnR` increments a counter for each first instance of a code page read while decrementing on execution. The application is terminated if some threshold is reached, presumably indicating that too many code page reads have occurred. Not only do the authors fail to provide a threshold value, but this thresholding is ill-conceived. We further elaborate on these problems in section §3 with a systematic evaluation of the effectiveness of the approach proposed by Backes et al. [4].

Gionta et al. [17] present another implementation of execute-only memory, but, unlike Backes et al. [4], dedicate significant effort to dealing with readable data in code sections. This approach requires making use of symbol information and disassembly to identify data in code sections prior to runtime. During runtime, execute-only memory is achieved by placing all data and all code in separate caches for executable and data memory. Unfortunately, many modern processors no longer have separate code and data caches, making this approach impractical. Additionally, the necessary static analysis can not always identify all possible pieces of data in an executable, potentially requiring manual identification of code and data for complex closed source binaries.

In Readactor, Crane et al. [12] also make use of execute-only memory, but guarantee code and data separation by instrumenting the LLVM compiler to emit code-only and data-only sections. Enforcement logic is therefore greatly simplified, as any read from a code section can be treated as malicious. Readactor makes use of Intel's Extended Page Tables (EPT) for hardware enforcement of execute-only memory. Being a compiler modification, however, means that this work requires access to source code and can not support legacy closed-source software.

Brookes et al. [8] also provide execute-only memory support, but has a slightly differing objective of protecting kernel code pages, and therefore would not prevent userspace just-in-time code reuse attacks. Like Readactor, ExOShim [8] makes use of Intel's Extended Page Tables to support execute-only memory.

Widespread Adoption Criteria: Despite decades of research into application defenses and a torrent of proposed approaches, only a handful of techniques have ever been widely adopted. Szekeres et al. [36] examine the criteria for widespread adoption and find three main factors affecting the outcome: (1) protection, (2) cost and (3) compatibility. Thus, we align our design and implementation with these factors. Specifically, we aim to enforce a strong protection policy that prevents control-flow hijacking with the combination of DEP, fine-grained ASLR and `NEAR`. Our design ensures that acceptable performance overhead is achievable. We also impose a requirement of *binary compatibility* with modularity support, as Szekeres et al. [36] note that defenses requiring source code, recompilation, or human intervention are impractical, unscalable and too costly. We note that no prior work meets all these criteria. Oxymoron [3] and `XnR` [4] do not strongly enforce the execute $\oplus$ read security policy, while the approach of Gionta et al. [17] requires human intervention for closed-source software and Readactor [12] requires source code and recompilation.

3. ON THE INEFFECTIVENESS OF `XnR`

To highlight the fundamental limitations of the `XnR` approach, we now present a detailed performance and security analysis of the approach described by Backes et al. [4]. To perform that analysis, we implemented a Windows kernel module following the guidelines given in the paper (as well as those observed in the Linux kernel patch made available by the authors); unfortunately, even after repeated requests, we were not able to obtain their Windows implementation. Our implementation includes support for 32-bit Windows because the just-in-time code reuse attacks presented by Snow et al. [34] targeted this platform.

3.1 On Performance

First, we revisit the performance analysis using the same CPU SPEC 2006 [20] benchmark programs listed in Figure 6 of the `XnR` paper [4]. The results of our tests are shown in Table 1.

The performance overhead that we observed for `bzip2, mcf, hmmer` and `astar` benchmarks is similar to the results presented by Backes et al. [4]. However, two of the benchmarks, namely `sjeng` and `h264ref`, incur a significantly higher performance penalty for the recommended window size of $n = 8$. Some indication that these programs cause higher overhead is depicted in the original paper (i.e., `sjeng` induces a 25% overhead at $n = 2$), but no details are given. We observed that the page faults caused by the data reads induced significantly higher overhead than the faults caused by code execution.

To help the reader better understand the cause of the increase in overhead, we present the distribution of page faults observed in a 20 second time frame after starting the benchmark. The results are shown as a heat-map in Figure 2 for a window size of $n = 8$. Each dot represents an access to the specific page outside of the "sliding window" that resulted in a page fault. The darker the color, the more page faults that were induced. Notice that there are clearly more than 8 frequently accessed pages active within a small window of time. The dark horizontal bands in Figure 2 represent the page faults on the pages that are rapidly evicted and added to

CPU SPEC Benchmark	XnR off	XnR-32 ($n = 8$)	Overhead	Execute Faults/s	Read Faults/s
bzip2	466	449	0.5%	325	101
mcf	247	249	1.1%	15898	26
astar	322	328	1.7%	16096	305
hmmer	591	605	2.7%	72302	1937
h264ref	524	1187	126.5%	876872	24964
sjeng	506	3166	526.5%	16548	196164

Table 1: CPU SPEC 2006 benchmark results

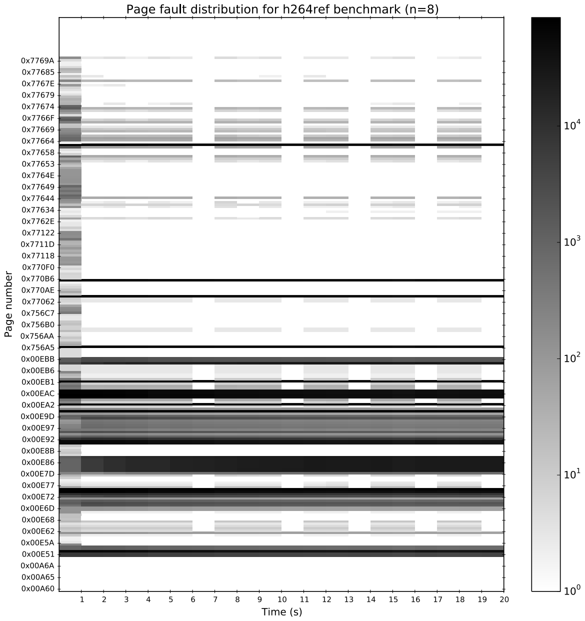


Figure 2: Fault distribution. Gaps in page numbers are due to the specific mapping of modules (e.g., h264ref vs ntdll).

the "sliding window" in every epoch. The presence of those bands demonstrates that the "sliding window" approach based on the code locality assumption is only valid for a small set of applications whose main code path fits completely within the "sliding window." In Figure 2, the code pages in the $0x00A60 - 0x00EBB$ range belong to the h264ref application, whereas pages in the range of $0x756A5 - 0x7769A$ belong to system libraries (ntdll.dll, kernelbase.dll) used by h264ref.

The observant reader would notice that read faults seem to incur a higher penalty that execute faults in Table 1. To better understand why the read faults cause higher performance overhead we performed a static analysis of the sjeng benchmark. We found that some of the jump tables being read by these programs are not located within the page containing the instructions referring to them. Examining the number of mis-predicted branches, we surmise that the high cost of read faults stem from the processor instruction pipeline flushes.[1] In short, these findings indicate that code locality plays a tremendous role in the performance of XnR.

3.2 On Security

Recall that from a security perspective, the heuristic of Backes et al. [4] (that allows code pages to be read, as long as the number of read operations does not exceed some unspecified threshold) claims to defeat just-in-time code reuse attacks. To assess this claim, we experimentally determined the threshold value for running Internet Explorer 10 and navigating to www.google.com. In this case, n is set to 8, as this is the smallest window size where the application correctly operates, and no threshold is chosen. The maximum value observed is then selected as the threshold needed by XnR. Our analysis (shown as the blue line in Figure 3) reveals that the counter quickly peaks to 2487, meaning that there are 2487 consecutive read operations from distinct pages made by a short section of code reading the imports and export directories for various libraries used by IE. The counter value fluctuates as instructions on accessed pages are read or executed. Deeper inspection revealed that the number of execute operations that decrease the counter value is on average 5.9 times higher than read operations. Per the methodology suggested by Backes et al. [4], we set the threshold value of **2487** for the window size $n = 8$ for the subsequent security analyses.

In accordance with Backes et al. [4], the idea is that if the threshold value is exceeded during a just-in-time code reuse attack, XnR will terminate the process and thwart the attack. To test this assertion, we recreated the just-in-time code reuse attack described by Snow et al. [34] by leveraging a memory disclosure vulnerability (CVE-2013-2551) in Internet Explorer 10 on Windows 8. The attack first sprays the heap with objects, then sets the length of an array within one of those objects to the maximum 32-bit integer (the vulnerability lies in the fact that one can set an arbitrary length for this array), thus enabling one to disclose memory at any address by indexing into the array. We read just past the array's original boundary—into data representing the subsequent object—to obtain a virtual table (vtable) address. The initial code pointer is disclosed from the vtable contents to initialize the memory mapping step described by Snow et al. [34]. The first page of code is read byte by byte, additional code pointers are identified from a disassembly of that page, and this process is recursively applied until no new pages are identified. The attack successfully discloses 599 pages, constructs the code reuse payload (which simply executes the Windows calculator) and transfers execution to the payload without ever approaching this threshold.

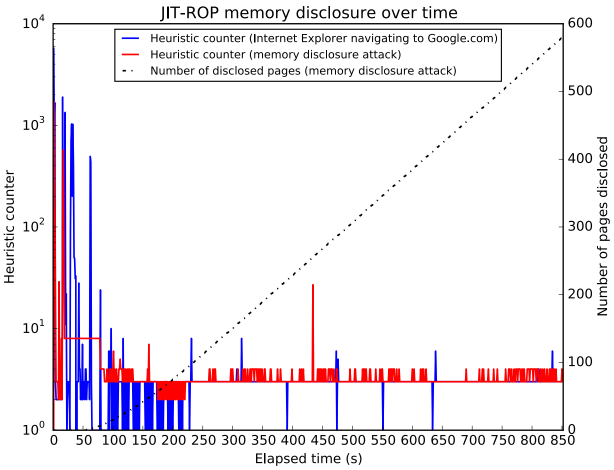


Figure 3: JIT-ROP memory disclosure over time;$n = 8$. Best viewed in color. Benign application (blue), JIT-ROP attack (red).

[1] To arrive at this finding we used the MIS-PREDICTED-BRANCHES counter available using the Intel Vtune Amplifier toolkit. [24]

The red line in Figure 3 shows the counter as the attack proceeds. The number of pages disclosed by the attack is also depicted on that graph. The first spike in the counter is similar to that observed when simply visiting `www.google.com`, while the second occurs during an early stage of the attack where the heap is manipulated. The counter never substantially increments during the attack due to the fact that there exists a code path consisting of more than 8 pages that is executed between subsequent disclosures of each byte. That is, while the counter increments once for each byte disclosed, it also decrements one or more times while executing code in-between those disclosures.

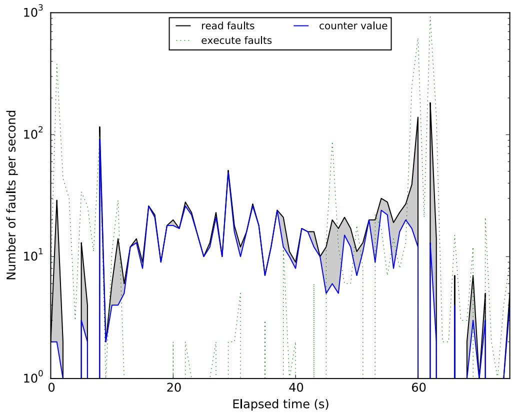


Figure 4: JIT-ROP attack that includes the execution of `nop` instructions per byte read during the attack. Best viewed in color.

Notice that even in this simplistic case with one benign application (IE), there is no threshold value that could be set to thwart the attack without incurring false positives with IE. Just visiting `www.youtube.com`, for example, the counter would need to be set to 4953 at $n = 8$. We experimented with a number of sliding window sizes, but found no threshold value that could be set for n up to 5000 that would thwart just-in-time code reuse.

Lastly, we modified the original JIT-ROP attack to additionally execute a series of `nop` instructions after every byte read. To focus in on just the read operations, we set n to 2500; which is the first value at which the movement of the counter mirrors the read faults. The results are shown in Figure 4. The shaded region shows that the counter never surpases the read faults (which directly influence `XnR`'s choice of a prescribed threshold value) because the execution of the `nop` instructions (green dotted lines) force the counter to decrease. The number of `nop` instructions executed can be arbitrarily set, thereby making it impossible to set an appropriate threshold that would not result in exceedingly high false positives, even for simple benign applications. Consequently, we argue that `XnR` offers little, if any, protection against just-in-time code reuse attacks.

4. GOALS AND ASSUMPTIONS

We now turn our attention to the overall goals and assumptions we impose on our approach. We deliberately attempt to align our goals with the criteria set forth by Szekeres et al. [36] and, in doing so, ensure our approach is practical for widespread adoption. Specifically, we design `NEAR` to have:

- **Strong Security Policy:** `NEAR`'s security policy should prevent all control-flow hijacking attacks that result in arbitrary code execution.
- **Strong Security Policy Enforcement:** Our approach should guarantee this security policy by using DEP to prevent code injection, fine-grained randomization to prevent code reuse, and `NEAR` to prevent just-in-time code reuse. Note that `NEAR` does not prevent code disclosure, but rather prevents the arbitrary code execution that occurs when the adversary attempts to execute their gadgets.
- **Binary and Module Compatibility:** `NEAR` should protect closed-source commercial off-the-shelf (COTS) binaries without the need for symbols or other debug information. Further, complex multi-threaded applications using shared libraries, such as web browsers and document readers should be fully supported. `JIT-ROP` operates on such applications, and so must we.
- **No Human Intervention:** `NEAR` should not require human intervention at any step. Once deployed, any application can be protected without the need for any application-specific pre-computation or manual analysis.
- **Negligible Performance Overhead:** `NEAR` should have an average performance overhead inline with the recommendations of Szekeres et al. [36]. Memory overhead should be negligible as well.
- **Application-Wide Protection:** The `NEAR` primitive should be tunable on a per-application basis such that only user-specified applications are protected to avoid a system-wide performance penalty.

We make several necessary assumptions in this work to support these goals. In particular, we assume the following application-level defenses are already in place for each protected process:

- **Non-Executable Memory:** We assume that writable data sections of application memory, such as the heap and stack, are non-executable and that executable regions are not writable. This protection is widely available–called DEP in Windows, or W⊕X in Linux. Thus, one can neither rewrite existing code or directly inject new code into a protected application.
- **JIT Mitigations:** We assume JIT-spraying [6, 27] mitigations such as JITDefender [11] are in place—that is, one cannot successfully convince an in-built JIT compiler (JavaScript for instance) within the vulnerable application to generate arbitrary malicious logic. Combined with non-executable memory, this assumption ensures that the adversary does not have any avenues for directly injecting malicious code.
- **Fine-grained ASLR:** We assume in-place fine-grained randomization is in use for preventing one from inferring the location of gadgets from a leaked code pointer alone. Furthermore, we assume once executable code has been loaded into memory and randomized using in-place fine-grained ASLR, it is not unloaded from memory.

On the Necessity of Fine-grained Randomization: We emphasize that the assumption of fine-grained randomization is not unique to our approach. Indeed, all the aforementioned works on execute ⊕ read [3, 4, 17, 12] implicitly or explicitly rely on a strong fine-grained randomization implementation. To understand why, consider that the goal of this line of defenses is to prevent one from disclosing instructions on code pages. However, without fine-grained randomization the adversary does not gain any new knowledge from reading the contents of code pages. That is, the use of *coarse-grained randomization*, which randomizes module locations as a single large block of memory, only forces the adversary to leak the value of a single code pointer from a data region. Gadget locations can then be computed with a relative offset from the disclosed pointer without reading any code page. Thus, any execute $\oplus$ read defense is moot without a strong fine-grained randomization implementation forcing the adversary to disclose code pages. We note, however, that fine-grained randomization is still an active area of research orthogonal to our own work and further discuss the state of this research in §7.

Given that non-executable memory and JIT mitigations are already widely deployed, and some practical prototype fine-grained randomization schemes have been described and publicly released [26], we argue that these assumptions are quite realistic. In turn, we define the attacker by assuming an adversary with extraordinary (but achievable) capabilities:

- **Repeated Memory Disclosure:** We assume the adversary can leak the value of any address in the vulnerable application's memory. Moreover, one can repeatedly apply this memory disclosure to reveal an arbitrary number of bytes. We impose no restrictions on how this memory is disclosed, *e.g.* it could be conveyed through in-built scripting (as with JIT-ROP), leaked over a network service (as with the HeartBleed SSL vulnerability), or through some other means.

- **Memory Map Knowledge:** We assume the adversary has a means of learning the overall memory layout of the active instantiation of the vulnerable application. That is, one can use the memory disclosure vulnerability without risk of inadvertently causing a memory access violation when reading an invalid address.

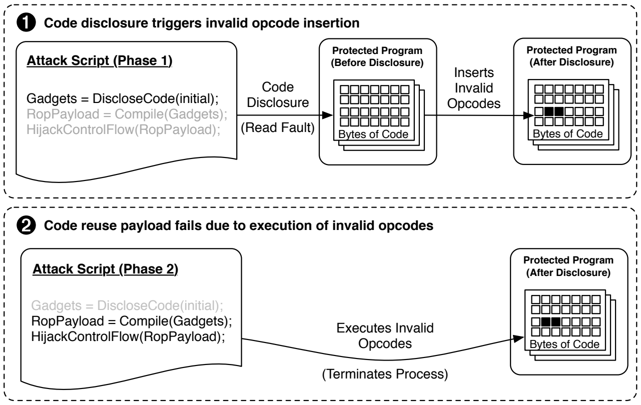
- **Control-Flow Hijacking:** We assume that the adversary can hijack program control-flow, *e.g.,* via overwriting a function return address, virtual table pointer, function callback pointer, or some other means. Further, we assume the adversary can use this hijacking to appropriately perform a *stack pivot* to attempt to execute any code reuse payload that may be generated.

Next, we describe our approach to achieving the aforementioned goals in face of a skilled adversary.

5. OUR APPROACH

As alluded to earlier, merely forbidding read access to program code regions fails to address the fact that commodity binaries and libraries contain data embedded within their code sections. Indeed, ignoring this challenge leads to a false sense of security, as demonstrated in §3. Thus, we take an approach that allows such data to be read, but prevents the subsequent execution of any data read from an executable region. Our memory permission primitive, coined *no-execute-after-read* (NEAR), ensures the following property: *code or data can be executed if and only if it has not been previously read.* As a result, gadgets revealed via memory disclosure cannot be later executed, hence preventing just-in-time code reuse. Independent from our work, Tang et al. [37] proposed a concept (albeit for the x86 platform only) that shares similar design principles with ours. We return to a comparison in §6.

While the NEAR principle is simple in theory, a number of challenges need to be overcome, not the least of which is the seeming lack of hardware features to support an efficient design. In what follows, we describe our design and implementation of NEAR to overcome these challenges and meet the goals set forth in §4.


❶ Code disclosure triggers invalid opcode insertion
Attack Script (Phase 1)
Code Disclosure
(Read Fault)
Protected Program (Before Disclosure)
Bytes of Code
Inserts Invalid Opcodes
Protected Program (After Disclosure)
Bytes of Code
❷ Code reuse payload fails due to execution of invalid opcodes
Attack Script (Phase 2)
Executes Invalid Opcodes
(Terminates Process)
Protected Program (After Disclosure)
Bytes of Code

```
Gadgets = DiscloseCode(initial);
RopPayload = Compile(Gadgets);
HijackControlFlow(RopPayload);
```

```
Gadgets = DiscloseCode(initial);
RopPayload = Compile(Gadgets);
HijackControlFlow(RopPayload);
```

Figure 5: Workflow of NEAR enforcing application protection.

5.1 Protected Program Workflow

Prior to detailing specific hardware features beneficial for our design, we first discuss the overall application workflow we aim to produce with a NEAR-protected application to convey the overall strategy. Figure 5 depicts this workflow, wherein a protected application both makes use of legitimate intermixed code and data and is also exploited by a just-in-time code reuse style attack wherein code is first disclosed, then later executed as a ROP gadget.

Beginning with a legitimate *code disclosure*, the NEAR primitive is notified that the application is attempting to read byte(s) on a code page. As we further elaborate on in the next section, this notification is implemented by forcing a *read exception* on every code page read. Our security policy is enforced by replacing the data read with invalid opcodes. To do so, we allow the read to occur, then force the CPU to single-step, *e.g.* via the machine trap flag feature. Upon regaining control, we read application register state and disassemble the instruction that performed the read. The instruction opcode and operands inform our system of the location and the size of the data read. Then, we save the original data bytes read (within a kernel data structure) and overwrite those bytes in application code with invalid opcodes. We call this process the *data-swap*. As a result, a later execution of those bytes causes an invalid opcode exception and the process is terminated, thus preventing the use of leaked code in a code reuse attack.

On the other hand, if those bytes are read again—due to legitimate program data embedded in the program code region, such as a jump table—we gain control once again through a read exception. At that point we swap the invalid opcodes with the original data and once again use the machine trap flag to allow reading the original data. Data-swapping allows us to gracefully handle legitimate embedded data in commodity applications. This does not compromise security because no code byte that is read at any point during process execution can later be executed and used in an attack. However, our approach introduces performance overhead associated with data reads in the code section. Indeed, each byte, word or double word of data that is read leads to costly read exceptions at run time. We examine the cost of data-swaps in Section 6, but now turn our attention to techniques for implementing code read exceptions on commodity hardware.

5.2 Code Read Notification

The goal of this step is to provide a mechanism in which our system is notified each time a code page is read. This presents a challenge, as the memory management units available with x86 and x86-64 CPUs do not provide permissions at this granularity. *That is, any memory page marked as executable is required to also* *be readable.* Past approaches have proposed marking pages as completely inaccessible to receive kernel-mode faults when code is read (as in the work of Backes et al. [4]). Unfortunately, that approach requires that at least one page, the currently executing page, is available for both execute and read. In practice, more than one page must be made available to remain performant. Further, we seek an approach that does not leave any code accessible to the adversary whatsoever, much less an entire page of code. In what follows, we describe several techniques for implementing code read notification on x86, x86-64, and ARM architectures.

5.2.1 *Enforcing* NEAR *on x86*

On this architecture, we take advantage of the fact that read, write, and execute permissions can be individually applied using the dedicated virtual machine extension called Extended Page Tables (EPT) [22]. EPT maps physical memory pages for hardware virtualized guest virtual machines (VMs).[2] Thus, this approach applies to hardware virtualized VMs, but non-virtualized systems can also leverage EPT by starting a thin hypervisor and transparently virtualizing the running system as the so-called "host domain."[3]

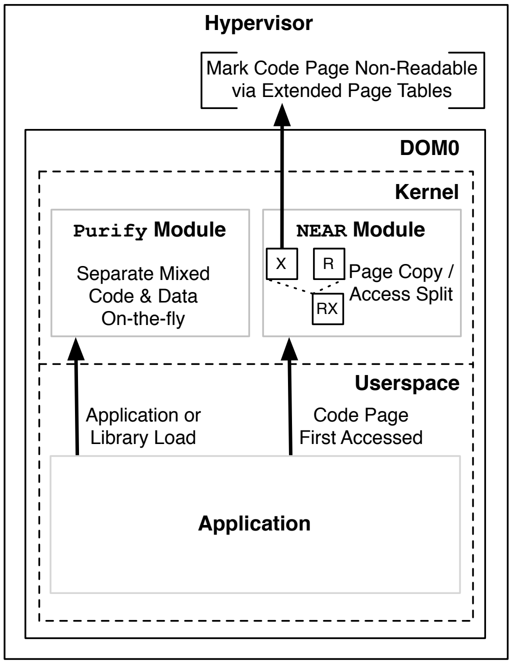

Hypervisor
Mark Code Page Non-Readable via Extended Page Tables
DOM0
Kernel
Purify Module
Separate Mixed Code & Data On-the-fly
NEAR Module
X
R
RX
Page Copy / Access Split
Userspace
Application or Library Load
Code Page First Accessed
Application

Figure 6: Using Extended Page Tables to implement support for code read notification on the x86 architecture.

This hypervisor approach, combined with a host-level kernel module, marks physical pages representing the code of protected processes as execute-only. We identify code regions to protect by first hooking the kernel's page fault handler. We use this hook to observe all page faults and identify code pages as they are made available to the process. We note faults that move a page from the not-present state to the present state, have the execute bit set, and belong to a process we wish to protect. When all these conditions occur we instruct the hypervisor to protect the given physical page. We protect the page by marking it as execute-only using EPT.

One problem with this approach is dealing with shared libraries. By protecting a shared page in one process, all other processes using this library would also be protected. Since our goal is to only protect a subset of processes, we require a solution to this problem. At first we attempted to allocate a copy of the faulting page and assign it to the protected process. This approach works on Linux (as a source code modification), but unfortunately the Windows kernel performs checks within strategic memory management functions to verify the consistency of pages used. When an inconsistent state is observed the OS faults and a reboot is required. This path is therefore infeasible without the ability to modify the closed-source Windows kernel. When a not-present fault occurs on Windows, we instead change the fault reason to a write fault and call the system's original fault handler. Since shared pages are marked with copy-on-write (COW), the system fault handler creates a new page copy on our behalf. In doing so, it also handles all the necessary changes to ensure a consistent page-mapping state.

Figure 6 gives a high-level overview of how this hypervisor-based approach is implemented on x86 Windows. We evaluate the performance of this implementation in §6. While this approach works in practice, the use of virtualization has performance implications and overly complicates the seemingly simple task of providing code read notifications. Fortunately, upcoming hardware features on the x86-64 architecture offer a more straightforward path towards this goal. Unlike prior work [37, 4, 12], we show how our approach can be readily adapted to other architectures.

5.2.2 *Enforcing* NEAR *on x86-64*

Upcoming 64-bit Intel processors include a feature called *Memory Protection Keys* (MPK) [19], which enables finer-grained control of memory protections by userspace processes. MPKs are applied to individual pages of process memory [23]. There are 16 possible keys, and each key supports enabling or disabling writing and accessing whichever pages it is applied to. As applications are unable to directly modify page tables, they cannot directly change which protection keys are applied to which pages. Applications can, however, modify the *meaning* of each of the 16 memory protection keys, by writing to an application-accessible 32-bit register, where each key is assigned one bit to enable or disable writing and another bit to enable or disable reading. Of particular importance, however, is that if both bits are cleared instruction fetches are still enabled, making the memory execute-only as long as the pages would have been otherwise executable. Thus, we anticipate that moving forward, MPKs provide a simpler and more efficient mechanism for implementing code read notification than the aforementioned hypervisor approach. Unfortunately, this hardware is not yet available. Instead, we noted that recent ARM processors include functionally similar features (for our purposes), and thus we implement an ARM version of NEAR to both showcase the portability of our approach and to elaborate on how one could leverage MPKs in the future.

5.2.3 *Enforcing* NEAR *on ARMv8*

The ARMv8-A architecture offers several important benefits over earlier ARM processors, but most importantly for NEAR, it is the first version of ARM processors that offers hardware support for execute-only pages. That is, the memory management unit directly supports the ability to mark memory as execute-no-read. This means that NEAR on ARMv8 only requires kernel modifications to enable the new protection (that is, no hypervisor is needed). While ARMv8, announced in 2011, is still a relatively new architecture, it is the first 64-bit ARM processor and seems likely to see increased usage in the future as mobile devices have increased pro-

[2] EPT is the part of the virtualization extensions VT-x and VT-d introduced by Intel.

[3] We note that NEAR is compatibile with the commodity hardware virtualization that supports nested virtual machines.

cessing requirements and support greater amounts of memory. Several devices are already available with ARMv8 processors, including Google's Nexus 9 tablet, which serves as our test platform, and the newest Apple iPad and iPhone devices.

Also of note is the fact that the ARMv8 MMU is usable regardless of whether we build an ARMv7 or ARMv8 version of the application. This means that despite ARMv7 and ARMv8 running in different compatibility modes on ARMv8 [2], applications for both architectures are still using the same memory management unit and have access to the same features. Thus, we can provide NEAR protection for both ARMv7 and ARMv8 applications, so long as the underlying hardware architecture is ARMv8-A.

In summary, the hardware primitives needed to support NEAR are highly portable, and support for these primitives appears to be improving in upcoming hardware revisions. Unfortunately, all of the primitives for supporting code read notification incur a non-negligible performance overhead in the form of a context-switch on each read, either at the kernel-level for x86-64 and ARMv8, or a virtual machine exit on x86. Therefore, it is imperative that we minimize the total number of code read notifications. To do so, we apply binary code analysis techniques to separate as much code as possible prior to launching an application.

5.3 Separating Mixed Code and Data

The goal of this step is to modify loaded executables on the fly, eliminating data from the executable regions of an application. We call this process *purification* (see Figure 7). This allows us to reduce runtime overhead incurred by programs with mixed code and data protected by NEAR by reducing the total number of code read notifications. Unfortunately, separating code and data on the x86 architecture is provably undecidable [40]. However, a significant amount of research (*e.g.* the work of Wartell et al. [40] and Shin et al. [32]) has been devoted to applying different mechanisms for correctly disassembling x86 binaries.

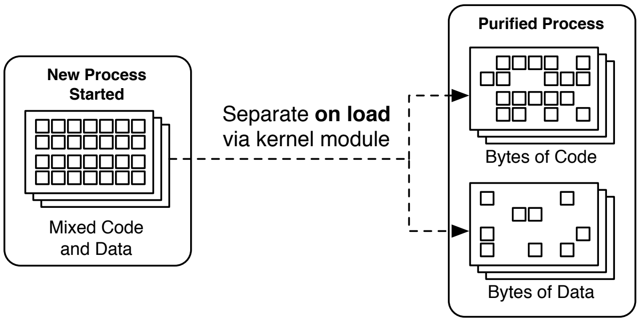


Figure 7: On-the-fly separation of code and data.

Mixed Code and Data: To highlight the problem of mixed code and data we examine two x86 Windows binaries and one ARM Android native library:

1. cryptbase.dll - A windows library providing cryptographic primitives to applications.
2. h264ref.exe - A benchmark derived from a reference implementation of H.264/AVC (Advanced Video Coding) (part of CPU SPEC2006).
3. libmozglue.so - The native library used to implement the vast majority of Firefox's functionality on Android.

The cryptbase.dll library provides various interfaces to DES and AES encryption routines. The library imports 52 symbols and exports 12 symbols. The executable region is 30882 bytes and contains 340 ROP gadgets.[4] A static analysis of the binary provides another interesting insight: the code region contains significant amounts of data. Indeed, 45% of the code region is composed of data—namely, the import address table, import table, export table and constant variables. The biggest contributor to data in the code region are constant *S-Boxes* used in the encryption routines.

The second binary, h264ref.exe, used in the SPEC2006 benchmark suite, is built on-demand on the benchmarked machine using SPEC's default Visual Studio compilation parameters. The binary is statically linked and does not contain any initialized data in its code region; it imports functions only from the NTDLL base system library. Despite that, enabling NEAR protection for this particular benchmark results in a significant performance overhead for the non-purified binary. The performance degradation is due to two jump tables located within the code region.

The last binary, libmozglue.so, serves to demonstrate that the problem of mixed code and data is not limited to the x86 architecture, or the Windows platform. On Android, for example, 8 of the 10 most popular applications make use of native libraries that contain data mixed into the library's code regions. Listing 1 highlights one example of this in the Firefox native library for ARM.

Listing 1: Native assembly code from Firefox for Android:

```
0001baa4 <alloc_set_oom_abort_handler>:
  ; Read a value relative to current PC.
  1baa4: 4b01   ldr r3, [pc, #4]
  1baa6: 447b   add r3, pc
  1baa8: 6018   str r0, [r3, #0]
  ; Returns from the function
  1baaa: 4770   bx lr
  ; Value to be read after return.
  1baac: .word 0x0003461e
```

Data Separation: We opted for a solution based on binary disassembly with additional guidance from metadata available in executable file headers. We implemented these techniques for x86 Portable Executable (PE) executables and libraries. In short, we identify and relocate function Exports and Imports, jump tables, and local function data that is often embedded in code directly adjacent to the function code (as highlighted by the preceding examples). The Exports directory contains the information about the exported symbols of an image. This data is accessed when a process loads a library and populates the Import Address Table and when a process is dynamically loading a library and looking up a specific symbol (for example using LoadLibrary(), GetProcAddress). The imports directory contains information about dependencies on external libraries. This data is used during the process load to populate the Import Address Table. The Import Address Table populated during the image load contains pointers to external functions imported from other dynamically linked libraries. The import address table is accessed during the process runtime every time an imported function is called.

Purification of the above mentioned structures depends on the known addresses of the data structures, which is stored in the binary image header. After the relocation of the Export and Import directories and the Import Address Table is complete, the PE header is

[4] Gadgets determined by the ROPShell tool available at http://www.ropshell.com/ropsearch?h=e97b4515fc3846cb5c6853c40e71ef28

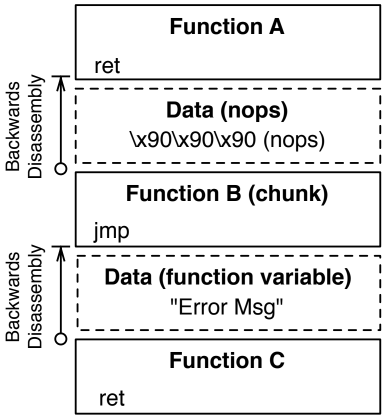


Figure 8: We identify function local variables by performing a backwards disassembly from function entry points (identified via recursive disassembly) until a control-flow instruction is reached.

updated to reflect the added sections and relocated data structures. Details of that process are further discussed in the next section.

Identification of jump tables proved to be more challenging. Jump tables contain the code pointers used to represent C-switch instructions. After inspecting a variety of applications, we observed that the array of pointers generated for a switch statement is usually accessed using the assembly construct of `jmp [base + offset * size_of_word]`. Jump targets located in the jump table are located in the code section to improve the instruction and data fetch locality. Our jump table search algorithm first scans the code section for clusters of two or more consecutive pointers to the analyzed code section. Those clusters are deemed "candidates." To verify the jump table "candidates" the code section is re-scanned in search for `jmp` instructions referencing the pointer cluster. Verified jump tables are relocated to a new, read-only section and the address used by the `jmp` instruction is adjusted to reflect the new location of the jump table.

Finally, we identify function-local data by performing a recursive disassembly of the entire binary. First, we identify gaps between known function code. Then, we identify references to data within these gaps and perform a backwards disassembly up to the first possible control-flow instruction (as in Figure 8), with the idea being that valid code preceding the identified data must invoke a control-flow instruction prior to valid data. We label all bytes between the control-flow instruction and the data reference as data. To relocate the data, we move it to a non-executable section and update any pre-existing references to it to point to that new region. Missing references to the data does not cause program errors, as it is still available for the program to read (but not execute) in its original location; any missed references will incur the performance penalty imbued by the code read notification operation.

We note that our code and data separation heuristics are by no means perfect. Thankfully, perfection is not required as imperfect separation has no bearing on the *security* of `NEAR`. Moreover, as we show in §6 our heuristics prove to be quite practical for meeting our performance and compatibility goals.

5.4 Platform Integration

To demonstrate the portability and practicality of `NEAR` on commodity systems, we implemented prototypes on Windows, Linux and Android for x86 and ARMv8. In what follows we detail the engineering of `NEAR` on each platform to ensure that future research efforts can reproduce our results.

5.4.1 Windows Prototype on x86

This prototype implementation operates on 32-bit Microsoft Windows 7, although nothing prohibits the overall approach from functioning on other platforms, *e.g.* Linux or with a 64-bit architecture. For simplicity, `NEAR` is implemented with two Windows kernel modules—the *hypervisor* module and the *purification* module. The hypervisor module implements a thin hypervisor that supports running the host as a "host domain," as described in §5.2. In addition to providing the hypervisor functionality, this module also hooks the system fault handler by installing a `JMP` hook at the *MmAccessFault* function entry. The Capstone library[5] provides kernel-mode disassembly which we used to determine the operand size of code read operations. Protected processes are configured via a Registry key entry specifying executable names to protect and, when started, are tracked at runtime via their process ID. The hypervisor module is composed of 2824 lines of new kernel-level `C` code.

The purification module handles all separation of mixed code and data as described in §5.3. As previously alluded to, we implemented purification as a kernel module in order to provide this functionality on-the-fly as processes and shared libraries are loaded. This on-load functionality is essential to its practicality, as manually purifying each binary and library offline (as done in Heisenbyte [37]) is prohibitive, due to possible human errors and sheer volume of binaries requiring purification. This module triggers its code purification routine using hooks on *ZwCreateFile*, *ZwOpenFile*, *ZwCreateSection*, *ZwOpenSection* and *ZwMapViewOfSection* system calls. The hooks are installed by replacing their entries in the System Service Descriptor Table (SSDT). `NTDLL` is purified early in the boot process by hooking *PspLocateSystemDll* with a `JMP` instruction at function entry. To do so, we replace the given location string with a path leading to our purified version of `NTDLL`. Finally, since the Windows kernel verifies integrity of loaded libraries, we temporarily disable these checks when loading a purified library by toggling the *g_CiEnabled* kernel variable. The purification module consists of 2500 lines of new kernel-level `C` code.

5.4.2 Android and Linux Prototype on ARM

To implement code read notifications on ARMv8-A we patched the Linux and Android (for Nexus 9) kernels to make use of the execute-only permission. To do so, we defined the execute-no-read permission bits in *pgtable.h*. Next, we inserted a call to our own code into `do_page_fault` in *fault.c*. We handle page faults only when they are due to an attempted read of an execute-only region of memory; all other cases are handled as before by the kernel. We also inserted our own code into `single_step_handler` in *debug_monitors.h* and `do_mmap_pgoff` in *mmap.c* to, respectively, handle single step faults and to apply execute-only permissions to executable sections of files as they are being loaded into memory. Our changes to the kernel consist of about 1000 lines of code.

The high-level procedure for handling a page fault in the `NEAR`-enabled ARM Android kernel is similar to the procedure used in our x86 Windows prototype–we restore the original copy of a page's data (or make a copy for a given page if we have not previously) and set the page's memory permissions to allow it to be read. Next, we set the userspace process to single step and return from the page fault. After the read completes, we catch the single step fault, restore the page's execute-only protection, and replace the data being read with invalid opcodes.

One area in which our ARMv8 prototype differs from our x86 version is that, for some simple instructions, we may choose to emulate the read operation directly in the page fault handler. This

[5] Available at http://www.capstone-engine.org/

improves performance of the emulated instructions because we no longer need to enable reading, single step, and restore execute-only permissions in order to perform a single read from memory, as all of these are relatively complex operations. The ARMv8 architecture is more conducive to the addition of basic emulation than x86 because instructions are fixed width and located at aligned offsets in memory. Additionally, the ARM architecture has a relatively limited set of instructions that are used to read from memory (generally grouped under the LDR assembler mnemonic). In our page fault handler, therefore, we read the instruction responsible for the faulting read and check it against a set of bitmasks to determine if it can be emulated. Simpler instructions, such as a single read from an unsigned PC-relative immediate offset, only require copying the data from the faulting address into a given register, which can be quickly determined by examining bits in the instruction. One convenient aspect of our approach is that we can leave more complicated instructions to be handled by the default single-step approach, rather than risk emulation mistakes or potential performance degradation in cases where emulation may be slower than single-stepping.

6. EVALUATION

For the experiments that follow, we evaluated our approach using a 32-bit version of Windows 7 Professional on a Dell Optiplex 990 powered by an Intel Core i7 CPU running at 3.4 GHz with 4GB of RAM. The system was configured to use only one physical core.

Recall that our goal is to provide a security guarantee wherein previously disclosed executable memory can not be executed. To test that the desired functionality was achieved, we implemented a naïve application that first reads memory from a location leaked via a memory disclosure and then attempts to directly invoke the function whose pointer was previously disclosed. Since the NEAR protection replaces the disclosed code with an invalid opcode (*i.e.,* HLT instruction), the operating system terminates the process raising the General Protection Fault (GPF) as soon as the control is transferred. Next, to further evaluate our ability to thwart advanced memory disclosure attacks, we also applied the same JIT-ROP style attack described in §3 that defeats XnR. In this case, Internet Explorer 10 and its prerequisite DLLs were protected using NEAR. The attack fails as soon as JIT-ROP [34] enters step ❷ of Figure 1.

The purification process separates different types of data embedded in the executable sections of programs. To gauge the impact that moving the different classes of data regions had on our runtime performance we re-examined all the CPU-SPEC 2006 benchmark programs. Curiously, we found that Imports and Exports directories and Import Address Tables were not present in the executable sections of these programs. This is because the C compiler provided with Microsoft Visual Studio 2013 places the aforementioned objects in a read-only section. Additionally, to our surprise, we observed no initialized data embedded in the text section of the benchmark binaries (unlike some of the real-world libraries (*e.g.*, the cryptbase.dll and libmozglue.so examples in §5)). Jump tables, on the other hand, were prevalent, and their movement had a significant impact on our runtime performance.

Table 2 shows the number of jump tables that were found and purified by our discovery algorithm (see §5.3), their size in bytes, and the number of EPT faults triggered by a purified versus non-purified binary. The reason for the significant amount of faults triggered by h264ref benchmark was explained in section §5.3. Notice that for the sjeng benchmark the amount of EPT faults was reduced by six orders of magnitude—which directly translates into a reduction in runtime overhead from 4084.7% to 1.10%.

Benchmark	EPT faults pre-purify	EPT faults post-purify	Jump tables	Purified bytes
400.perlbench	1.7B	164,495	183	12651
401.bzip2	2.5M	136,623	18	544
403.gcc	600M	2.3M	1067	73927
429.mcf	175,562	2376	14	320
445.gobmk	1.1M	127,343	51	2770
456.hmmer	2.5M	1.6M	28	1012
458.sjeng	1.6B	2405	31	948
464.h264ref	317M	25M	33	828
471.omnetpp	2.2M	100,349	38	1285
473.astar	689,231	322,440	15	376
483.xalancbmk	1.7B	3.3M	146	5437
IE v10	16.6M	5.9M	1926	106679

Table 2: Impact of purification on the selected programs.

CPU SPEC benchmark	Heisenbyte (enforcement)	NEAR (enforcement)
400.perlbench	61.57%	3.04%
401.bzip2	0.0%	1.21%
403.gcc	35.64%	19.88%
429.mcf	19.26%	4.04%
445.gobmk	0.85%	0.99%
456.hmmer	6.86%	1.81%
458.sjeng	1.06%	1.10%
464.h264ref	0.23%	7.63%
471.omnetpp	11.44%	2.32%
473.astar	5.41%	2.13%
483.xalancbmk	36.15%	5.51%
Average:	16.48%	5.72%

Table 3: Performance overhead on x86.

Table 3 provides a performance comparison to the concurrent work reported by Tang et al. [37]. Our performance is significantly better than that of Heisenbyte, which we attribute to our efforts to better understand the intricacies of code and data intermingling, and take action to rectify that impact whenever possible. Our overhead ranges from 0.99% (for 445.gobmk) to 19.88% (gcc).

To provide the reader with a better feel for the overhead of NEAR on real-world applications, we used an online browser benchmarking tool known as Peacekeeper. The Peacekeeper benchmark measures the performance of several aspects of a browser, including how quickly it renders HTML5 video, how well it performs 3D rendering, the speed of commonly used text parsing operations, and the time it takes to navigate to dynamically generated pages. In short, its tests measure the browser's ability to render and modify specific elements used in typical web pages, and does so by manipulating the DOM tree in real-time[6]. The results in Table 4 shows that our overhead (of 4.7%, on average) is barely noticeable on x86 for common browser-related tasks.

Peacekeeper Benchmark	Baseline	NEAR Enforcement	% overhead
Internet Explorer 10	2133	2020	5.2%
Google Chrome v45.0.2454	3963	3788	4.4%
Mozilla Firefox v41.0.11	4719	4285	4.6%

Table 4: Browser performance benchmark results

[6] See http://peacekeeper.futuremark.com/faq.action for more info.

Lastly, our empirical results in Table 5 show that NEAR incurs a modest memory overhead for protected processes. Recall that each executable page of a protected process must have a read-only copy created when the page is first rolled into memory. In the set of tested applications, we observed between 100 and 500 additional pages of memory created per process, which is insignificant when observed in the context of the size of the working set of these processes (*i.e.*, from 60 to 933 megabytes).

CPU SPEC benchmark	Total memory (kB)	Allocated due to NEAR (kB)
400.perlbench	594,944	1352 (0.22%)
401.bzip2	876,544	448 (0.05%)
403.gcc	954,368	1828 (0.19%)
429.mcf	864,256	424 (0.04%)
445.gobmk	29,696	1072 (3.6%)
456.hmmer	61,440	596 (0.97%)
458.sjeng	184,320	488 (0.26%)
464.h264ref	69,632	764 (1.09%)
471.omnetpp	124,928	1108 (0.88%)
473.astar	320,512	520 (0.16%)
483.xalancbmk	348,160	1628 (0.46%)

Table 5: Memory overhead

7. LIMITATIONS

Our proof-of-concept does not support writable pages and so does not presently offer protection for self modifying programs. That said, we expect that the protection model offered by NEAR may be extended to allow for writable code pages without changing the invariant that code cannot be executed after it was read.

Similar to XnR [4] and Heisenbyte [37], NEAR does not prevent indirect memory disclosure attacks such as return to libc [14] or COOP [28]. Protection against such attacks could be introduced with a significant performance impact using binary instrumentation to rewrite all the code pointers on the stack and the heap. However, NEAR offers an indirect protection against Blind ROP attacks [5]. The threat model of Blind ROP attack is different than JIT-ROP (*i.e.*, multiple disclosures crashing the process vs multiple disclosures without crashing a process). An adversary could potentially discover a useful ROP gadget, but attempts to execute the gadget would be thwarted by fine grained ASLR. That said, if the assumption in §4 about unloading and reloading of code from memory is violated, then both Heisenbyte and NEAR are subject to the code-inference attacks proposed recently by Snow et al. [33].

While NEAR delivers a solution to address the issue of mixed code and data, more research and engineering effort could be directed at the process of generating binaries. Previously, Backes et al. [4] suggested modifying the Linux linker to remove the non-executable header from executable section, while Crane et al. [12] proposed modifying the LLVM compiler to change the code emitted for jump tables. If such modifications are available, then our protection mechanism could be simplified even further.

8. CONCLUSION

We present a novel technique that prevents the execution of previously disclosed code. Our approach, dubbed *No-Execute-After-Read* (NEAR) thwarts a dominant class of attacks that rely on memory disclosures to disassemble code in memory in order to build a code-reuse attack [34]. We showed how using existing hardware primitives for x86 processors—as well as hardware support for ARMv8 memory permissions and recently introduced Intel Memory Protection Keys—NEAR can meet the three main factors that limit the wide spread adoption of security technologies [36]. Our protection is based on a strong security foundation, and our runtime overhead is comparable to, or better than, contemporary approaches that also aim to thwart memory disclosure attacks.

9. CODE AVAILABILITY

To enable reproducibility of our results, and to encourage further research in this area, the thin hypervisor we developed for this work is available under an open source license at *github.com/uncseclab*

10. ACKNOWLEDGMENTS

We express our gratitude to Alex Blate, Michael Deakin, Kedrian James, Vance Miller, Micah Morton and Teryl Taylor for insightful discussions. We also thank the anonymous reviewers for their suggestions on how to improve the paper. This work is supported in part by the National Science Foundation under awards 1421703 and 1127361 (with a supplement from the Department of Homeland Security under its Transition to Practice program), and the Office of Naval Research under award N00014-15-1-2378. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Department of Homeland Security, or the Office of Naval Research.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and Systems Security*, vol. 13, no. 1, pp. 1–40, 2009.

[2] *ARM Cortex-A57 MPCore Processor Technical Reference Manual*, ARM, 2013.

[3] M. Backes and S. Nürnberger, "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing," in *USENIX Security Symposium*, 2014, pp. 433–447.

[4] 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*, 2014, pp. 1342–1353.

[5] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *IEEE Symposium on Security and Privacy*, 2014, pp. 227–242.

[6] D. Blazakis, "Interpreter exploitation: Pointer inference and jit spraying," in *Black Hat DC*, 2010.

[7] T. K. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack." in *ACM Symposium on Information, Computer and Communications Security*, 2011.

[8] S. Brookes, R. Denz, M. Osterloh, and S. Taylor, "Exoshim: Preventing memory disclosure using execute-only kernel code," in *International Conference on Cyber Warfare and Security*, 2016.

[9] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *USENIX Security Symposium*, 2015, pp. 161–176.

[10] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 559–572.

[11] P. Chen, Y. Fang, B. Mao, and L. Xie, "JITDefender: A defense against jit spraying attacks," in *IFIP International Information Security Conference*, 2011, pp. 142–153.

[12] 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 *IEEE Symposium on Security and Privacy*, 2015, pp. 763 – 780.

[13] 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 *Symposium on Network and Distributed System Security*, 2015.

[14] S. Designer, "return-to-libc attack," *Bugtraq, Aug*, 1997.

[15] 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*, 2015, pp. 901–913.

[16] S. Forrest, A. Somayaji, and D. Ackley, "Building diverse computer systems," in *Hot Topics in Operating Systems*, 1997, p. 67.

[17] J. Gionta, W. Enck, and P. Ning, "Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities," in *Proceedings of the ACM Conference on Data and Application Security and Privacy*, 2015, pp. 325–336.

[18] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *USENIX Security Symposium*, 2012, pp. 475–490.

[19] D. Hansen, "[RFC] x86: Memory protection keys," 2015.

[20] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[21] J. D. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd my gadgets go?" in *IEEE Symposium on Security and Privacy*, 2012, pp. 571–585.

[22] *Best Practices for Paravirtualization Enhancements from Intel Virtualization Technology: EPT and VT-d*, Intel, 2015.

[23] *Intel ®64 and IA-32 Architectures Software Developer's Manual: Combined Volumes 1, 2A, 2B, 2C, 3A, 3B and 3C*, Intel, 2015.

[24] *Intel VTune Amplifier 2016*, Intel, 2015.

[25] L. Liu, J. Han, D. Gao, J. Jing, and D. Zha, "Launching return-oriented programming attacks against randomized relocatable executables," in *IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 2011, pp. 37 – 44.

[26] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *IEEE Symposium on Security and Privacy*, 2012, pp. 601–615.

[27] C. Rohlf and Y. Ivnitskiy, "Attacking clientside JIT compilers," in *Black Hat USA*, 2011.

[28] 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 *IEEE Symposium on Security and Privacy*, 2015, pp. 745 – 762.

[29] F. J. Serna, "The info leak era on software exploitation," in *Black Hat USA*, 2012.

[30] 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*, 2007, pp. 552–561.

[31] H. Shacham, E. jin Goh, N. Modadugu, B. Pfaff, and D. Boneh, "On the effectiveness of address-space randomization," in *ACM Conference on Computer and Communications Security*, 2004, pp. 298–307.

[32] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *USENIX Security Symposium*, 2015, pp. 611–626.

[33] K. 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 *IEEE Symposium on Security and Privacy*, 2016.

[34] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *IEEE Symposium on Security and Privacy*, 2013, pp. 574–588.

[35] A. Sotirov and M. Dowd, "Bypassing browser memory protections in Windows Vista," 2008.

[36] L. Szekeres, M. Payer, T. Wei, and D. Song, "SOK: Eternal War in Memory," in *IEEE Symposium on Security and Privacy*, 2013, pp. 48–62.

[37] A. Tang, S. Sethumadhavan, and S. Stolfo, "Heisenbyte: Thwarting memory disclosure attacks using destructive code reads," in *ACM Conference on Computer and Communications Security*, 2015, pp. 256–267.

[38] P. Team, "Pax address space layout randomization (aslr)," Sep. 2015.

[39] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *ACM Conference on Computer and Communications Security*, 2002, pp. 255–264.

[40] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham, "Differentiating code from data in x86 binaries," in *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases*, 2011, pp. 522–536.

[41] 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*, 2012, pp. 157–168.

APPENDIX

A. BENCHMARK CONFIGURATION

We compiled the CPU SPEC2006 benchmark programs with the Microsoft Visual Studio 2013 C/C++ compiler using the default compiler and linker options listed in the benchmark suite. To allow for direct comparisons with prior work, we used the "base" configuration which is an unoptimized variant. For our empirical evaluations, we ran three iterations of all the eleven benchmark programs and measured the average execution time. Since we also need to report performance results for the "purified" version of the benchmark programs, we disabled the integrity check for the binaries by setting the configuration switch `check_md5` to zero.