2015 IEEE Symposium on Security and Privacy

Readactor: Practical Code Randomization Resilient to Memory Disclosure

Stephen Crane*, Christopher Liebchen†, Andrei Homescu*, Lucas Davi†,
Per Larsen*, Ahmad-Reza Sadeghi†, Stefan Brunthaler*, Michael Franz*
*University of California, Irvine.
{sjcrane,ahomescu,perl,s.brunthaler,franz}@uci.edu
†CASED/Technische Universität Darmstadt, Germany.
{lucas.davi,christopher.liebchen,ahmad.sadeghi}@trust.cased.de

***Abstract*—Code-reuse attacks such as return-oriented programming (ROP) pose a severe threat to modern software. Designing practical and effective defenses against code-reuse attacks is highly challenging. One line of defense builds upon fine-grained code diversification to prevent the adversary from constructing a reliable code-reuse attack. However, *all* solutions proposed so far are either vulnerable to memory disclosure or are impractical for deployment on commodity systems.**

In this paper, we address the deficiencies of existing solutions and present the first practical, fine-grained code randomization defense, called Readactor, resilient to both static and dynamic ROP attacks. We distinguish between *direct* memory disclosure, where the attacker reads code pages, and *indirect* memory disclosure, where attackers use code pointers on data pages to infer the code layout without reading code pages. Unlike previous work, Readactor resists both types of memory disclosure. Moreover, our technique protects both statically and dynamically generated code. We use a new compiler-based code generation paradigm that uses hardware features provided by modern CPUs to enable execute-only memory and hide code pointers from leakage to the adversary. Finally, our extensive evaluation shows that our approach is practical—we protect the entire Google Chromium browser and its V8 JIT compiler—and efficient with an average SPEC CPU2006 performance overhead of only 6.4%.

I. Introduction

Design and implementation of practical and resilient defenses against code-reuse attacks is challenging, and many defenses have been proposed over the last few years. So far, these defense mechanisms can roughly be classified into two primary categories: control-flow integrity (CFI) and code randomization. CFI, when properly implemented [2], prevents attackers from executing control-flow edges outside a static control-flow graph (CFG) [3]. However, fine-grained CFI solutions suffer from performance problems and the precision of the CFI policy is only as good as that of the underlying CFG. Obtaining a completely precise CFG is generally not possible, even with source code. Recent work on control-flow integrity has therefore focused on coarse-grained solutions that trade security for performance [17, 25, 50, 67, 69]. Unfortunately, all of these solutions have been successfully bypassed due to their imprecise CFI policies [13, 20, 29, 30, 55].

Code randomization (see [40] for an overview), on the other hand, has suffered a blow from information disclosure, which breaks the fundamental memory secrecy assumption of randomization, namely, that the code layout of a running program is unknown to attackers [61]. We distinguish between two types of memory disclosure: *direct* and *indirect*. In a direct memory disclosure attack, the adversary reads code pages directly and mounts a return-oriented programming (ROP) attack based on the leakage of code pointers embedded in instructions residing on code pages, as shown in the just-in-time code-reuse (JIT-ROP) attack [59]. In an indirect memory disclosure attack, the adversary reads multiple code pointers that are located on data pages (e.g., stack and heap) to infer the memory layout of an application (as we show in an experiment in Section III).

Since randomization is known to be efficient [22, 34], recently proposed defenses [6, 7] focus on reducing or eliminating memory disclosure. For instance, Oxymoron [6] aims at hiding direct code and data references in instructions, whereas Execute-no-Read (XnR) marks all memory pages (except a sliding window) as non-accessible to prevent memory pages from being dynamically read and disassembled [7]. However, information disclosure is surprisingly hard to prevent. As we explain in Section III, none of these techniques provide sufficient protection against memory disclosure and can be bypassed. They are also not sufficiently practical to protect complex applications such as web browsers that contain just-in-time compilers. Finally, we note that Szekeres et al. [62] propose a different approach called Code-Pointer Integrity (CPI) which separates code pointers from non-control data. Kuznetsov et al. [39] implement CPI by placing all code pointers in a secure region which (in 64-bit mode) is hidden by randomizing its offset in the virtual address space. However, Evans et al. [23] successfully bypass this CPI implementation using side-channel attacks enabled by the large size of the secure region.

Goals and contributions. In this paper, we focus on code randomization. Our goal is to tackle the shortcomings of existing defenses by closing memory disclosure channels while using a reasonable granularity of code randomization. We classify information disclosure sources into direct and indirect memory leakage. We then present the design and implementation of Readactor, the first practical fine-grained code randomization defense that resists both classes of memory disclosure attacks. Our defense combines novel compiler transformations with a hardware-based enforcement mechanism that prevents adversaries from reading any code. Specifically, we use virtualization support in current, commodity Intel processors to enforce execute-only pages [35]. This support allows us to avoid two important shortcomings of prior work [7, 27]: either requiring a sliding window of readable code or legacy hardware, respectively. Our main contributions are:

- **Comprehensive ROP resilience.** Readactor prevents all existing ROP attacks: conventional ROP [58], ROP without returns [14], and dynamic ROP [9, 59]. Most importantly, Readactor improves the state of the art

DOI 10.1109/SP.2015.52

in JIT-ROP defenses by preventing indirect memory disclosure through code-pointer hiding.

- **Novel techniques.** We introduce compiler transformations that extend execute-only memory to protect against the new class of indirect information disclosure. We also present a new way to implement execute-only memory that leverages hardware-accelerated memory protections.
- **Covering statically & dynamically generated code.** We introduce the first technique that extends coverage of execute-only memory to secure just-in-time (JIT) compiled code.
- **Realistic and extensive evaluation.** We provide a full-fledged prototype implementation of Readactor that diversifies applications, and present the results of a detailed evaluation. We report an average overhead of 6.4% on compute-intensive benchmarks. Moreover, our solution scales beyond benchmarks to programs as complex as Google's popular Chromium web browser.

II. Return-Oriented Programming

In general, code-reuse attacks execute benign and legitimate code to perform illegal actions. To do so, the adversary exploits a memory corruption error (such as a buffer overflow) to transfer control to existing instruction sequences that are chained together to perform the malicious behavior.

The most common code-reuse technique is return-oriented programming (ROP) [53]. The basic idea of ROP is to invoke short instruction sequences (gadgets, in ROP parlance) one after another. To successfully launch an attack, the adversary first needs to identify—using an offline static analysis phase—which gadgets and library functions satisfy the attack goal. Once all gadgets are identified, the adversary injects pointers into the data area of the application, where each pointer references a gadget.

For a conventional stack-overflow vulnerability, the adversary writes the pointers onto the stack and overwrites the return address of the vulnerable function with the address of the first gadget. This can be achieved by overflowing a stack-allocated buffer and writing a new pointer address to the stack slot containing the return address.

Once the vulnerable function executes a return instruction, the control flow is redirected to the first gadget, which itself ends with a return instruction. Return instructions play an important role[1] as they are responsible for chaining multiple sequences together. This attack principle has been shown to be Turing-complete, meaning that the adversary can perform arbitrary, malicious computations [53].

III. The Threat of Memory Disclosure

Simple code randomization such as address space layout randomization (ASLR) complicates ROP attacks by randomizing the base addresses of code segments. Hence, the adversary must guess where the required instruction sequences reside in memory. Recent research has shown that randomization

[1]ROP does not necessarily require return instructions, but can leverage indirect jumps or calls to execute a chain of ROP gadgets [12, 14].

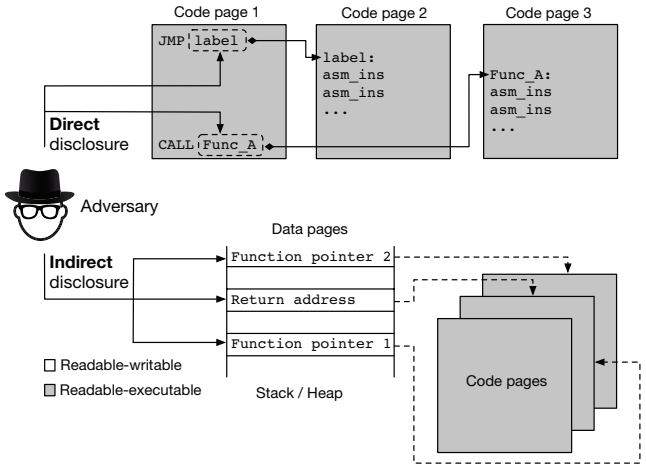

Figure 1: Direct and indirect memory disclosure.

at the level of functions, basic blocks, or individual instructions enhances security (see [40] for a detailed overview of fine-grained code randomization) relative to ASLR because these approaches randomize the internal code structure of an application.

However, the adversary can sometimes use memory disclosure vulnerabilities to learn the memory layout and randomized locations of machine code in an application. Using this information, the adversary can reliably infer the runtime addresses of instruction sequences and bypass the underlying code randomization. In general, the adversary can launch direct and indirect memory disclosure attacks; Figure 1 illustrates both classes of disclosure.

In a direct memory disclosure attack, the adversary is able to directly read code pointers from code pages. Such pointers are typically embedded in direct branch instructions such as direct jumps and calls. The top of Figure 1 shows how the adversary can access a single code page (code page 1), dynamically disassemble it, and identify other code pages (pages 2 and 3) via direct call and jump instructions. By performing this recursive disassembly process on-the-fly, the adversary can directly disclose all gadgets needed to relocate a ROP attack to match the diversified code [59].

Two protection methods have been proposed to prevent direct memory disclosure: rewriting inter-page references and redirecting attempts to read code pages. In the first approach, direct code references in calls and jumps between code pages are replaced by indirect branches to prevent the adversary from following these code pointers [6]. A conceptually simpler alternative is to prevent read access to code pages that are not currently executing [7], e.g., code page 2 and 3 in Figure 1.

Unfortunately, obfuscating code pointers between pages does not prevent *indirect* memory disclosure attacks, where the adversary only harvests code pointers stored on the data pages of the application which are necessarily readable (e.g., the stack and heap). Examples of such pointers are return addresses and function pointers on the stack, and code pointers in C++ virtual method tables (vtables). We conducted experiments that indicate that the adversary can bypass countermeasures that

only hide code pointers in direct calls and jumps. We collected code addresses from virtual table pointers on the heap, and disassembled code pages to identify useful gadgets, similar to the original JIT-ROP attack. We found 144 virtual function pointers pointing to 74 code pages in IE 8 and showed that it is possible to construct a JIT-ROP attack from those 74 code pages [21]. We call this updated attack *indirect JIT-ROP* to distinguish it from the original JIT-ROP attack that directly reads the code layout.

We must also consider whether preventing read access to code pages suffices to protect against indirect memory disclosure vulnerabilities. Since code pages are not readable, the adversary cannot disassemble code to construct an attack. The adversary still gains information from leaked code pointers, however, as our experiment on indirect JIT-ROP demonstrates. By leaking pointers to known code locations, the adversary is able to infer the contents of code surrounding the pointer targets. The severity of this threat depends on the type of code randomization that is deployed in conjunction with non-readable code pages. For example, if function permutation is used, each leaked code pointer allows the adversary to correctly infer the location and the entire content of the function surrounding the leaked code address, since there is no randomization used within functions. Thus, the security of making code pages non-readable depends on the granularity of the code randomization.

IV. Adversary Model and Assumptions

Our defense against all known variants of return-oriented programming attacks builds on the following assumptions and adversary model:

- The target system provides built-in protection against code injection attacks. Today, all modern processors and operating systems support data execution prevention (DEP) to prevent code injection.
- The adversary cannot tamper with our implementation of Readactor.
- The adversary has no a priori knowledge of the in-memory code layout. We ensure this through the use of fine-grained diversification.
- The target program suffers at least from one memory corruption vulnerability which allows the adversary to hijack the control-flow.
- The adversary knows the software configuration and defenses on the target platform, as well as the source code of the target application.
- The adversary is able to read and analyze any readable memory location in the target process.

Our adversary model is consistent with prior offensive and defensive work, particularly the powerful model introduced in JIT-ROP [59].

We cannot rule out the existence of timing, cache, and fault side channels that can leak information about the code layout to attackers. Although information disclosure through side-channels is outside the scope of this paper we note that Readactor mitigates recent remote side-channel attacks against diversified code since they also involve direct memory disclosure [9, 57].

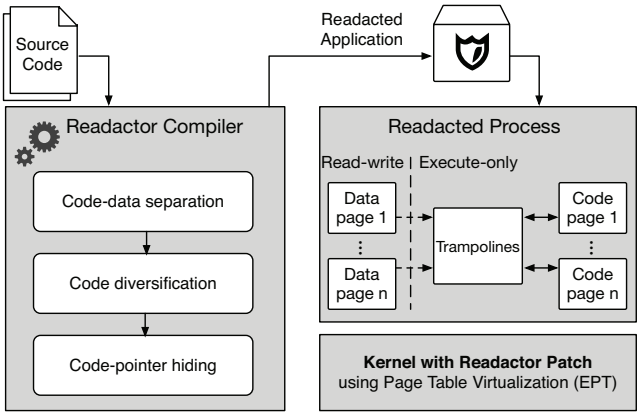

Figure 2: System overview. Our compiler generates diversified code that can be mapped with execute-only permissions and inserts trampolines to hide code pointers. We modify the kernel to use EPT permissions to enable execute-only pages.

V. Readactor Design

Overview: Readactor protects against both direct and indirect disclosure (see Section III). To handle attacks based on direct disclosure, it leverages the virtualization capabilities in commodity x86 processors to map code pages with execute-only permissions at *all* times. Hence, in contrast to previous related work [7], the adversary cannot read or disassemble a code page at *any* time during program execution. To prevent indirect disclosure attacks, Readactor hides the targets of all function pointers and return addresses. We hide code pointers by converting these into direct branches stored in a dedicated *trampoline* code area with execute-only permissions. Code-pointer hiding allows the use of practical and efficient fine-grained code randomization, while maintaining security against indirect memory disclosure.

Figure 2 shows the overall architecture of Readactor. Since our approach benefits from precise control-flow information, which binary analysis cannot provide, we opt for a compiler-based solution. Our choice to use a compiler also improves the efficiency and practicality of our solution. In particular, our technique scales to complex, real-world software: the Chromium web browser and its V8 JavaScript engine (see Section X). However, diversification could instead be done at program load-time, protecting against theft of the on-disk representation of the program.

As shown on the left of Figure 2, our compiler converts unmodified source code into a *readacted* application. It does so by (i) separating code and data to eliminate benign read accesses to code pages, (ii) randomizing the code layout, and (iii) emitting trampoline code to hide code pointers from the adversary. The right side of Figure 2 illustrates how our patched kernel maps all executable code pages with execute-only permissions at runtime. We do not alter the permissions of data areas, including the stack and heap. Hence, these are still readable and writable.

Code-Pointer Hiding: In an ideal fine-grained code randomization scheme, the content and location of every single instruction is random. Execute-only pages offer sufficient protec-

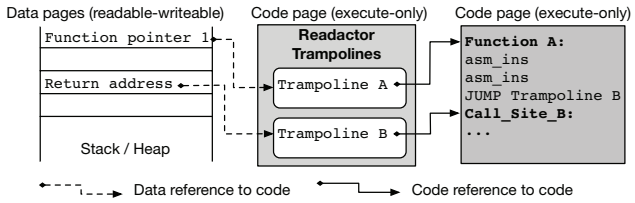

Figure 3: Readacted applications replace code pointers in readable memory with trampoline pointers. The trampoline layout is not correlated with the function layout. Therefore, trampoline addresses do not leak information about the code to which they point.

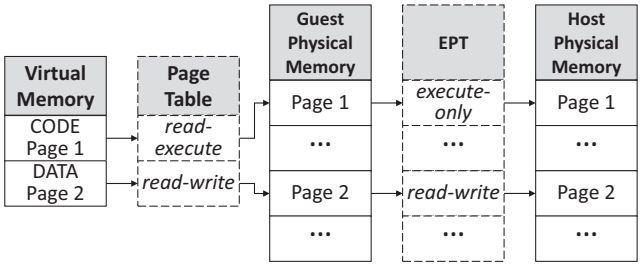

Figure 4: Relation between virtual, guest physical, and host physical memory. Page tables and the EPT contain the access permissions that are enforced during the address translation.

tion against all forms of memory disclosure at this granularity, since indirect disclosure of a code address gives the adversary no information about the location of any other instruction. However, ideal fine-grained randomization is inefficient and does not allow code sharing between processes. Hence, practical protection schemes randomize code at a coarser granularity to reduce the performance overhead [40]. Efficient use of modern processor instruction caches requires that frequently executed instructions are adjacent, e.g., in sequential basic blocks. Furthermore, randomization schemes such as Oxymoron [6] that allow code pages to be shared between processes lead to significantly lower memory usage but randomize at an even coarser granularity (i.e., page-level randomization).

To relax the requirement of ideal fine-grained code randomization, we observe that indirect JIT-ROP relies on disclosing code pointers in readable memory. The sources of code pointers in data pages are (i) C++ virtual tables, (ii) function pointers stored on the stack and heap, (iii) return addresses, (iv) dynamic linker structures (i.e., the global offset table on Linux), and (v) C++ exception handling. Our prototype system currently handles sources (i)-(iv); protecting code pointers related to C++ exceptions is an ongoing effort requiring additional compiler modifications which we discuss in Section VII-C.

Figure 3 illustrates our high-level technique to hide code pointers from readable memory pages. Whenever the program takes the address of a code location to store in readable memory, we instead store a pointer to a corresponding trampoline. Function pointers, for example, now point to trampolines rather than functions. When a call is made via `Function pointer 1` in Figure 3, the execution is redirected to a Readactor trampoline (`Trampoline A`), which then branches directly to `Function A`.

Because trampolines are located in execute-only memory and because the trampoline layout is not correlated with the layout of functions, trampoline addresses do not leak information about non-trampoline code. Hence, trampolines protect the original code pages from indirect memory disclosure (see Section VII-C for details). This combination allows us to use a more practical fine-grained randomization scheme, e.g., function permutation and register randomization, which adds negligible performance overhead and aligns with current cache models.

For a more detailed pictorial overview of the design of Readactor, see Appendix A. In the following sections, we describe each component of Readactor in detail. First, we describe how we enable hardware-assisted execute-only permission on code pages (Section VI). We then present our augmented compiler that implements fine-grained code randomization and code-pointer hiding (Section VII). Finally, in Section VIII we explain how we extended our approach to also protect just-in-time compiled code.

VI. Readactor – Execute-Only Memory

Enforcing execute-only memory for all executable code is one of the key components of our system. Below we discuss the challenges of implementing hardware enforced execute-only memory on the x86 architecture.

A. Extended Page Tables

The x86 architecture provides two hardware mechanisms to enforce memory protection: segmentation and paging. Segmentation is a legacy feature and is fully supported only in 32-bit mode. In contrast, paging is used by modern operating systems to enforce memory protection. While modern x86 CPUs include a permission to mark memory as non-executable [4, 35], it used to be impossible to mark memory as executable and non-readable at the same time. This changed in late 2008 when Intel introduced a new virtualization feature called *Extended Page Tables* (EPTs) [35]. Modern AMD processors contain a similar feature called *Rapid Virtualization Indexing*.

Readactor uses EPTs to enforce execute-only page permissions in hardware. EPTs add an additional abstraction layer during the memory translation. Just as standard paging translates virtual memory addresses to physical addresses, EPTs translate the physical addresses of a virtual machine (VM)—the so-called *guest physical memory*—to real physical addresses or *host physical memory*. The access permissions of each page are enforced during the respective translations. Hence, the final permission is determined by the intersection of the permissions of both translations. EPTs conveniently allow us to enforce (non-)readable, (non-)writable, and (non-)executable memory permissions independently, thereby enabling efficient enforcement of execute-only code pages.

Figure 4 shows the role of the page table and the EPT during the translation from a virtual page to a host physical page. In this example, the loaded application consists of two pages: a code page, marked execute-only, and a data page marked as readable and writable. These page permissions are set by the

compiler and linker. If a code page is labeled with only execute permission, the operating system sets the page to point to a page marked execute-only in the EPT. Note that access control is enforced for each translation step. Hence, a read operation on the code page is allowed during the translation of the virtual to the guest physical page. But, when the guest physical page is translated to the host physical page, an access fault is generated, because the EPT permission is set to execute-only. Similar to the access permissions of a standard x86 page table, the EPT permissions cannot be bypassed by software. However, EPTs are only available when the operating system is executing as a virtualized guest. The next section describes how we addressed this challenge.

B. Hypervisor

Our approach can be used in two different scenarios: software already operating inside a virtualized environment, and software executing directly on physical hardware. For the former case, common in cloud computing environments, the execute-only interface can be implemented as an extension to an existing hypervisor [43, 47, 48, 64, 66]. We chose to focus on the second, non-virtualized scenario for two reasons: First, while standard virtualization is common for cloud computing, we want a more general approach that does not require the use of a conventional hypervisor (and its associated overhead). Many of the attacks we defend against (including our indirect JIT-ROP attack in Section III) require some form of scripting capability [15, 16, 20, 59] and therefore target software like browsers and document viewers running on non-virtualized end-user systems. Second, implementing a thin hypervisor allows us to measure the overhead of our technique with greater precision.

Our hypervisor is designed to transparently transfer the currently running operating system into a virtual environment on-the-fly. Our thin hypervisor design is inspired by hypervisor rootkits that transparently switch an operating system from executing on physical hardware to executing inside a virtual environment that hides the rootkit [38, 54]. Unlike rootkits, however, our hypervisor interfaces with the operating system it hosts by providing an interface to manage EPT permissions and to forward EPT access violations to the OS. Our hypervisor also has the capability to revert the virtualized operating system back to direct hardware execution without rebooting if needed for testing or error handling. For performance and security reasons, we keep our hypervisor as small as possible; it uses less than 500 lines of C code.

Figure 5 shows how we enable execute-only page permissions by creating two mappings of the host physical memory: a *normal* and a *readacted* mapping. The EPT permissions for the normal mapping allow the conventional page table to fully control the effective page permissions. As previously mentioned, the final permission for a page is the intersection of the page table permission and the EPT permission. Hence, setting the EPT permissions to RWX for the normal mapping means that only the permissions of the regular page table are enforced. We set the EPT permissions for the readacted mapping to execute-only so that any read or write access to an address using this mapping results in an access fault. The operating system can map virtual memory to physical memory using either of these mappings. When a block of memory is mapped through the readacted mapping, execute-only permissions are enforced.

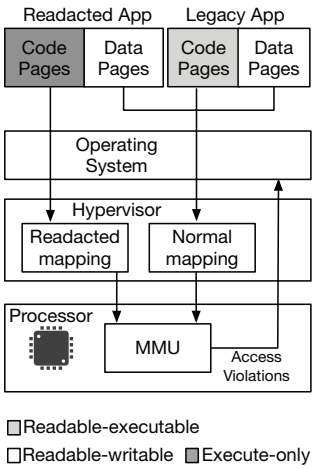

Figure 5: Readactor uses a thin hypervisor to enable the extended page tables feature of modern x86 processors. Virtual memory addresses of protected applications (top left) are translated to physical memory using a *readacted* mapping to allow execute-only permissions whereas legacy applications (top right) use a *normal* mapping to preserve compatibility. The hypervisor informs the operating systems of access violations.

When the normal mapping is used, executable memory is also readable.

Our use of extended page tables is fully compatible with legacy applications. Legacy applications can execute without any modification when Readactor is active, because the normal mapping is used by default. Readactor also supports code sharing between legacy and readacted applications. Legacy applications accessing readacted libraries will receive an execute-only mapping of the library, thus securing the library from disclosure. Readacted applications that require a legacy, un-readacted library can load it normally, but the legacy library will still be vulnerable to information disclosure.

C. Operating System

To simplify implementation and testing, our prototype uses the Linux kernel. However, our fundamental approach is operating system agnostic and can be ported to other operating systems. We keep our patches to the Linux kernel as small as possible (the patch contains 82 lines of code and simply supports the mapping of execute-only pages). Our patch changes how the Linux kernel writes page table entries. When a readacted application requests execute-only memory, we set the guest physical address to point to the readacted mapping rather than the normal mapping.

VII. Readactor – Compiler Instrumentation

To support the Readactor protections, we modified the LLVM compiler infrastructure [41] to (i) generate diversified code, (ii) prevent benign code from reading data residing in code pages, and (iii) prevent the adversary from exploiting code pointers to perform indirect disclosure attacks.

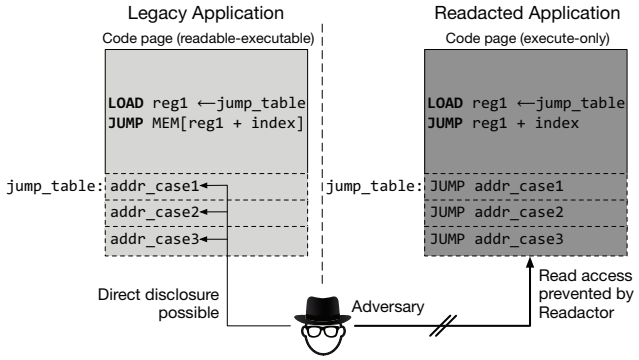

Figure 6: We rewrite switch-case tables to be executable instructions, rather than data embedded in executable code.

A. Fine-grained Code Diversification

Our compiler supports several fine-grained code randomization techniques: function permutation, basic-block insertion, NOP (no-operation) insertion, instruction schedule randomization, equivalent instruction substitution, register allocation randomization, and callee-saved register save slot reordering. The last technique randomizes the stack locations that a function uses to save and restore register values that it must preserve during its execution. In our prototype implementation of Readactor, we use function permutation [37], register allocation randomization, and callee-saved register save slot reordering [49]. We selected these transformations because they permute the code layout effectively, have low performance impact, and make the dataflow between gadgets unpredictable.

Our prototype implementation performs fine-grained code randomization at compile-time. With additional implementation effort, we can make the compiler emit binaries that randomize themselves at load-time [8, 44, 65]. Self-randomizing binaries eliminate the need to generate and distribute multiple distinct binaries, which improves the practicality of diversity. However, the security properties of compile-time and load-time solutions are largely similar. Hence, we focus on how to randomize programs and how to protect the code from disclosure irrespective of when randomization happens.

B. Code and Data Separation

To increase efficiency, compilers sometimes intersperse code and data. Since Readactor enforces execute-only permissions for code pages, we must prevent the compiler from embedding data in the code it generates. That is, we must generate Harvard-architecture compatible code. If we do not strictly separate code and data, we run the risk of raising false alarms as a result of benign attempts to read data from code pages.

We found that the LLVM compiler only emits data in the executable `.text` section of x86 binaries when optimizing a switch-case statement. LLVM emits the basic block address corresponding to each switch-case in a table after the current function. As shown in the left part of Figure 6, the switch statement is then implemented as a load from this table and an indirect branch to the loaded address.

Our compiler translates switch statements to a table of direct branches rather than a list of code pointers that an attacker can read. Each direct branch targets the first basic block corresponding to a switch-case. The switch statement is then generated as an indirect branch into the sequence of direct branches rather than an indirect load and branch sequence. This entirely avoids emitting the switch-case pointers as data, thereby making LLVM generate x86 code compatible with execute-only permissions. Figure 6 shows how code pointers (`addr_case1...addr_case3`) are converted to direct jumps in an example switch-case statement. We quantify the impact of this compiler transformation in Section X.

While examining x86 binaries on Linux, we noticed that contemporary linkers include both the readable ELF header data and executable code on the first page of the mapped ELF file. Hence, we created a patch for both the BFD and Gold linkers to start the executable code on a separate page from the readable ELF headers and to adjust the page permissions appropriately. This separation allows the ELF loader to map the headers as readable while mapping the first code page as execute-only.

C. Code-Pointer Hiding

Making code non-readable prevents the original JIT-ROP attack but not indirect JIT-ROP. In the latter attack, an attacker combines pointer harvesting with partial a priori knowledge of the code layout, e.g., the layout of individual code pages or functions (cf. Section III). To thwart indirect JIT-ROP, we hide code pointers so they are no longer stored in readable memory pages.

We protect against the sources of indirect code disclosure identified in Section V by adding a level of indirection to code pointers. The two steps in code-pointer hiding are (i) creating trampolines for each instruction reachable through an indirect branch and (ii) replacing all code pointers in readable memory with trampoline pointers. We use two kinds of trampolines: *jump trampolines* and *call trampolines*, to protect function addresses and call sites respectively.

We generate a *jump trampoline* for each function that has its address taken. Figure 7 shows how we replace a vtable and function pointer with pointers to jump trampolines. For example, when a call is made through funcPtr_trampoline, execution is redirected to the original target of the call: Function_B.

The call trampolines that hide return addresses on the stack are shown in Figure 8. Normally, a call to Method_A will push the address of the following instruction (call_site_1) onto the stack. The Readactor compiler moves the call into a call trampoline such that the return address that is pushed onto the stack points to the call trampoline rather than the calling function. When the callee returns to the call trampoline, a direct branch transfers execution back to the original caller. Like jump trampolines, call trampolines cannot be read by attackers and therefore do not leak information about the function layout.

A final source of indirect code leakage is related to C++ exception handling. When an exception occurs, the C++ runtime library must unwind the stack, which is the process of walking back up the call chain and destroying locally allocated objects until an appropriate exception handler is found. Modern C++

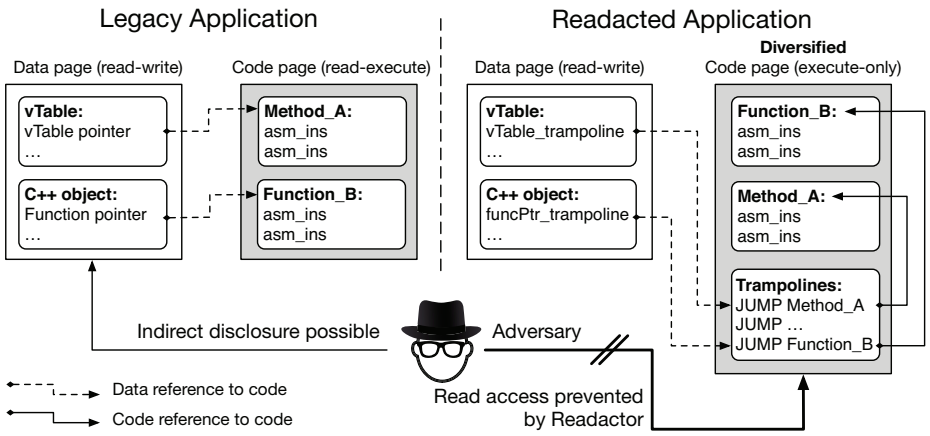

Figure 7: Hiding code pointers stored in the heap and in C++ vtables. Without Readactor, pointers to functions and methods may leak (left). With Readactor, only pointers to jump trampolines may leak and the layouts of functions and jump trampolines are randomized (right).

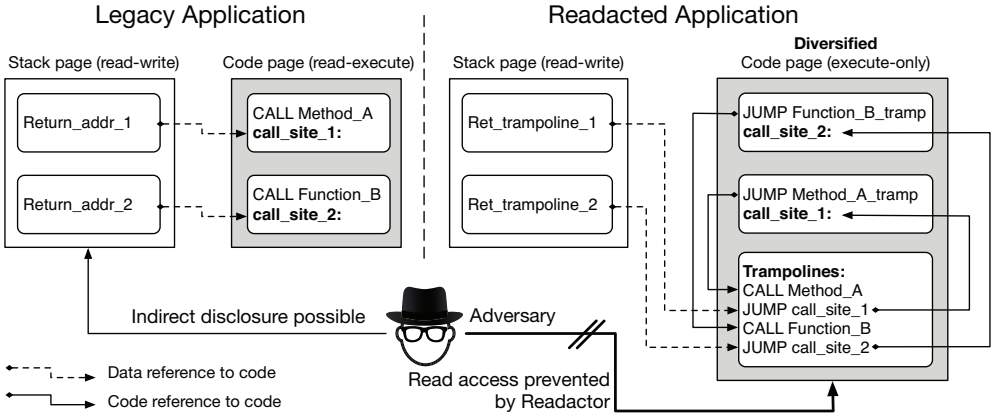

Figure 8: Hiding return addresses stored on the machine stack. Without Readactor, each activation frame on the stack leaks the location of a function (left). With Readactor, calls go through call trampolines so the return addresses pushed on the stack can only leak trampoline locations – not return sites (right).

compilers implement efficient exception handling by generating an exception handling (EH) table that informs the unwinding routine of the stack contents. These data tables are stored in readable memory during execution and contain the range of code addresses for each function and the information to unwind each stack frame. During stack unwinding, the C++ runtime library locates the exception handling entry for each return address encountered on the stack. Since our call trampolines push the address of a trampoline onto the stack rather than the real return address, the runtime will try to locate the address of the call trampoline in the exception handling tables. Hence, we need to replace the real function bounds in the EH table with the bounds of the trampolines for that function.

Our prototype compiler implementation does not rewrite the EH table to refer to trampolines; however, doing so is merely a matter of engineering effort. No aspect of our approach prevents us from correctly supporting C++ exception handling. We found that disabling C++ exception handling was not a critical limitation in practice, since many C++ projects, including Chromium, choose not to use C++ exceptions for performance or compatibility reasons.

Handwritten assembly routines are occasionally used to optimize performance critical program sections where standard C/C++ code is insufficient. To prevent this assembly code from leaking code pointers to the stack, we can rewrite it to use trampolines at all call sites. Additionally, we can guarantee that no code pointers are stored into readable memory from assembly code. Our current implementation does not perform such analysis and rewriting of handwritten assembly code but again, doing so would involve no additional research.

While code pointers are hidden from adversaries, trampoline pointers are stored in readable memory as shown on the right-hand sides of Figures 7 and 8. Therefore, we must carefully consider whether leaked trampoline pointers are useful to adversaries. If the layout of trampolines is correlated with the function layout, knowing the trampoline pointers informs

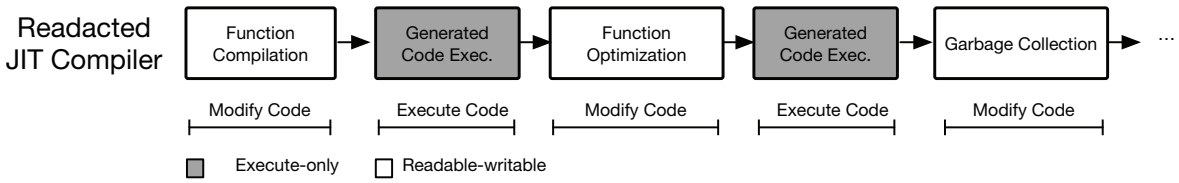

Figure 9: Timeline of the execution of a JIT-compiled program. Execution switches between the JIT compiler and generated code.

adversaries of the code layout. To ensure there is no correlation between the layout of trampolines and functions, we permute the list of trampolines. We also insert *dummy* entries into the list of trampolines that consist of privileged instructions. Hence, any attempts to guess the contents of the trampoline list by executing them in a row will eventually trigger an exception [19].

Each trampoline entry contains a direct jump that consists of an opcode and an immediate operand encoding the destination of the jump. Because we permute the order of functions, the value of the immediate operand is randomized as a side effect. This makes the contents of trampolines unpredictable to attackers and prevents use of any unintended gadgets contained in the immediate operands of direct jumps.

An attacker can use trampoline pointers just as they are used by legitimate code: to execute the target of the trampoline pointer without knowledge of the target address. Because we only create trampolines for functions that have their address taken and for all return sites, our code mechanism restricts the attacker to reuse only function-entry gadgets and call-preceded gadgets. Note that CFI-based defenses constrain attackers similarly and some CFI implementations use trampoline mechanisms similar to ours [63, 67, 69]. Coarse-grained CFI defenses are vulnerable to gadget stitching attacks where adversaries construct ROP chains out of the types of gadgets that are reachable via trampoline pointers [20, 29]. Although gadget stitching attacks against Readactor will be hard to construct because the required trampoline pointers may be non-trivial to leak, we still included protection against such attacks. We observe that gadget chains (including those that bypass CFI) pass information from one gadget to another via stack slots and registers. Because we perform register allocation randomization and callee-saved register save slot reordering, the attacker cannot predict how data flows through the gadgets reachable via trampolines.

Modern processors have deep pipelines and fetch instructions long before they may be needed. On the one hand, pipelining can hide the cost of the additional indirection introduced by trampolines. On the other hand, we must ensure that our use of trampolines to hide code pointers does not increase the number of costly branch predictor misses that stall the processor pipeline until the correct instructions are fetched. Thanks to the use of a dedicated return-address stack in modern branch predictor units, targets of function returns are rarely mispredicted as long as function calls and returns are paired. By preserving this pairing, we ensure that our trampolines do not introduce additional branch predictor misses. We evaluate the performance impact of jump and call trampolines in Section X.

VIII. Readactor – JIT Compiler Protection

Web pages embed JavaScript code that must be executed by the browser. The most effective way to execute JavaScript (and other so called dynamic languages) is via JIT compilation, which all major web browsers support. What sets just-in-time compilers apart from their ahead-of-time counterparts is that code generation happens dynamically at runtime. Consequently, our compile-time techniques described in Section VII will not protect dynamically generated code. To make our defense practical and comprehensive, we extended our execute-only memory to support dynamically compiled code. This section describes how this was achieved for the V8 JavaScript engine which is part of the Chromium web browser. We believe that the method we used generalizes to other JIT compilers.

Modern JavaScript engines are tiered, which means that they contain several JIT compilers. The V8 engine contains three JIT compilers: a baseline compiler that produces unoptimized code quickly and two optimizing compilers called CrankShaft and TurboFan. Having multiple JIT compilers lets the JavaScript engine focus the optimization effort on the most frequently executed code. This matters because the time spent on code optimization adds to the overall running time.

JIT engines cache the generated code to avoid continually recompiling the same methods. Frequently executed methods in the code cache are recompiled and replaced with optimized versions. When the code cache grows beyond a certain size, it is garbage collected by removing the least recently executed methods. Since the code cache is continually updated, JIT compilers typically allocate the code cache on pages with RWX permissions. This means that, unlike statically generated code, there is no easy way to eliminate reads and writes to dynamically generated code without incurring a high performance impact.

We apply the Readactor approach to dynamically generated code in two steps. First, we modify the JIT compilers to separate code and data in their output. Other V8 security extensions [5, 46, 60] require this separation as well to prevent code injection attacks on the code cache, and Ansel et al. [5] also implement it. Second, with code and data separated on different pages, we then identify and modify all operations that require reads and writes of generated code. The following sections discuss these two steps in greater detail. Figure 9 shows the permissions of the code cache over time after we modified the V8 engine. Our changes to the V8 JavaScript engine adds a total of 1053 new lines of C++ code across 67 different source files.

A. Separating Code and Data

The unmodified V8 JIT compiler translates one JavaScript function at a time and places the results in a code cache

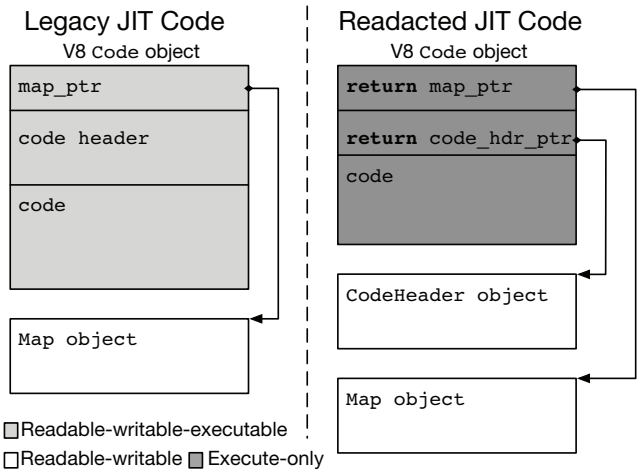

Figure 10: Transforming V8 `Code` objects to separate code and data.

backed by pages with RWX permissions. Each translated function is represented by a `Code` object in V8 as shown on the left side of Figure 10. Each `Code` object contains the generated sequence of native machine instructions, a code header containing information about the machine code, and a pointer to a `Map` object common to all V8 JavaScript objects.

To store `Code` objects on execute-only memory pages, we move their data contents to separate data pages and make this data accessible by adding new getter functions to the `Code` object. To secure the code header, we move its contents into a separate `CodeHeader` object located on page(s) with RW permissions. We add a getter method to `Code` objects that returns a pointer (`code_hdr_ptr`) to the `CodeHeader` object. Similarly, we replace the map pointer (`map_ptr`) with a getter that returns the map pointer when invoked. These changes eliminate all read and write accesses to `Code` objects (except during object creation and garbage collection) so they can now be stored in execute-only memory; a transformed `Code` object is shown on the right side of Figure 10.

B. Switching Between Execute-Only and RW Permissions

With this separation between code and data inside `Code` objects, we guarantee that no JavaScript code needs to modify the contents of executable pages. However, the JIT compiler still needs to change code frequently. As Figure 9 shows, execution alternates between the compiler and JavaScript code, and changes to code can only come from the compiler. Completely eliminating code writes from the compiler would require a significant refactoring of V8 (due to extensive use of inline caches, relocations and recompilation, which are hard to completely eliminate), as well as incur a significant performance hit. Instead, we observe that the generated code is either executed or suspended so that it can be updated. During execution, we map code with execute-only permissions, and when execution is suspended, we temporarily remap it with RW permissions. For both performance and security reasons, we minimize the number of times we re-map pages, as well as the length of time a page stays accessible. Each time a `Code` object becomes writable, it provides a window for the attacker to inject malicious code into that object.

Song et al. [60] recently demonstrated that an attack during this window is feasible. They propose a defense based on process separation, where the JIT compiler is located in a separate process from the untrusted browser, and only the JIT process can write to the generated code. This successfully protects against code injection attacks against the code cache, but not against disclosure of the generated code. In the untrusted process, the generated code is mapped as read-only and executable, but could instead be mapped as execute-only for use with Readactor. We believe their solution is fully compatible with and complementary to ours, and can be used to protect the JIT from code injection.

IX. Security Evaluation

The main goal of Readactor is to prevent code-reuse attacks constructed using either direct or indirect disclosure vulnerabilities. Thus, we have analyzed and tested its effectiveness based on five different variants of code-reuse attacks, namely (i) static ROP attacks using direct and indirect disclosure, (ii) just-in-time ROP attacks using direct disclosure, (iii) just-in-time ROP attacks using indirect disclosure, (iv) ROP attacks on just-in-time generated code, and (v) return-into-libc attacks. We present a detailed discussion on each type of code-reuse attack and then evaluate the effectiveness of Readactor using a sophisticated proof-of-concept JIT-ROP exploit.

a) Static ROP: To launch a traditional ROP attack [14, 58], the adversary must know the runtime memory layout of an application and identify ROP gadgets based on an offline analysis phase. To defeat regular ASLR the adversary needs to leak a single runtime address through either direct or indirect disclosure. Afterwards, the addresses of all target gadgets can be reliably determined.

Since Readactor performs fine-grained randomization using function permutation, the static adversary can only guess the addresses of the target gadgets. In other words, the underlying fine-grained randomization ensures that an adversary can no longer statically determine the addresses of all gadgets as offsets from the runtime address of a single leaked function pointer. In addition, we randomize register allocation and the ordering of stack locations where registers are saved to ensure that the adversary cannot predict the runtime effects of gadgets. Using these fine-grained diversifications, Readactor fully prevents static ROP attacks.

b) JIT-ROP with direct disclosure: JIT-ROP attacks bypass fine-grained code randomization schemes by disassembling code pages and identifying ROP gadgets dynamically at runtime. One way to identify a set of useful gadgets for a ROP attack is to exploit direct references in call and jump instructions [59]. Readactor prevents this attack by marking all code pages as non-readable, i.e., execute-only. This differs from a recent proposal, XnR [7], that always leaves a window of one or more pages readable to the adversary. Readactor prevents all reading and disassembly of code pages by design.

c) JIT-ROP with indirect disclosure: Preventing JIT-ROP attacks that rely on direct disclosure is insufficient, since advanced attacks can exploit indirect disclosure, i.e.,

harvesting code pointers from the program's heap and stack (see Section III). Readactor defends against these attacks with a combination of fine-grained code randomization and code-pointer hiding. Recall that pointer hiding ensures that the adversary can access only trampoline addresses but cannot disclose actual runtime addresses of functions and call sites (see Section V). Hence, even if trampoline addresses are leaked and known to the adversary, it is not possible to use arbitrary gadgets inside a function because the original function addresses are hidden in execute-only trampoline pages. As discussed in Section VII-C, code-pointer hiding effectively provides at least the same protection as coarse-grained CFI, since only valid address-taken function entries and call-sites can be reused by an attacker. However, our scheme is strictly more secure, since the adversary must disclose the address of each trampoline from the stack or heap before he can reuse the function or call-site. In addition, we strengthen our protection by employing fine-grained diversifications to randomize the dataflow of this limited set of control-flow targets.

Specifically, when exploiting an indirect call (i.e., using knowledge of a trampoline address corresponding to a function pointer), the adversary can only redirect execution to the trampoline but not to other gadgets located inside the corresponding function. In other words, we restrict the adversary who has disclosed a function pointer to whole-function reuse.

On the other hand, disclosing a call trampoline allows the adversary to redirect execution to a valid call site (e.g., call-preceded instruction). However, this still does not allow the adversary to mount the same ROP attacks that have been recently launched against coarse-grained CFI schemes [13, 20, 29, 55], because the adversary only knows the trampoline address and not the actual runtime address of the call site. Hence, leaking one return address does not help to determine the runtime addresses of other useful call sites inside the address space of the application. Furthermore, the adversary is restricted to only those return trampoline addresses that are leaked from the program's stack. Not every return trampoline address will be present on the stack, only those that are actually used and executed by the program are potentially available. This reduces the number of valid call sites that the adversary can target, in contrast to the recent CFI attacks, where the adversary can redirect execution to *every* call site in the address space of the application without needing any disclosure.

Finally, to further protect call-site gadgets from reuse through call trampolines, we use two fine-grained diversifications proposed by Pappas et al. [49] to randomize the dataflow between gadgets: register allocation and stack slot randomization. Randomizing register allocation causes gadgets to have varying sets of input and output registers, thus disrupting how data can flow between gadgets. We also randomly reorder the stack slots used to preserve registers across calls. The program's application binary interface (ABI) specifies a set of callee-saved registers that functions must save and restore before returning to their caller. In the function epilogue, the program restores register values from the stack into the appropriate registers. By randomizing the storage order of these registers, we randomize the dataflow of attacker-controlled values from the stack into registers in function epilogues.

d) ROP on just-in-time generated code: In contrast to many other recent defenses (e.g., [6, 7, 44]), Readactor also applies its protection mechanisms to dynamically-generated code. This coverage is important since many well-known programs feature scripting facilities with just-in-time (JIT) code generation (e.g., Internet Explorer, Firefox, Adobe Reader, and Microsoft Word). Typically, dynamic code is of particular interest to the adversary, as it is usually mapped as read-write-executable (RWX).

Hence, several exploits use a technique called JIT-spraying [11]. In this attack, the adversary writes a script (e.g., JavaScript) that emits shellcode as unintended instruction sequences into the address space of an application. A well-known example is the XOR instruction that can be exploited to hide shellcode bytes as an immediate value. Google's V8 JIT engine mitigates this specific instantiation of JIT-spraying by XOR'ing random values with the immediate values. However, another way to exploit JIT-compiled code (RWX) memory is to disclose its address, overwrite the existing code with shellcode, and execute it. The adversary sprays a large number of shellcode copies abusing the JIT compiler. After the shellcode has been emitted, the adversary simply needs to exploit a vulnerability and redirect execution to the shellcode through memory corruption.

Readactor prevents this classic JIT-spraying attack as well as any attack that attempts to identify useful code in the JIT-compiled code area through direct memory disclosure. We achieve this by marking the JIT code area as execute-only and use V8's built-in coarse-grained randomization (similar to ASLR). Hence, the adversary can neither search for injected shellcode nor find other useful ROP code sequences. On the other hand, given V8's coarse-grained randomization, it is still possible for the adversary to guess the address of the injected shellcode. To tackle guessing attacks, we are currently working on integrating fine-grained randomization inside V8 (inspired by the ideas used in librando [33]).

We also tested whether indirect disclosure of JIT-compiled code is feasible. Our experiments revealed that V8's JIT code cache contains several code pointers referencing JIT-compiled code. Hence, the adversary could exploit these pointers to infer the code layout of the JIT memory area. To protect against such attacks, these code pointers need to be indirected through execute-only trampolines (our standard jump trampolines). We can store these trampolines in a separate execute-only area away from the actual code. To add support for code-pointer hiding, we would need to modify both the JITted code entry points from the JavaScript runtime and all JavaScript-to-JavaScript function calls to call trampolines instead. This work is an engineering effort that is currently ongoing, as it requires porting our LLVM compiler changes over to the V8 JIT.

e) Return-into-libc: Most of the papers dealing with code-reuse attacks do not provide a security analysis of classic return-into-libc attacks [45], i.e., attacks that only invoke entire functions rather than short ROP code sequences. In general, it is very hard to prevent return-into-libc attacks, since they target legitimate addresses, such as exported library functions. In Readactor, we limit the attack surface for return-into-libc attacks.

To launch a return-into-libc attack, the adversary needs to identify code pointers to functions of interest, e.g., `system` or `mprotect` on Linux. Typically, this is done by disclosing

the function address from a known position within either code or data sections. We prevent disclosure from code because Readactor maps code pages as execute-only. On the other hand, code pointers in data sections, e.g. pointers in the import address table (IAT) in Windows or the global offset table (GOT) in Linux which are used for shared library calls, can be exploited in an indirect disclosure attack. Since Readactor hides code pointers in trampolines and also performs function permutation, the adversary cannot exploit a IAT/GOT entry to determine the address of a function of interest in a readacted library. However, if the function of interest is imported by the program then the adversary can exploit the GOT entry containing the corresponding trampoline addresses to directly invoke the function.

In summary, Readactor provides a high degree of protection against code reuse attacks of all kinds, while being practical and efficient at the same time, as we demonstrate in the next subsection. First, we describe our protection against proof-of-concept exploit targeting the JavaScript JIT.

f) Proof-of-concept exploit: To demonstrate the effectiveness of our protection, we introduced an artificial vulnerability into V8 that allows an attacker to read and write arbitrary memory. This vulnerability is similar to a vulnerability in V8[2] that was used during the 2014 Pwnium contest to get arbitrary code execution in the Chrome browser. In an unprotected version of V8, the exploitation of the introduced vulnerability is straightforward. From JavaScript code, we first disclose the address of a function that resides in the JIT-compiled code memory. Next, we use our capability to write arbitrary memory to overwrite the function with our shellcode. This is possible because the JIT-compiled code memory is mapped as RWX in the unprotected version of V8. Finally, we call the overwritten function, which executes our shellcode instead of the original function. This attack fails under Readactor because the attacker can no longer write shellcode to the JIT-compiled code memory, since we set all JIT-compiled code pages execute-only. Further, we prevent any JIT-ROP like attack that first discloses the content of JIT-compiled code memory, because that memory is not readable. We tested this by using a modified version of the attack that reads and discloses the contents of a code object. Readactor successfully prevented this disclosure by terminating execution of the JavaScript program when it attempted to read the code.

X. Performance Evaluation

We rigorously evaluated the performance impact of Readactor on both the SPEC CPU2006 benchmark suite and a large real-world application, the Chromium browser. We also evaluated our changes to the V8 JavaScript engine using standard JavaScript benchmarks.

1) SPEC CPU2006: The SPEC CPU2006 benchmark suite contains CPU-intensive programs which are ideal to test the worst-case overhead of our compiler transformations and hypervisor. To fully understand the impact of each of the components that make up the Readactor system, we measured and report their performance impact independently.

We performed all evaluations using Ubuntu 14.04 with Linux kernel version 3.13.0. We primarily evaluated SPEC on an Intel Core i5-2400 desktop CPU running at 3.1 GHz with dynamic voltage and frequency scaling (Turbo Boost) enabled. We also independently verified this evaluation using an Intel Xeon E5-2660 server CPU running at 2.20 GHz with Turbo Boost disabled, and observed identical trends and nearly identical performance (within one percent on all averages). We summarize our SPEC measurements in Figure 11. Overall, we found that Readactor, with all protections enabled, incurs an average performance overhead of just 6.4% for SPEC CPU2006.

a) Code-Data Separation: First we evaluated the performance overhead of separating code from data by rewriting how the compiler emits switch tables in code (see Section VII-B). We found the impact of transforming switch table data into executable code to be less than half of a percent on average, with a maximum overhead of 1.1%. This overhead is minimal because we maintain good cache locality by keeping the switch table close to its use. In addition, modern processors can prefetch instructions past direct jumps, which means these jumps have a low performance impact. We omit this experiment from Figure 11 for clarity, since it showed such minimal overheads.

b) Code-Pointer Hiding: We then evaluated full code pointer protection, with hiding of both function pointers and return addresses enabled. We found that code-pointer hiding resulted in a performance slowdown of 4.1% on average over all benchmarks (*Pointer Hiding* in Figure 11). This protection introduces two extra direct jumps for each method call and one direct jump when de-referencing function pointers. On closer inspection using hardware performance counters, we observed that these jumps back and forth from the regular code section to the trampolines slightly increased the instruction-cache pressure, resulting in more instruction-cache misses.

We hypothesized that the bulk of code-pointer hiding overhead was due to call trampolines, which are far more common than function pointers. To verify this, we disabled return address hiding while keeping function pointer protection enabled. We found that function pointer protection by itself incurred an average overhead of only 0.2% on SPEC, with no benchmark exceeding 2%. Thus, most of the overhead for code-pointer hiding is caused by the frequent use of call trampolines. This effect is amplified in benchmarks which make many function calls, such as *xalancbmk*.

c) Hypervisor: To understand the performance impact of the hypervisor layer, including the additional EPT translation overhead, we ran SPEC under our hypervisor without any execute-only page protections or code-pointer hiding enabled (*Hypervisor* in Figure 11). We observed that the virtualization overhead was 1.1% on average. Since we allow the virtual processor full control of all hardware and registers, this overhead is mainly caused by the extra memory translation overhead from translating guest physical addresses to host physical addresses through the EPT. Even though we use an identity mapping from guest physical to host physical addresses, the processor must still walk through the whole EPT whenever translating a new memory address. The larger overhead observed for the *mcf* benchmark supports this theory, as it has a higher and more stable memory footprint (845Mib) than the other benchmarks [31]. This results in more swapping in and out of the cache, which in turn triggers more EPT address translations.

[2] CVE-2014-1705

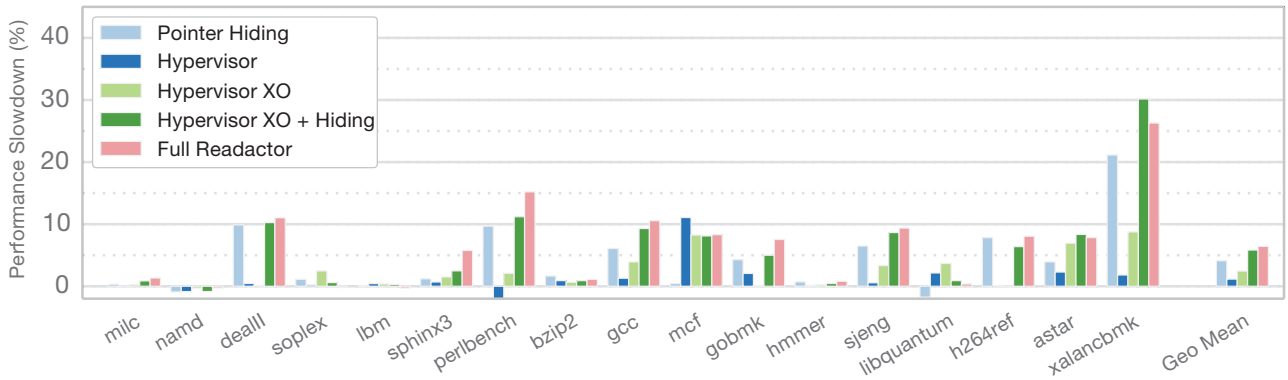

Figure 11: Performance overhead for SPEC CPU2006 with Readactor enabled relative to an unprotected baseline build.

After compiling SPEC with separation of code and data and marking code pages as execute-only during linking, we ran the benchmarks under the hypervisor, enforcing execute-only page permissions (*Hypervisor XO* in Figure 11). This configuration incurred a performance slowdown of 2.5%, somewhat higher than the overhead of the hypervisor itself. Much of this overhead difference is due to the separation of code and data, which de-optimizes execution slightly. We attribute the rest of this difference to measurement variance, since the hypervisor itself should not add any significant overhead when enforcing execute-only pages versus legacy readable and executable pages. In either case the processor must still translate all addresses through the EPT when the hypervisor is enabled.

d) Full Readactor: Enabling code-pointer hiding along with page protections provided by the hypervisor resulted in a slowdown of 5.8% (*Hypervisor XO + Hiding* in Figure 11). This overhead is approximately the sum of the overheads of both components of the system, the execute-only hypervisor enforcement and pointer hiding. This confirms our hypothesis that each component of the Readactor system is orthogonal with respect to performance.

With the addition of our fine-grained diversity scheme (function, register, and callee-saved register slot permutation) we now have all components of Readactor in place. For the final integration benchmark we built and ran SPEC using three different random seeds to capture the effects of different code layouts. Altogether we observed that the full Readactor system incurred a geometric mean performance overhead of 6.4% (*Full Readactor* in Figure 11). This shows the overhead of applying our full protection scheme to a realistic worst-case scenario of CPU-intensive code, which bounds the overhead of our system in practice.

2) Chromium Browser: To test the performance impact of our protections on complex, real-world software, we compiled and tested the Chromium browser, which is the open-source variant of Google's Chrome browser. Chromium is a highly complex application, consisting of over 16 million lines of code [10]. We were able to easily apply all our protections to Chromium with the few minor changes described below. Overall, we found that the perceived performance impact on web browsing with the protected Chromium, as measured by Chromium's internal UI smoothness benchmark, was 4.0%, which is in line with the average slowdown we observed for SPEC.

Since our protection system interferes with conventional stack walking, we had to disable debugging components of Chromium that use stack walking. We found that the optimized memory allocator used in Chromium, TCMalloc, uses stack tracing to provide detailed memory profiling information to developers. We disabled this functionality, which is not needed for normal execution. We also observed that Chromium gathers stack traces at tracing points during execution, again for debugging. Conveniently, we could disable this stack tracing with a single-line source code change. With these minor modifications we could compile and test the current development version[3] of Chromium with our LLVM-based Readactor compiler.

To understand the perceived performance impact during normal web browsing we benchmarked page scrolling smoothness with Chromium's internal performance testing framework. We ran the scrolling smoothness benchmark from the Chromium source tree on the Top 25 sites selected by Google as representatives of popular websites. This list includes 13 of the Alexa USA Top 25 sites including Google properties such as Google search, GMail and Youtube, Facebook, and news websites such as CNN and Yahoo. The Chromium scrolling benchmark quantifies perceived smoothness by computing the mean time to render each frame while automatically scrolling down the page. We report the average slowdown as time per frame averaged over 3 runs of the benchmark suite to account for random variance.

Overall, we found that the slowdown in rendering speed for our full Readactor system was about 4.0%, averaged over 3 different diversified builds of Chromium. This overhead is slightly lower than what we found for SPEC, which is natural considering that browser rendering is not as CPU-intensive as the SPEC benchmarks. However, browser smoothness and responsiveness are critical factors for daily web browsing, rather than raw computing performance.

We also evaluated the performance impact of our techniques on Chromium using the extensive Dromaeo benchmark suite to give a worst-case estimate for browser performance. This suite, composed of 55 individual benchmarks, includes standard

[3]Chromium sources checked out on 2014-11-04.

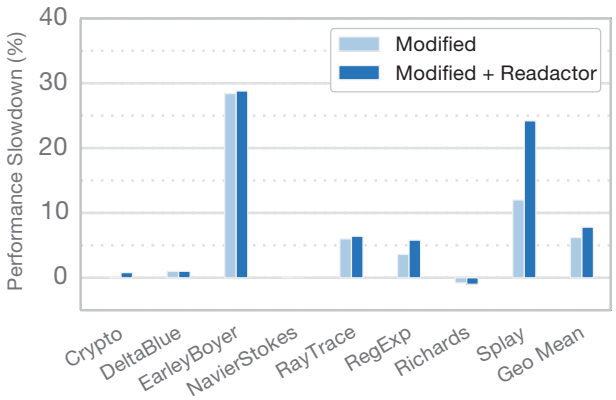

Figure 12: Performance of modified V8 running under Readactor, relative to a vanilla build and to a modified build running natively.

JavaScript benchmarks such as the Sunspider and V8 JavaScript benchmarks, as well as benchmarks that exercise DOM and CSS processing. Dromaeo is comprehensive, and hence, ideal to evaluate the overall impact of our protections on performance-critical browser components.

We found that execute-only code protection alone, without code-pointer hiding, introduced a 2.8% overall performance slowdown on Dromaeo. Combining the hypervisor execute-only code pages along with code-pointer hiding resulted in a 12% performance slowdown. We attribute this higher overhead to increased instruction cache pressure caused by our call pointer protection. However, Dromaeo represents a worst-case performance test, and rendering smoothness on real websites is a far more important factor in browsing.

3) V8 JavaScript JIT: We evaluated the performance impact of our changes to the V8 JavaScript engine, as well as the overhead of running the JIT compiler under Readactor. To get a more accurate sample, we benchmarked the V8 engine alone, outside of the browser. Figure 12 shows the results of our evaluation. We see small to negligible overhead for most benchmarks, with the exception of two benchmarks which put significant pressure on the memory allocator: *EarleyBoyer* and *Splay*. Both benchmarks allocate large numbers of temporary objects and trigger frequent garbage collection cycles, which become much more expensive due to our repeated re-mapping of pages. For another benchmark—*Richards*—we observe a very small speedup of 1%, which we attribute to measurement noise. Overall, the performance penalty of our changes comes to 6.2% when running natively and 7.8% with Readactor enabled.

We also added our V8 JIT compiler patches to the full Chromium browser to evaluate their impact on the whole browsing experience. We observed a higher impact on scrolling smoothness, with an average time per frame slowdown of 13.8% versus 4.0% without the JIT compiler patches. This extra slowdown is due to the overhead of separating JIT generated data from code to allow the JIT to map generated code pages as execute-only.

XI. Related Work

Most code-reuse exploit mitigation approaches are based on either program randomization or some form of integrity checking. In contrast, Readactor combines a probabilistic defense against ROP (code layout randomization) with integrity checks (execute-only page permissions) against disclosure to comprehensively thwart code-reuse attacks. We discuss probabilistic and integrity-checking defenses separately. We summarize the main difference between our approach and closely related work, namely Oxymoron [6], XnR [7], and HideM [27] in Table I. Readactor is the only defense that provides protection against all known variants of ROP attacks (traditional ROP, direct and indirect JIT-ROP), while performing efficiently and protecting JIT-compiled code.

A. Code Randomization Defenses

Cohen was first to explore program protection using diversity [18]. Forrest et al. [24] later demonstrated stack-layout randomization against stack smashing. Address space layout permutation (ASLP) [37] randomizes the code layout at function granularity; adversaries must therefore disclose more than one code pointer to bypass ASLP. Binary Stirring [65] permutes both functions and basic blocks within functions, and Instruction Layout Randomization (ILR) [32] randomizes the code layout at the instruction level. Larsen et al. [40] compare these and additional approaches to automatic software diversity.

Unfortunately, these defenses remain vulnerable to memory disclosure attacks. The appearance of JIT-ROP attacks convincingly demonstrated that probabilistic defenses cannot tacitly assume that attackers cannot leak code memory layout at runtime [59]. Blind ROP [9], another way to bypass fine-grained code randomization, further underscores the threat of memory disclosure.

In response to JIT-ROP, Backes and Nürnberger proposed Oxymoron [6] which randomizes the code layout at a granularity of 4KB memory pages. This preserves the ability to share code pages between multiple protected applications running on a single system. Oxymoron seeks to make code references in direct calls and jumps that span code page boundaries opaque to attackers. Internally, Oxymoron uses segmentation and redirects inter-page calls and jumps through a dedicated hidden table. This prevents direct memory disclosure, i.e., it prevents the recursive-disassembly step in the original JIT-ROP attack. Unfortunately, Oxymoron can be bypassed via indirect memory disclosure attacks as we have described in Section III. In contrast to Readactor, Oxymoron does not protect JIT-compiled code. Readactor offers more comprehensive protection against both direct and indirect code disclosure and protects JIT-compiled code against direct disclosure.

Another defense against JIT-ROP, Execute-no-Read (XnR) by Backes et al. [7], is conceptually similar to Readactor as it is also based on execute-only pages. However, it only emulates execute-only pages in software by keeping a sliding window of n pages as both readable and executable, while all other pages are marked as *non-present*. XnR does not fully protect against direct memory disclosure because pages in execution are readable. Hence, the adversary can disclose function addresses (see Section III), and force XnR to map a target page as readable by calling the target function through an embedded script. This

Property	Note	Oxymoron [6]	XnR [7]	HideM [27]	**Readactor execute-only**	**Full Readactor**
Prevents disclosure of:						
References to other code pages		✓				✓
Code pages			✓	✓	✓	✓
Return addresses						✓
Jump table pointers		✓	✓	✓	✓	✓
C++ vtable and function pointers						✓
Security:						
ROP [58]		✓	✓	✓	✓	✓
JIT-ROP [59]		✓	✓	✓	✓	✓
Blind ROP [9]	Direct Memory Disclosure		✓	✓	✓	✓
Indirect JIT-ROP (Section III)	Indirect Memory Disclosure					✓
Coverage:						
Static compiled code		✓	✓	✓	✓	✓
JIT compiled code						✓
Efficiency:						
Avg. Runtime	SPEC CPU2006	2.5%	2.2%	1.5%	2.5%	6.4%
Max. Runtime	SPEC CPU2006	11%	15%	6.5%	8.8%	26%
Avg. Runtime	V8 Benchmarks					7.8%

Table I: Comparison of randomizing defenses against attacks combining memory disclosure and ROP such as JIT-ROP. Readactor offers best-in-class security and is the only mechanism that covers both statically and dynamically compiled code.

can be repeated until the disclosed code base is large enough to perform a JIT-ROP attack. In Readactor, such an attack is not possible by design, because code pages are always execute-only. Further, it remains unclear how well XnR can protect against indirect memory disclosure. First, it only assumes a code randomization scheme without implementing and evaluating one. Second, as we have discussed in detail in Section V, defending against indirect code disclosure with randomization alone (i.e., without code-pointer hiding) requires XnR to use a very fine-grained and unpractical code randomization scheme. Moreover, in contrast to XnR, we evaluate Readactor against complex software such as the Google Chromium web-browser and also extend our protections to JIT-compiled code.

HideM by Gionta et al. [27] also implements execute-only pages. Rather than supporting execute-only pages by unmapping code pages when they are not actively executing, HideM uses split translation lookaside buffers (TLBs) to direct instruction fetches and data reads to different physical memory locations. HideM allows instruction fetches of code but prevents data accesses except whitelisted reads of embedded constants. This is the same technique PaX [51] used to implement W$\oplus$X memory before processors supported RX memory natively. However, the split TLB technique does not work on recent x86 hardware because most processors released after 2008 contain unified second-level TLBs.

Giuffrida et al. [28] evaluated a comprehensive, compiler-based software diversifier that allows live re-randomization of a micro-kernel. Frequent re-randomization may render any knowledge gleaned through memory disclosure useless. However, the entire JIT-ROP attack can run in as little as 2.3 seconds while re-randomization at 2 second intervals add an overhead of about 20%. Moreover, it remains unknown how well this approach scales to complex applications containing JIT compilers and modern operating systems with monolithic kernels.

Whereas Oxymoron, XnR, and HideM seek to *hide* the code layout, Opaque CFI (O-CFI) [44] is designed to *tolerate* certain kinds of memory disclosure. Similar to Readactor, O-CFI combines code randomization and integrity checks. It tolerates code layout disclosure by bounding the target of each indirect control flow transfer. Since the code layout is randomized at load time, the bounds for each indirect jump are randomized too. The bounds are stored in a small table which is protected from disclosure using x86 segmentation. O-CFI uses binary rewriting and stores two copies of the program code in memory to detect disassembly errors. Hence, it requires more program memory than Readactor. In contrast, the trampolines added by Readactor require very little extra memory. Apart from the fact that O-CFI requires precise binary static analysis as it aims to statically resolve return addresses, indirect jumps, and calls, the adversary may be able to disassemble the code, and reconstruct (parts of) the control-flow graph at runtime. Hence, the adversary could dynamically disclose how the control-flow is bounded.

Davi et al. [21] also propose a defense mechanism, Isomeron, that tolerates full disclosure of the code layout. To do so, Isomeron keeps two isomers (clones) of all functions in memory; one isomer retains the original program layout while the other is diversified. On each function call, Isomeron randomly determines whether the return instruction should switch execution to the other isomer or keep executing code in the current isomer. Upon each function return, the result of the random trial is retrieved, and if a decision to switch was made, an offset (the distance between the calling function f and its isomer f') is added to the return address. Since the attacker does not know which return addresses will have an offset added and which will not, return addresses injected during a ROP attack will no longer be used as is and instead, the ROP attack becomes unreliable due to the possible addition of offsets to the injected gadget addresses. Since Isomeron is implemented as dynamic binary instrumentation framework its runtime and memory overheads are substantially greater than those of Readactor.

Lastly, our work is related to a randomization approach specifically targeting JIT-compiled code called librando [33]. It deploys two randomization techniques: NOP insertion and constant blinding. In fact, as we mentioned before, Readactor can benefit from librando to enforce code randomization on

JIT-code beyond the coarse-grained randomization offered by the Google V8 engine. However, librando only targets JIT-code and not static code. Further, it provides no protection against direct disclosure attacks, and may slow down code by as much as 3.5x.

B. Integrity-checking Defenses

After DEP, Control-flow integrity (CFI) [1, 3] is the most prominent type of integrity-based defense. CFI constrains the indirect branches in a binary such that they can only reach a statically identified set of targets. Since CFI does not rely on randomization, it cannot be bypassed using memory disclosure attacks. However, it turns out that precise enforcement of control-flow properties invariably comes at the price of high performance overheads on commodity hardware. In addition, it is challenging (if not impossible) to always resolve valid branch addresses for indirect jumps and calls.

As a result, researchers have traded off security for performance by relaxing the precision of the integrity checks. CFI for COTS binaries [69] relies on static binary rewriting to identify all potential targets for indirect branches (including returns) and instruments all branches to go through a validation routine. CFI for COTS binaries merely ensures that branch targets are either call-preceded or target an address-taken basic block. Similar policies are enforced by Microsoft's security tool called EMET [42], which builds upon ROPGuard [25].

Compact control-flow integrity and randomization (CCFIR) is another coarse-grained CFI approach based on static binary rewriting. CCFIR collects all indirect branch targets into a *springboard* section and ensures that all indirect branches target a springboard entry. Our code-pointer hiding technique has similarities with the use of trampolines in CCFIR but the purposes differ. The springboard is part of the CFI enforcement mechanism whereas our trampolines are used to prevent indirect memory closure. Although the layout of springboard entries is randomized to prevent traditional ROP attacks, CCFIR does not include countermeasures against JIT-ROP.

A number of approaches have near-zero overheads because they use existing hardware features to constrain the control-flow before potentially dangerous system calls. In particular, x86 processors contain a last branch record (LBR) register which kBouncer [50], and ROPecker [17] use to inspect a small window of recently executed indirect branches.

Since CFI does not randomize the code layout, attackers can inspect the code layout ahead of time and carefully choose gadgets that adhere to a coarse-grained CFI policy [13, 29, 30].

SafeDispatch [36] and forward-edge CFI [63] are two compiler-based implementations of fine-grained forward CFI, which is CFI applied to indirect calls. The former prevents virtual table hijacking by instrumenting all virtual method call sites to check that the target is a valid method for the calling object. It offers low runtime overhead (2.1%) but only protects virtual method calls. Forward CFI is a set of similar techniques which protect both virtual method calls and calls through function pointers. It maintains tables to store trusted code pointers and halt program execution whenever the target of an indirect branch is not found in one of these tables. Even though both techniques add only minimal performance overhead, these approaches do not protect against attacks that use `ret`-terminated gadgets. Moreover, as no code randomization is applied, the adversary can easily invoke critical functions in complex programs that are also legitimately used by the program.

A number of recent CFI approaches focus on analyzing and protecting vtables in binaries created from C++ code [26, 52, 68]. Although these approaches do not require source code access, the CFI policy is not as fine-grained as their compiler-based counterparts. A novel attack technique for C++ applications, *counterfeit object-oriented programming* (COOP), undermines the protection of these binary instrumentation-based defenses by invoking a chain of virtual methods through legitimate call sites to induce malicious program behavior [56].

Code-Pointer Integrity (CPI) was identified as an alternative to CFI by Szekeres et al. [62] and was first implemented by Kuznetsov et al. [39]. CPI prevents the first step of control-flow hijacking during which the adversary overwrites a code pointer. In contrast, CFI verifies code pointers *after* they may have been overwritten. CPI protects control data such as code pointers, pointers to code pointers, etc. by separating them from non-sensitive data. The results of a static analysis are used to partition the memory space into a normal area and a safe region. Code pointers and other sensitive control data are stored in the safe region. The safe region also includes meta-data such as the sizes of buffers. Loads and stores that may affect sensitive control data are instrumented and checked using meta-data stored in the safe region. By ensuring that accesses to potentially dangerous objects, e.g., buffers, cannot overwrite control data, CPI provides spatial safety for code pointers. In 32-bit mode, CPI uses x86 segmentation to restrict access to the safe region, the same is not possible in 64-bit mode so the safe region is merely hidden from attackers whenever segmentation is not fully supported. Evans et al. [23] demonstrated that (in 64-bit mode) the size of the safe region is (in some implementations) so large that an attacker can use a corrupted data pointer to locate it and thereby bypass the CPI enforcement mechanism using an extension of the side-channel attack by Siebert et al. [57]. Unlike our approach, it remains to be seen whether CPI can be extended to protect dynamically generated code without degrading the JIT compilation latency and performance.

Recently, a new CFI solution has been proposed by Niu and Tan that also applies CFI to JIT-compiled code [46]. Their RockJIT framework enforces fine-grained CFI policies for static code and coarse-grained CFI policies for JIT-compiled code. Similarly to how we double-map physical host pages in the guest physical space, RockJIT maps the physical pages containing JIT-compiled code twice in virtual memory: once as readable and executable, and once as a readable and writable for the JIT compiler to modify—a *shadow code heap* accessible only to the JIT. This protects JIT-compiled code from code injection and tampering attacks, in addition to the protections provided by CFI enforcement. However, since only coarse-grained CFI is applied, the adversary can still leverage memory disclosure attacks to identify valid gadgets in JIT-compiled code and redirect execution to critical call sites in static code (i.e., calls that legitimately invoke a dangerous API function or a system call) to induce malicious program behavior.

XII. Conclusion

Numerous papers demonstrate that code randomization is a practical and efficient mitigation against code-reuse attacks. However, memory leakage poses a threat to all these probabilistic defenses. Without resistance to such leaks, code randomization loses much of its appeal. This motivates our efforts to construct a code randomization defense that is not only practical but also resilient to all recent bypasses.

We built a fully-fledged prototype system, Readactor, to prevent attackers from disclosing the code layout directly by reading code pages and indirectly by harvesting code pointers from the data areas of a program. We prevent direct disclosure by implementing hardware-enforced execute-only memory and prevent indirect disclosure through code-pointer hiding.

Our careful and detailed evaluation verifies the security properties of our approach and shows that it scales beyond simple benchmarks to complex, real-world software such as Google's Chromium web browser and its V8 JavaScript engine. Compared to prior JIT-ROP mitigations, Readactor provides comprehensive and efficient protection against direct disclosure, is the first defense to address indirect disclosure, and is also the first technique to provide uniform protection for both statically and dynamically compiled code.

We hope that forthcoming defenses will focus on countering return-into-libc and future variants of code reuse by building on our foundation of memory-leakage resilient software diversity.

Acknowledgment

The authors thank the anonymous reviewers, Mathias Payer, Robert Turner, and Mark Murphy for their detailed and constructive feedback.

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts D11PC20024 and N660001-1-2-4014 and by gifts from Oracle and Mozilla.

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 has been co-funded by the German Science Foundation as part of project S2 within the CRC 1119 CROSSING and the European Union's Seventh Framework Programme under grant agreement No. 609611, PRACTICE project.

References

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

[2] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. A theory of secure control flow. In *7th International Conference on Formal Engineering Methods*, ICFEM, 2005.

[3] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information System Security*, 13, 2009.

[4] AMD. Intel 64 and IA-32 architectures software developer's manual - Chapter 15 Secure Virtual Machine nested paging. http://developer.amd.com/resources/documentation-articles/developer-guides-manuals, 2012.

[5] J. Ansel, P. Marchenko, Ú. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. Biffle, and B. Yee. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2011.

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

[7] 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 SIGSAC Conference on Computer and Communications Security*, CCS, 2014.

[8] S. Bhatkar and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *14th USENIX Security Symposium*, 2005.

[9] 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.

[10] Black Duck Software, Inc. Chromium project on open hub. https://www.openhub.net/p/chrome, 2014.

[11] D. Blazakis. Interpreter exploitation: Pointer inference and JIT spraying. BlackHat DC, 2010.

[12] T. K. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS, 2011.

[13] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium*, 2014.

[14] S. Checkoway, L. Davi, A. Dmitrienko, A. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2010.

[15] X. Chen and D. Caselden. CVE-2013-3346/5065 technical analysis. http://www.fireeye.com/blog/technical/cyber-exploits/2013/12/cve-2013-33465065-technical-analysis.html, 2013.

[16] X. Chen, D. Caselden, and M. Scott. The dual use exploit: CVE-2013-3906 used in both targeted attacks and crimeware campaigns. http://www.fireeye.com/blog/technical/cyber-exploits/2013/11/the-dual-use-exploit-cve-2013-3906-used-in-both-targeted-attacks-and-crimeware-campaigns.html, 2013.

[17] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *21st Annual Network and Distributed System Security Symposium*, NDSS, 2014.

[18] F. B. Cohen. Operating system protection through program evolution. *Computers & Security*, 12, 1993.

[19] S. Crane, P. Larsen, S. Brunthaler, and M. Franz. Booby trapping software. In *New Security Paradigms Workshop*, NSPW, 2013.

[20] L. Davi, A. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium*, 2014.

[21] 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.

[22] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A. Sadeghi. Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *8th ACM Symposium on Information, Computer and Communications Security*, ASIACCS, 2013.

[23] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point: On the effectiveness of code pointer integrity. In *36th IEEE Symposium on Security and Privacy*, S&P, 2015.

[24] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *The 6th Workshop on Hot Topics in Operating Systems*, HotOS-VI, 1997.

[25] I. Fratric. ROPGuard: Runtime prevention of return-oriented programming attacks. http://www.ieee.hr/_download/repository/Ivan_Fratric.pdf, 2012.

[26] R. Gawlik and T. Holz. Towards automated integrity protection of C++ virtual function tables in binary programs. In *30th Annual Computer Security Applications Conference*, ACSAC, 2014.

[27] 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.

[28] 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*, 2012.

[29] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *35th IEEE Symposium on Security and Privacy*, S&P, 2014.

[30] E. Göktas, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *23rd USENIX Security Symposium*, 2014.

[31] J. L. Henning. SPEC CPU2006 memory footprint. *SIGARCH Computer Architecture News*, 35, 2007.

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

[33] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Librando: transparent code randomization for just-in-time compilers. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2013.

[34] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automatic software diversity. In *IEEE/ACM International Symposium on Code Generation and Optimization*, CGO, 2013.

[35] Intel. Intel 64 and IA-32 architectures software developer's manual - Chapter 28 VMX support for address translation. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[36] D. Jang, Z. Tatlock, and S. Lerner. SafeDispatch: Securing C++ virtual calls from memory corruption attacks. In *21st Annual Network and Distributed System Security Symposium*, NDSS, 2014.

[37] 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.

[38] S. T. King, P. M. Chen, Y. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. Subvirt: Implementing malware with virtual machines. In *27th IEEE Symposium on Security and Privacy*, S&P, 2006.

[39] 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.

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

[41] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *IEEE/ACM International Symposium on Code Generation and Optimization*, CGO, 2004.

[42] Microsoft. Enhanced Mitigation Experience Toolkit. https://www.microsoft.com/emet, 2015.

[43] Microsoft. Hyper-V. http://www.microsoft.com/hyper-v, 2015.

[44] V. Mohan, P. Larsen, S. Brunthaler, K. Hamlen, and M. Franz. Opaque control-flow integrity. In *22nd Annual Network and Distributed System Security Symposium*, NDSS, 2015.

[45] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine*, 11, 2001.

[46] B. Niu and G. Tan. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2014.

[47] Open Virtualization Alliance. KVM - kernel based virtual machine. http://www.linux-kvm.org.

[48] Oracle Corporation. VirtualBox. http://www.virtualbox.org.

[49] 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.

[50] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *22nd USENIX Security Symposium*, 2013.

[51] PaX Team. *Homepage of The PaX Team*, 2001. http://pax.grsecurity.net.

[52] A. Prakash, X. Hu, and H. Yin. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *22nd Annual Network and Distributed System Security Symposium*, NDSS, 2015.

[53] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information System Security*, 15, 2012.

[54] J. Rutkowska and A. Tereshkin. IsGameOver() anyone? In *BlackHat USA*, 2007.

[55] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. Evaluating the effectiveness of current anti-ROP defenses. In *17th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID, 2014.

[56] 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.

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

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

[59] 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.

[60] C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski. Exploiting and protecting dynamic code generation. In *22nd Annual Network and Distributed System Security Symposium*, NDSS, 2015.

[61] 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.

[62] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *34th IEEE Symposium on Security and Privacy*, S&P, 2013.

[63] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *23rd USENIX Security Symposium*, 2014.

[64] VMware, Inc. VMware ESX. http://www.vmware.com/products/esxi-and-esx/overview.

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

[66] Xen Project. Xen. http://www.xenproject.org.

[67] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *34th IEEE Symposium on Security and Privacy*, S&P, 2013.

[68] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song. VTint: Defending virtual function tables' integrity. In *22nd Annual Network and Distributed System Security Symposium*, NDSS, 2015.

[69] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *22nd USENIX Security Symposium*, 2013.

APPENDIX

Figure A provides a detailed overview of Readactor.

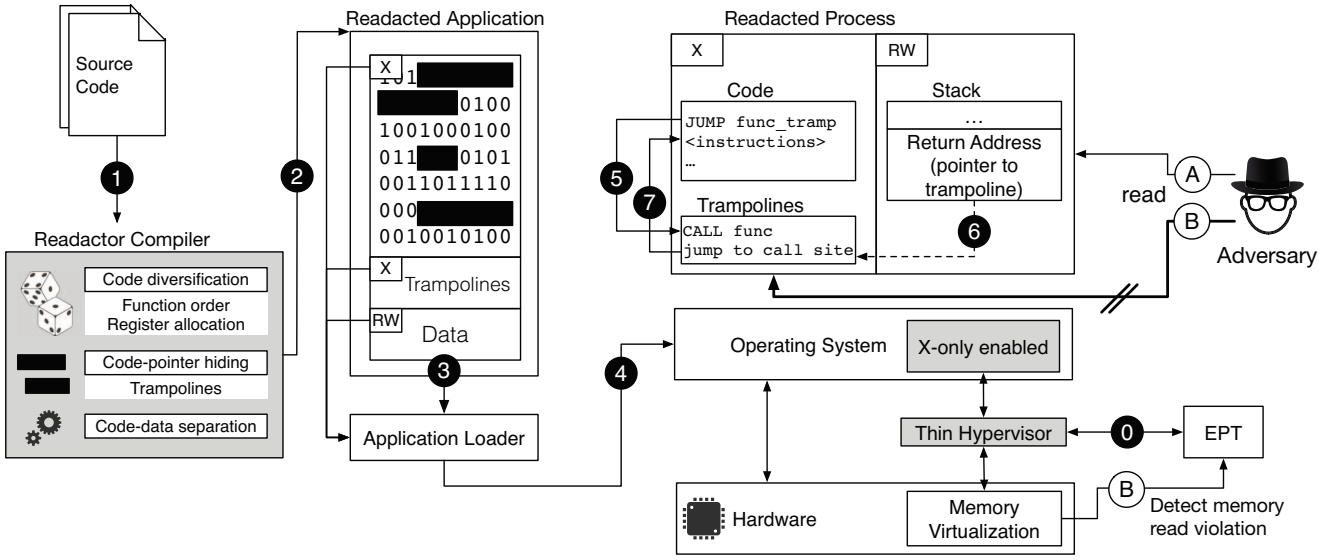

Figure A: Overview of Readactor. Components marked in gray are part of Readactor.

System Components:

❶ Compilation: The compiler takes the source code of an arbitrary program and creates a binary. The compiler (i) strictly separates code and data, (ii) applies code diversification in the form of function permutation, register allocation randomization and save slot reordering, and (iii) implements code-pointer hiding, by creating a trampoline, as `jmp <dst>` instruction for every code pointer (e.g., return address destinations).

❷ Binary: The output binary contains different sections for code, trampolines and data. The linker marks appropriate access permissions for each section.

❸ Loader: The loader reads the size and permissions bits of each section and allocates the respective memory regions, protecting them with the requested permissions (❹).

❺ Code-pointer hiding: In order to hide code pointers during runtime, calls are substituted with a `jmp` instruction to the corresponding trampoline. The trampoline will then call the original function, which pushes a return address on the stack. However, the return address (❻) will not point into the code section, but to the trampoline section. As described in Section IX, disclosing trampoline pointers will not provide any knowledge to the adversary about the layout or content of the code section. Once the called function returns to the trampoline, control flow is returned to the original call site through another `jmp` instruction (❼). We similarly protect function pointers with `jmp` trampolines (not shown).

Attack Scenarios:

Ⓐ Reading data memory: Data regions remain readable and writable. Hence, the adversary can disclose and modify code pointers. However, the disclosed code pointers do not provide any information about the applied code randomization and can therefore not be used to create a ROP gadget chain (cf. Section IX).

Ⓑ Reading code memory: Code regions are set to execute-only. Any attempt to read these regions causes an EPT exception which is forwarded to the operating system. An application that causes an execute-only exception by attempting to read or write a code region is immediately killed by the operating system. Since the execute-only permission is enforced in hardware it cannot be bypassed by software.