# Return to the Zombie Gadgets: Undermining Destructive Code Reads via Code Inference Attacks

Kevin Z. Snow, Roman Rogowski, Fabian Monrose
*Department of Computer Science*
*University of North Carolina at Chapel Hill, USA*
*Email: kzsnow,rogowski,fabian@cs.unc.edu*

Jan Werner
*Renaissance Computing Institute (RENCI)*
*Chapel Hill, USA*
*Email: jjwerner@cs.unc.edu*

Hyungjoon Koo, Michalis Polychronakis
*Department of Computer Science*
*Stony Brook University, USA*
*Email: hykoo,mikepo@cs.stonybrook.edu*

***Abstract*—The concept of destructive code reads is a new defensive strategy that prevents code reuse attacks by coupling fine-grained address space layout randomization with a mitigation for *online* knowledge gathering that destroys potentially useful gadgets as they are disclosed by an adversary. The intuition is that by destroying code as it is read, an adversary is left with no *usable* gadgets to reuse in a control-flow hijacking attack. In this paper, we examine the security of this new mitigation. We show that while the concept initially appeared promising, there are several unforeseen attack tactics that render destructive code reads ineffective in practice.**

**Specifically, we introduce techniques for leveraging *constructive reloads*, wherein multiple copies of native code are loaded into a process' address space (either side-by-side or one-after-another). Constructive reloads allow the adversary to disclose one code copy, destroying it in the process, then use another code copy for their code reuse payload. For situations where constructive reloads are not viable, we show that an alternative, and equally powerful, strategy exists: leveraging *code association via implicit reads*, which allows an adversary to undo in-place code randomization by inferring the layout of code that follows already disclosed bytes. As a result, the implicitly learned code is not destroyed, and can be used in the adversary's code reuse attack. We demonstrate the effectiveness of our techniques with concrete instantiations of these attacks against popular applications. In light of our successes, we argue that the code inference strategies presented herein paint a cautionary tale for defensive approaches whose security blindly rests on the perceived inability to undo the application of in-place randomization.**

***Index Terms*—memory disclosure; code reuse; return-oriented programming; application security; fine-grained randomization**

## I. Introduction

Despite decades of research into application-level defenses, a multitude of vulnerabilities continue to be discovered and exploited in today's feature-rich software ecosystem, including web browsers, email clients, and document readers. A seemingly endless cycle marches onwards—defenses address known problems, new problems arise or prior assumptions are invalidated, then attacks quickly bypass those previous defenses, ad infinitum. For instance, the *no-execute* (NX) primitive was introduced to prevent the execution of malicious *code injection* into exploited program memory. As a result, attackers turned to constructing their malicious program logic by chaining together existing code fragments within the exploited program. These *code reuse* attacks either link together entire functions (*i.e.*, return-to-libc attacks) or combine short instruction sequences (dubbed gadgets) ending with `ret`, `call`, or `jmp` instructions.

In computer security parlance, this is called return-oriented programming (ROP) [40] or jump-oriented programming (JOP) [15, 10], depending on the type of gadgets used. To thwart these attacks, *address-space layout randomization* (ASLR) was widely adopted over the past decade as a means to obfuscate the location of code necessary to construct these code reuse payloads. Soon thereafter, ASLR was bypassed by leveraging a *memory disclosure* vulnerability to disclose a code pointer, then subsequently using this information to align malicious code pointers with gadgets in the newly discovered code region.

To address the shortcomings of contemporary ASLR, Bhatkar et al. [6] proposed diversifying software between subsequent runs using an instantiation of what is now called *fine-grained* randomization. In short, the idea is to not only randomize code region locations, but also the functions within those regions, the basic blocks within those functions, as well as the instructions and registers therein. In doing so, leaked function pointers presumably provide too little information to derive the location of useful ROP or JOP gadgets, thus preventing this form of code reuse. Given the promise of this idea, a plethora of fine-grained ASLR schemes have appeared in the academic literature [35, 24, 47, 23, 28, 7, 19], each with their own advantages and disadvantages. In general, however, all of the various instantiations of fine-grained randomization achieve their goal by enforcing a policy that attempts to prevent the adversary from inferring the location of gadgets given knowledge of leaked code pointers. However, applications have since evolved to commonly provide dynamic features, such as interactive scripting, that fundamentally alter the adversarial model. That is, ASLR depends on the secrecy of code region addresses, but a script controlled by the adversary provides an opportunity for leaking this secret prior to exploitation. Unfortunately, shortly after these works


DOI 10.1109/SP.2016.61

began gaining traction, Snow et al. [42] presented a technique called *just-in-time code reuse* (JIT-ROP), which expands this technique to leak not just code pointers, but also the code itself by following those pointers and recursively disclosing each discovered page of code. Afterwards, the leaked code is inspected, gadgets are identified, and a payload is compiled just-in-time.

Spurred by the just-in-time attack paradigm, numerous defenses that adapt, or improve upon, fine-grained ASLR have recently been proposed to better fortify applications against these attacks. Generally speaking, the approaches they take can be categorized as either attempting to prevent the disclosure of code [3, 4, 21, 16, 11], or attempting to prevent the execution of disclosed code [45, 48]. Of these approaches, only the work of Backes et al. [4], called `XnR`, and the concept of destructive reads proposed by Tang et al. [45] (dubbed `Heisenbyte`) and Werner et al. [48] (called `NEAR`) are specifically designed to support commodity software, *i.e.,* they forgo the requirement of source code access and recompilation.

In this work, we shed light on the difficulty of designing binary-compatible techniques for *preventing the execution of disclosed code*, especially when real-world constraints are taken into consideration. For example, the use of dynamically loaded shared libraries, just-in-time code generation engines (*e.g.,* JavaScript or ActionScript), and the adversary's ability to control some aspects of program operation (such as opening new browser tabs) are often neglected in the design of application-level defenses, but are requisite features to support performance and user experience goals. We focus on the approaches of Tang et al. [45] and Werner et al. [48] which both attempt to ensure that *code or data can be executed if and only if it has not been previously read*. The instantiation of this relaxed property is meant to avoid disruption of any legitimate reads to data embedded within code segments, which cannot be precisely identified during static analysis, due to the complexity of many closed-source applications. At the same time, this property ensures that gadgets revealed via a memory disclosure cannot be later executed, as they have been "destroyed" in the execute-only mapping of the same locations. Sadly, we show that this primitive fails to account for two new classes of attacks that are relatively easy to mount.

Our specific contributions are as follows:

- We introduce three new security properties that we argue must be considered by application defenses that prevent the execution of disclosed code; namely, code *persistence*, *singularity* and *dis-association*.
- We show that implementing the notion of destructive code reads for commodity applications is more difficult than first thought, especially in light of the adversary's ability to break each of the aforementioned properties in recently proposed work [45, 48].
- We explore novel methods of undermining binary compatible fine-grained randomization, which has potential implications for the larger body of work that relies on the notion of fine-grained randomization in general.
- We highlight the need for a series of potential new directions for application defenses, such as the need for binary-compatible randomization schemes, that prevent code inference and the use of re-randomization at strategic "trigger points."

## II. Background and Related Work

For pedagogical reasons, we quickly review two concepts that are key to the remainder of this paper, namely fine-grained layout randomization and just-in-time code reuse attacks.

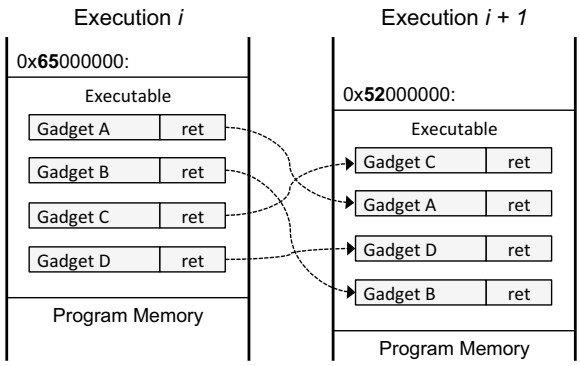


Fig. 1. Fine-grained randomization changes the location of the gadgets present in a code segment.

Fine-grained randomization [6, 35, 24, 47, 23, 28, 7, 19] attempts to address well-known weaknesses [32, 22, 41, 38, 44, 26, 5] of contemporary ASLR by not only randomizing the locations of memory regions, but also shuffling functions, basic blocks, instructions, registers, and even the overall structure of code and data. The outcome of this diversification process is that the locations of any previously pinpointed gadgets are arbitrarily different in each instance of the same code segment, as illustrated in Figure 1. Even so, Snow et al. [42] showed that fine-grained randomization alone, even assuming a perfect implementation, does not prevent all control-flow hijacking attacks that leverage code reuse. Consider, for instance, a leaked code pointer that is not used to infer the location of gadgets, but is rather used along with a memory disclosure vulnerability to reveal the actual code bytes at the referenced location.

As depicted in Figure 2, a just-in-time code reuse attack uses the initially disclosed code pointer to recursively map out as many code pages as possible in step ❶. As it does so, gadgets and system API pointers are identified, then a ROP payload is compiled on-the-spot, during the course of exploitation. The payload is returned to the adversary's script and control flow is hijacked to execute the payload in step ❷. The attack is fundamentally enabled by the adversary's ability to repeatedly leverage a memory disclosure to leak actual code bytes, rather than only code pointers. In response, a number of works have emerged that attempt to mitigate just-in-time code reuse attacks by either (*i*) making code regions executable but non-readable—thus preventing code disclosure in step ❶, or

(*ii*) allowing code to be disclosed, but preventing the execution of that disclosed code in step ❷.

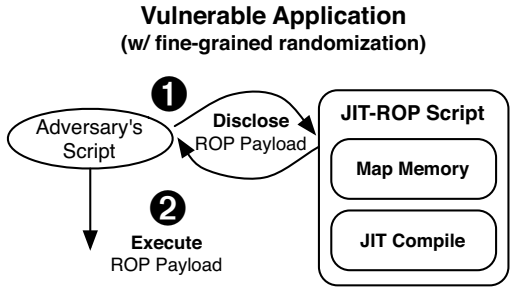


Fig. 2. In a JIT-ROP attack, code is disclosed to generate a ROP payload on-the-fly, and then program control flow is redirected to execute the gadgets of the assembled payload.

### A. Preventing Code Disclosure

The approaches that attempt to stop attack progression at step ❶ are, for the most part, instantiations of the long-standing idea of execute-only memory (XOM) [31] but applied to contemporary OSes. Backes and Nürnberger [3], for example, propose an approximation of XOM by eliminating all code-to-code pointer references—thus preventing the recursive code memory mapping step of just-in-time code reuse. To do so, a special translation table mediates all function calls. Unfortunately, such an approach requires heavy program instrumentation, and the necessary program transformations are only demonstrated to be achievable given source-level instrumentation. Additionally, Davi et al. [18] later showed that just-in-time payloads can still be constructed even in the absence of code-to-code pointers.

Crane et al. [16, 17], Brookes et al. [12], and Braden et al. [11] approach the problem of execute-only memory by starting with the requirement of source-level access. Hence, the many challenges that arise due to computed jumps and the intermingling of code and data in commodity (stripped) binaries, are alleviated. Readactor [16], for example, relies on a modified compiler to ensure that all code and data is appropriately separated. Execute-only memory is then enforced by leveraging finer-grained memory permission bits made available by extended page tables (EPT) in recent processors. Likewise, Brookes et al. [12] also leverage Intel's EPT to provide an execute-only memory approach (called ExOShim), but their approach does not provide the strong guarantees of Crane et al. [16, 17].

Gionta et al. [21] also approach execute-only memory with consideration for intermixed code and data. To do so, data embedded in code sections is first identified via binary analysis (with symbol information), then split into separate code-only and data-only memory pages. At runtime, instruction fetches are directed to code-only pages, while reads are directed to data-only pages. Conditional page direction is implemented by loading split code and data translation look-aside buffers (TLBs) with different page addresses. One drawback of this approach is that all recent processors now make use of a unified second-level TLB, rendering this approach incompatible with modern hardware. Moreover, static code analysis on binaries is a difficult problem, leaving no other choice but to rely on human intervention to separate code and data in closed-source software. Thus, the security guarantees imbued by that approach are only as reliable as the human in the loop.

Unlike the aforementioned works that require source code [3, 16, 17], debug symbols, or human intervention [21], Backes et al. [4] take a different approach that is geared toward protection for commercial off-the-shelf (COTS) binaries. A software implementation of execute-only-memory is provided wherein all code regions are initially made inaccessible, which causes a kernel-mode memory fault when code is executed or read. A process is terminated if it attempts to *read* from a code section, but is made accessible for code *execution*. The accessible code page is again made inaccessible when a fault occurs on a different page. Unfortunately, rather than confronting the challenges of intermixed code and data head-on, the approach of Backes et al. [4] allows for a limited number of pages to be accessible at any point in time, and increments a counter for each first instance of a code page read while decrementing on execution. An application is terminated if some undefined threshold is reached, presumably indicating that too many reads of bytes of code have occurred. Given the lack of guidance on how an appropriate threshold can be set in practice, the security guarantees afforded by `XnR` remain unclear. Note that systems like XnR [4], which temporarily allow an attacker to non-destructively read significant parts of code, can also be bypassed by simply gathering gadgets from those temporarily available pages in a JIT-ROP fashion by reading them directly.

### B. Preventing Disclosed Code Execution

Heisenbyte [45] takes a radically different approach that instead focuses on the concept of *destructive reads*, whereby code is garbled after it is read. By taking advantage of existing virtualization support (*i.e.,* EPT) and focusing solely on thwarting the execution of disclosed code bytes, Heisenbyte's use of destructive code reads sidesteps the many problems that arise due to incomplete disassembly in binaries, and thereby affords protection of complex close-sourced COTS binaries. Similarly, `NEAR` [48] implements a so-called *no-execute-after-read* memory primitive using EPT on *x86* Windows and other hardware primitives on *x86-64* and *ARMv8* which, instead of randomly garbling code, substitutes fixed invalid instruction(s), hence ensuring that subsequent execution always terminates the application. `NEAR` also demonstrates how valid data within code sections can be automatically and reliably relocated on-load, without the use of source code or debug symbols, which significantly reduces average runtime overhead from 16.48% in Heisenbyte to 5.72% in `NEAR`.

Both Tang et al. [45] and Werner et al. [48] provide an excellent overview of how destructive reads can be implemented by leveraging EPT and conservatively relocating intermingled code and data during an offline analysis phase. When a

protected application loads, a duplicate copy of its executable memory pages is maintained, and that copy is used in the event of a memory read operation. To detect read operations in an executable memory page, the page is originally marked as execute-only. For the purposes of this paper, the techniques used to separate code and data are not of particular importance, but interested readers are referred to Tang et al. [45][§4] for more details.

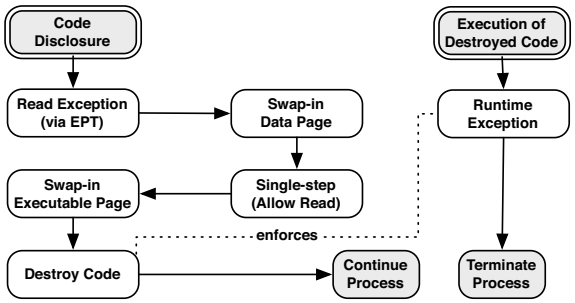


Fig. 3. Overview of destructive reads.

In our evaluations we use the NEAR implementation for the 32-bit Microsoft Windows platform[1]. We emphasize, however, that the security implications outlined in this paper are equally applicable to both approaches. The high-level overview is shown in Figure 3. NEAR also leverages EPT for marking code pages as execute-only for the sole purpose of intercepting the disclosure of code. Similar to Heisenbyte, a kernel module is used to hook the system page fault handler, identify newly loaded code pages, and communicate with the hypervisor module to mark those pages as execute-only. The act of reading a byte of code results in an invalid opcode being written to that location in memory, which would generate an exception in the event that the previously disclosed code is later executed. Further, we reiterate that fine-grained ASLR is a *prerequisite* for execute-only memory protections, as without randomization there is no need for an adversary to dynamically disclose code at runtime. To fulfill this requirement, we use the publicly available implementation of in-place code randomization [35].

## III. Assumptions

As noted earlier, unlike the recent body of work on preventing code disclosure attacks [3, 21, 16, 17, 18, 12], Tang et al. [45] suggest a new defensive model wherein preventing the execution of disclosed code is the ultimate goal. To be able to achieve this goal for commodity software (*i.e.,* without requiring access to source code), several important assumptions underscore the design of destructive code reads. Specifically, it is assumed that a suite of application defenses are in place that prevent all control-flow hijacking attacks, with the exception of so-called just-in-time code reuse attacks (JIT-ROP) [42]. Destructive reads, therefore, are the proposed solution to mitigating such attacks. Hence, in this paper, we also assume that the following mitigations are in place:

- **Non-Executable memory**: The stack(s) and heap(s) of the protected program are non-executable, thus preventing an attacker from directly injecting new executable code into data regions. Furthermore, all executable regions (including those of shared libraries) are non-writable; thus the modification of existing code is not possible.
- **Fine-grained randomization**: Program and library code regions are randomized using binary-compatible transformations on the ordering of registers and instructions within basic blocks. To achieve binary compatibility, we assume the techniques of Pappas et al. [35] are employed, as no other proposed approach meets these goals without auxiliary information (*i.e.,* source code or debug symbols) for complex COTS software.
- **JIT mitigations**: Browser-specific defenses against JIT-spraying instructions useful for code injection and code reuse attacks are in use. For example, Internet Explorer includes countermeasures that share commonalities with Librando [25].
- **Destructive Code Reads**: We assume that the act of reading *any* byte of code immediately precludes that specific byte of code from being executed later.

The aforementioned assumptions are in accordance with those of both the just-in-time code reuse paradigm given by Snow et al. [42] and the destructive code read defense presented by Tang et al. [45] and Werner et al. [48]. We remind the reader that the fact that binary compatibility is required, means that only in-place fine-grained code diversification can be assumed. We elaborate on the reasons for that next.

### A. Implications of Binary-compatible Fine-grained ASLR

Binary compatible fine-grained ASLR transformations are (thus far) only able to reliably achieve a subset of those transformations possible in schemes that randomize code by utilizing program source code (or debug symbols). The reason for this disparity is that binary code analysis is a provably undecidable problem [46]. Because of this fact, the use of more aggressive fine-grained randomization strategies, such as randomizing the location of functions or basic blocks, or randomly inserting inaccessible guard regions between pages of code, remains a significant challenge. Existing attempts in doing so at the binary level, such as Binary Stirring [47], which rearranges all basic blocks present in a code segment, rely on fragile heuristics for the recovery of jump tables, computed jump targets, callback routines, and other code intricacies that complicate code disassembly. Indeed, Wartell et al. [47] demonstrate the applicability of Binary Stirring using solely main executables (not libraries) of simple utility programs. Note that although binary metadata such as relocation information or export tables are leveraged by state-of-the-art disassemblers to improve the coverage and accuracy of code identification and control flow graph extraction, imprecisions still exist due to the above code intricacies.

[1]The thin hypervisor implemented by Werner et al. [48] is available under an open source license at *github.com/uncseclab*

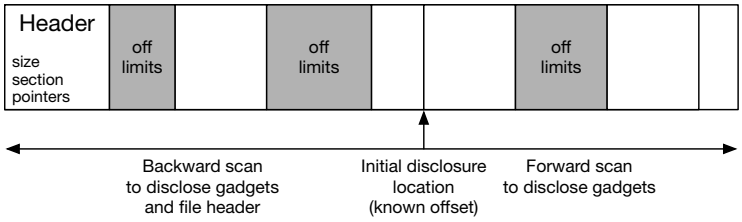


Fig. 4. In contrast to the original JIT-ROP attack [42], the absence of guard pages allows for linear scanning of memory, and the disclosure of any code that is not within the program's normal execution path (represented by the gray areas).

Instead, binary compatible fine-grained ASLR schemes [35, 29] perform code transformations in-place, *i.e.,* as in rearranging code within a basic block, without changing its location, rather than globally moving large swaths of code, which requires identifying and rewriting *all* code that interacts with the moved sections; again, an undecidable problem [46].

Thus, one can conclude that a just-in-time code reuse attack can take a simpler approach to identifying code pages than that presented by Snow et al. [42]. In that work, the existence of guard pages (*i.e.,* randomly introduced in-accessible pages) were dealt with by disassembling each known page of code and only queuing new pages observed as instruction operands in the first page. Instead, in our case, the adversary can precompute the offset from the initially disclosed function pointer to the beginning of the module that contains a function, as is routinely done by exploits in-the-wild as the first step in bypassing coarse-grained ASLR. Furthermore, even if functions are randomized (which has not been shown to be practical to date for complex COTS binaries), the adversary could walk backwards in memory, page-by-page, until the first few bytes of the page match the binary-format header for the target platform.

Figure 4 illustrates these simplified strategies. In the module depicted, the initially disclosed pointer remains at a fixed offset between randomizations with in-place randomization. Hence, one either computes the offset of that initial pointer offline prior to attack, or steps backwards carefully avoiding off-limits pages at runtime while searching for the binary header. In face of destructive code reads, the off-limits regions represent portions of code that would be executed during normal program operation, and hence we must avoid destroying that code.

Given that the adversary now knows the base address of a given module, one can parse the information in that header to determine the module layout, including start and end addresses, as well as the location of import tables. Thus, one can use this information to directly disclose the *entire* code region of a module, as well as obtain references to all other modules referenced by that module. In short, binary compatible fine-grained ASLR provides the attacker with the opportunity to acquire a superset of the gadgets obtained via the so-called `JIT-ROP` attack [42]. However, up to this point, destructive code reads still prevent the adversary from using any discovered gadgets by destroying the code as it is read.

### B. Adversarial Assumptions

Similar to Snow et al. [42] and Tang et al. [45], we assume an adversary who can read and write arbitrary memory of the vulnerable process. Additionally, we assume that the adversary is capable of running scripted code within the limits of the attacked application (*e.g.,* JavaScript or ActionScript code) and storing the gathered information either locally, *e.g.,* in cookies or in HTML5 Local Storage [1], or on a remote server.

At this point, it is prudent to note that the concept of destructive reads only works in cases where the following (implicit) assumptions hold:

- **Code Persistence**: Code may not be loaded and unloaded by the adversary. This assumption guarantees that an adversary may not restore destroyed code after learning its layout.
- **Code Singularity**: The process may not contain any duplicate code sections. This assumption guarantees that an adversary may not infer any information about the code in process memory by reading another existing copy of that code.
- **Code Dis-association**: Any information discovered during an attempted attack can not be relied upon in subsequent attacks. This assumption is needed to ensure that an adversary cannot mount an incremental attack against an application disclosing partial information and then reusing it in the next stage of an attack.

Unfortunately, these assumptions are easily broken in practice, especially in scripting environments. In what follows, we explore how breaking any of those implicit assumptions results in the complete compromise of the security afforded by destructive code reads, thereby allowing the attacker to re-enable the use of just-in-time code reuse attacks.

## IV. Undermining Destructive Reads

Next, we detail four distinct strategies to just-in-time disclose a usable code reuse payload in face of destructive code reads. Each of these strategies breaks one of the three implicit assumptions detailed in section §III-B to accomplish the stated goal. In short, we first demonstrate the need for code persistence by building code reuse payloads from shared library code that can be unloaded and *reloaded* by the adversary on demand through scripting or creating a new process. We also explore the need for code singularity by demonstrating that

an adversary can load similar copies of JIT-compiled code that contains fixed usable gadgets. Lastly, we detail a more concerning attack that entails *implicit* reading of code, thus avoiding code destruction altogether. This attack turned out to be more powerful than we first envisioned, and motivates the need for a stronger dis-association property that is not present in any binary-compatible fine-grained randomization scheme we are aware of.

### A. Code Cloning via JIT Script Engines

The first strategy for defeating destructive code reads (which we refer to as *code cloning*) targets the weak assumption of code singularity. That is, the strategy of destroying code after it is read only provides a benefit if multiple copies of the target code can not co-exist in memory. This assumption holds in most cases (*e.g.,* program code sections are unique, only one copy of shared libraries are loaded in a process), but when it does not, attackers can disclose and destroy one copy of the code while using the second copy of the code to execute their just-in-time payload.

Unfortunately, there exists at least one ubiquitous practice that breaks the code singularity assumption—*just-in-time (JIT) compilation*. Web browsers and document readers provide JIT compilers for JavaScript (as well as ActionScript and Java). Worse yet, the JIT-compilation process can be precisely controlled by the adversary through scripts embedded in web pages or documents [36, 9]. To this end, we designed and implemented a JavaScript JIT code cloner that enables an adversary to create two native code regions from the same source JavaScript code, then to disclose gadgets from the first code region which are immediately destroyed, and finally to execute those same gadgets using a payload that references the gadgets still available in the second code region.

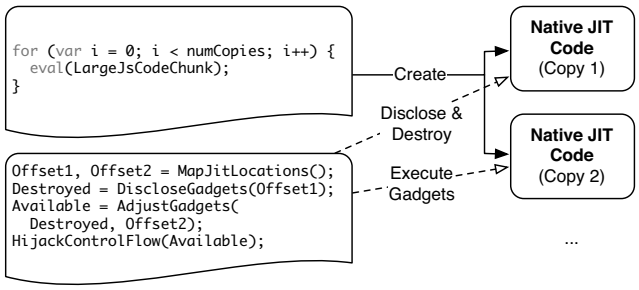


Fig. 5. Countering destructive code reads with JavaScript JIT cloning.

The high-level work flow is depicted in Figure 5. The adversary first triggers compilation of a large segment of JavaScript code using an *eval* statement. After the code is generated, loaded in memory, and protected by destructive reads, the adversary must identify the location of these JIT-compiled code regions.

For pedagogical reasons, the sample code shown in Figure 5 was used to create an abundance of JIT-compiled copies to aid our understanding of how an adversary can initially navigate to the JIT-compiled regions using a memory disclosure vulnerability. In short, we found that in both Internet Explorer and Firefox, the global data section of a shared library related to JavaScript functionality contains a pointer (at a fixed offset) to a heap region that in turn points to the JIT-compiled code region(s). Thus, an adversary would simply use the memory disclosure vulnerability to follow a series of data pointers to eventually arrive at the beginning of each JIT-compiled region.

Once there, the attack continues with a full code disclosure of *one* of these regions, which triggers the destruction of the code that was read. After disclosing enough code, the learned knowledge of gadgets from the destroyed section is used by adjusting their address by a fixed offset (the difference in the JIT-compiled code region base addresses) and executing the payload in the code region clone. In Section V, we elaborate on the gadgets that can be generated and discovered via JIT-cloning, as well as the effects of existing JIT-spraying mitigations.

### B. Code Non-Persistence via Shared Library Reloading

The JIT-cloning attack subverts destructive code reads based on the observation that program code regions are not always unique. We now turn our attention to whether the assumption of code persistence holds in practice. That is, we consider which techniques one might apply to modify (or *restore*) code that has previously been destroyed. Unfortunately, we need not look far, as during our preliminary explorations we observed shared libraries being dynamically loaded and unloaded during normal program operation. Indeed, the dynamic load and unload capability of shared libraries provides programs with several benefits, such as the ability to support third-party plug-ins and the ability to transiently leverage infrequently used features, which minimizes the application memory footprint.

The problem, of course, is that if one can load and unload libraries on demand, destructive code reads are immediately subverted. To see why, consider an adversary that first loads a library of interest, then discloses and destroys the library code, triggers the unloading of the library, and finally triggers the loading of a fresh copy once again. Practically, however, one cannot directly load and unload libraries within the context of an embedded script. Nevertheless, there exist a multitude of opportunities for *indirectly* reloading shared libraries.

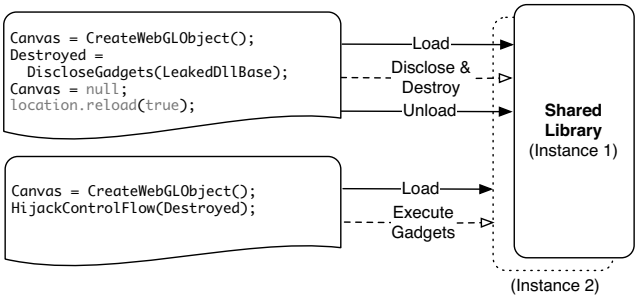


Fig. 6. Countering destructive reads with shared library reloading via embedded JavaScript.

Without much effort, we were able to identify an instance of indirect library reloading by launching a web benchmark suite in a browser while monitoring all library loading and unloading operations. The benchmark tests a number of auxiliary features, including an analysis of how well the browser renders a WebGL graphic canvas using JavaScript by dynamically writing the WebGL object to the rendered HTML. During that test, we observed that Microsoft Internet Explorer 11 loads the graphics library `d3d10warp.dll` to handle rendering, and then unloads that library when the benchmark proceeds to the next test (via a page refresh).

We leverage this observation in the attack depicted in Figure 6. The behavior is not unique to IE, as we found that Firefox (version 41) transiently loads a number of libraries, including `urlmon.dll`, to handle the download of a compressed file. The download action can be automatically and transparently invoked via a snippet of JavaScript code embedded in a web page controlled by the adversary. These libraries are also unloaded when the page is refreshed, and thus enable one to disclose, destroy, refresh, and then execute those disclosed gadgets.

Library reloading enables the disclosure of gadgets from one library at a time, but an even more direct approach that allows the reloading of *all* program code at once exists. The strategy for doing so is explored next.

### C. Code Non-Persistence via Process Reloads

Modern applications, such as web browsers, empower the adversary with a startling level of control over their target system, even prior to hijacking control-flow. As previously shown, we can load and unload specific libraries in the target process. We can also create new processes on the target system, for example, by embedding a small snippet of JavaScript code (*e.g.,window.open(url)*) in a web page, which renders the target web page in a new browser tab. Modern browsers attempt to strongly enforce the same-origin policy and sandbox potential exploits by launching these tabs in completely separate processes.[2] In countering destructive reads, one can view these adversary-created tabs as *disposable sources for gadget disclosure*.

The high-level work flow of an attack that takes advantage of multiple processes is presented in Figure 7. First, the attacker triggers a new process (Process 3) by creating a separate browser tab via JavaScript, then discloses gadgets in all accessible code regions, destroying that code in the process and relaying the gadget information on-the-fly to the original process. This can be achieved either using an adversary-controlled web server as an intermediary, or locally by leveraging capabilities of the HTML5 local storage API [1].

Next, the payload is built and executed in the original process whose code has not been destroyed. The challenge is in taking care to avoid disclosing and destroying code in the tabbed process that is necessarily executed in the work flow of relaying the disclosed gadget information. However, that can be easily overcome by recording the requisite code paths offline, prior to deploying the exploit, and then simply skipping disclosing any code bytes in that path during the just-in-time disclosure phase. As binary compatible fine-grained ASLR (*i.e.,* in-place randomization) is assumed, we can accurately blacklist byte ranges to avoid those that remain consistent across different instances of the same randomized library.

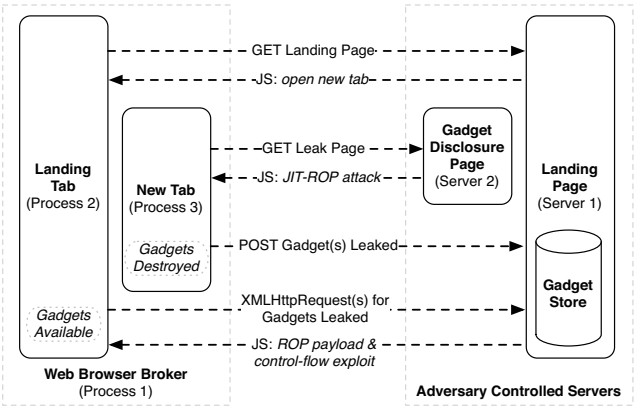


Fig. 7. Countering destructive code reads with process reloading using multiple browser tabs.

We later show in Section V that even when avoiding the disclosure of code in the work flow execution path, one can identify gadgets of all the requisite types needed to just-in-time compile a code reuse payload. However, before doing so, we present a more insidious attack that takes advantage of specific limitations of the *combination* of destructive code reads with binary-compatible code randomization.

### D. Code Association via Implicit Reads

To allow for precise differentiation between code and data embedded within code segments, execute-only memory using destructive reads is enforced at a byte-level granularity. Although this approach effectively prevents the execution of code that has been previously read, its implications regarding an attacker's ability to *infer* the layout of code that *follows* already disclosed bytes requires careful consideration. It is conceivable that depending on the applied code randomization strategy, reading only a few bytes of existing code might be enough for making an informed guess about the instructions that follow the disclosed code without actually reading them.

This issue is particularly pertinent for binary-only defenses that rely on specific, fine-grained code randomization techniques due to the imprecision of code disassembly and static analysis. In-place code randomization [35], for instance, employs a set of narrow-scoped code transformations that probabilistically alter the functionality of short instruction sequences that can be used as gadgets. In what follows, we discuss how an attacker can infer the particular randomized instance of a gadget for each of the four code transformations used by in-place code randomization [35], as well as the recent instruction

[2]In practice, some browsers heuristically decide when a tab should be created as a separate process.

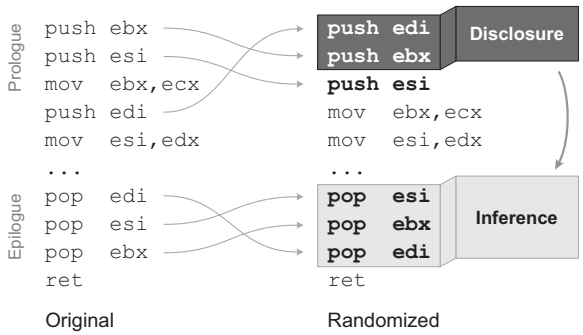


Fig. 8. Example of indirect disclosure against register preservation code reordering. By disclosing the first two instructions of a randomized function's prologue, an attacker can precisely infer the structure of the randomized gadget at the function's epilogue.

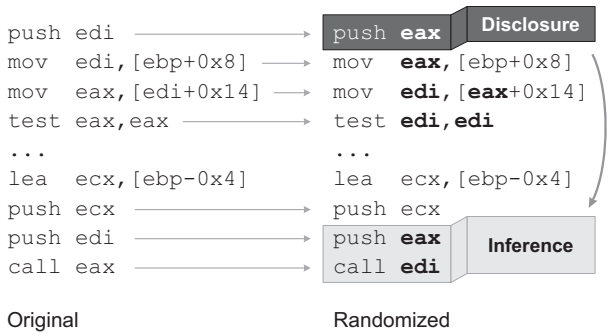


Fig. 9. Example of indirect disclosure against register reassignment. By disclosing just one instruction that involves a reassigned register at the beginning of its live region, an attacker can precisely infer what registers are used in the randomized instructions of the gadget.

displacement technique [29]. We have evaluated the feasibility of indirect code disclosure against the two most effective (in terms of gadget coverage) of these transformations, and as we show in Section V-C, we can infer the randomized state of the vast majority of all randomized gadgets.

*Instruction substitution* is a randomization strategy that replaces existing instructions with functionally-equivalent ones, with the goal of altering any overlapping instructions that are part of a gadget. Given that the original binary code of a program and the sets of equivalent instructions are common knowledge, an attacker knows a priori all instructions that are candidates for substitution. By just reading the opcode byte of a candidate-for-substitution instruction in the randomized instance of a program, an attacker can precisely infer the sequence of bytes that follow the opcode byte (*i.e.,* the instruction's operands), and consequently, the state of any overlapping randomized gadget. If the disclosed opcode is also part of the randomized gadget, however, the part of the gadget that starts after the opcode byte will remain usable.

*Basic block instruction reordering* is another common code randomization approach that changes the order of instructions within a basic block according to an alternative, functionally equivalent instruction scheduling. By precomputing all possible orderings of a given basic block, an attacker may be able to infer the order of instructions towards the end of the block by just reading a few instructions from the beginning of the block. The feasibility of this inference approach for a given gadget depends on the size of the basic block in which the gadget is contained, the location of the gadget within the block, and the number of possible instruction orderings.

*Register preservation code reordering* changes the order of the `push` and `pop` instructions that are often found at a function's prologue and epilogue, respectively. These instructions are used to preserve the values of callee-saved registers that would otherwise be overwritten by the current function. Since the registers are restored in the reverse order in which they were `push`'ed, an attacker can precisely infer the structure of "`pop; pop; ret;`" gadgets that are part of an epilogue by reading the randomized code of the corresponding prologue, as illustrated in Figure 8. Register preservation code reordering has the highest coverage among the four transformations, altering about half of all available gadgets in a code segment [35].

*Register reassignment* swaps the register operands of instructions throughout overlapping live ranges (a *live range* begins with an instruction that defines a register and comprises all instructions in which that definition is live). Given that an attacker can precompute all live regions in the original code, reading even a single instruction at the beginning of a live region might be enough to infer the structure of gadgets towards the end of that region, as illustrated in Figure 9. Register reassignment has the second highest coverage among the four transformations, altering more than 40% of the gadgets in a code segment [35].

*Instruction displacement* [29] relocates sequences of instructions that contain gadgets into random locations within a different code segment, and overwrites the original code with trap instructions. The semantics of the code are preserved by patching the starting address of a moved code region with a direct `jmp` instruction to the new location. Consequently, an attacker can read the operand of the `jmp` instruction, and infer the location of a displaced gadget.

Note that even if we assume that a more aggressive code diversification technique such as function and basic block reordering [47] is applied, code inference may still be possible. If only the location—but not the internal structure—of basic blocks (or even easier, whole functions) is randomized, then the disclosure of a long-enough unique sequence of bytes will be enough to infer the rest of a basic block's (or function's code). Given that basic block reordering is currently not applicable for the complex COTS software targeted by the binary-level execute-only memory protections considered in this work, we leave the evaluation of such fingerprinting-based inference attacks as part of our future work.

### *E. Implementation*

Our prototype implements one instantiation of each of the four attack strategies. The resulting proof-of-concept consists

of a single HTML page with four buttons, one that launches each of the distinct attack strategies using JavaScipt embedded in the page. Our attack uses the same Internet Explorer memory disclosure exploit as used by Snow et al. [42].

*1) Enhanced* JIT-ROP*:* Due to the relaxed assumption of binary compatible fine-grained ASLR (see section §III), we take a different approach to mapping out code regions. That is, we do not recursively disassemble code pages by identifying call sites pointing to different pages. In fact, this step is not required; since all randomization must be done *in-place*, a single disclosed function pointer gives us enough information to infer the start and end address of that entire code module. Thus, we can linearly inspect the code in each identified code region and recursively reach other shared libraries by examining the *import table* in each known module. This technique results in a superset of the gadgets identified using the JIT-ROP technique [42].

*2) Destructive Code Reads:* To test our attacks against the destructive code read paradigm, we utilized the implementation discussed in Section II-B. This implementation works with complex commodity software such as Internet Explorer, Chrome, Firefox and Adobe Acrobat without any compatibility issues. Further, we verified that the destructive code read implementation functions correctly by using a memory disclosure vulnerability in Internet Explorer and observing that the execution of disclosed gadgets resulted in the application terminating.

## V. Evaluation

We now evaluate the practicality of our attacks against destructive code reads by using them in conjunction with the same real-world exploit against Internet Explorer as in [42] on Windows 8 (32-bit), as well as simulated exploits against Mozilla Firefox (version 41) and several versions of Adobe Acrobat. We also provide empirical evaluations of the usable (that is, not destroyed) gadgets that can be identified in our attacks and demonstrate that just-in-time ROP payloads can be successfully constructed and executed.

### A. On Availability of JavaScript JIT-Cloning Gadgets

To determine the feasibility of JIT-Cloning, *i.e.,* disclosing code on one JIT-compiled region and later executing discovered gadgets using a copy of the destroyed region, we must evaluate several factors. First, we must determine both how many gadgets we can persuade the JIT engine to produce. Without the use of destructive code reads, all of these gadgets would be available for use by the adversary. When destructive code reads are in use, however, we may only use the subset of the gadgets we identify that are consistently available across all copies of that JIT-compiled region. Since some JIT-spray mitigations are in place among commodity web browsers, we would expect that JIT code copies are, in practice, not identical, but rather they are similar to one another. Hence, we must determine if these copies are similar enough to provide one with a consistent set of available gadgets.

```
start = 'for ( var i = 0; i <100000; ++ i ) {'
contents = ''
for (var j = 0; j < nConstants; j++) {
  var randGadget = Math.floor(
    Math.random() * 0xffffffff); // 4-byte gadgets
  contents += 'g' + j + ' = ' + randGadget + ';';
}
end = '}'

eval(start+contents+end) // generate copy 1
eval(start+contents+end) // generate copy 2
```

Listing 1. Snippet of naive JavaScript code used to spray random gadgets. By evaluating this code twice we generate two similar JIT-compiled regions.

To explore this issue, we evaluated two copies of the same snippet of JavaScript code, depicted in Listing 1. The outer loop attempts to coax the heuristics of the JavaScript engine to JIT the code within the loop. Inside the loop a long series of random constant assignments are made. Intuitively, gadgets are randomly generated within the operands of instructions assigning those constant values. Since we just-in-time disclose gadgets, we do not necessarily care exactly how the JIT engine interprets and compiles the code, so long as gadgets are in fact generated.

The JavaScript was launched by embedding it into a web page and browsing to that page with Mozilla Firefox (32-bit, version 41). Our analysis shows that two (nearly) identical copies of JIT-compiled code are loaded into program memory at the same time. Moreover, those copies contained identical gadgets at identical offsets within the JIT-compiled regions. Since an arbitrary number of gadgets may be generated by evaluating larger amounts of code, and prior work has already demonstrated far more advanced strategies for JIT-spraying gadgets [37, 39, 2], we leave more intricate techniques for the generation of gadgets as an exercise for future work.

We also experimented with the code of Listing 1 using Internet Explorer 10 (32-bit). Unfortunately, the listed JavaScript did not appear to invoke the JavaScript engine's JIT-compilation. Upon further investigation, we found that IE has an undisclosed set of criteria for triggering JIT-compilation. Additionally, as a response to attacks that force a JIT engine to generate on-demand the desired shellcode or ROP gadgets, IE and other JIT engines have started employing code diversification techniques such as NOP insertion and constant blinding [27, 25]. That said, Athanasakis et al. [2] and Song et al. [43] have shown that the existing state-of-the-art mitigations are only able to complicate gadget spraying, but not prevent it altogether. Internet Explorer's Chakra engine, for instance, which is the most advanced in terms of employed protections, uses NOP insertion and blinding of 4-byte constants, both of which have already been circumvented. We also note that constant blinding does not hinder JIT-cloning, as we can determine, at runtime, exactly where those randomized constants are located in the first copy of the JIT region and subsequently ignore those gadgets when making use of the second region.

Although the insertion of NOPs may shift the locations of generated gadgets within a segment, the *sequence* of gadgets

TABLE I
THE TOTAL GADGETS DISCOVERED, AS WELL AS THE NUMBER OF GADGETS THAT ARE AVAILABLE TO AN ATTACKER AFTER DESTRUCTIVE CODE READS (*i.e.,* GADGETS THAT ARE NOT IN THE NORMAL PROGRAM EXECUTION PATH).

| Gadget Engine | Gadget Type(s) | d3d10warp.dll (% available) | urlmon.dll (% available) | vgx.dll (% available) | 42 of 109 libraries reachable from vgx.dll (% available) |
|---|---|---|---|---|---|
| ROPSHELL | Unique | 86% | 96% | 98% | 77% |
| JIT-ROP | MovRegG | 60% | 100% | 93% | 76% |
| | LoadRegG | 82% | 97% | 97% | 78% |
| | LoadMemG | 75% | 100% | 100% | 84% |
| | StoreMemG | 54% | 100% | 100% | 63% |
| | ArithmeticLoadG | 91% | 100% | 90% | 67% |
| | ArithmeticStoreG | 85% | 92% | 97% | 51% |
| | ArithmeticG | 73% | 100% | 98% | 64% |
| | StackPivotG | 100% | 100% | 100% | 76% |
| | JumpG | 96% | 100% | 100% | 70% |
| **Can build payload?** | | Yes (manual) | Yes (manual) | Yes (manual) | Yes (automatic) |

within the segment remains the same. By leveraging multiple copies of the same generated code, our attack can destructively read the generated code until the preamble of a gadget—which is known in advance—is found. Destroying some gadgets of interest during this process is not a problem, as they can be located in other JIT'ed copies of the same generated code by following the same strategy. At the same time, constant blinding of only 4-byte constants leaves enough room for the generation of useful gadgets by exploiting 1-byte and 2-byte constants, and overlapping instructions [2]. Simply extending constant blinding to smaller constants is not an attractive solution, as the associated overhead has been shown to be prohibitively high [2]. Additionally, Evans et al. [20] and Carlini et al. [14] have shown that protections based on CFI at the JIT engine level [34] may still allow an attacker to generate enough permitted control flow paths for the construction of a functional ROP payload.

### B. On Reloaded Code Gadgets Available

We now turn our attention to the feasibility of our attacks leveraging the lack of code persistence in destructive code reads, namely the library reload and process reload attacks described in section §V. These attacks more closely follow the original methodology of just-in-time code reuse attacks in that we leak an initial code pointer from a virtual function pointer table (vtable) by leveraging a memory disclosure attack, then proceed to disclose gadgets in that library module. Note, however, that in the case of *shared library reloads*, we are limited to disclosing only those gadgets within the libraries we can load and unload. That is, disclosing any other bytes of code will leave that code destroyed with no way to restore it to it's original form. Thus, Table I provides an analysis of the gadgets available in the specific libraries we control in our example attack (`d3d10warp.dll` and `urlmon.dll`).

Specifically, we analyzed gadgets in the libraries with both the `ROPSHELL`[3] gadget engine and the `JIT-ROP` gadget engine. Examining those unique gadgets more closely, we found a number of *useful* gadgets in each category in both libraries. Note that even after eliminating those gadgets located in execution paths for loading and unloading the libraries, we can still identify gadgets of each type. Unfortunately, even with all gadget types being available, the `JIT-ROP` compiler was unable to automatically generate a payload using only the gadgets identified in each distinct library. However, we were able to construct functional code reuse payloads by inspecting the available gadget set and manually chaining together gadgets in ways not supported by the automated compiler. One could, of course, update the `JIT-ROP` compiler to recognize and make use of those combinations automatically.

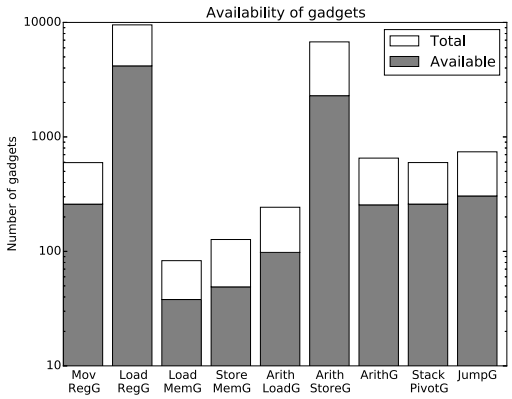


Fig. 10. The number of gadgets found in each JIT-ROP category across all reachable libraries that are also available for disclosure (*i.e.,* those gadgets that are not executed during normal program execution).

Our *process reload* technique is much more flexible in terms of where gadgets may be disclosed than the library reload attack. That is, we may recursively disclose gadgets in any library that the initially disclosed library depends upon. To do so, we simply follow function pointers in the module's import section to navigate to new shared library modules. Indeed, we found that when using `vgx.dll` as the starting point (as given by the specific exploit we used), we could ultimately navigate to 42 of the 109 loaded libraries by recursively

[3]See the interface available at ropshell.com.

following imports using the same memory disclosure. Table I also reports both the gadgets found in `vgx.dll` as well as the cumulative percent of gadgets available in all reachable (and loaded) libraries.

```
LoadReg:  | pop edi | ret
    (value to load)
LoadReg:  | pop esi | ret
    (value to load)
LoadReg:  | pop ebp | retn 0xc
    (value to load)
    (padding)
    (padding)
    (padding)
LoadReg:  | pop ebx | ret
    (value to load)
PushA: LOADLIBRARYW* | pusha  | ret
    String data ('kernel32')
    String data
    String data
    String data
    String data
LoadReg:  | pop edi | ret
    (value to load)
LoadReg:  | pop esi | ret
    (value to load)
MovReg:   | xchg eax,ebp | ret
LoadReg:  | pop ebx | ret
    (value to load)
PushA: GETPROCADDRESS* | pusha  | ret
    String data ('WinExec')
    String data
LoadReg:  | pop edi | ret
    (value to load)
MovReg:   | xchg eax,esi | ret
LoadReg:  | pop ebp | retn 0xc
    (value to load)
    (padding)
    (padding)
    (padding)
LoadReg:  | pop ebx | ret
    (value to load)
LoadReg:  | pop edx | retn 0x8
    (value to load)
    (padding)
    (padding)
PushA: WINEXEC* | pusha  | ret
    String data ('calc')
    String data
```

Listing 2. High-level representation of the ROP payload automatically generated by JIT-ROP using gadgets available in the reachable libraries.

The disclosure of only the initial shared library, `vgx.dll`, yields results similar to that of the shared library reload technique. We observe significantly more gadgets when considering that, in practice, the adversary will be able to disclose the code of all libraries dependent on that initial library. As shown in Table I and Figure 10, we are able to identify a plethora of gadgets in each of the requisite categories that are useful for building a malicious code reuse payload. Indeed, the `JIT-ROP` compiler is able to automatically construct a code reuse payload in this case that launches the calculator process and cleanly exits the exploited process. Listing 2 depicts one of the many possible payloads that could be automatically generated by the `JIT-ROP` compiler. In short, each line of the listing represents one 8-byte value to be loaded on the program stack after a control-flow hijacking followed by a stack pivot. Gadgets execute, in order, from the first line to the last, with data and padding intermingled between the gadget pointers.

The most important take-away of Table I and Figure 10, however, is not the number of gadgets and types that we are able to identify in this particular exploit instance, or even whether we are able to build a payload from those gadgets. Indeed, one can likely discover more gadgets using different techniques from those employed in this paper, such as using jump [10] or call-oriented programming [13] or use more sophisticated techniques for identifying standard ROP gadgets with or without side-effects. Instead, our results conservatively describe the effectiveness of destructive code reads empirically by noting the average *reduction* of gadgets across all of these experiments.

In doing so we conclude that, conservatively, destructive code reads offer only a 30% reduction in the gadgets available to the attacker, and this reduction is wholly attributable to eliminating gadgets that exist within the normal program execution path. That is, disclosing a gadget located in the normal program execution path results in the destruction of that code, and ultimately the program crashes before an exploit can complete.

### C. On Implicit Code Reads

To assess the feasibility of implicit code disclosure against the in-place code randomization technique of Pappas et al. [35], we evaluated how many randomized gadgets can be implicitly read by following the inference strategy outlined in Section IV-D. As discussed, the effectiveness of implicit disclosure in the face of instruction substitution and basic block instruction reordering depends on multiple factors related to the specifics of the particular randomized gadget, including (but not limited to) the size of the basic block in which it is contained and the location of substitutable instructions.

Due to the complexity involved in that analysis, and more importantly, the lower coverage of those two transformations, we opted instead to focus on register reassignment and register preservation code reordering. Given the nature of these two transformations, precise inference of the state of these randomized gadgets is always possible without having to destructively read even a single byte of the gadget being revealed—only memory locations preceding the actual gadget must be disclosed.

As an example, Figure 11 shows the bytes of the randomized instance of the code depicted in Figure 8. By destructively reading just two bytes, corresponding to the two out of the three instructions that can be reordered at the function's prologue, an attacker can infer the values of the randomized bytes at the function's epilogue, *i.e.,* which of the six possible orderings of the `pop` instructions of the gadget is actually used. Similarly, as shown in Figure 12 (which corresponds to the example of Figure 9), by destructively reading just a single byte, an attacker can infer which of the two possible register operand combinations are used in the gadget.

For our empirical analyses, we used a set of 47 libraries from Adobe Reader v9.3 and Adobe Acrobat Reader DC,

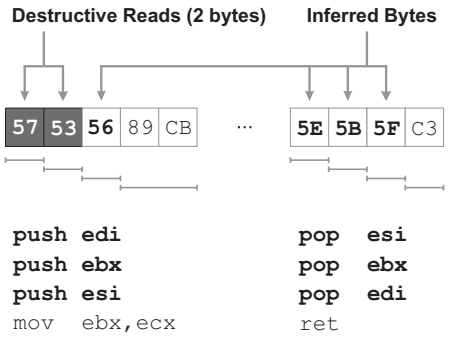


Fig. 11. Destructively reading two bytes in the function's prologue allows an attacker to infer the values of the randomized bytes of the gadget, *i.e.,* which of the six possible orderings of the `pop` instructions actually used.

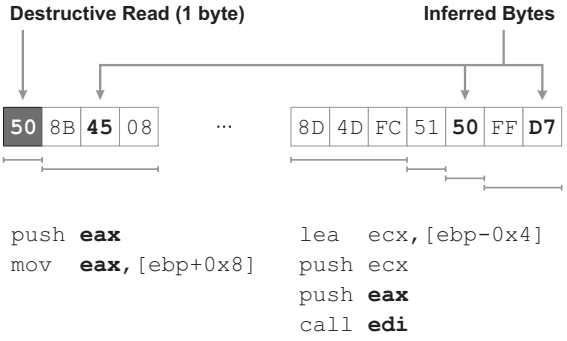


Fig. 12. Destructively reading a single byte in an instruction that involves a reassigned register allows an attacker to infer the state of the randomized gadget.

which in total contain 628,907 gadgets. We used the publicly available implementation of in-place code randomization [35] to randomize the libraries. Figure 13 shows the percentage of gadgets that can be randomized by each of the four randomization techniques. Note that a given gadget can be randomized by more than one technique. The combination of all techniques randomizes 78.28% of all gadgets found in the analyzed code. We found that similar to the results reported by Pappas et al. [35], instruction substitution and basic block instruction reordering achieve the lowest randomization coverage (21.43% and 33.98%, respectively).

The two more effective transformations, which happen to always allow for implicit code disclosure, achieve a combined coverage of 68.72%. In other words, by focusing only on register reassignment and register preservation code reordering, *an attacker can infer the state of 90.44% of all randomized gadgets* (*i.e.,* including the 21.72% of the gadgets that cannot be randomized by any of the transformations of Pappas et al. [35]). Based on these results, and considering that further inference against instruction substitution and basic block reordering is likely possible, we conclude that in-place code randomization is not sufficient for use in conjunction with binary-level execute-only memory protections.

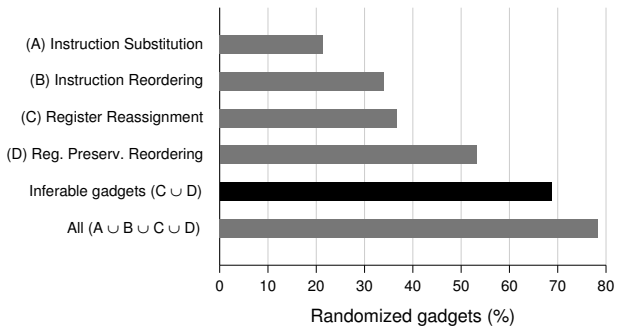


Fig. 13. Randomization coverage achieved by the different transformations of in-place code randomization. The state of randomized gadgets due to register reassignment (C) and register preservation code reordering (D) can always be inferred through indirect disclosure. This means that an extra 68.72% of all gadgets (C ∪ D) can be safely used by an attacker.

## VI. Discussion and Possible Mitigations

Many of the problems associated with binary-compatible execute-only or destructive read mitigations can be solved by using source and compiler-level techniques. For instance, the work of Crane et al. [16] provides an execute-only primitive and fully separates code and data at the compiler-level, which eliminates the need for destructive reads. Unfortunately, these techniques require one to either recompile open source software (instead of using the binary distribution) or leave end-users at the mercy of application developers for closed-source software. Binary-compatible mitigations instead transparently mitigate attacks on the (closed or open source) applications that are already in use. We believe the development and wide-spread adoption of mitigation tools such as Microsoft's Enhanced Mitigation Experience Toolkit (EMET) [33] aptly demonstrate the demand for binary-compatible mitigations.

Clearly, moving forward, any security analysis of binary-level execute-only memory protections that rely on destructive code reads must take into consideration the underlying guarantees of the code diversification technique being relied upon. Given that achieving complete randomization coverage (*e.g.,* using basic block reordering [47]) is challenging for complex closed-source applications (which systems such as Heisenbyte [45] and `NEAR` [48] are meant to protect), best-effort techniques such as in-place code randomization are the only available options. Sadly, as our results have shown, such schemes are not sufficient when destructive reads are the principal protection limiting the partial inference of code bytes. Other protections are needed. Thus, we hope that this work motivates the need for additional research into more advanced binary-level code diversification techniques that can withstand code association attacks of the types presented in this paper.

Fortunately, more straightforward solutions are in reach for mitigating the attacks that exploit code non-persistence via library reloads. The simplest solution is to disallow the unloading of libraries, even if doing so comes at a price of higher memory utilization and less flexibility for complex applications. An alternative, but more difficult, solution would

be to re-randomize each time a library is loaded. Of course, such re-randomization would need to be performed in a way that does not hinder one's ability to support shared libraries.

More challenging still is the design and implementation of practical techniques for re-randomizing all libraries each time a process loads (for example, in a new browser tab). The use of position independent code [30] will likely be required to achieve that goal, though the challenges of supporting shared libraries seem rather demanding. An alternative might be to use the frameworks suggested by Crane et al. [17] and Bigelow et al. [8] as a base for frequent re-randomizations. While these approaches appear to offer at least one path forward, they currently offer no support for closed-source applications.

## VII. Conclusion

The emergence of JIT-ROP attacks that leverage memory leak vulnerabilities to bypass code diversification protections has prompted active research on defenses that enforce execute-only memory, which prevents the runtime disclosure of code by prohibiting read accesses on executable memory. From a practical perspective, only a few of those approaches [45, 4] can be applied for the protection of the complex COTS programs that are being targeted by current in-the-wild exploits, such as closed-source browsers and document viewers.

In this paper, we show that the recently proposed notion of destructive code reads [45, 48], which enforces a relaxed property of allowing data reads from executable memory, but prevents the subsequent execution of previously read data, is at best fragile. Although this approach seemingly strikes a balance between compatibility with complex binaries and protection against runtime code disclosure attacks, giving an attacker the ability to perform reads even in a destructive way is enough to undermine any offered protection.

To demonstrate this, we presented four ways in which an attacker can pinpoint the location and state of gadgets. In particular, code cloning via JIT code generation and code non-persistence via shared library or process reloading rely on destructively reading copies of generated or existing code, allowing the reuse of gadgets that have not been previously read, while code inference relies on reading preceding bytes related to the randomized state of gadgets. These techniques highlight the need for further research in binary-compatible code randomization schemes tailored for use in conjunction with execute-only memory protection, that will prevent gadget inference through code cloning or implicit reads.

## VIII. Acknowledgments

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

## References

[1] "Intercom: Cross-window message broadcast interface." [Online]. Available: https://github.com/diy/intercom.js

[2] M. Athanasakis, E. Athanasopoulos, M. Polychronakis, G. Portokalidis, and S. Ioannidis, "The devil is in the constants: Bypassing defenses in browser JIT engines," in *Symposium on Network and Distributed System Security*, 2015.

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

[4] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, "You can run but you can't read: Preventing disclosure exploits in executable code," in *ACM Conference on Computer and Communications Security*, 2014, pp. 1342–1353.

[5] A. Barresi, K. Razavi, M. Payer, and T. R. Gross, "Cain: Silently breaking ASLR in the cloud," in *USENIX Workshop on Offensive Technologies*, 2015.

[6] E. Bhatkar, D. C. Duvarney, and R. Sekar, "Address obfuscation: an efficient approach to combat a broad range of memory error exploits," in *USENIX Security Symposium*, 2003, pp. 105–120.

[7] S. Bhatkar, R. Sekar, and D. C. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits," in *USENIX Security Symposium*, 2005, pp. 17–17.

[8] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *ACM Conference on Computer and Communications Security*, 2015, pp. 268–279.

[9] D. Blazakis, "Interpreter exploitation," in *USENIX Workshop on Offensive Technologies*, 2010, pp. 1–9.

[10] T. K. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack." in *ACM Asia Conference on Computer and Communications Security*, 2011, pp. 30–40.

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

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

[13] N. Carlini and D. Wagner, "ROP is still dangerous: Breaking modern defenses," in *USENIX Security Symposium*, 2014, pp. 385–399.

[14] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of

control-flow integrity," in *USENIX Security Symposium*, 2015, pp. 161–176.

[15] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *ACM Conference on Computer and Communications Security*, 2010, pp. 559–572.

[16] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *IEEE Symposium on Security and Privacy*, 2015, pp. 763 – 780.

[17] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz, "It's a trap: Table randomization and protection against function-reuse attacks," in *ACM Conference on Computer and Communications Security*, 2015, pp. 243–255.

[18] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming," in *Symposium on Network and Distributed System Security*, 2015.

[19] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi, "Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and ARM," in *ACM Asia Conference on Computer and Communications Security*, 2013, pp. 299–310.

[20] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *ACM Conference on Computer and Communications Security*, 2015, pp. 901–913.

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

[22] H. Gisbert and I. Ripoll, "On the effectiveness of nx, ssp, renewssp, and aslr against stack buffer overflows," in *IEEE International Symposium on Network Computing and Applications*, 2014, pp. 145–152.

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

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

[25] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Librando: transparent code randomization for just-in-time compilers," in *ACM Conference on Computer and Communications Security*, 2013, pp. 993–1004.

[26] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR," in *IEEE Symposium on Security and Privacy*, May 2013, pp. 191–205.

[27] A. Jangda, M. Mishra, and B. De Sutter, "Adaptive just-in-time code diversification," in *Proceedings of the Second ACM Workshop on Moving Target Defense (MTD)*, 2015, pp. 49–53.

[28] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software," in *Annual Computer Security Applications Conference*, 2006, pp. 339 –348.

[29] H. Koo and M. Polychronakis, "Juggling the gadgets: Binary-level code randomization using instruction displacement," in *ACM Asia Conference on Computer and Communications Security*, May 2016.

[30] J. R. Levine, *Chapter 8: Loading and overlays. Linkers and Loaders.* San Francisco: Morgan-Kauffman, 1999.

[31] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000, pp. 168–177.

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

[33] Microsoft, "The enhanced mitigation experience toolkit." 2016. [Online]. Available: https://support.microsoft.com/en-us/kb/2458544/

[34] B. Niu and G. Tan, "Rockjit: Securing just-in-time compilation using modular control-flow integrity," in *ACM Conference on Computer and Communications Security*, 2014, pp. 1317–1328.

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

[36] C. Rohlf and Y. Ivnitskiy, "The security challenges of client-side just-in-time engines," *IEEE Security & Privacy*, vol. 10, no. 2, pp. 84–86, March/April 2012.

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

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

[39] F. J. Serna, "Flash JIT - spraying for info leak gadgets," 2013.

[40] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *ACM Conference on Computer and Communications Security*, 2007, pp. 552–561.

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

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

[43] C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski, "Exploiting and protecting dynamic code generation," in *Symposium on Network and Distributed System Security*, 2015.
[44] A. Sotirov and M. Dowd, "Bypassing browser memory protections in Windows Vista," 2008.
[45] A. Tang, S. Sethumadhavan, and S. Stolfo, "Heisenbyte: Thwarting memory disclosure attacks using destructive code reads," in *ACM Conference on Computer and Communications Security*, 2015, pp. 256–267.
[46] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham, "Differentiating code from data in x86 binaries," in *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases*, 2011, pp. 522–536.
[47] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *ACM Conference on Computer and Communications Security*, 2012, pp. 157–168.
[48] J. Werner, G. Baltas, R. Dallara, N. Otterness, K. Z. Snow, F. Monrose, and M. Polychronakis, "No-execute-after-read: Preventing code disclosure in commodity software." in *ACM Asia Conference on Computer and Communications Security*, 2016.