Defeating Zombie Gadgets by Re-randomizing Code Upon Disclosure

Micah Morton,[1] Hyungjoon Koo,[2] Forrest Li,[1] Kevin Z. Snow,[3]
Michalis Polychronakis,[2] and Fabian Monrose[1]

[1] University of North Carolina at Chapel Hill
[2] Stony Brook University
[3] ZeroPoint Dynamics

Abstract. Over the past few years, return-oriented programming (ROP) attacks have emerged as a prominent strategy for hijacking control of software. The full power and flexibility of ROP attacks was recently demonstrated using *just-in-time* ROP tactics (`JIT-ROP`), whereby an adversary repeatedly leverages a memory disclosure vulnerability to identify useful instruction sequences and compile them into a functional ROP payload at runtime. Since the advent of just-in-time code reuse attacks, numerous proposals have surfaced for mitigating them, the most practical of which involve the re-randomization of code at runtime or the destruction of gadgets upon their disclosure. Even so, several avenues exist for performing *code inference*, which allows `JIT-ROP` attacks to infer values at specific code locations without directly reading the memory contents of those bytes. This is done by reloading code of interest or implicitly determining the state of randomized code. These so-called "*zombie gadgets*" completely undermine defenses that rely on destroying code bytes once they are read. To mitigate these attacks, we present a low-overhead, binary-compatible defense which ensures an attacker is unable to execute gadgets that were identified through code reloading or code inference. We have implemented a prototype of the proposed defense for closed-source Windows binaries, and demonstrate that our approach effectively prevents zombie gadget attacks with negligible runtime overhead.

Keywords: Code Reuse, `JIT-ROP`, Code Inference, Destructive Reads

1 Introduction

In recent years, memory corruption attacks have become increasingly sophisticated. For example, present day exploits on commodity systems must circumvent Address Space Layout Randomization (ASLR), a widely deployed defense which requires the adversary to use *memory disclosure* to compute the addresses of useful gadgets in a program before repurposing them for malicious means. Researchers and practitioners recently proposed further approaches to harden vulnerable applications against memory disclosure, through focusing on more fine-grained forms of code diversification [26, 31]. In turn, attackers responded with the development of "just-in-time" ROP (`JIT-ROP`), a style of attack that leverages the dynamic scripting capabilities of document renderers and web browsers to repeatedly disclose memory in order to build exploit payloads at runtime, all the while making no assumptions about the layout of code and thus circumventing fine-grained ASLR [28].

Not willing to be outdone, defenders developed several new mechanisms in order to stay one step ahead of attackers armed with scripted memory disclosure capabilities. In this vein, proposed compile-time defenses [7, 12] effectively mitigate `JIT-ROP` attacks by enforcing code memory to be executable but not readable—eliminating an attacker's ability to use memory disclosure to enumerate and read code pages. In an effort to be more readily deployable to closed-source software, other binary-compatible defenses have attempted to apply on-demand code randomization [6, 10, 33] or gadget destruction [30, 32] at *runtime* in order to protect against `JIT-ROP`.

Unfortunately, the most promising `JIT-ROP` defenses either have major hurdles to overcome in achieving widespread deployability [29], due to their reliance on source code access and compiler support [7, 12, 13], or have been shown to be vulnerable to ingenious advancements in `JIT-ROP` capabilities. In particular, Snow et al. [27] demonstrated that existing execute-only memory protections applicable at the binary level based on the concept of destructive code reads [30, 32] can be bypassed using code reloading, JIT code generation, or implicit code disclosure attacks. As explained in §4, these clever evasion techniques are made possible through the attacker's ability to re-load a given code module multiple times or to deduce the values of certain code bytes based on values of related instructions. That said, the shortcomings of previous binary-compatible defenses do not indicate that the task of defending against code reuse is insurmountable. Rather, in this work we propose further advancements to existing defense paradigms that aptly harden them against these powerful code reloading and code inference strategies.

In this paper, we identify two concepts as the pillars of any effective `JIT-ROP` defense that seeks to prevent the execution of disclosed gadgets. We refer to these as the *trigger* and *countermeasure* of a defense, respectively. These terms come from the fact that part of the defense must be *triggered* when an attacker has disclosed potentially useful executable bytes in memory, and some subsequent *countermeasure* must be taken to ensure those bytes cannot be leveraged by an attacker for hijacking control of the application. The purpose of this work is to adapt ideas put forth by existing defenses that implement runtime gadget destruction [30, 32] by making novel extensions to both the trigger and countermeasure components. At its core, our defense features the ability to efficiently and robustly re-randomize program instructions in response to their code bytes being disclosed by an attacker. Specifically, to deal with code reloading attacks, our approach detects when code modules that could contain fresh usable copies of gadgets (that were previously disclosed and destroyed in another instance of that module) are about to be loaded, and replaces them with new randomized versions. In addition, to deal with the more sophisticated code inference attacks, our approach re-randomizes upon each destructive read the subset of the code that could potentially be implicitly disclosed.

2 Goals and Adversarial Model

Our goal is to provide a binary-compatible defense against just-in-time code reuse attacks [28]. We are particularly interested in sound defenses that have low runtime overhead and are applicable to real world programs and their complexities. We assume the attacker has full power of scripted arbitrary memory disclosure as well as the ability to cause arbitrary code modules to be loaded or unloaded, per the attacks recently presented by Snow et al. [27]. Specifically, we assume that *i.* Data Execution Prevention, *ii.* Fine-grained Address Space Layout Randomization (*e.g.,* [7, 15, 19, 26, 31]) and *iii.* Destructive read capabilities that leverage execute-only memory (*e.g.,* [8, 30, 32]) are in use. We also assume that adversary have at their disposal a memory disclosure vulnerability that allows them to read and write arbitrary memory locations. By now, these assumptions are commonly accepted as being no stronger than the capabilities already leveraged by skilled adversaries (*e.g.,* [6, 9, 12, 18, 28]) to defeat contemporary ASLR.

3 Background & Related Work

Over the past several years, a number of defenses (*e.g.,* [4, 6, 7, 10, 11, 12, 15, 16, 19, 20, 21, 30, 32, 33]) have been suggested as ways to curtail the power of just-in-time code reuse attacks. Interested readers are referred to Crane et al. [13] for an excellent review of the current state of the art in return oriented programming attacks. Here, we instead focus on existing defensive strategies in terms of the *triggers* they utilize and their runtime *countermeasures*. To date, a myriad of triggers have been proposed, such as invoking countermeasures as a result of file I/O [6], process forking [24], or elapsed wall-time [10, 33]. At the same time, the runtime countermeasures involve either re-randomizing code layout [6, 10, 24, 33] or overwriting disclosed code bytes as a way to ensure they cannot be leveraged in a ROP payload [30, 32].

Unfortunately, the existing approaches all have significant shortcomings. For example, Bigelow et al. [6] assumes that scripting environments are out of scope, and so their approach cannot protect widely used applications like modern browsers or document renderers; the work of Chen et al. [10] only offers a probabilistic defense, and the recent proposals of Tang et al. [30] and Werner et al. [32] have been undermined using only modest enhancements to the original `JIT-ROP` framework [27]. Additionally, many of these proposed defenses suffer from shortcomings in terms of real-world applicability (*e.g.,* such as poor performance guarantees or lack of compatibility with multi-threaded programs) or require the ability to re-compile code from source in order to enable the proposed protections [7, 12, 13]. Although the ability to recompile software for added security enforcement is often an ideal avenue for mitigating software threats, such defenses that require access to source code are not positioned for near-term deployability in the same way as binary-compatible defenses. Since binary-compatible defenses only require updating core system components rather than all commodity software running on a device, wide-spread deployability is more feasible. Our defense is motivated by the need to protect vulnerable systems in the near-term, and so many of our design choices prioritize deployability.

One related work which shares similar deployability goals is a defense proposed by Williams-King et al. [33] which offers a binary-compatible solution for constantly re-randomizing code at prescribed intervals (of wall-clock time) in order to break `JIT-ROP` payloads. Their approach requires relocating functions using complex pointer tracking techniques in order to avoid creating stale pointers that can no longer be safely dereferenced; however, such analysis is known to be an unsolvable problem in the general case, and raises a slew of challenges for real-world deployment. The approach of Williams-King et al. [33] makes strides in advancing the robustness of pointer tracking based defenses by leveraging program analysis and assuming access to debug symbols in order to bolster the accuracy of moving functions around at runtime within the address space of a protected process. Unfortunately, there are corner cases in deployable pointer tracking that are not handled by their work, thereby lessening the near-term deployability of the defense. For instance, even when instructing the compiler/linker to preserve as much information as possible, certain information is not retained, such as locations of static functions (for which the known offset within a module can be hard-coded by the compiler), alignment of jump tables, or existence of functions which implicitly fall through to the next function. This lack of information complicates the prospect of reordering functions in applications that feature these program constructs. Certain aspects of data flow tracking are also not supported (*e.g.,* when an object is initialized in one library and `memcpy`'d to a different library). Our work does not share these drawbacks, as we avoid pointer tracking altogether. In another related code-shuffling style approach, Chen et al. [10] attempt to provide a probabilistic defense against `JIT-ROP` attacks by applying time-based binary stirring [31] to a process in an attempt to re-randomize all code in the entire program. Unfortunately, since Chen et al. [10] provide no guidance on how to determine realistic intervals for triggering their defense, it remains unclear what the incurred overhead is for thwarting real-world `JIT-ROP` attacks that have a lifetime of a few seconds [28].

Destructive Reads Of late, several defenses that rely on the notion of execute-only memory (*i.e.,* to enforce that any given location in a code section can be either read or executed—*but not both*) have been suggested as a mechanism for preventing code reuse attacks. Indeed, instead of attempting to solve the difficult problem of separating code and data and preventing code from being read recursively (*e.g.,* [4, 5, 19]), the idea behind *destructive reads* [30, 32] is to allow all code to be disclosed, but to prevent any disclosed code from subsequently being executed. Sadly, while the notion of destructive reads was thought to be an effective technique for mitigating just-in-time code reuse attacks as originally proposed by Snow et al. [28], several ingenious attacks have surfaced that leverage the ability to load and unload modules at will—or for selectively disclosing bytes of memory as a means of inferring surrounding gadgets—to undermine any afforded protection [27].

Even with these attacks in mind, we show how the notion of destructive reads can be effectively combined with load-time randomization to provide strong protection against powerful code reuse attacks. Our solution for doing so is discussed next.

4 Approach

In what follows, we propose a practical defense against just-in-time code reuse attacks that take advantage of an adversary's ability to disclose and execute code bytes whose values were learned by loading and unloading code modules, or performing so-called *code inference attacks* like those recently presented by Snow et al. [27]. At a high-level, our approach centers around the ability to place randomized versions of code in a process at key trigger points during the execution of a just-in-time code reuse attack. Specifically, we replace the code upon which an attack relies with logically equivalent code of a different form that will break the attacker's ROP payload. To achieve this, we apply binary-compatible *in-place* randomization to code modules in order to obtain multiple diversified copies of the code which are kept in kernel-space memory where they are not accessible to user-level processes. With swappable versions of a module available at our disposal, we can then efficiently replace disclosed code at runtime with minimal complexities while assuring correct program execution. Specifically, when a module is loaded into memory from disk, we ensure that a randomized copy of that module is mapped into the user-space process, thwarting code reloading attacks. Furthermore, individual reads to executable addresses in the module trigger our system to swap localized code sequences within functions for semantically-equivalent randomized code sequences from one of the alternate versions maintained in kernel-space. This technique prevents adversaries from making use of individually disclosed gadgets, while not requiring any re-routing of control flow or swapping of entire code modules.

Both `Heisenbyte` [30] and `NEAR` [32] provide solid foundations for detecting the most straightforward way an attacker can learn the values of code bytes—*i.e.,* by directly reading their values in memory—and are able to prevent the execution of those exact bytes at a later time. Unfortunately, attacks are still possible when scripting environments can be used to cause modules to be loaded or unloaded at will, exchanging destroyed gadgets for fresh versions of the previously disclosed bytes and thus rendering destructive code reads ineffective. Indeed, the most concerning attacks suggested by Snow et al. [27] involve the use of *implicit code reads* that allow an adversary to infer the values of code bytes *indirectly*, based on the directly read values of related code bytes. These two orthogonal attack approaches necessitate two orthogonal components of our defense: one for thwarting *code reloading* and another for defending against *code inference*. Figure 1 shows an overview of the proposed approach, depicting how our defense ensures different randomized copies of code modules get mapped into memory on image load, as well as ensuring code bytes that are disclosed by an attacker get swapped for a different randomized version of those bytes before they can be leveraged in an exploit payload.

These two components of our defense extend destructive read capabilities presented in previous work, which we briefly explain before relaying the specifics of our contributions. `Heisenbyte` [30] and `NEAR` [32] both implement what is called a *thin hypervisor*, allowing them to leverage hardware virtualization support for Extended Page Tables (EPT) to intercept read accesses to executable sections of a given process. While this may seem like a drastic means by which to simply mark code pages as execute-only, at present, it is the only feasible approach for contemporary Intel processors.[4] In both `Heisenbyte` and `NEAR`, enforcing *destructive reads* involves registering an EPT fault handler that, when invoked, assures the byte values where a fault takes place can never be subsequently executed (*e.g.,* by overwriting the bytes with invalid opcodes). In addition, the byte values at those offsets must be preserved so that they can be made available in the event that an application wants to read the data again at a later time.

4.1 Defeating Code Reloading

The first of the two components of our defense aims to combat so called *code reloading* attacks. The approach we take here is straightforward: ensure that adversaries are faced

[4] That said, as hardware support [22] is added for more fine-grained control of the memory protections applied to individual pages, we expect the hypervisor component of execute-only-memory based defenses to become obsolete.

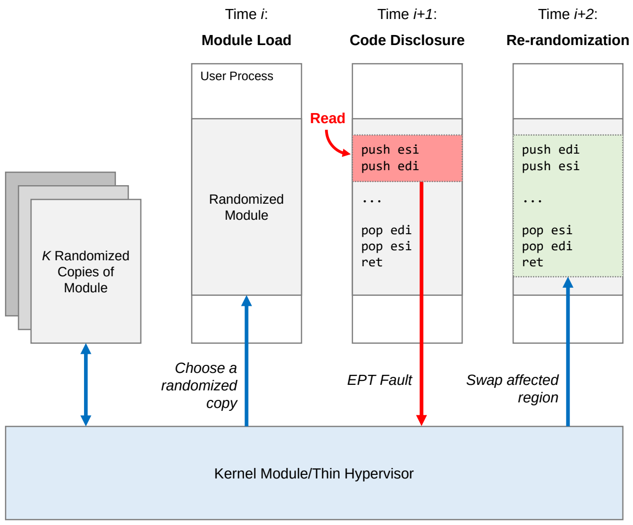


Fig. 1: At load time, one among many randomized versions of a module is picked at random. Whenever a potential code disclosure event occurs (due to a destructive read operation), the locally surrounding region is re-randomized by swapping it with a different randomized instance.

with a different randomized copy of a module each time the module is loaded, thereby preventing them from disclosing gadgets in one copy of a module and executing them in a different identical copy. To do this, we apply in-place randomization (using the techniques of Pappas et al. [26]) to k distinct versions of the module and map the different versions into process memory on load. As shown on the left side of Figure 1, this can be done by hooking the operating system functions that map executable images into memory and redirecting the associated *OpenFile* call to any of the k randomized versions of the binary that reside on disk. While this straightforward countermeasure by itself eliminates a significant subset of the attacks presented in the work by Snow et al. [27], additional special attention must be taken to avoid the pitfalls that opened the door to attacks based on code inference via *implicit* disclosure.

4.2 Defeating Code Inference

The idea behind code inference attacks is that in-place code randomization [26] applies code transformations at such a local level, that reading even one byte where randomization has been applied is often enough information to infer how other related nearby bytes (which may actually contain useful gadgets) have been randomized. The problem this poses for defenses that leverage destructive code reads is that simply destroying the code byte that was directly read does nothing to prevent an attacker from leveraging gadgets that were discovered through *implicit* code disclosure, enabled by *explicit* code disclosure [27].

The ability of attackers to mount code inference attacks stems from the nature of the code transformations applied by binary-compatible fine-grained randomization techniques [23, 26], and specifically their narrow scope. Specifically, in-place randomization [26] applies the following four code transformations: *instruction substitution*, which replaces existing instructions with functionally-equivalent ones; *basic block instruction reordering*, which applies a functionally equivalent instruction ordering within a basic block by maintaining any data dependencies; *register preservation code reordering*, which reorders the *push* and *pop* instructions of a function's prologue and epilogue; and *register reassignment*, which swaps register operands throughout overlapping live regions.

By disclosing a few instructions, an attacker is able to infer the state of related instructions that are part of a ROP gadget. For example, an adversary could use code inference to implicitly learn the precise structure of a gadget that has been randomized using register preservation code reordering—which involves reordering the push and pop instructions of a function's prologue and epilogue. By (destructively) reading instructions in the prologue that are affected by the transformation, but which are not part of the actual gadget, an attacker can accurately infer the structure of the gadget in the function epilogue. Concretely, if the attacker knows that registers are saved onto the stack by a function, the order by which these registers are popped in the epilogue is the reverse order in which they were pushed during the prologue, so reading the prologue allows the adversary to infer the exact gadget contained in the function epilogue. Since the actual disclosure by the adversary was aimed at the prologue, destructive read enforcement will only protect those bytes, leaving the epilogue to be freely used as a useful gadget for the adversary. Similar code inference attacks against the rest of the transformations are discussed by Snow et al. [27], all of which are mitigated by our defense.

Crafting a countermeasure that renders implicit code reads ineffective turned out to be more difficult than it appeared on first blush. The reason is that an important criterion of ours was to allow for runtime re-randomization *without* having to deal with unsound and cumbersome pointer tracking. As noted in §3, other approaches to runtime re-randomization (*e.g.,* [6, 24, 33]) have also turned to ASLR-style code relocation at runtime, but these works needed to apply heuristics to deal with the problem of stale pointers. Re-randomization schemes can introduce stale pointers into a program if they do not carefully adjust every pointer that references a given code section when that section is relocated at runtime. The tracking of all pointers is rife with challenges and it remains an active area of research.

We choose not to introduce such complexity into our work. As an alternative to moving around large chunks of code in process memory, we opted for a more localized solution that guarantees that any—*explicitly* or *implicitly*—disclosed bytes are re-randomized in response to disclosure, while simplifying as much as possible the problem of ensuring correct program continuation. As shown on the right side of Figure 1, we detect when a code disclosure occurs (causing an EPT fault, which is intercepted by our *thin hypervisor*) and replace only the part of the code that was disclosed with a different randomized version. As in our approach for combating code reloading attacks, we must maintain k different randomized versions of the program code, so we can randomly select from k different versions (which reside in kernel module memory) of the disclosed code to swap in at runtime. The intricacies of ensuring program correctness when swapping code ranges are discussed further in Section 5.

Critically, to deal with code inference attacks, we ensure that the part of code that is randomized includes not only the destructively read bytes, but also *all other instructions that could potentially be inferred*. The choice of using in-place randomization was a driving factor in simplifying our solution: with in-place randomization we can know the exact range of code that is vulnerable to implicit code disclosure for any given explicit code read. Importantly, we do not have to swap an entire randomized version of the program every time a disclosure happens, as the vast majority of randomized locations in a program cannot be inferred from a single disclosure. In other words, every explicit code read carries with it the potential to infer the values of other code bytes without actually reading them, but the range of code bytes that can be inferred is limited and is easy to compute in advance. We refer to this range of addresses as the *scope* of randomization, and return to a discussion thereof in Section 4.2.

What is important to understand at this stage is that through offline analysis we are able to compute the scope of randomization for each byte in a code module. Thus, when we intercept an explicit code read at a given location, we can look up the scope of randomization for that byte and swap the code in that range of addresses for a different randomized version. This necessitates only swapping out localized ranges of code, and in that way, we sidestep the issue of using broadly scoped runtime re-randomization techniques that incur a large overhead and rely on complex pointer tracking [6, 10, 33]. Moreover, in-place randomization guarantees that different randomized versions of the same code will always be the same size. Hence, these bytes can be interchanged without having to worry about making room for a larger version of logically equivalent code when the swap takes place.

Scope of In-Place Randomization The exact range of code that must be swapped out for a given EPT fault is directly dependent on type of transformation that was applied by in-place randomization to the surrounding code bytes. For the remaining discussion, we use the randomization technique of Pappas et al. [26] (called ORP) as an example since it is representative of the state of the art in this domain. The three possible scopes for a given transformation include randomizing at the opcode, basic block, or function level. Thus, if byte x in a module has to be randomized, the randomization may involve simply rewriting the opcode containing x, or could involve altering an entire function's worth of code that includes x.

There are two ways to think about the *scope of randomization* for a given byte in a code module. In one sense it can be considered the range of code bytes that are potentially vulnerable to implicit disclosure should that code byte be explicitly disclosed. In another sense this term represents the smallest range of code bytes that can be swapped for a different randomized version while still maintaining correct execution of the program. The three different scopes at which ORP applies code randomization determine the range of bytes that need to be swapped as a result of a given byte being directly read by an attacker. Note that we never have to swap out more than an entire function's worth of bytes for a single EPT violation, since no ORP transformations are applied at a broader scope than the function level. One caveat with using binary-compatible randomization techniques like ORP is that it may not always be possible to randomize all bytes in a given program. This is due to the fact that commodity binaries can include data in their executable code sections and disassembly of closed-source software can be imprecise. That said, the coverage of existing tools is high enough [23] that this limitation does not significantly weaken the security assurance that our defense offers. We discuss this further in §6.

Notice too that as long as we safely swap the correct amount of code (based on the type of randomization applied), we can ensure correct program execution. That said, the preceding discussion assumes that function in which we intend to replace code does not already have an activation record on the stack. In other words, if a function has been invoked but not yet returned at the time that a code disclosure targets that function, we cannot safely change the bytes of that function without potentially causing a crash when program execution returns to the function. For example, two different randomized versions of the same function could potentially save and restore registers to the stack in a different order for their respective function prologues/epilogues. If the registers are saved in one order during the function prologue and the code of the function gets randomized before that function invocation has returned, the newly randomized code may restore registers in a different order than they were saved in the prologue. If we allow this to happen, we could introduce failures into otherwise correct programs.

To ensure this does not happen, we take a conservative approach and do not randomize any span of bytes that are referenced by a pointer on one of the program stacks at the time of the disclosure. This simplification ensures that no functions where execution has already started but not yet completed will be randomized. The approach is conservative as it assumes that *every* word on each of the program stacks is a pointer, but in practice, this is certainly not the case. This conservative approach would seem to be a weakness, but it turns out not to be the case for two reasons: first, the code and data separation techniques we leverage from `NEAR` and `Heisenbyte` are highly effective in minimizing legitimate code reads that occur during normal program execution (*i.e.,* by moving data that does not need to be executed out of the code section of a binary). Second, code reads that cannot be eliminated by the "purification" steps of Werner et al. [32] or Tang et al. [30] are not likely to trigger a stack lookup because they will be referencing code bytes that cannot be randomized by our approach anyway.

To see why, it is important to keep in mind that the majority of reads directed at code sections by a program are for reading data that has been embedded in the code section, rather than reading actual machine instructions. But, as ORP's conservative offline disassembly should not identify this data as executable instructions, ORP will not randomize this data and our defense will, by extension, not be able to randomize data bytes in code sections. Our empirical analysis (§6) confirms that is the case. Nonetheless, this restriction could be relaxed by employing more accurate stack unwinding [17] or shadow stack techniques (implemented either in software or hardware) [2, 14] that do not assume (as we do here for simplicity) that every word on the stack is a pointer.

5 Implementation

As a proof of concept, we chose to build this system for 32-bit x86 Windows, with all implementation contained in a single loadable kernel module.[5] The kernel module is comprised of code for setting up the *thin hypervisor* and reacting to EPT faults, as well as code for hooking operating system routines to ensure that each load results in a randomized copy of the image being mapped into a process' memory. In what follows, we discuss some key decisions we made during our design, as well as implementation challenges we encountered along the way.

5.1 Adapting Offline Randomization Techniques for Online Defense

One challenge we faced arose due to the fact that all the existing in-place randomization techniques we are aware of only work offline (*e.g.,* ORP randomizes code instructions in an executable file). Since swapping out code at runtime is central to our defense, we needed a way to access different randomized versions of a given range of program code when a destructive read occurs. Our solution was to create a single aggregate binary with k distinct randomized versions of the `.text` section. Thus, when the binary is loaded, multiple different randomized versions of the code for the module will be brought into memory and their instructions can be swapped into the executing `.text` section whenever necessary. This involves storing additional metadata specifying the ranges of swappable code that have been randomized, which is used to determine whether to swap out memory in response to a given EPT fault (*i.e.,* yes if the fault is in a range randomized by ORP, no otherwise).[6] On a final note, we must ensure that the extra `.text` sections maintained by our defense are only accessible to the operating system, so their contents cannot be disclosed by exploits as part of attacking a user space process. We achieve this by only mapping one of the k code sections as accessible to user space when the binary is loaded into memory.

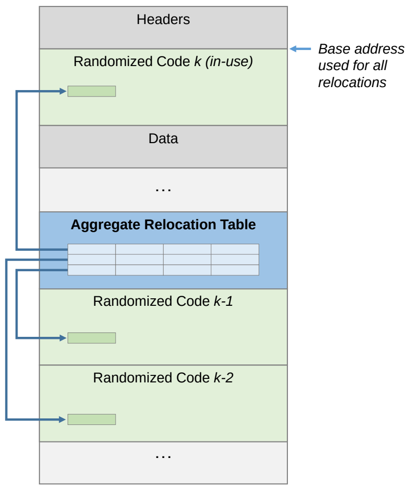
5.2 Handling Relocatable Code

Another technical challenge is dealing with the problem of relocatable binaries. In Windows, for example, binaries are loaded into process memory in such a way that it not uncommon that hard-coded addresses in the binary must be adjusted before the module can execute properly. This absolute addressing (in contrast to Linux-style position-independent code) assumes a specific load address in process memory called the preferred offset. If the binary is loaded at an address other than the preferred offset, all hard-coded addresses in the executable need to be adjusted during the loading process. Our runtime re-randomization approach may introduce incorrect execution if we simply swap bytes of a program out for the corresponding bytes from a different randomized binary whose hard-coded addresses were not correctly adjusted.

[5] Our thin hypervisor and kernel module are built upon the code provided by Werner et al. [32] as part of their work on destructive reads.

[6] Failure of our system to swap a given range of code indicates that this range was not randomized through ORP, and thus is not vulnerable to inference attacks. Note that destructive read enforcement still protects these memory areas.

To address this problem, we coerce the loader to adjust all the hard-coded addresses in each of the k code sections *as if they are all being loaded at the same code section offset* within the module (*e.g.,* offset 0x1000 is typical in the PE format). In Windows, for example, each relocatable binary contains a table that specifies the hard coded addresses in the binary that need to be adjusted if it is not loaded at its preferred offset. As shown in Figure 2, our approach takes executable file `a` and packages it with k randomized versions of `a`, so we also must combine the relocation tables from each randomized file into one large relocation table in the aggregate binary. In this way, we can safely start the module off executing from a randomly chosen version, i, of the binary, but when a destructive read is triggered, we then randomly select one of the other $k-1$ variants and swap the bytes that are in the scope of the destructive read.


Headers
Randomized Code k (in-use)
Base address used for all relocations
Data
…
Aggregate Relocation Table
Randomized Code k-1
Randomized Code k-2
…
Fig. 2: Load time adjustments

6 Evaluation

To test the runtime overhead of our defense, we ran the same selection of SPEC benchmark programs that were used to evaluate the performance of destructive read enforcement [32]. Similar to Werner et al. [32], we compiled the CPU SPEC2006 benchmark programs with the Microsoft Visual Studio 2013 C/C++ compiler using the default compiler and linker options listed in the benchmark suite. We used the unoptimized "base" configuration.

The negligible performance penalty incurred by our solution on top of destructive read enforcement (shown in Table 1) can be attributed to the fact that for these benchmarks, all observed EPT faults were directed at data embedded in code sections, rather than actual code instructions that could be randomized. As such, a hash table lookup was enough to decide that these ranges could not be swapped, so no further action was necessary. Of course, as soon as an attacker starts disclosing actual machine instructions in the code, runtime overhead would likely increase due to the need to repeatedly unwind the program stacks and copy appropriate ranges of bytes surrounding the EPT faulting addresses.

As for memory overhead, our solution involves mapping k extra copies of the `.text` section into memory for each protected code module. To help understand the tradeoff between security assurance and memory overhead, we consider that the probability of success for a ROP exploit comprised of n gadgets that attempts to randomly guess which gadgets exist at given locations would be $1/k^n$. Thus, choosing a k value even as low as two or three can still provide strong assurance against ROP chains working as expected by the adversary. It may seem that storing the k additional `.text`

SPEC CPU benchmark	Destructive Reads	Re-randomization	Total
400.perlbench	3.04%	0.1%	3.14%
401.bzip2	1.21%	0.03%	1.24%
403.gcc	19.88%	0.2%	20.08%
429.mcf	4.04%	0.3%	4.34%
445.gobmk	0.99%	0%	0.99%
456.hmmer	1.81%	0%	1.81%
458.sjeng	1.10%	0.09%	1.19%
464.h264ref	7.63%	0%	7.63%
417.omnetpp	2.32%	0.13%	2.45%
473.astar	2.13%	0.8%	2.83%
483.xalancbmk	5.51%	0.25%	5.76%
Average	4.51%	0.17%	4.68%

Table 1: End-to-end runtime overhead.

sections in memory would incur a large memory overhead when considering that the `.text` sections of the benchmark programs we tested range from 65KB to 2.4MB. This is not the case, however, as in-place randomization only alters a small fraction of bytes in the code section (about 3% on average), and thus only these transformed bytes would need to be stored in memory to be swapped in as needed.

6.1 Security and Correctness

To show that we can reliably thwart code inference attacks without introducing incorrectness into programs, we ran a runtime stress test, forcing all possible randomizable code ranges to be swapped. This allows us to confirm that we can correctly swap in and out all parts of the program that are marked as randomizable. Our results confirm that in all cases swapping these code ranges worked as expected and did not alter the correctness of the code compared to the original binary. The average time to perform each swap incurred a low runtime overhead, at only $0.105ms$.

Moreover, to demonstrate that our solution thwarts code reloading attacks, we took the same approach outlined by Snow et al. [27] for generating exploit payloads from gadgets in commonly reloaded DLLs. We thoroughly inspected one of these DLLs (`vgx.dll`) to confirm that the constructed exploit payloads are broken by in-place randomization of the respective code modules, and consequently that the resulting payloads could not be reliably used in an attack. Indeed, approximately 70% (9,622 out of 13,729) of the gadgets in the module identified by the automated gadget finding tool ROPEME [1] were swappable. The remaining gadget discrepancy mainly arises from different parameter settings used by ROPEME and the build-in gadget discovery module in ORP (*e.g.,* the number of look-ahead bytes or gadget depth during the gadget generation process).

With k being a finite number of different program variations, the reader may be curious as to whether our system would be vulnerable to some type of fingerprinting attack that seeks to infer the version of code that has been swapped into user-space process memory. This would not be feasible, however, for a few reasons. First, since every randomizable gadget in a program is a degree of freedom for in-place randomization, an attacker would be forced to disclose all randomizable sections in order to uniquely identify one of the k versions. The footprint of such a brute-force probing attack would be so substantial that it would be trivially detectable and would constitute on its own a clear trigger for detection. Note that k can easily be tuned to make such an attack practically unrealistic. Furthermore, k can be increased to a much higher number than what is allowable by the available memory on the system. Indeed, rotating k randomized in-memory instances is only one option. The system could easily generate a higher number of new randomized instances in the background once the pool of ready-to-use instances in the kernel module is running low. We leave the implementation of this additional functionality for future work.

6.2 Function Randomization Variability and Coverage

The in-place code randomization of Pappas et al. [26] uses a combination of four different transformation techniques of different spatial granularity (instruction, basic block, and whole function) to generate alternative representations of a program's code. For a given code disclosure, multiple transformations may have been applied to the code area surrounding the address which caused an EPT fault. We use the coarsest randomization scope (*i.e.,* the function that contains the disclosed code bytes) as the unit of re-randomization because function scope randomizations tend to offer the highest level of variability for a given range of code. That said, when we swap the bytes of a given function for a randomized copy, the contained bytes may have been altered by any combination of the four transformation techniques, so our solution still fully benefits from all four transformation tactics. It is crucial to evaluate whether re-randomization at the function level allows for enough randomization variability to prevent attackers from guessing or inferring the structure of the code to be swapped in. Specifically, according to the definition by Pappas et al. [26], we define *function randomization variability* to be the number of possible randomized instances that can be generated for a given function.

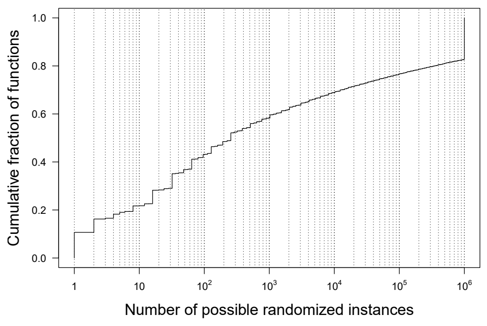


Fig. 3: Function randomization variability.

To gain a better understanding of the resulting randomization variability, we performed an empirical evaluation based on more than 1.5 million functions from 2,566 PE files from both Windows 7 and Windows 8.1. Figure 3 shows the number of possible randomized instances of a function (including its original form), as a cumulative fraction of all 1.5M functions contained in the analyzed PE files. Notice that 10% of the functions have a variability value of one (*i.e.,* just their original instance), meaning that in-place randomization cannot generate any variants for them. The next 4% have only two possible instances, and then the variability for the rest of the functions increases exponentially. For ease of exposition, we cap the calculation of all possible variants to 100,000. As explained in §6, note that just two versions of a function could be enough to foil an attacker, since randomly choosing which version of the code to swap at runtime means that the success rate for the attacker diminishes rapidly.

In general, these 10% of functions cannot be randomized due to their tiny size, often a consequence of compiler intricacies such as basic block sharing, wrapper functions, and other performance optimizations. In fact, our data on Windows binaries shows that about 15% of functions are at most 10 bytes in size, whereas only half of them are larger than 50 bytes. Moreover, 40% of functions consist of a single basic block, while 62% have five or fewer basic blocks. Our findings confirm the observations of Pappas et al. [26] in that the 10% of non-randomizable functions consists mostly of such tiny functions. Overall, we found that roughly 80% of gadgets can be probabilistically broken. Although the possibility remains that a functional payload might still be constructed based solely on non-randomizable gadgets, Pappas et al. [26] showed that this was not possible using state-of-the-art ROP compilers—even without considering recent work by Koo et al.[23] that increases gadget randomization coverage even further. Moreover, additional improvements to in-place randomization techniques could be easily adopted by our system, as this is an orthogonal research topic.

7 Limitations

One limitation of our approach is that it leaves open the possibility of code bytes being disclosed from functions currently on the call stack without those disclosed bytes being re-randomized immediately. In this restricted scenario, the `JIT-ROP` strategy of following code pointers to disclose code pages en masse (with hopes of leaking enough data to be able to compile a ROP payload on demand) can no longer be followed. Indeed, a completely different approach must be taken, involving somehow knowing which functions have been called but not yet returned at the time the exploit is underway—and then only disclosing code in those functions. While such an attack may be conceivable in theory, our expectation is that non-trivial enhancements to the principal of just-in-time code reuse would need to be made before such an attack vector would be feasible. Even if such attacks were possible, the fact that destructive reads targeted at randomizable ranges in a process are rare in normal programs[7] means it would be highly suspicious if one of these reads were to a function that has an activation record on the stack. Since an attacker would need to leverage this rare event many times during said hypothetical attack (*i.e.,* to be able to disclose the requisite amount of code in order to build a ROP payload), the repeated observance of this phenomena could signal that an attack is underway and the offending process would simply be terminated.

Finally, among the attacks against destructive code reads presented by Snow et al. [27], we currently do not deal with code cloning via JIT code generation, as this would require substantial changes into the JIT engine of each protected application. We do not consider this as a significant issue since modern browsers have already adopted constant blinding techniques [3, 25] to prevent the generation of malicious code or ROP gadgets, and thus thwart this type of attack.

[7] In our evaluation with the benchmark programs there was not a single instance were there was an EPT fault in a range of code that was marked by ORP as randomizable code.

8 Conclusion

Over the past year, defenses that leverage the concept of *destructive reads* (*e.g.,* [30, 32]) have been shown to offer a readily deployable mitigation against the threat of just-in-time code reuse attacks. The initial attraction of the promise of destructive reads as a defensive measure stems from the fact that it offers a solution that is compatible with closed-source applications, has low overhead, and has well-defined security properties—three factors that Szekeres et al. [29] argue promote wide-spread adoption of security technologies. Unfortunately, very recent attacks by Snow et al. [27] shed light on inherent weaknesses in adversarial assumptions that did not account for the possibility of *code re-loading* and *code inference* techniques, which can be used to undermine the security guarantees provided by destructive reads.

To address these weaknesses, we provide a solution that strengthens the applicability of destructive reads by eliminating the threats posed by code reloading and code inference attacks. Our defense includes two orthogonal components: one for mitigating code reloading and the other for preventing code inference. In particular, we demonstrate a novel solution for loading randomized copies of an executable any time the image is loaded, thereby preventing an entire class of code reloading attacks. In addition, we take advantage of the localized approach for code diversification used for in-place randomization [19, 26, 31] to enable efficient and robust runtime re-randomization of code that has been disclosed *implicitly* through code inference. Our solution is practical, and offers the first protection (we are aware of) against the ingenious use of zombie gadgets as disclosed by Snow et al. [27].

9 Acknowledgments

We are grateful to the anonymous reviewers and our shepherd, Stefan Brunthaler, for their insightful comments. This work was supported in part by the Office of Naval Research (ONR) under award no. N00014-15-1-2378, and the National Science Foundation (NSF) awards no. 1421703 and 1617902. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the view of the US government, ONR or NSF.

References

1. ROPEME - ROP exploit made easy, 2016. URL `https://github.com/packz/ropeme`.
2. Control-flow enforcement technology preview, 2016. URL `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf`.
3. 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.
4. M. Backes and S. Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *USENIX Security Symposium*, pages 433–447, 2014.
5. 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*, pages 1342–1353, 2014.
6. 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*, pages 268–279. ACM, 2015.

7. 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.
8. S. Brookes, R. Denz, M. Osterloh, and S. Taylor. Exoshim: Preventing memory disclosure using execute-only kernel code. In *International Conference on Cyber Warfare and Security*, page To appear, 2016.
9. P. Chen, J. Xu, J. Wang, and P. Liu. Instantly obsoleting the address-code associations: A new principle for defending advanced code reuse attack. *arXiv preprint arXiv:1507.02786*, 2015.
10. Y. Chen, Z. Wang, D. Whalley, and L. Lu. Remix: On-demand live randomization. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 50–61. ACM, 2016.
11. S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Thwarting cache side-channel attacks through dynamic software diversity. In *Symposium on Network and Distributed System Security*, 2015.
12. S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *IEEE Symposium on Security and Privacy*, pages 763 – 780, 2015.
13. 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*, pages 243–255, 2015.
14. T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *ACM Asia Conference on Computer and Communications Security*, pages 555–566, 2015.
15. 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.
16. D. Evans, A. Nguyen-Tuong, and J. Knight. *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*. Springer, 2011.
17. Y. Fu, J. Rhee, Z. Lin, Z. Li, H. Zhang, and G. Jiang. Detecting stack layout corruptions with robust stack unwinding. In *Symposium on Recent Advances in Intrusion Detection*. 2016.
18. R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz. Enabling client-side crash-resistance to overcome diversification and information hiding. In *Symposium on Network and Distributed System Security*, 2016.
19. 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*, pages 325–336, 2015.
20. C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium*, pages 475–490, 2012. URL https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/giuffrida.
21. C. L. Goues, A. Nguyen-Tuong, H. Chen, J. W. Davidson, S. Forrest, J. D. Hiser, J. C. Knight, and M. Van Gundy. *Moving Target Defenses in the Helix Self-Regenerative Architecture*, pages 117–149. Springer New York, New York, NY, 2013. ISBN 978-1-4614-5416-8. doi: 10.1007/978-1-4614-5416-8_7. URL http://dx.doi.org/10.1007/978-1-4614-5416-8_7.
22. D. Hansen. [rfc] x86: Memory protection keys, 2015. URL https://lwn.net/Articles/643617/.

23. 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.
24. K. Lu, S. Nürnberger, M. Backes, and W. Lee. How to make aslr win the clone wars: Runtime re-randomization. In *Symposium on Network and Distributed System Security*, 2016.
25. G. Maisuradze, M. Backes, and C. Rossow. What cannot be read, cannot be leveraged? revisiting assumptions of JIT-ROP defenses. In *USENIX Security Symposium*, 2016.
26. V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE Symposium on Security and Privacy*, pages 601–615, 2012.
27. K. Snow, R. Rogowski, J. Werner, H. Koo, F. Monrose, and M. Polychronakis. Return to the zombie gadgets: Undermining destructive code reads via code inference attacks. In *IEEE Symposium on Security and Privacy*, 2016.
28. 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*, pages 574–588, 2013.
29. L. Szekeres, M. Payer, T. Wei, and D. Song. SOK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy*, pages 48–62, 2013.
30. A. Tang, S. Sethumadhavan, and S. Stolfo. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *ACM Conference on Computer and Communications Security*, pages 256–267, 2015.
31. 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*, pages 157–168, 2012.
32. 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.
33. D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello. Shuffler: Fast and deployable continuous code re-randomization. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 367–382, 2016.