Juggling the Gadgets: Binary-level Code Randomization using Instruction Displacement

Hyungjoon Koo
Stony Brook University
hykoo@cs.stonybrook.edu

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

ABSTRACT

Code diversification is an effective mitigation against return-oriented programming attacks, which breaks the assumptions of attackers about the location and structure of useful instruction sequences, known as "gadgets." Although a wide range of code diversification techniques of varying levels of granularity exist, most of them rely on the availability of source code, debug symbols, or the assumption of fully precise code disassembly, limiting their practical applicability for the protection of closed-source third-party applications. In-place code randomization has been proposed as an alternative binary-compatible diversification technique that is tolerant of partial disassembly coverage, in the expense though of leaving some gadgets intact, at the disposal of attackers. Consequently, the possibility of constructing robust ROP payloads using only the remaining non-randomized gadgets is still open.

In this paper we present *instruction displacement*, a code diversification technique based on static binary instrumentation that does not rely on complete code disassembly coverage. Instruction displacement aims to improve the randomization coverage and entropy of existing binary-level code diversification techniques by displacing any remaining non-randomized gadgets to random locations. The results of our experimental evaluation demonstrate that instruction displacement reduces the number of non-randomized gadgets in the extracted code regions from 15.04% for standalone in-place code randomization, to 2.77% for the combination of both techniques. At the same time, the additional indirection introduced due to displacement incurs a negligible runtime overhead of 0.36% on average for the SPEC CPU2006 benchmarks.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

Keywords

Code diversification; return-oriented programming

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASIA CCS '16, May 30-June 03, 2016, Xi'an, China
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4233-9/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2897845.2897863

1. INTRODUCTION

The deployment of non-executable page protections in recent operating systems has prompted a shift from code injection to code reuse attacks [23, 35, 50]. In a code reuse attack, after the control flow of a vulnerable process is hijacked, execution is diverted to code that already exists in the address space of the vulnerable process, instead of externally introduced code. Return-oriented programming (ROP) [50] has become the de facto code reuse technique, as the stitching of short instruction sequences (called "gadgets") allows for increased flexibility in achieving arbitrary code execution, even in the presence of additional protection mechanisms such as control flow integrity [15, 16, 21, 24, 28, 47].

Constructing a functional ROP exploit requires precise knowledge of the location and structure of the code in a vulnerable process, and thus various protections aim to break these two assumptions. Address space layout randomization (ASLR) [44] randomizes the load address of shared libraries and main executables to prevent the reuse of code from known locations. Although incomplete ASLR coverage often leaves enough code mapped in static locations to allow the construction of functional ROP payloads [26, 32, 43, 59], even when a process is fully randomized, memory disclosure bugs can be used to leak the base address of code segments. This allows exploits to dynamically adjust gadget offsets in the ROP payload before executing it, improving this way reliability for frequently updated target applications, such as browsers and document viewers [6, 7, 10, 34, 37, 49].

To mitigate the effect of ASLR bypasses, code diversification techniques [11, 12, 22, 30, 33, 41, 55] change not only the location but also the structure of code, breaking the assumptions of attackers about existing gadgets. Even if the offsets of some gadgets in the original program are known, the same offsets in a diversified instance of the same program will correspond to arbitrary instruction sequences, rendering any ROP payloads constructed based on the original code image unusable. In feature-rich applications with scripting support, however, malicious script code can leverage a memory leak to dynamically scan the code segments of a process, pinpoint useful gadgets, and synthesize them into a functional ROP payload at runtime. Such "just-in-time" ROP (JIT-ROP) attacks [52] can be used to effectively bypass code diversification protections. Code diversification can also be bypassed under certain circumstances by remotely leaking [13] or inferring [48] what code exists at a given memory location.

A crucial requirement for the successful operation of a JIT-ROP exploit is the ability to *read* the executable memory segments of the vulnerable process through a memory disclosure vulnerability. Based on this observation, recent works have proposed the enforcement of an "execute-no-read" policy to allow instruction fetches but prevent memory reads from code pages, and thus block any on-the-fly gadget discovery attempt. As the x86 and x86-64 architectures provide only "write" and "execute" memory page protection bits, execute-only policies are enforced in other ways, including page table manipulation [8], split TLBs [27], virtualization extensions [19, 54, 57], or techniques based on a lightweight form of software fault isolation [14].

Either as a standalone defense, or as a prerequisite of execute-only memory protections, code diversification is an effective defense against ROP exploits. From a practical perspective, however, the applicability of most of the existing techniques for the protection of third-party applications depends on the availability of source code [1, 11, 12, 33], debug symbols [3, 4], or the assumption of accurate code disassembly [30,55,58]. Unfortunately, achieving full disassembly coverage and precision is a challenging proposition, especially for the complex closed-source programs that have been plagued by ROP exploits in the wild, such as Windows browsers and document viewers.

In-place code randomization [41], on the other hand, is a code diversification technique that can be applied on stripped binaries even without complete disassembly coverage. This is achieved through a set of narrow-scoped code transformations that eliminate or probabilistically alter the functionality of short instruction sequences that can be used as ROP gadgets, without changing the location or size of basic blocks. Unavoidably, however, the opportunistic nature of the applied transformations results in incomplete randomization coverage, leaving many unaffected gadgets to the disposal of attackers. Based on the results reported by Pappas et al. [41], 82% of the gadgets in the correctly disassembled code of various Windows binaries could be modified on average. Although the authors report that two automated ROP payload construction tools were unable to construct a functional payload using solely the remaining unmodifiable gadgets, this does not exclude the possibility that a functional payload could still be constructed using only non-randomized gadgets, e.g., in a manual way or using an advanced ROP compiler.

Code diversification techniques that do not rely on fully precise code disassembly are an attractive defense due to their practical applicability on even complex binaries. Increasing their randomization coverage in the face of imprecise disassembly is important to improve resilience against attacks that may rely on unmodifiable gadgets for the construction of ROP payloads. Furthermore, recent work on binary-level execute-only memory protections against JIT-ROP attacks [8, 54, 57] necessitates the development of effective code diversification techniques compatible with complex binary executables—without enough code diversification coverage, a functional JIT-ROP exploit may still be possible.

As a step towards improving the current state of the art in binary-level code diversification techniques for COTS software, in this paper we present *instruction displacement*, a new code randomization technique based on static binary instrumentation that does not rely on complete code disassembly coverage. Instruction displacement relocates sequences of instructions that contain gadgets into random locations, and overwrites the original code with trap instructions, effectively preventing their use by an attacker. The end goal of the proposed technique is to improve the randomization coverage and entropy achieved by existing code diversification techniques with a minimal performance impact.

We have implemented a prototype of instruction displacement for Windows binaries, and applied it on a wide range of closed source applications, such as Microsoft Office and Adobe Reader. The results of our experimental evaluation demonstrate that instruction displacement reduces the number of non-randomized gadgets from 15.04% for standalone in-place code randomization, to 2.77% for the combination of both techniques (or from 21.45% to 8.96%, when also considering the non-disassembled code regions). At the same time, the additional indirection introduced due to displacement incurs a negligible runtime overhead of 0.36% on average for the SPEC CPU2006 benchmarks.

In summary, our work makes the following main contributions:

- We present *instruction displacement*, a *practical* code diversification technique for stripped binary executables, applicable even with partial code disassembly coverage.

- We have implemented a prototype of the proposed instruction displacement technique for Windows binaries, and describe in detail its design and implementation.

- We have experimentally evaluated of our prototype implementation, and demonstrate that it reduces the number of non-randomized gadgets from 15.04% for standalone in-place code randomization, to 2.77% for the combination of both techniques, while incurring a negligible runtime overhead of 0.36% for the SPEC CPU2006 benchmarks.

2. BACKGROUND AND MOTIVATION

The complexity of static binary code analysis when dealing with complex stripped executables poses challenges for code diversification protections. Being a provably undecidable problem [56], accurate code disassembly and complete control flow graph extraction is complicated due to intermixed code and data, jump tables, computed jumps, callback and exception handling routines, and other code intricacies. Although at the source code level (or when debug symbols are available) it is possible to perform extensive transformations that effectively randomize all available gadgets [1, 11, 12, 20, 33], at the binary level it is challenging to apply aggressive fine-grained code diversification, such as randomizing the location of functions or basic blocks.

Existing attempts to achieve this, such as Binary Stirring [55], rely on various heuristics to fully and precisely extract all code and code references, so that after randomization all appropriate points can be fixed appropriately. Unfortunately, however, although such approaches may work well for relatively simple executables, they do not scale for large and complex COTS software, such the vulnerable Windows browsers and document viewers that are being targeted in the wild. Indeed, Wartell et al. [55] evaluate Binary Stirring using only main executables (not dynamic libraries) taken from simple utility programs. Introducing a runtime component after static analysis [30], on the other hand, can allow for the randomization of arbitrarily complex programs, in the expense though of increased runtime overhead.

From a practical perspective, a different compromise can be made by accepting the imprecision of static code analysis, and developing binary-compatible code diversification techniques that can tolerate partial code extraction in the expense of the achieved randomization coverage. In-place code randomization (IPR) [41], for instance, uses four different narrow-scoped code transformations that probabilistically alter the functionality of (or eliminate completely) short instruction sequences that can be used as gadgets.

Specifically, *instruction substitution* replaces existing instructions with functionally-equivalent ones (of the same or smaller length), to alter any overlapping instructions that may be part of a gadget. *Basic block instruction reordering* changes the order of instructions within a basic block according to an alternative, functionally equivalent instruction scheduling, again affecting any overlapping gadgets. *Register preservation code reordering* changes the order of the `push <reg>` and `pop <reg>` instructions that are often used at function prologues and epilogues, respectively, to alter the semantics of any useful "`pop; pop; ret;`" gadgets that are often found at function epilogues. Lastly, *register reassignment* swaps the register operands of instructions throughout overlapping live ranges, again with the goal to alter the semantics of any gadgets that involve those registers.

By not altering the location and size of basic blocks and functions, IPR diversifies only the accurately extracted parts of the code, enabling compatibility with third-party stripped binaries. The achieved *partial* code randomization, however, unavoidably leaves a fraction of gadgets completely unaffected by the applied randomization. Specifically, Pappas et al. [41] report that on average, 18% of the gadgets located in the extracted code regions remained unmodified. When also considering the executable regions that were left out due to incomplete disassembly coverage, this percentage increases to 23.1% of all gadgets in the binary. Although the authors demonstrate that two automated ROP payload construction tools did not manage to construct a functional ROP payload using solely the remaining 23.1% of the gadgets, as they admit, this does not preclude that an attacker could manually construct a robust payload using solely unmodifiable gadgets.

Furthermore, some of the randomized gadgets are affected only in a minimal and predictable way that may still allow for their use. For instance, an attacker could still use a reordered function epilogue gadget by initializing the register operands of all `pop` instructions in the gadget with the same value, and then reliably using any one of the initialized registers. Consequently, it is also desirable to increase the entropy of randomization, so that guessing or inferring the state of a randomized gadget becomes much harder.

In this work, we aim to improve both the *coverage* and *entropy* of binary-level code diversification, so that the percentage of any reliably usable (i.e., non-randomized) gadgets is reduced even further.

Threat Model

Code diversification techniques rely on the assumption that an attacker cannot read or leak a diversified instance of the protected code. Experience though has shown that under certain conditions this is possible by reading [52], leaking [13], or inferring [48] the code of a vulnerable process. Although instruction displacement makes the gadgets "disappear" from their original locations, they are still available in some other random location. Consequently, as any code diversification technique, it cannot defend against JIT-ROP [52] and other code leakage attacks.

These can be tackled by recent execute-only memory protections [8, 14, 19, 27, 54, 57], which operate under the assumption that protected code has been properly diversified. For binary-compatible approaches [8, 54, 57], instruction displacement can be crucial in ensuring that adequate randomization coverage has been achieved. When execute-only memory enforcement is implemented using the concept of "destructive reads" [54, 57], however, an attacker may be able to infer the structure of a randomized gadget by (destructively) reading a few preceding bytes [53]. As is also the case with previous in-place code transformations [41], in such a setting where an attacker can disclose arbitrary bytes of the randomized code, instruction displacement can be undermined. For instance, by (destructively) reading the bytes of the inserted jump instructions, a JIT-ROP exploit can pinpoint at runtime the actual address of the displaced gadgets and then use them as part of a dynamically constructed ROP payload. [53]

3. INSTRUCTION DISPLACEMENT

The goal of instruction displacement is to randomize the locations of gadgets so that their starting addresses become unknown to an attacker. In contrast to in-place code randomization, which leaves the randomized instructions in their original locations, instruction displacement relocates sequences of instructions that contain gadgets from their original locations to a newly allocated code segment. Due to ASLR and additional random padding, the base address of this separate segment in the address space of a process is completely random, and thus the locations of all displaced gadgets become unpredictable.

In the rest of this section, we first provide an overview of the overall displacement approach and various constraints that must be satisfied, and then describe in detail the displacement strategy that we follow.

3.1 Overall Approach

Any code diversification approach must maintain the semantics of the original program. In addition, given our assumption that parts of the original code may not have been extracted or disassembled properly, an additional constraint that must be followed is that the location and size of any correctly identified basic blocks must not be altered.

Changing the location of a basic block requires adjusting *all* instructions in the rest of the code that transfer control to that basic block—including computed jumps—to point to the new location. In our case, given that a complete view of the control flow graph is not available, moving a basic block may break the semantics of the code, since any control transfer from non-extracted code to that basic block will become stale, as it will still point to the original location. Similarly, changing the size of a basic block, e.g., in order to add more instructions for diversification purposes, requires shifting any code that immediately follows the expanded basic block. In essence, this means that all basic blocks following the modified one must be moved, which again is not possible.

Given the above constraints, we observe that although we cannot change the boundaries of a basic block, we can still perform arbitrary modifications *within a basic block*, as long as the semantics of the code remain the same (e.g., as is the case with the intra basic block instruction reordering transformation of in-place code randomization [41]). Furthermore, although patching an arbitrary location of a binary executable is not possible, we can safely patch any location within a basic block (assuming there is enough space), as long as the basic block's boundaries have been properly identified.

Based on these two observations, instruction displacement uses code patching to selectively relocate (some of) the instructions of a basic block to a random location. The overall approach is illustrated in Figure 1. The upper part of the figure shows the original code of a basic block (rectangles represent variable-length instructions), and the lower part shows the modified version of the code, with some of its instructions displaced into a new code region. In this example, the basic block contains two ROP gadgets, G1 and G2, located at addresses `addr1` and `addr2`, respectively. The first (G1) is an *unintended* gadget that begins from the middle of an existing instruction and ends with a `ret` instruction (opcode 0xC3) located again within an existing instruction. The second (G2) is an *intended* gadget that ends with a `call eax` instruction that is part of the program's code.

In the modified version of the code, the instructions at the beginning of the basic block have been overwritten by a relative `jmp` instruction that points to the overwritten instructions, which have now been copied into a random location, along with some of their following instructions. Note that the `jmp` instruction takes five bytes (one-byte opcode plus four bytes for its immediate operand), and thus instructions contained in basic blocks shorter than five bytes cannot be displaced in the general case. Although a smaller 2-byte relative `jmp` instruction could be used, its 8-bit displacement usually

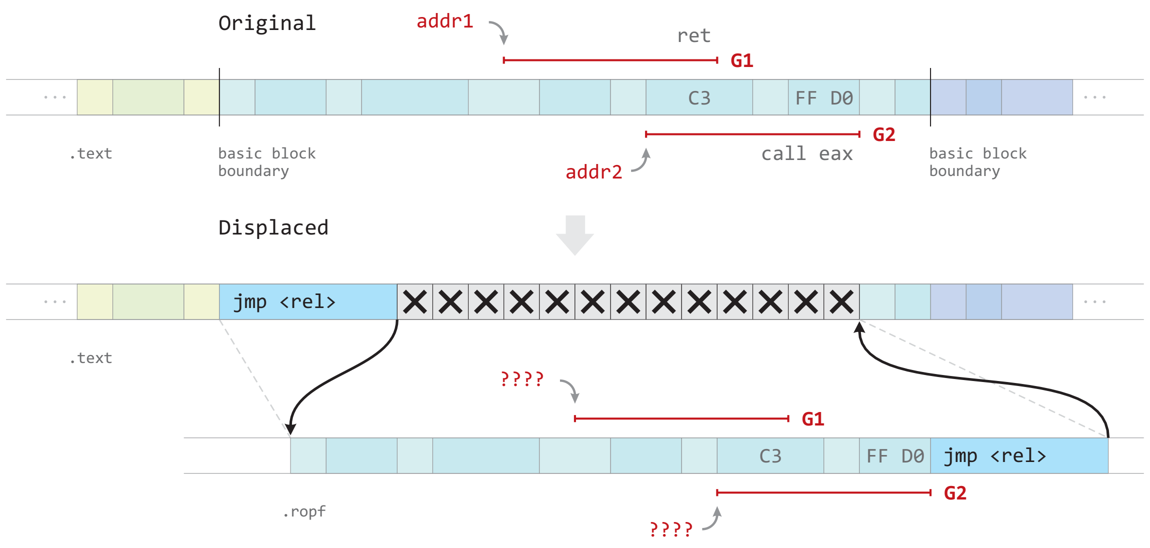


Figure 1: High-level view of instruction displacement. By moving part of the original basic block's code in a random location, the starting addresses of the two gadgets become unpredictable. To maintain the original semantics of the code, the displaced instructions are linked with the rest of the code using relative jumps.

cannot "reach" far enough for transferring control to the area that contains the displaced instructions. For such cases, an alternative approach would be to insert a smaller trap instruction and achieve indirection through an appropriate handler routine. Unfortunately, the associated runtime overhead of such a solution would be prohibitively high. As we discuss in Section 5.1.2, the percentage of gadgets in such small basic blocks is very low, in the order of 0.83%, and thus we have chosen to ignore them.

Recall that a basic block is defined as a straight-line sequence of instructions with only one entry point and only one exit. Consequently, we can safely patch with a `jmp` instruction *any* location within a basic block that corresponds to the address of an existing instruction. To preserve the semantics of the basic block's code, all that remains to be done is to transfer control back to the original location after the execution of the displaced instructions. This can be achieved again with a relative `jmp` placed right after the final displaced instruction.

By moving the instructions that contain the two gadgets in a randomly chosen location, an attacker cannot rely on them anymore based on their original addresses. The original code right after the patched location is overwritten with instructions that will crash the program or trap execution (e.g., privileged or interrupt instructions), and thus any transfer to the original locations of the gadgets (`addr1` and `addr2`) is ruled out. At the same time, the starting addresses of the displaced gadgets are now random, so an attacker cannot guess them (proper ASLR implementations, e.g., the one used in the latest versions of Windows, and additional random padding at the beginning of the segment that contains the displaced code fragments achieve enough entropy for that purpose).

3.2 Displacement Strategy

Although the address of a displaced gadget is not known to an attacker, the location of the inserted `jmp` can be easier to predict, and thus an attacker can still use it as the starting address for reaching a gadget. Depending on whether a gadget is intended or unintended, we can follow a different displacement strategy while trying to minimize the number of displaced instructions.

3.2.1 Intended Gadgets

Due to the inserted `jmp`, among all intended gadgets in a displaced code region, the one that (in the original code) begins with the patched instruction is still usable—the attacker can still rely on its original address, and the inserted `jmp` at that location will unconditionally transfer control to it. Depending on the location of the gadget within the basic block, however, this means that the attacker now must use a longer gadget, which is likely to have many more side effects in terms of register and memory state changes (a given indirect branch instruction is generally the "end" of several nested gadgets extending backwards from it). Although the use of longer-than-usual gadgets is possible [16, 21, 28, 47], it complicates significantly the construction of ROP payloads due to the additional side effects of the extra non-essential instructions.

To increase the complexity of any remaining usable gadgets, a displaced sequence of instructions begins as "far" as possible from any contained gadgets—in most cases, this means the beginning of the respective basic block. Given that it is desirable to minimize the number of displaced instructions, for very large basic blocks, we have set a limit of displacing up to 20 instructions from the end of a target gadget. In the example of Figure 1, the `jmp` is inserted at the beginning of the basic block, and the three instructions of gadget G2 can now be used only if seven additional instructions are executed before them.

Given that the percentage of the remaining usable gadgets by following the "entry point" of a displaced region is very low (0.6% in our experiments, as discussed in Section 5.1.2), we have chosen to not take any further action about them. We should note, however, that instruction displacement opens up more possibilities for randomizing or eliminating altogether the displaced gadgets. Indeed, once a sequence of instructions has been displaced, further transformations on the displaced gadgets can be applied. Fortunately though, in contrast to the general case, we can now fully disassemble the displaced instructions, and there is no space constraint due to previous or following basic blocks, as we have full control over the region where the displaced instructions are copied, and the placement of individual code fragments within that region. This means that we can apply more aggressive code transformations, beyond what is possible using in-place code randomization, such as splitting an existing instruction into two or more instructions. As an alternative example, we can apply transformations similar to the ones used by G-Free [40] to completely eliminate the displaced gadgets.

3.2.2 *Unintended Gadgets*

Unintended gadgets begin only from unaligned instructions, and may end with an either aligned or unaligned instruction (if the first instruction is an aligned one, then there is no way to "escape" from the intended instruction stream due to the unambiguous nature of instruction decoding). Consequently, the "predictable entry point" issue discussed above does not apply when a displaced instruction sequence contains solely unintended gadgets—by following the inserted `jmp`, an attacker still cannot reach the unintended gadget (as is the case with gadget G1 in Figure 1). This makes the decision on which location to patch much simpler: it is enough to patch the intended instruction that contains the opcode byte of the first unintended instruction of the gadget. The location of that opcode byte in the displaced instruction will be random, and by following the `jmp` the attacker will be forced to execute the intended instruction stream, without being able to reach the unintended gadget.

Especially for unintended gadgets, this approach is quite effective even when a gadget spans two consecutive basic blocks. In such cases, although we cannot displace the whole gadget (due to our restriction in maintaining basic block boundaries intact), it is enough to displace even just the first instruction of the gadget to make the whole gadget unusable. This is possible when the first overlapping instruction is located towards the end of the first basic block, in which case it can be safely displaced.

In essence, instruction displacement enforces a coarse-grained control flow integrity constraint in a probabilistic and selective way. For intended gadgets, control flow is allowed to reach only the entry point of the basic block that contains a gadget (or, for very large basic blocks, the first of a sufficiently large number of instructions preceding the gadget). For unintended gadgets, control flow cannot "escape" from the intended instruction stream and reach any of the unaligned instructions of the gadget.

3.2.3 *Combining Instruction Displacement with In-Place Code Randomization*

Each displaced code region results in a slight increase in memory space and CPU overhead, due to the copied code, the extra indirection, and the disruption of code locality (although the latter some times has a positive impact, as discussed in Section 5.4). It is thus desirable to minimize the number of displaced regions whenever possible. Given that the end goal of the proposed technique is to improve the coverage and entropy achieved by existing code diversification techniques, we can combine instruction displacement with in-place code randomization, and apply the former only for gadgets that cannot be randomized by any of the existing code transformations of IPR (and optionally, also for gadgets that are randomized with insufficient entropy).

To that end, each binary is first analyzed to pinpoint all existing gadgets, and IPR is applied to randomize or eliminate as many gadgets as possible. Then, a second instruction displacement pass considers all remaining unmodifiable gadgets, and attempts to displace them whenever possible. In many cases, a basic block might contain several gadgets, some of which might be affected by IPR, and some not. To increase randomization coverage as much as possible, we follow a conservative approach and apply displacement even if only a single out of several gadgets within the same instruction sequence cannot be randomized by IPR.

3.3 Putting It All Together

We discuss a few remaining issues and optimizations by looking at a real example of applying instruction displacement. Figure 2 shows a basic block from Adobe Reader's BIB.dll that contains several (nested) gadgets ending with a `ret` instruction. In particular, "`pop; pop; ret;`" gadgets are quite useful in assembling ROP payloads, while the `call`-preceded gadget starting with the `lea` instruction can be used to bypass coarse-grained CFI protections [16, 21, 28, 47]. After instruction displacement, the `push` instruction at address 0x7002806 has been replaced by a direct `jmp` to the displaced instructions, which now reside at a random location within a new `.ropf` section of the binary (detailed in the following section). All remaining original instructions are overwritten with `int3` instructions. The only option that is now left for an attacker is to use the code of the whole basic block, starting with the `push` instruction. This might not be desirable, as it involves the execution of another function, which may have disastrous side effects. All other (intended and unintended) gadget starting locations within the basic block become unpredictable.

This example illustrates a common case in which the ending instruction of a gadget is also the final instruction of a basic block. We can exploit this fact to reduce the number of indirections needed due to instruction displacement. Depending on the type of branch at the end of a basic block, a `jmp` back to the original location may not be needed at all. As the most common case, all indirect branch instructions (i.e., those that can be the ending instructions of gadgets), will transfer control to the intended target no matter whether they have been displaced or not. In this example, the `ret` instruction will always transfer control to the return address that will be read from the stack, irrespectively of the actual location of the `ret`. Consequently, an extra `jmp` for transferring control back to the original location is not needed. The same is true for any unconditional branches, but care must be taken to adjust any relative displacement operands accordingly. Unfortunately, the same strategy cannot be applied for conditional branches, as we do not have control of the fall-through target.

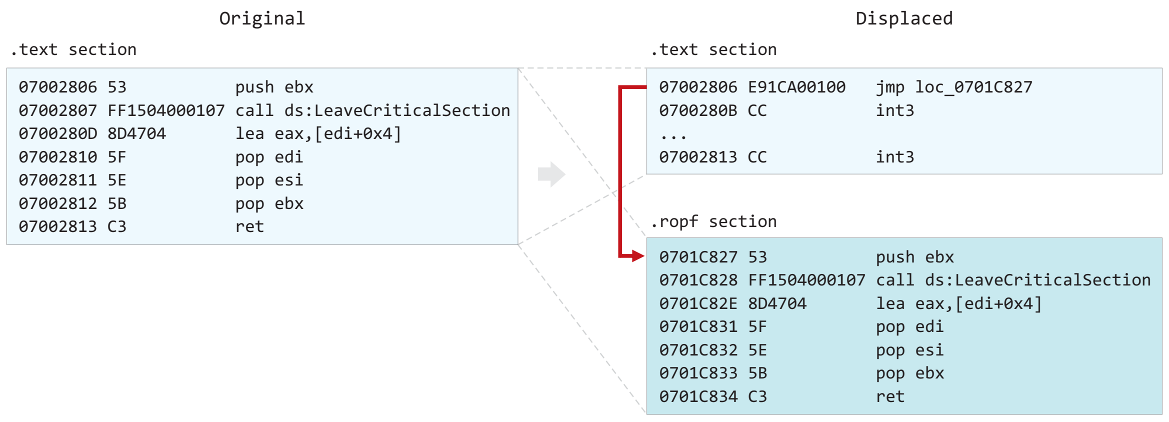
Any other instructions that involve relative address operands must also be adjusted accordingly after the randomly chosen location of the displaced code region is picked. Besides relative `call` instructions and the like, this includes PC-relative memory accesses for 64-bit programs.

4. IMPLEMENTATION

To demonstrate the effectiveness of instruction displacement, we have developed a prototype implementation for Windows binaries. Our prototype supports 32-bit PE binaries (both main executables and dynamic link libraries), without relying on any debug or symbolic information (e.g., PDB files). To randomize a binary, a three-phase process is followed: i) identification of candidate gadgets for displacement, ii) modification of the PE executable to add a new code section for the displaced instructions, and iii) binary instrumentation for actually displacing the selected gadgets. In the following, we discuss these three phases in detail.

4.1 Gadget Identification

The first phase aims to identify the code regions that will be displaced. A necessary condition for any candidate region is to fall within the boundaries of a basic block, and thus a first necessary step is to extract the code and identify as many functions and basic blocks as possible. This is achieved using IDA Pro [29], a state-of-the-art code disassembler that achieves decent accuracy when dealing with regular (non-obfuscated) PE executables. IDA Pro leverages the relocation information present in Windows DLLs,



```
07002806 53            push ebx
07002807 FF1504000107  call ds:LeaveCriticalSection
0700280D 8D4704        lea eax,[edi+0x4]
07002810 5F            pop edi
07002811 5E            pop esi
07002812 5B            pop ebx
07002813 C3            ret
```

```
07002806 E91CA00100    jmp loc_0701C827
0700280B CC            int3
...
07002813 CC            int3
```

```
0701C827 53            push ebx
0701C828 FF1504000107  call ds:LeaveCriticalSection
0701C82E 8D4704        lea eax,[edi+0x4]
0701C831 5F            pop edi
0701C832 5E            pop esi
0701C833 5B            pop ebx
0701C834 C3            ret
```

Figure 2: A real example of gadget displacement taken from Adobe Reader's BIB.dll module.

and identifies compiler-specific code constructs and optimizations, such as basic block sharing [31]. We should note, however, that as in previous works [41], we do not take into account IDA Pro's speculative disassembly results, e.g., for embedded data and code regions that are reached only through computed jumps or which are part of signal handling routines. These rely on heuristics that are prone to errors, and thus we follow a conservative approach to prevent any correctness issues with the instrumented code due to falsely identified code regions.

Our code extraction module is based on the open-source implementation of in-place code randomization [2], which we also use to pinpoint all remaining gadgets after the application of IPR. We have extended the implementation to consider gadgets comprising up to 15 instructions, from the just five instructions in the original implementation. We use IPR with maximum coverage settings, so as to reduce the number of displacements. An analysis pass then identifies all remaining unmodified gadgets and calculates the appropriate code regions to displace as many gadgets as possible. Gadgets contained in basic blocks smaller than five bytes are left intact, as they cannot be safely patched. Depending on the proximity of different gadgets (ending with different indirect branch instructions) within the same basic block, separate candidate regions are merged to minimize the required instrumentation in terms of additional `jmp` instructions. The final boundaries of each region are computed based on the strategy described in Section 3.2.

4.2 PE File Layout Modification

Once all to-be-displaced code regions have been identified, the PE file is augmented with a new code section, named `.ropf`, in which the displaced regions will be moved. The executable is modified using the `pefile` python library [17]. First, we define a new section header in accordance with the `IMAGE_SECTION_HEADER` structure, which is inserted into the section headers array, between the last existing header and the first data section. For simplicity, the new section is appended at the end of the file, so that the rest of the sections remain intact. Although more complex layouts could be studied to keep displaced instructions closer to their original locations and facilitate patching using two-byte `jmp` instructions (e.g., by identifying and reusing any unused regions within existing code segments), the resulting increase in coverage would still be minimal (due to the small percentage of less-than-5-byte basic blocks, as well as the limited reach of the 8-bit displacement), so the added complexity is not justified.

Besides the addition of the above entry, some existing information related to the overall PE image must be updated accordingly. Specifically, the following entries in the `IMAGE_OPTIONAL_HEADER` structure need to be updated: size of code and image, size of initialized data and uninitialized data, and the checksum of the binary. The size of the `.ropf` section is calculated based on the identified code regions, and by provisioning some extra room for the added `jmp` instructions for transferring control back to the original code, as well as some padding space.

4.3 Binary Instrumentation

With the `.ropf` section ready to host the displaced instructions, the actual patching of the original code and the copying of the displaced instructions can begin. The identified code regions are copied and placed in the `.ropf` section in a randomly chosen order (an additional small random gap can be added between successive regions if needed). As regions located within the same basic block or function of the original code end up in close proximity after displacement, this some times has a positive impact in terms of runtime overhead due to code locality, as discussed in Section 5.4. More sophisticated ordering schemes could also be explored, especially when taking into consideration hot spots and code locality, e.g., based on prior profiling information. To diversify the locations of gadgets even further, a large padding area of a randomly selected size is allocated at the beginning of the `.ropf` section.

For the code disassembly and reassembly operations needed to patch the original code locations, adjust the operands of displaced instructions, and insert additional `jmp` instructions at the end of displaced regions (whenever necessary), we use the Capstone framework [46]. We have also employed several optimizations using bit-level operations to speed up the instrumentation phase. Care must be taken while generating the `jmp` instructions for patching the original code so that any immediate operands do not result in accidental generation of new potentially useful gadgets (e.g., due to embedded 0xC3 bytes). This is avoided by adjusting the destination address of the displaced instructions by a few bytes in case an immediate contains an indirect branch opcode.

Finally, a final important step for ensuring the correct operation of the resulting binary is to update the PE file's relocation information for all affected code locations. To enable loading of modules at arbitrary addresses, PE files contain a `.reloc` section that contains a list of offsets (relative to each PE section), known as "fixups" [45, 51]. At load time, these entries specify the absolute code or data

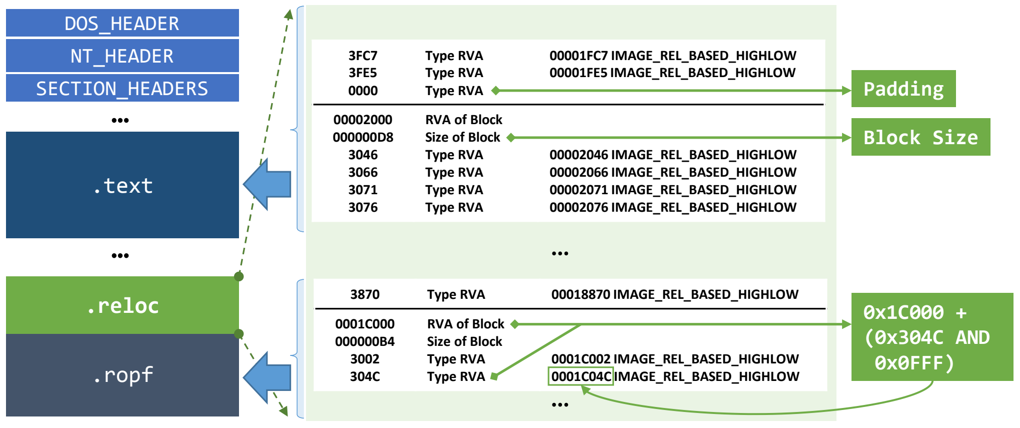

DOS_HEADER
NT_HEADER
SECTION_HEADERS
...
.text
...
.reloc
.ropf
3FC7 Type RVA 00001FC7 IMAGE_REL_BASED_HIGHLOW
3FE5 Type RVA 00001FE5 IMAGE_REL_BASED_HIGHLOW
0000 Type RVA
Padding
00002000 RVA of Block
000000D8 Size of Block
Block Size
3046 Type RVA 00002046 IMAGE_REL_BASED_HIGHLOW
3066 Type RVA 00002066 IMAGE_REL_BASED_HIGHLOW
3071 Type RVA 00002071 IMAGE_REL_BASED_HIGHLOW
3076 Type RVA 00002076 IMAGE_REL_BASED_HIGHLOW
...
3870 Type RVA 00018870 IMAGE_REL_BASED_HIGHLOW
0001C000 RVA of Block
000000B4 Size of Block
3002 Type RVA 0001C002 IMAGE_REL_BASED_HIGHLOW
304C Type RVA 0001C04C IMAGE_REL_BASED_HIGHLOW
...
0x1C000 + (0x304C AND 0x0FFF)

Figure 3: Rewriting the relocation section of a PE file for both the original (`.text`) and the new (`.ropf`) code sections.

addresses within the module that must be adjusted according to the module's load address (which is usually randomly selected, due to ASLR).

As Figure 3 shows, the relocation table consists of a series of blocks grouped according to their relative virtual address (RVA). Each block begins with the RVA, the size of the block, the actual relocation entries, and some padding bytes for alignment. Each relocation entry consists of two bytes. The first four bits of the entry are set to 0x3, which represents the most common type of fixup transformation (`IMAGE_REL_BASED_HIGHLOW`). The following 12 bits represent the offset from the RVA of the corresponding block. The relocatable address can be calculated by adding the RVA and the offset, making it relative to the new base address of the segment instead of its original (preferred) one [51].

A crucial detail here is that any relocation entries regarding locations in the original code regions (that have now been displaced) must be removed from the respective block. The reason for this is that any stale entries can lead to corruption of the inserted `jmp` instructions, e.g., in case any of the overwritten instructions happened to involve RVAs with corresponding `.reloc` entries. Thus, not only new entries for the `.ropf` section must be created, but also the corresponding entries for the `.text` section must be removed, resulting in a total number of relocation entries equal to the number of entries in the original binary.

5. EXPERIMENTAL EVALUATION

In this section we present the results of the experimental evaluation of our prototype implementation in terms of randomization coverage, file size increase, correct execution, and performance overhead. Our tests were performed on a 64-bit Windows 10 system equipped with an Intel Core i5-4590 3.3GHz processor and 16GB of RAM. For the evaluation of randomization coverage, we used a set of 2,695 PE files (both main executables and DLLs) from two different versions of Adobe Reader (Reader v9.3 and Acrobat Reader DC), Microsoft Office 2013, two Windows 7 and Windows 8.1 installations, and other programs and utilities, as detailed in Table 1. For correctness and performance evaluation, we used a set of core Windows DLLs, as well as the Windows version of the SPEC CPU2006 benchmark suite.

5.1 Randomization Coverage

We begin our evaluation with the goal of assessing the improvement in terms of randomization coverage that instruction displacement can achieve. To that end, we compare the randomization coverage of in-place code randomization [41], instruction displacement, and the combination of the two techniques, as described in Section 3.2.3. In our initial experiments we use a maximum gadget length of five instructions, so that our results are comparable with the results reported by Pappas et al. [41]. In Section 5.1.3, we present further results when considering gadgets of size up to 10 and 15 instructions.

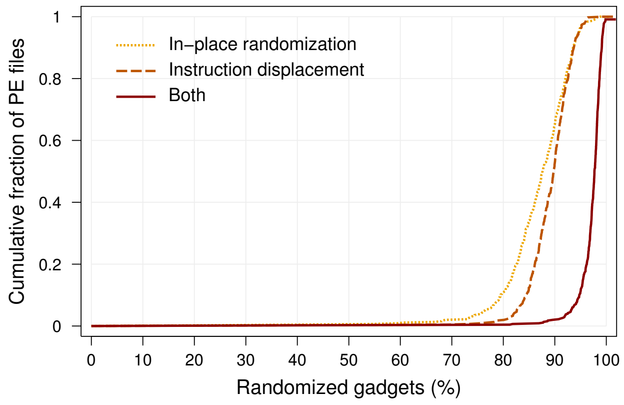


Figure 4: Randomized gadgets per PE file due to in-place code randomization, instruction displacement, and the combination of both techniques.

Table 1 summarizes key statistics about the distribution of gadgets in the tested binaries, and the randomization coverage of the two techniques. The 2,695 executables contain a total of approximately 13 million gadgets, 58.52% of which are unintended. The "unreachable" column refers to gadgets located in regions that cannot be properly disassembled, and thus are left untouched (by both techniques). These amount to 6.37% of all gadgets on average. In the rest of this section, unless specified otherwise, percentages of randomized gadgets are calculated over the number of gadgets located only within the properly disassembled code regions.

5.1.1 Coverage Improvement

Figure 4 shows the percentage of randomized gadgets in each PE file achieved by in-place code randomization, instruction displacement, and the combination of both techniques, as a cumulative frac-

Applications		Gadget Distribution			Randomized Gadgets			Other
Name	Files	Total	Unintended	Unreachable	IPR	Disp.	Both	File Increase
Adobe Reader	50	677,689	55.24%	4.61%	82.16%	88.98%	96.69%	2.18%
MS Office 2013	18	195,774	55.04%	4.93%	83.02%	88.71%	97.25%	2.98%
Windows 7	1,224	5,595,031	53.97%	6.11%	83.95%	89.11%	97.41%	1.94%
Windows 8.1	1,341	6,077,543	63.46%	6.90%	86.43%	91.14%	97.15%	1.42%
Various	62	496,749	55.15%	5.79%	83.23%	89.21%	96.83%	1.79%
Total	2,695	13,042,786	58.52%	6.37%	84.96%	90.04%	97.23%	1.68%

Table 1: Data set of PE files used for randomization coverage analysis.

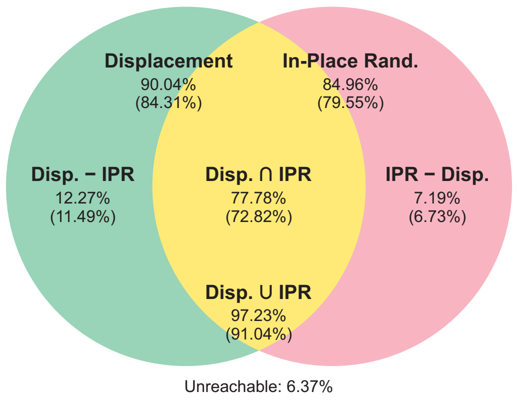


Figure 5: Randomization coverage of each technique in relation to each other. Instruction displacement increases the coverage achieved by in-place randomization alone from 84.96% to 97.23%.

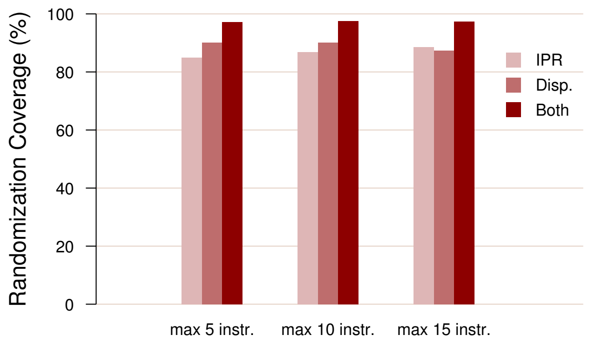


Figure 6: Randomization coverage for different maximum gadget lengths.

tion of all PE files in our data set. Although both techniques achieve comparable coverage, their combination manages to randomize a greater number of gadgets, and this is true for most executables, as evident from the slightly steeper curve. Specifically, as shown in Table 1, in-place code randomization on average affects 84.96% of the gadgets, instruction displacement affects 90.04% of the gadgets, while their combined use randomizes 97.23% of all gadgets in the properly disassembled code regions.

The Venn diagram in Figure 5 sheds more light into how each of the two techniques contributes in randomizing gadgets. While a majority of 77.78% can be randomized by both techniques, instruction displacement affects an extra 12.27%, increasing significantly the overall randomization coverage. When considering the whole binary, including the areas that cannot be disassembled, the randomization coverage is improved from 79.55% to 91.04%.

5.1.2 Gadget Analysis

The achieved randomization coverage of 97.23% leaves only a remaining 2.77% of gadgets that cannot be randomized by either of the two techniques. There are several reasons why instruction displacement cannot affect those gadgets. Among them, 0.83% are contained within basic blocks smaller than five bytes, and thus cannot be displaced due to the restriction of having to use a 5-byte `jmp` instruction for patching. Another 0.6% correspond to "basic block entry" gadgets that remain usable by following the inserted `jmp` instruction. The rest 1.34% cannot be displaced due to various other corner cases related to basic block alignment.

We also looked specifically into `call`-preceded gadgets, as they can be particularly useful for an attacker that wants to bypass any deployed coarse-grained CFI protections [16, 21, 28, 47]. The percentage of `call`-preceded among all gadgets, including the areas that cannot be disassembled, is 5.76%. After randomization using both techniques, their number is reduced to just 0.16% of all gadgets, with 0.14% being located in unreachable regions. This means that the achieved randomization coverage is enough to affect the vast majority of `call`-preceded gadgets.

5.1.3 Longer Gadgets

Given that an attacker may be able to use longer gadgets by spending some additional effort, we explored how the randomization coverage is affected when considering longer gadgets. To that end, we repeated our experiments using a maximum gadget length of 10 and 15 instructions. In all cases we follow the same approach as before, i.e., we first apply each diversification technique separately, followed by their combination.

Although the total number of discovered gadgets when considering a maximum length of 10 instructions increases by about 18%, the percentage of randomized gadgets using both techniques remains almost the same, at around 97.4%, and so does also for 15-instructions-long gadgets, as shown in Figure 6. As the gadget size increases, IPR affects a slightly larger percentage of gadgets, moving from 84.96% to 86.78% and 88.59%, respectively. This result is expected, as the longer the gadget, the more opportunities for the different code transformations of IPR to affect some of its instructions. In contrast, as longer gadgets are more likely to span consecutive basic blocks, the coverage of instruction displacement drops slightly from 90.11% for 10 instructions to 87.42% for 15 instructions. It still contributes though an additional 9% in coverage when combined with IPR.

5.2 File Size Increase

Instruction displacement unavoidably incurs an increase in the size of the randomized PE files. Based on our experiments, the size of the `.ropf` section that hosts the displaced gadgets was verified to increase proportionally to the ratio of displaced code regions. As shown in Table 1 (last column), the average increase over the original PE file is minimal, at about 2.35%.

The total size of the displaced code regions is slightly larger than the original displaced code due to the additional *jmp* instructions that sometimes are appended at the end of displaced regions, and more rarely, due to larger displacement offsets in some operands. From all displaced regions, only 43.54% require a pair of jumps—in the rest of the cases, the region ends with an indirect branch instruction that takes care of transferring control to the appropriate location. Some additional spaced is also consumed to the random padding at the beginning of the `.ropf` section.

5.3 Correctness

Any static binary instrumentation technique should preserve the original semantics of the instrumented program. To ensure that our transformations do not break the functionality of complex binaries, we first performed some manual testing with real-world applications, such as Adobe Reader. After randomization, we verified that a variety of PDF documents would open and render properly. Furthermore, when running diversified versions of the SPEC benchmarks, as described below, we did not encounter any issues with erroneous output.

As an attempt to exercise a more significant amount of code, we also used an automated testing approach based on the test suite of Wine [5], as similarly done by previous works [41, 42]. Wine is a compatibility layer capable of running Windows applications on several POSIX-compliant operating systems. To validate that the ported APIs provided by Wine function as expected, Wine comes with an extensive test suite that covers the implementations of most functions exported by the core Windows DLLs. We ported to Windows some of Wine's test suites for 27 system DLLs, comprising a total of 10,036 different test cases, and used them repeatedly with randomized versions of those 27 actual Windows DLLs. By checking the outcome between various inputs and expected outputs, we could confirm that the randomized versions of the DLLs always worked correctly.

5.4 Performance Overhead

Finally, our last set of experiments focused on evaluating the performance overhead of instruction displacement. Since the technique involves extensive code patching and indirection, we expect to observe an increase in CPU overhead due to the extra executed `jmp` instructions and different code locality patterns. To get a better understanding of the performance implications, we performed two sets of experiments. First, we used a subset of the DLLs and Wine test cases used for the correctness evaluation, leaving aside any tests that involved the creation of files and other operations that would mask out any CPU overhead. For each DLL, we measured the overall CPU user time for the completion of all relevant tests by taking the average time across multiple runs, using both the original and the randomized versions of the DLL. Second, we used the Windows-compatible subset of the standard SPEC CPU2006 benchmark suite.

Figures 7 and 8 show the runtime overhead of instruction displacement (when used in conjunction with in-place code randomization) over native execution for the Wine and SPEC experiments, respectively. The average overhead across all Wine tests is 0.48%, with a maximum of 1.87%. Surprisingly, some of the test cases exhibit

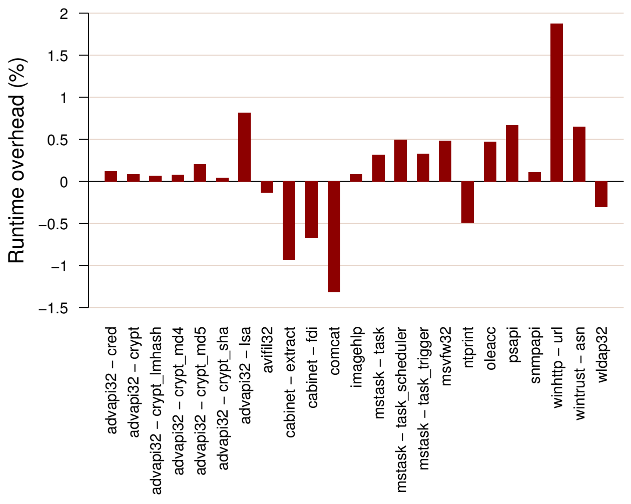


Figure 7: Runtime overhead over native execution for diversified versions of Windows system DLLs, driven by test cases ported from Wine's test suite. The average overhead across all tests is 0.48%.

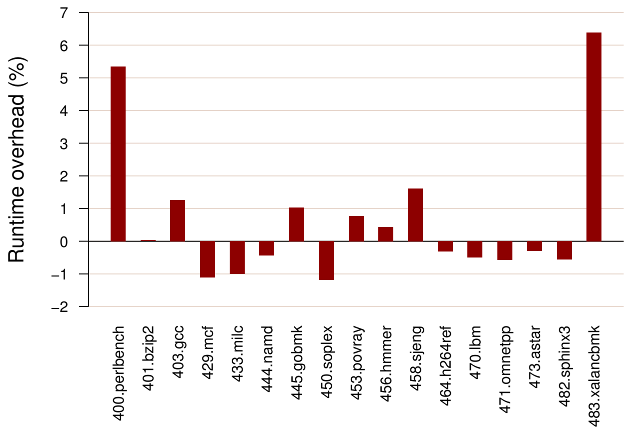


Figure 8: Runtime overhead for the SPEC CPU2006 benchmarks. The average overhead across all benchmarks is 0.36%.

a negative overhead, meaning that the diversified code ran faster than the original. We observed the same behavior in a stable and repeatable way across many iterations of the same experiment, with different instances of randomized binaries. We attribute this speedup in better caching behavior due to better code locality in the `.ropf` section, as different "hot" basic blocks may now be brought in close proximity.

For the SPEC benchmarks, the average overhead was 0.36%. The two benchmarks with the highest overhead are `xalancbmk` and `perlbench` (6.38% and 5.34%, respectively), which is expected given that they are among the largest and more complex ones. A few other benchmarks exhibited the same negative overhead behavior that was also observed before, again in a consistent way across many repetitions.

We analyzed further the Wine and SPEC test cases that exhibited negative overheads using statistical hypothesis testing. With the null hypothesis that the mean CPU times for the original and randomized binaries are identical, Welch's two-sample *t*-test failed to reject it.

That is, the means of the two distributions of CPU times for the original and randomized binaries in each case are not significantly different from each other with a 95% confidence interval, implying that these differences fall within the margin of measurement error.

We also explored the overhead of instruction displacement when used as a standalone technique, without the prior application of IPR. That is, when all 90.04% of the gadgets that can potentially be displaced are actually displaced, as opposed to just 12.27% when used in conjunction with IPR. The average overhead across all SPEC benchmarks in that case was just 2.06%, denoting that even extensive but focused patching can still incur a minimal performance overhead.

6. DISCUSSION AND LIMITATIONS

The two main limiting factors for instruction displacement in terms of randomization coverage are the precision of code extraction, and the size of existing basic blocks. Even when using a state-of-the-art disassembler like IDA Pro, some parts of the code cannot be extracted, and thus any gadgets in those regions remain unmodified. As our experiments have shown (Figure 5), when considering all available gadgets in a binary, instruction displacement reduces the number of unmodifiable gadgets from 21.45% for standalone in-place randomization, to 8.96% for the combination of both techniques. Given that the majority of them are located in unreachable regions (6.37%), a more accurate code extraction technique would allow for improved coverage. On the other hand, only a fraction (0.83%) of all extracted gadgets could not be displaced because they reside in small basic blocks. For "entry point" gadgets that still remain available after displacement, we plan to explore further transformations that can be applied on the displaced instructions, as discussed in Section 3.2.1.

Given the best-effort nature of our approach, we still cannot exclude the possibility of an attacker being able to assemble a functional ROP payload using solely the remaining fraction of unmodifiable gadgets. An indication about the complexity of ROP payload construction when working with a limited set of gadgets was provided by Pappas et al. [41], who showed that two automated ROP payload construction frameworks were unable to construct a functional payload using only the remaining unmodifiable gadgets by IPR. With the application of instruction displacement on top of IPR, this set of gadgets is significantly reduced even further (from 21.45% to 8.96%), and thus it is reasonable to assume that automated construction becomes even harder.

Besides the significant increase in coverage, instruction displacement also offers an additional benefit over in-place randomization in terms of the achieved randomization entropy. Although 77.78% of the gadgets can be randomized by both techniques, the randomization achieved through instruction displacement is qualitatively different. For some gadgets, IPR affects only a few of their instructions (or even just some of the instructions' operands), and often gadgets may exist in one out of just two possible states, leaving open the possibility of them being still usable after making the right assumptions. On the other hand, displaced gadgets end up in random locations that are infeasible to predict. Although in this work we have restricted the use of instruction displacement only for gadgets that are not randomized at all by IPR, in the future we plan to explore more aggressive combinations of the two techniques to improve randomization entropy even further. As we showed in Section 5.4, the associated overhead when displacing all possible gadgets is still modest, at 2.1%, so a small increase in the current number of displaced regions would have a negligible impact in the overall overhead.

7. RELATED WORK

The concept of diversification has been the basis of a wide range of software protections against code injection and code reuse attacks [18, 25, 36]. Address space layout randomization (ASLR) [39, 44] is probably one of the most widely deployed protections against code reuse attacks. Besides frequent weaknesses related to incomplete ASLR coverage [26, 32, 43, 59], however, even proper ASLR can often be bypassed using memory disclosure bugs that leak the base address of loaded modules [6, 7, 10, 34, 37, 49, 52].

More fine-grained forms of randomization, complementary to ASLR, aim to diversify even further the layout and structure of a process' code. Randomization can be performed at the function [1, 11, 12, 33], memory page [9], basic block [3, 4, 55], or instruction level [22, 30, 41], breaking the assumptions of attackers about the location and structure of gadgets based on the original code image. From a deployment perspective, most of the techniques that fully randomize all code segments depend on the availability of source code [1, 11, 12, 20, 33], debug symbols [3, 4], the use of heavyweight dynamic binary instrumentation [20, 30] or the assumption of accurate code disassembly [9, 55, 58]. In contrast, in-place code randomization [41], can be applied on stripped binaries even with partial disassembly coverage.

Another line of compile-time approaches prevent the construction of ROP code by generating machine code that does not contain unintended gadgets, and which safeguards any remaining intended gadgets using additional indirection [38, 40].

Oxymoron [9] applies fine grained code randomization that is compatible with code sharing. It offers some resistance to JIT-ROP attacks [20, 52] by replacing direct branches with indirect branches. This makes it harder for attackers to harvest code pages by following the flow of control. Other recent research efforts protect against JIT-ROP attacks by making code executable but not readable. This can be achieved by relying on page table manipulation [8], split TLBs [27], hardware virtualization extensions [19, 54, 57], or techniques based on a lightweight form of software fault isolation [14].

The binary-compatible of these approaches [8, 54, 57] can benefit from the improved randomization coverage achieved by our proposed instruction displacement technique.

8. CONCLUSION AND FUTURE WORK

The emergence of return-oriented programming attacks and the limitations of ASLR against their advanced forms have prompted active research on defenses based on code diversification. From a practical perspective, however, only a few of those approaches [41] can be applied for the protection of the complex COTS software that is being targeted by current in-the-wild exploits, such as closed-source browsers and document viewers.

In this paper, we focused on the limitations of current binary-compatible code diversification techniques, and in particular on in-place code randomization (IPR) [41]. Our main goal is to improve its randomization coverage, which currently leaves many gadgets unaffected (and others with insufficient randomization entroy), and thus at the disposal of attackers for the construction of randomization-resistant ROP payloads. To that end, we have presented *instruction displacement*, a code diversification technique based on static binary instrumentation that (similarly to IPR) does not rely on complete code disassembly coverage. We have demonstrated that the proposed technique reduces the number of non-randomized gadgets from 15.04% for standalone in-place code randomization, to 2.77% for the combination of both techniques (or from 21.45% to 8.96%, when also considering the non-disassembled code regions), while it introduces a negligible performance overhead of 0.36%.

RIGHTSLINK
Copyright Clearance Center

As part of our future work, we plan to explore more aggressive combinations of instruction displacement and IPR, facilitated by the increased flexibility in altering existing code once it has been displaced, to further improve randomization entropy. We will also investigate other patching techniques or alterations that will reduce even further the currently small fraction of remaining non-randomized gadgets.

Availability

Our prototype implementation is publicly available at: https://github.com/kevinkoo001/ropf

Acknowledgments

This work was supported by the Office of Naval Research through contract N00014-15-1-2378. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the Office of Naval Research.

9. REFERENCES

[1] /ORDER (put functions in order). http://msdn.microsoft.com/en-us/library/00kh39zz.aspx.

[2] Orp: in-place binary code randomizer. http://nsl.cs.columbia.edu/projects/orp/.

[3] Profile-guided optimizations. http://msdn.microsoft.com/en-us/library/e7k32f4k.aspx.

[4] Syzygy - profile guided, post-link executable reordering. http://code.google.com/p/syzygy/wiki/SyzygyDesign.

[5] Wine. http://www.winehq.org.

[6] MWR Labs Pwn2Own 2013 Write-up - Webkit Exploit, 2013. https://labs.mwrinfosecurity.com/blog/mwr-labs-pwn2own-2013-write-up-webkit-exploit/.

[7] B. Antoniewicz. Analysis of a Malware ROP Chain, Oct. 2013. http://blog.opensecurityresearch.com/2013/10/analysis-of-malware-rop-chain.html.

[8] 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 *Proceedings of the 21st ACM Conference on Computer and year = 2014, Communications Security (CCS)*, pages 1342–1353.

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

[10] J. Bennett, Y. Lin, and T. Haq. The Number of the Beast, 2013. http://blog.fireeye.com/research/2013/02/the-number-of-the-beast.html.

[11] E. Bhatkar, D. C. Duvarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *In Proceedings of the 12th USENIX Security Symposium*, 2003.

[12] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, August 2005.

[13] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, pages 227–242, 2014.

[14] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi. Leakage-resilient layout randomization for mobile devices. In *Proceedings of the 2016 Network and Distributed System Security (NDSS) Symposium*, 2016.

[15] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Security Symposium*, pages 161–176, 2015.

[16] N. Carlini and D. Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *Proceedings of the 23rd USENIX Security Symposium*, pages 385–399, Aug. 2014.

[17] E. Carrera. pefile. https://github.com/erocarrera/pefile.

[18] F. B. Cohen. Operating system protection through program evolution. *Computers and Security*, 12:565–584, Oct. 1993.

[19] 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 *Proceedings of the 36th IEEE Symposium on Security and Privacy*, May 2015.

[20] 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 *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[21] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium*, pages 401–416, Aug. 2014.

[22] 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 *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 299–310, 2013.

[23] S. Designer. Getting around non-executable stack (and fix). http://seclists.org/bugtraq/1997/Aug/63.

[24] 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 *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, pages 901–913, 2015.

[25] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, 1997.

[26] G. Fresi Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)*, 2009.

[27] J. Gionta, W. Enck, and P. Ning. Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 325–336, 2015.

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

[29] Hex-Rays. IDA Pro Disassembler. http://www.hex-rays.com/idapro/.

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

[31] X. Hu, T.-c. Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and Communications Security (CCS)*, 2009.

[32] R. Johnson. A castle made of sand: Adobe Reader X sandbox. CanSecWest, 2011.

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

[34] V. Kotov. Dissecting the newest IE10 0-day exploit (CVE-2014-0322), Feb. 2014. http://labs.bromium.com/2014/02/25/dissecting-the-newest-ie10-0-day-exploit-cve-2014-0322/.

[35] S. Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. http://www.suse.de/~krahmer/no-nx.pdf.

[36] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. Sok: Automated software diversity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, pages 276–291, 2014.

[37] H. Li. Understanding and exploiting Flash ActionScript vulnerabilities. CanSecWest, 2011.

[38] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *Proceedings of the 5th European conference on Computer Systems (EuroSys)*, 2010.

[39] M. Miller, T. Burrell, and M. Howard. Mitigating software vulnerabilities, July 2011. http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=26788.

[40] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-Free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, 2010.

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

[42] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Security Symposium*, pages 447–462, August 2013.

[43] Parvez. Bypassing Microsoft Windows ASLR with a little help by MS-Help, Aug. 2012. http://www.greyhathacker.net/?p=585.

[44] PaX Team. Address space layout randomization. http://pax.grsecurity.net/docs/aslr.txt.

[45] M. Pietrek. An in-depth look into the Win32 portable executable file format, part 2, 1994. https://msdn.microsoft.com/en-us/library/ms809762.aspx.

[46] N. A. Quynh. Capstone: Next-gen disassembly framework. Black Hat USA, 2014.

[47] E. G. s, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proceedings of the 23rd USENIX Security Symposium*, pages 417–432, August 2014.

[48] J. Seibert, H. Okhravi, and E. Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, pages 54–65, 2014.

[49] F. J. Serna. CVE-2012-0769, the case of the perfect info leak, Feb. 2012. http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf.

[50] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and Communications security*, pages 552–61, October 2007.

[51] Skape. Locreate: An anagram for relocate. *Uninformed*, 6, 2007.

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

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

[54] A. Tang, S. Sethumadhavan, and S. Stolfo. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, pages 256–267, 2015.

[55] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pages 157–168, October 2012.

[56] 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*, pages 522–536, 2011.

[57] J. Werner, G. Baltas, R. Dallara, N. Otternes, K. Snow, F. Monrose, and M. Polychronakis. No-execute-after-read: Preventing code disclosure in commodity software. In *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security (ASIACCS)*, May 2016.

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

[59] D. A. D. Zovi. Practical return-oriented programming. SOURCE Boston, 2010.