Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization

Vasilis Pappas
Columbia University
vpappas@cs.columbia.edu

Michalis Polychronakis
Columbia University
mikepo@cs.columbia.edu

Angelos D. Keromytis
Columbia University
angelos@cs.columbia.edu

***Abstract*—The wide adoption of non-executable page protections in recent versions of popular operating systems has given rise to attacks that employ return-oriented programming (ROP) to achieve arbitrary code execution without the injection of any code. Existing defenses against ROP exploits either require source code or symbolic debugging information, or impose a significant runtime overhead, which limits their applicability for the protection of third-party applications.**

In this paper we present *in-place code randomization*, a practical mitigation technique against ROP attacks that can be applied directly on third-party software. Our method uses various narrow-scope code transformations that can be applied statically, without changing the location of basic blocks, allowing the safe randomization of stripped binaries even with partial disassembly coverage. These transformations effectively eliminate about 10%, and probabilistically break about 80% of the useful instruction sequences found in a large set of PE files. Since no additional code is inserted, in-place code randomization does not incur any measurable runtime overhead, enabling it to be easily used in tandem with existing exploit mitigations such as address space layout randomization. Our evaluation using publicly available ROP exploits and two ROP code generation toolkits demonstrates that our technique prevents the exploitation of the tested vulnerable Windows 7 applications, including Adobe Reader, as well as the automated construction of alternative ROP payloads that aim to circumvent in-place code randomization using solely any remaining unaffected instruction sequences.

I. Introduction

Attack prevention technologies based on the No eXecute (NX) memory page protection bit, which prevent the execution of malicious code that has been injected into a process, are now supported by most recent CPUs and operating systems [1]. The wide adoption of these protection mechanisms has given rise to a new exploitation technique, widely known as *return-oriented programming* (ROP) [2], which allows an attacker to circumvent non-executable page protections without injecting any code. Using return-oriented programming, the attacker can link together small fragments of code, known as *gadgets*, that already exist in the process image of the vulnerable application. Each gadget ends with an indirect control transfer instruction, which transfers control to the next gadget according to a sequence of gadget addresses injected on the stack or some other memory area. In essence, instead of injecting binary code, the attacker injects just data, which include the addresses of the gadgets to be executed, along with any required data arguments.

Several research works have demonstrated the great potential of this technique for bypassing defenses such as read-only memory [3], kernel code integrity protections [4], and non-executable memory implementations in mobile devices [5] and operating systems [6]–[9]. Consequently, it was only a matter of time for ROP to be employed in real-world attacks. Recent exploits against popular applications use ROP code to bypass exploit mitigations even in the latest OS versions, including Windows 7 SP1. ROP exploits are included in the most common exploit packs [10], [11], and are actively used in the wild for mounting drive-by download attacks.

Attackers are able to a priori pick the right code pieces because parts of the code image of the vulnerable application remain static across different installations. Address space layout randomization (ASLR) [1] is meant to prevent this kind of code reuse by randomizing the locations of the executable segments of a running process. However, in both Linux and Windows, parts of the address space do not change due to executables with fixed load addresses [12], or shared libraries incompatible with ASLR [6]. Furthermore, in some exploits, the base address of a DLL can be either calculated dynamically through a leaked pointer [9], [13], or brute-forced [14].

Other defenses against code-reuse attacks complementary to ASLR include compiler extensions [15], [16], code randomization [17]–[19], control-flow integrity [20], and runtime solutions [21]–[23]. In practice, though, most of these approaches are almost never applied for the protection of the COTS software currently targeted by ROP attacks, either due to the lack of source code or debugging information, or due to their increased overhead. In particular, from the above techniques, those that operate directly on compiled binaries, e.g., by permuting the order of functions [18], [19] or through binary instrumentation [20], require precise and complete extraction of all code and data in the executable sections of the binary. This is possible only if the corresponding symbolic debugging information is available, which however is typically stripped from production binaries. On the other hand, techniques that do work on stripped binary executables using dynamic binary instrumentation [21]–[23], incur a significant runtime overhead that limits their adoption. At the same time, instruction set randomization (ISR) [24], [25] cannot prevent code-reuse attacks, and current implementations also rely on heavyweight runtime instrumentation or code emulation frameworks.

Starting with the goal of a practical mitigation against the recent spate of ROP attacks, in this paper we present a novel code randomization method that can harden third-party applications against return-oriented programming. Our

approach is based on narrow-scope modifications in the code segments of executables using an array of code transformation techniques, to which we collectively refer as *in-place code randomization*. These transformations are applied statically, in a conservative manner, and modify only the code that can be safely extracted from compiled binaries, without relying on symbolic debugging information. By preserving the length of instructions and basic blocks, these modifications do not break the semantics of the code, and enable the randomization of stripped binaries even without complete disassembly coverage. The goal of this randomization process is to eliminate or probabilistically modify as many of the gadgets that are available in the address space of a vulnerable process as possible. Since ROP code relies on the correct execution of all chained gadgets, altering the outcome of even a few of them will likely render the ROP code ineffective.

Our evaluation using real-world ROP exploits against widely used applications, such as Adobe Reader, shows the effectiveness and practicality of our approach, as in all cases the randomized versions of the applications rendered the exploits non-functional. When aiming to circumvent the applied code randomization, Q [26] and Mona [27], two automated ROP payload construction tools, were unable to generate functional exploit code by relying solely on any remaining non-randomized gadgets.

Although quite effective as a standalone mitigation, in-place code randomization is not meant to be a complete prevention solution, as it offers probabilistic protection and thus cannot deliver any protection guarantees. However, it can be applied in tandem with existing randomization techniques to increase process diversification. This is facilitated by the practically zero overhead of the applied transformations, and the ease with which they can be applied on existing third-party executables.

Our work makes the following main contributions:

- We present in-place code randomization, a novel and practical approach for hardening third-party software against ROP attacks. We describe in detail various narrow-scope code transformations that do not change the semantics of existing code, and which can be safely applied on compiled binaries without symbolic debugging information.
- We have implemented in-place code randomization for x86 PE executables, and have experimentally verified the safety of the applied code transformations with extensive runtime code coverage tests using third-party executables.
- We provide a detailed analysis of how in-place code randomization affects available gadgets using a large set of 5,235 PE files. On average, the applied transformations effectively eliminate about 10%, and probabilistically break about 80% of the gadgets in the tested files.
- We evaluate our approach using publicly available ROP exploits and generic ROP payloads, as well as two ROP payload construction toolkits. In all cases, the randomized versions of the executables break the malicious ROP code, and prevent the automated construction of alternative payloads using the remaining unaffected gadgets.

II. Background

The introduction of non-executable memory page protections led to the development of the return-to-libc exploitation technique [28]. Using this method, a memory corruption vulnerability can be exploited by transferring control to code that already exists in the address space of the vulnerable process. By jumping to the beginning of a library function such as `system()`, the attacker can for example spawn a shell without the need to inject any code. Frequently though, especially for remote exploitation, calling a single function is not enough. In these cases, multiple return-to-libc calls can be "chained" together by first returning to a short instruction sequence such as `pop reg; pop reg; ret;` [29], [30]. When arguments need to be passed through registers, a few short instruction sequences ending with a `ret` instruction can be chained directly to set the proper registers with the desired arguments, before calling the library function [31].

In the above code-reuse techniques, the executed code consists of one or a few short instruction sequences followed by a large block of code belonging to a library function. Hovav Shacham demonstrated that using only a carefully selected set of short instruction sequences ending with a `ret` instruction, known as *gadgets*, it is possible to achieve arbitrary computation, obviating the need for calling library functions [2]. This powerful technique, dubbed *return-oriented programming*, in essence gives the attacker the same level of flexibility offered by arbitrary code injection without injecting any code at all—the injected payload comprises just a sequence of gadget addresses intermixed with any necessary data arguments.

In a typical ROP exploit, the attacker needs to control both the program counter and the stack pointer: the former for executing the first gadget, and the latter for allowing its `ret` instruction to transfer control to subsequent gadgets. Depending on the vulnerability, if the ROP payload is injected in a memory area other than the stack, then the stack pointer must first be adjusted to the beginning of the payload through a stack pivot [6], [32]. In a follow up work [33], Checkoway et al. demonstrated that the gadgets used in a ROP exploit need not necessarily end with a `ret` instruction, but with any other indirect control transfer instruction.

The ROP code used in recent exploits against Windows applications is mostly based on gadgets ending with `ret` instructions, which conveniently manipulate both the program counter and the stack pointer, although a couple of gadgets ending with `call` or `jmp` are also used for calling library functions. In all publicly available Windows exploits so far, attackers do not have to rely on a fully ROP-based implementation for the whole malicious code that needs to be executed. Instead, ROP code is used only as a first stage for bypassing DEP [1]. Typically, once control flow has been hijacked, the ROP code allocates a memory area with write and execute permissions by calling a library function like `VirtualAlloc`, copies into it some plain shellcode included in the attack vector, and finally jumps to the copied shellcode which now has execute permission [32].

III. Approach

Our approach is based on the randomization of the code sections of binary executable files that are part of third-party applications, using an array of binary code transformation techniques. The objective of this randomization process is to break the code semantics of the gadgets that are present in the executable memory segments of a running process, without affecting the semantics of the actual program code.

The execution of a gadget has a certain set of consequences to the CPU and memory state of the exploited process. The attacker chooses how to link the different gadgets together based on which registers, flags, or memory locations each gadget modifies, and in what way. Consequently, the execution of a subsequent gadget depends on the outcome of all previously executed gadgets. Even if the execution of a single gadget has a different outcome than the one anticipated by the attacker, then this will affect the execution of all subsequent gadgets, and it is likely that the logic of the malicious return-oriented code will be severely impacted.

A. Why In-Place?

The concept of software diversification [34] is the basis for a wide range of protections against the exploitation of memory corruption vulnerabilities. Besides address space layout randomization [1], many techniques focus on the internal randomization of the code segments of executable, and can be combined with ASLR to increase process diversity [17]. Metamorphic transformations [35] can shift gadgets from their original offsets and alter many of their instructions, rendering them unusable. Another simpler and probably more effective approach is to rearrange existing blocks of code either at the function level [18], [19], [36], [37], or with finer granularity, at the basic block level [38], [39]. If all blocks of code are reordered so that no one resides at its original location, then all the offsets of the gadgets that the attacker would assume to be present in the code sections of the process will now correspond to completely different code.

These transformations require a precise view of all the code and data objects contained in the executable sections of a PE file, including their cross-references, as existing code needs to be shifted or moved. Due to computed jumps and intermixed data [40], complete disassembly coverage is possible only if the binary contains relocation and symbolic debugging information (e.g., PDB files) [19], [41], [42]. Unfortunately, debugging information is typically stripped from release builds for compactness and intellectual property protection.

For Windows software, in particular, PE files (both DLL and EXE) usually do retain relocation information even if no debugging information has been retained [43]. The loader needs this information in case a DLL must be loaded at an address other than its preferred base address, e.g., because another library has already been mapped to that location. or for ASLR. In contrast to Linux shared libraries and PIC executables, which contain position-independent code, Windows binaries contain absolute addresses, e.g., as immediate instruction operands or initialized data pointers, that are valid only if the executable has been loaded at its preferred base address. The `.reloc` section of PE files contains a list of offsets relatively to each PE section that correspond to all absolute addresses at which a delta value needs to be added in case the actual load address is different [44].

Relocation information *alone*, however, does not suffice for extracting a complete view of the code within the executable sections of a PE file [38], [41]. Without the symbolic debugging information contained in PDB files, although the location of objects that are reached *only* via indirect jumps *can* be extracted from relocation information, their actual type—code or data—still remains unknown. In some cases, the actual type of these objects could be inferred using heuristics based on constant propagation, but such methods are usually prone to misidentifications of data as code and vice versa. Even a slight shift or size increase of a single object within a PE section will incur cascading shifts to its following objects. Typically, an unidentified object that actually contains code will include PC-relative branches to other code objects. In the absence of the debugging information contained in PDB files, moving such an unidentified code block (or any of its relatively referenced objects) without fixing the displacements of all its relative branch instructions that reference other objects, will result to incorrect code.

Given the above constraints, we choose to use only binary code transformations that do not alter the size and location of code and data objects within the executable, allowing the randomization of third-party PE files *without* symbolic debugging information. Although this restriction does not allow us to apply extensive code transformations like basic block reordering or metamorphism, we can still achieve partial code randomization using narrow-scope modifications that can be *safely* applied even without complete disassembly coverage. This can be achieved through slight, in-place code modifications to the correctly identified parts of the code, that do not change the overall structure of basic blocks or functions, but which are enough to alter the outcome of short instruction sequences that can be used as gadgets.

B. Code Extraction and Modification

Although completely accurate disassembly of stripped x86 binaries is not possible, state-of-the-art disassemblers achieve decent coverage for code generated by the most commonly used compilers, using a combination of different disassembly algorithms [40], the identification of specific code constructs [45], and simple data flow analysis [46]. For our prototype implementation, we use IDA Pro [47] to extract the code and identify the functions of PE executables. IDA Pro is effective in the identification of function boundaries, even for functions with non-contiguous code and extensive use of basic block sharing [48], and also takes advantage of the relocation information present in Windows DLLs.

Typically, however, without the symbolic information of PDB files, a fraction of the functions in a PE executable are not identified, and parts of code remain undiscovered. Our code transformations are applied conservatively, only

on parts of the code for which we can be confident that have been accurately disassembled. For instance, IDA Pro speculatively disassembles code blocks that are reached only through computed jumps, taking advantage of the relocation information contained in PE files. However, we do not enable such heuristic code extraction methods in order to avoid any disastrous modifications due to potentially misidentified code. In practice, for the code generated by most compilers, relocation information also ensures that the correctly identified basic blocks have no entry point other than their first instruction. Similarly, some transformations that rely on the proper identification of functions are applied only on the code of correctly recognized functions. Our implementation is separate from the actual code extraction framework used, which means that IDA Pro can be replaced or assisted by alternative code extraction approaches [41], [49], [50], providing better disassembly coverage.

After code extraction, disassembled instructions are first converted to our own internal representation, which holds additional information such as any implicitly used registers, and the registers and flags read or written by the instruction. For correctness, we also track the use of general purpose registers even in floating point, MMX, and SSE instructions. Although these type of instructions have their own set of registers, they do use general purpose registers for memory references (e.g., as the `fmul` instruction in Fig. 1). We then proceed and apply the in-place code transformations discussed in the following section. These are applied only on the parts of the executable segments that contain (intended or unintended [2]) instruction sequences that can be used as gadgets. As a result of some of the transformations, instructions may be moved from their original locations within the same basic block. In these cases, for instructions that contain an absolute address in some of their operands, the corresponding entries in the `.reloc` sections of the randomized PE file are updated with the new offsets where these absolute addresses are now located.

Our prototype implementation processes each PE file individually, and generates multiple randomized copies that can then replace the original. Given the complexity of the analysis required for generating a set of randomized instances of an input file (in the order of a few minutes on average for the PEs used in our tests), this allows the off-line generation of a pool of randomized PE files for a given application. Note that for most of the tested Windows applications, only some of the DLLs need to be randomized, as the rest are usually ASLR-enabled (although they can also be randomized for increased protection). In a production deployment, a system service or a modified loader can then pick a different randomized version of the required PEs each time the application is launched, following the same way of operation as tools like EMET [51].

IV. In-Place Code Transformations

In this section we present in detail the different code transformations used for in-place code randomization. Although some of the transformations such as instruction reordering and register reassignment are also used by compilers and polymorphic code engines for code optimization [52] and obfuscation [35], applying them at the binary level—without having access to the higher-level structural and semantic information available in these settings—poses significant challenges.

A. Atomic Instruction Substitution

One of the basic concepts of code obfuscation and metamorphism [35] is that the exact same computation can be achieved using a countless number of different instruction combinations. When applied for code randomization, substituting the instructions of a gadget with a functionally-equivalent—but different—sequence of instructions would not affect any ROP code that uses that gadget, since its outcome would be the same. However, by modifying the instructions of the original program code, this transformation in essence modifies certain bytes in the code image of the program, and consequently, can drastically alter the structure of non-intended instruction sequences that overlap with the substituted instructions.

Many of the gadgets used in ROP code consist of unaligned instructions that have not been emitted by the compiler, but which happen to be present in the code image of the process due to the density and variable-length nature of the x86 instruction set. In the example of Fig. 1(a), the actual code generated by the compiler consists of the instructions `mov; cmp; lea;` starting at byte `B0`.[1] However, when disassembling from the next byte, a useful non-intended gadget ending with `ret` is found.

Compiled code is highly optimized, and thus the replacement of even a single instruction in the original program code usually requires either a longer instruction, or a combination of more than one instruction, for achieving the same purpose. Given that our aim is to randomize the code of stripped binaries, even a slight increase in the size of a basic block is not possible, which makes the most commonly used instruction substitution techniques unsuitable for our purpose.

In certain cases though, it is possible to replace an instruction with a single, functionally-equivalent instruction of the *same* length, thanks to the flexibility offered by the extensive x86 instruction set. Besides obvious candidates based on replacing addition with negative subtraction and inversely, there are also some instructions that come in different forms, with different opcodes, depending on the supported operand types. For example, `add r/m32,r32` stores the result of the addition in a register *or* memory operand (r/m32), while `add r32,r/m32` stores the result in a register (r32). Although these two forms have different opcodes, the two instructions are equivalent when both operands happen to be registers. Many arithmetic and logical instructions have such dual equivalent forms, while in some cases there can be up to five equivalent instructions (e.g., `test r/m8,r8`, `or r/m8,r8`, `or r8, r/m8`, `and r/m8,r8`, `and r8,r/m8`, affect the flags of the EFLAGS register in the same way when both operands are

[1] The code of all examples throughout the paper comes from icucnv36.dll, included in Adobe Reader v9.3.4. This DLL was used for the ROP code of a DEP-bypass exploit for CVE-2010-2883 [53] (see Table II).

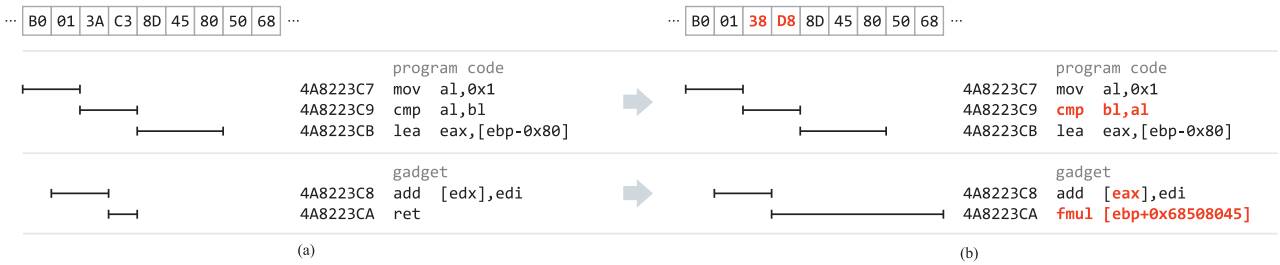

Figure 1. Example of atomic instruction substitution. The equivalent, but different form of the `cmp` instruction does not change the original program code (a), but renders the non-intended gadget unusable (b).

the *same* register). In our prototype implementation we use the sets of equivalent instructions used in Hydan [54], a tool for hiding information in x86 executables, with the addition of one more set that includes the equivalent versions of the `xchg` instruction.

As shown in Fig. 1(b), both operands of the `cmp` instruction are registers, and thus it can be replaced by its equivalent form, which has different opcode and ModR/M bytes [55]. Although the actual program code does not change, the `ret` instruction that was "included" in the original `cmp` instruction has now disappeared, rendering the gadget unusable. In this case, the transformation completely *eliminates* the gadget, and thus will be applied in all instances of the randomized binary. In contrast, when a substitution does not affect the gadget's final indirect jump, then it is applied probabilistically.

B. Instruction Reordering

In certain cases, it is possible to reorder the instructions of small self-contained code fragments without affecting the correct operation of the program. This transformation can significantly impact the structure of non-intended gadgets, but can also break the attacker's assumptions about gadgets that are part of the actual machine code.

1) Intra Basic Block Reordering: The actual instruction scheduling chosen during the code generation phase of a compiler depends on many factors, including the cost of instructions in cycles, and the applied code optimization techniques [52]. Consequently, the code of a basic block is often just one among several possible instruction orderings that are all equivalent in terms of correctness. Based on this observation, we can partially modify the code within a basic block by reordering some of its instructions according to an alternative instruction scheduling.

The basis for deriving an alternative instruction scheduling is to determine the ordering relationships among the instructions, which must always be satisfied to maintain code correctness. The *dependence graph* of a basic block represents the instruction interdependencies that constrain the possible instruction schedules [56]. Since a basic block contains straight-line code, its dependence graph is a directed acyclic graph with machine instructions as vertices, and dependencies between instructions as edges. We apply dependence analysis on the code of disassembled basic blocks to build their dependence graph using an adaptation of a standard dependence DAG construction algorithm [56, Fig. 9.6] for machine code. Applying dependence analysis directly on machine code requires a careful treatment of the dependencies between x86 instructions. Compared to the analysis of code expressed in an intermediate representation form, this includes the identification of data dependencies not only between register and memory operands, but also between CPU flags and implicitly used registers and memory locations.

For each instruction i, we derive the sets $use[i]$ and $def[i]$ with the registers used and defined by the instruction. Besides register operands and registers used as part of effective address computations, this includes any implicitly used registers. For example, the use and def sets for `pop eax` are $\{esp\}$ and $\{eax,\ esp\}$, while for `rep stosb`[2] are $\{ecx, eax, edi\}$ and $\{ecx,\ edi\}$, respectively. We initially assume that all instructions in the basic block depend on each other, and then check each pair for read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) dependencies. For example, i_1 and i_2 have a RAW dependency if any of the following conditions is true: i) $def[i_1] \cap use[i_2] \neq \emptyset$, ii) the destination operand of i_1 and the source operand of i_2 are both a memory location, iii) i_1 writes at least one flag read by i_2.

Note that condition ii) is quite conservative, given that i_2 will actually depend on i_1 only if i_2 reads the *same* memory location written by i_1. However, unless both memory operands use absolute addresses, it is hard to determine statically if the two effective addresses point to the same memory location. In our future work, we plan to use simple data flow analysis to relax this condition. Besides instructions with memory operands, this condition should also be checked for instructions with implicitly accessed memory locations, e.g., `push` and `pop`. The conditions for WAR and WAW dependencies are analogous. If no conflict is found between two instructions, then there is no constraint in their execution order.

Figure 2(a) shows the code of a basic block that contains a non-intended gadget, and Fig. 3 its corresponding dependence DAG. Instructions not connected via a direct edge are independent, and have no constraint in their relative execution order. Given the dependence DAG of a basic block, the possible orderings of its instructions correspond to the different

[2] `stosb` (Store Byte to String) copies the least significant byte from the `eax` register to the memory location pointed by the `edi` register and increments `edi`'s value by one. The `rep` prefix repeats this instruction until `ecx`'s value reaches zero, while decreasing it after each repetition.

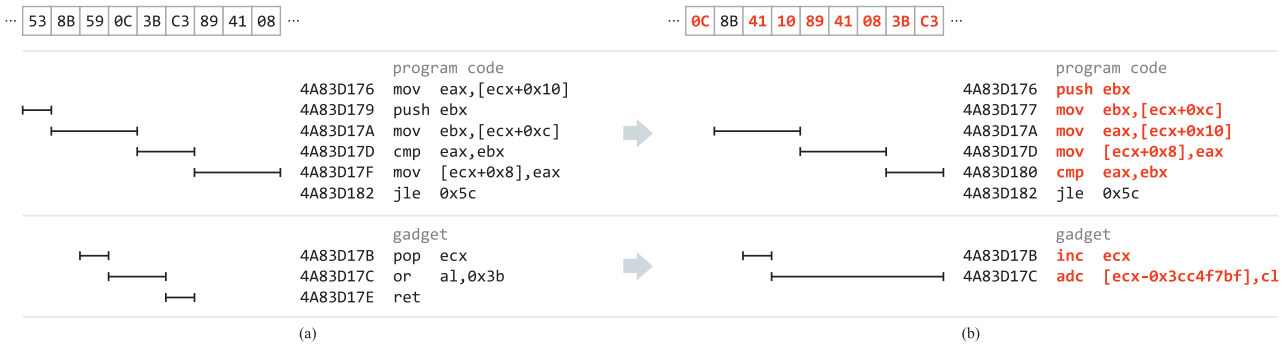

Figure 2. Example of how intra basic block instruction reordering can affect a non-intended gadget.

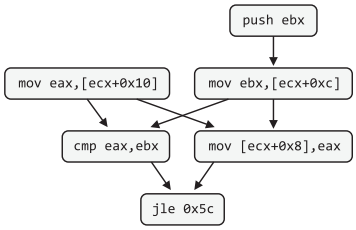

Figure 3. Dependence graph for the code of Fig. 2.

```
4A834B3B   0  push ebx              4A834B3B  push edi
4A834B3C  -4  push esi              4A834B3C  push ebx
4A834B3D  -8  mov  ebx,ecx          4A834B3D  push esi
4A834B3F  -8  push edi              4A834B3E  mov  ebx,ecx
4A834B40  -C  mov  esi,edx          4A834B40  mov  esi,edx
...                                 ...
4A834B7C  -C  pop  edi              4A834B7C  pop  esi
4A834B7D  -8  pop  esi              4A834B7D  pop  ebx
4A834B7E  -4  pop  ebx              4A834B7E  pop  edi
4A834B7F   0  ret                   4A834B7F  ret
```

(a) (b)

Figure 4. Example of register preservation code reordering.

topological sorting arrangements of the graph [57]. Fig. 2(b) shows one of the possible alternative orderings of the original code. The locations of all but one of the instructions and the values of all but one of the bytes have changed, eliminating the non-intended gadget contained in the original code. Although a new gadget has appeared a few bytes further into the block, (ending again with a `ret` instruction at byte `C3`), an attacker cannot depend on it since alternative orderings will shift it to other locations, and some of its internal instructions will always change (e.g., in this example, the useful `pop ecx` is gone). In fact, the `ret` instruction can be eliminated altogether using atomic instruction substitution.

An underlying assumption we make here is that basic block boundaries will not change at runtime. If a computed control transfer instruction targets a basic block instruction other than its first, then reordering may break the semantics of the code. Although this may seem restrictive, we note that throughout our evaluation we did not encounter any such case. For compiler-generated code, IDA Pro is able to compute all jump targets even for computed jumps based on the PE relocation information. In the most conservative case, users may choose to disable instruction reordering and still benefit from the randomization of the other techniques—Section V includes results for each technique individually.

2) Reordering of Register Preservation Code: The calling convention followed by the majority of compilers for Windows on x86 architectures, similarly to Linux, specifies that the `ebx`, `esi`, `edi`, and `ebp` registers are callee-saved [58]. The remaining general purpose registers, known as scratch or volatile registers, are free for use by the callee without restrictions. Typically, a function that needs to use more than the available scratch registers, preserves any non-volatile registers before modifying them by storing their values on the stack. This is usually done at the function prologue through a series of `push` instructions, as in the example of Fig. 4(a), which shows the very first and last instructions of a function. At the function epilogue, a corresponding series of `pop` instructions restores the saved values from the stack, right before returning to the caller. Sequences that contain `pop` instructions followed by `ret` are among the most widely used gadgets found in ROP exploits, since they allow the attacker to load registers with values that are supplied as part of the injected payload [59]. The order of the `pop` instructions is crucial for initializing each register with the appropriate value.

As seen in the function prologue, the compiler stores the values of the callee-saved registers in arbitrary order, and sometimes the relevant `push` instructions are interleaved with instructions that use previously-preserved registers. At the function epilogue, the saved values are `pop`'ed from the stack in *reverse* order, so that they end up to the proper register. Consequently, as long as the saved values are restored in the right order, their actual order on the stack is irrelevant. Based on this observation, we can randomize the order of the `push` and `pop` instructions of register preservation code by maintaining the first-in-last-out order of the stored values, as shown in Fig. 4(b). In this example, there are six possible orderings of the three `pop` instructions, which means that any assumption that the attacker may make about which registers will hold the two supplied values, will be correct with a probability of one in six (or one in three, if only one register needs to be initialized). In case only two registers are preserved, there are two possible orderings, allowing the gadget to operate correctly half of the time.

This transformation is applied conservatively, only to func-

tions with accurately disassembled prologue and epilogue code. To make sure that we properly match the `push` and `pop` instructions that preserve a given register, we monitor the stack pointer delta throughout the whole function, as shown in the second column of Fig. 4(a). If the deltas at the prologue and epilogue do not match, e.g., due to call sites with unknown calling conventions throughout the function, or indirect manipulation of the stack pointer, then no randomization is applied. As shown in Fig. 4(b), any non-preservation instructions in the function prologue are reordered along with the `push` instructions by maintaining any interdependencies, as discussed in the previous section. For functions with multiple exit points, the preservation code at all epilogues should match the function's prologue. Note that there can be multiple `push` and `pop` pairs for the same register, in case the register is preserved only throughout some of the execution paths of a function.

C. Register Reassignment

Although the program points at which a certain variable should be stored in a register or spilled into memory are chosen by the compiler using sophisticated allocation algorithms, the actual name of the general purpose register that will hold a particular variable is mostly an arbitrary choice. Based on this observation, we can reassign the names of the register operands in the existing code according to a different—but equivalent—register assignment, without affecting the semantics of the original code. When considering each gadget as an autonomous code sequence, this transformation can alter the outcome of many gadgets, which will now read or modify different registers than those assumed by the attacker.

Due to the much higher cost of memory accesses compared to register accesses, compilers strive to map as many variables as possible to the available registers. Consequently, at any point in a large program, multiple registers are usually in use, or *live* at the same time. Given the control flow graph (CFG) of a compiled program, a register r is *live* at a program point p iff there is a path from p to a use of r that does not go through a definition of r. The *live range* of r is defined as the set of program points where r is live, and can be represented as a subgraph of the CFG [60]. Since the same register can hold different variables at different points in the program, a register can have multiple disjoint live regions in the same CFG.

For each correctly identified function, we compute the live ranges of all registers used in its body by performing liveness analysis [52] directly on the machine code. Given the CFG of the function and the sets $use[i]$ and $def[i]$ for each instruction i, we derive the sets $in[i]$ and $out[i]$ with the registers that are *live-in* and *live-out* at each instruction. For this purpose, we use a modified version of a standard live-variable analysis algorithm [52, Fig. 9.16] that computes the in and out sets at the instruction level, instead of the basic block level. The algorithm computes the two sets by iteratively reaching a fixed point for the following data-flow equations: $in[i] = use[i] \cup (out[i] - def[i])$ and $out[i] = \bigcup\{in[s] : s \in succ[i]\}$, were $succ[i]$ is the set of all possible successors of instruction i.

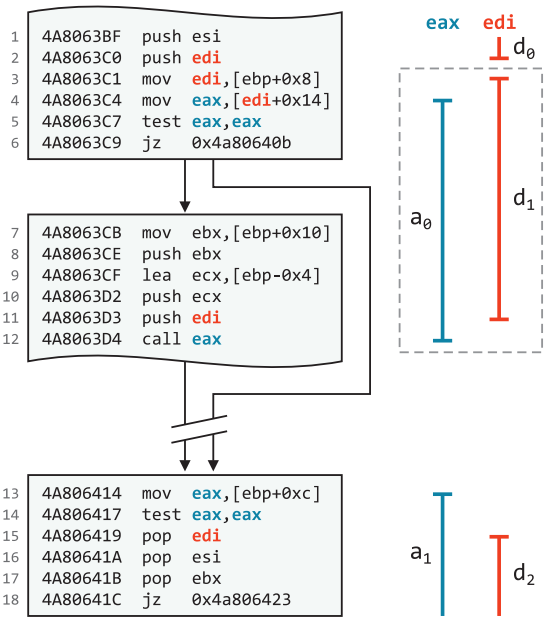

Figure 5. The live ranges of `eax` and `edi` in part of a function. The two registers can be swapped in all instructions throughout their parallel, self-contained regions a_0 and d_1 (lines 3–12).

Figure 5 shows part of the CFG of a function and the corresponding live ranges for `eax` and `edi`. Initially, we assume that all registers are live, since some of them may hold values that have been set by the caller. In this example, `edi` is live when entering the function, and the `push` instruction at line 2 stores (uses) its current value on the stack. The following `mov` instruction initializes (defines) `edi`, ending its previous live range (d_0). Note that although a live range is a sub-graph of the CFG, we illustrate and refer to the different live ranges as linear regions for the sake of convenience.

The next definition of `edi` is at line 15, which means that the last use of its previous value at line 11 also ends its previous live region d_1. Region d_1 is a *self-contained* region, within which we can be confident that `edi` holds the same variable. The `eax` register also has a self-contained live region (a_0) that runs in parallel with d_1. Conceptually, the two live ranges can be extended to share the same boundaries. Therefore, the two registers can be swapped across all the instructions located within the boundaries of the two regions, without altering the semantics of the code.

The `call eax` instruction at line 12 can be conveniently used by an attacker for calling a library function or another gadget. By reassigning `eax` and `edi` across their parallel live regions, any ROP code that would depend on `eax` for transferring control to the next piece of code, will now jump to an incorrect memory location, and probably crash. For code fragments with just two parallel live regions, an attacker can guess the right register half of the times. In many cases though, there are three or more general purpose registers with parallel live regions, or other available registers that are live before or after another register's live region, allowing for a higher number of possible register assignments.

The registers used in the original code can be reassigned by modifying the ModR/M and sometimes the SIB byte of the relevant instructions. As in previous code transformations, besides altering the operands of instructions in the existing code, these modifications can also affect overlapping instructions that may be part of non-intended gadgets. Note that implicitly used registers in certain instructions cannot be replaced. For example, the one-byte "move data from string to string" instruction (`movs`) always uses `esi` and `edi` as its source and destination operands, and there is no other one-byte instruction for achieving the same operation using a different set of registers [55]. Consequently, if such an instruction is part of the live region of one of its implicitly used registers, then this register cannot be reassigned throughout that region. For the same reason, we exclude `esp` from liveness analysis. Finally, although calling conventions are followed for most of the functions, this is not always the case, as compilers are free to use any custom calling convention for private or static functions. Most of these cases are conservatively covered through a bottom-up call analysis that discovers custom register arguments and return value registers.

First, all the external function definitions found in the import table of the DLL are marked as level-0 functions. IDA Pro can effectively distinguish between different calling conventions that these external functions may follow, and reports their declaration in the C language. Thus, in most cases, the register arguments and the return value register (if any) for each of the level-0 functions are known. For any `call` instruction to a level-0 function, its register arguments are added to `call`'s set of implicitly read registers, and its return value registers are added to `call`'s set of implicitly written registers.

In the next phase, level-1 functions are identified as the set of functions that call only level-0 functions or no other function. Any registers read by a level-1 function, without prior writing them, are marked as its register arguments. Similarly, any registers written and not read before a return instruction are marked as return value registers. Again, the sets of implicitly read and written register of all the `call` instructions to level-1 functions are updated accordingly. Similarly, level-2 functions are the ones that call level-1 or level-0 functions, or no other function, and so on. The same process is repeated until no more function levels can be computed. The intuition behind this approach is that private functions, which may use non-standard calling conventions, are called by other functions in the same DLL and, in most cases, not through computed call instructions.

V. Experimental Evaluation

A. Randomization Analysis

1) Coverage: A crucial aspect for the effectiveness of in-place code randomization is the randomization coverage in terms of what percentage of the gadgets found in an executable can be safely randomized. A gadget may remain intact for one of the following reasons: i) it is part of data embedded in a code segment, ii) it is part of code that could not be disassembled, or iii) it is not affected by any of our transformations. In this section, we explore the randomization coverage of our prototype implementation using a large data set of 5,235 PE files (both DLL and EXE), detailed in Table I.

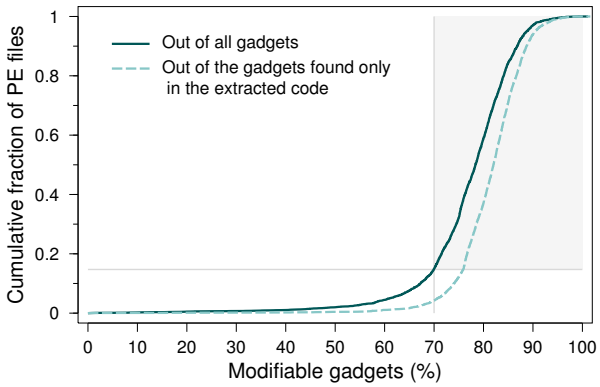

Figure 6. Percentage of modifiable gadgets for a set of 5,235 PE files. Indicatively, for the upper 85% of the files, more than 70% of *all* gadgets in the executable segments of each PE file can be modified (shaded area).

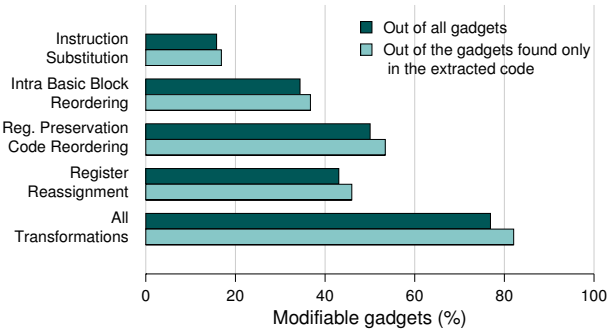

Figure 7. Percentage of modifiable gadgets according to the different code transformations.

We consider as a gadget [2] any intended or unintended instruction sequence that ends with an indirect control transfer instruction, and which does not contain i) a privileged or invalid instruction (can occur in non-intended instruction sequences), and ii) a control transfer instruction other than its final one, with the exception of indirect `call` (can be used in the middle of a gadget for calling a library function). We assume a maximum gadget length of five instructions, which is typical for existing ROP code implementations [2], [33]. For larger gadgets, it is possible that the modified part of the gadget may be irrelevant for the purpose of the attacker. For example, if only the first instruction of the gadget `inc eax; pop ebx; ret;` is randomized, this will not affect any ROP code that either does not rely on the value of `eax` at that point, or uses the shorter gadget `pop ebx; ret;` directly. For this reason, we consider *all* different subsequences with length between two to five instructions as separate gadgets.

Figure 6 shows the percentage of modifiable gadgets out of *all* gadgets found in the executable sections of each PE file (solid line), as a cumulative fraction of all PE files in the data set. In about 85% of the PE files, more that 70% of the gadgets can be randomized by our code transformations. Many

Table I
MODIFIABLE (ELIMINATED VS. BROKEN) GADGETS FOR A COLLECTION OF VARIOUS PE FILES.

Software	PE Files	Code (MB)	Total	Modifiable (%)	Eliminated (%)	Broken (%)
Adobe Reader 9	43	6.7	1,250,959	943,506 (75.4)	108,614 (8.7)	834,892 (66.7)
Firefox 4	28	3.5	458,760	381,011 (83.0)	56,800 (12.4)	324,211 (70.6)
iTunes 10	75	3.7	396,478	293,392 (74.0)	31,779 (8.0)	261,613 (66.0)
Windows XP SP3	1,698	134.4	8,305,177	6,452,895 (77.7)	770,589 (9.3)	5,682,306 (68.4)
Windows 7 SP1	3,391	324.8	16,951,300	12,970,844 (76.5)	1,637,082 (9.7)	11,333,762 (66.8)
Total	5,235	473.1	27,362,674	21,041,648 (**76.9**)	2,604,864 (**9.5**)	18,436,784 (**67.4**)

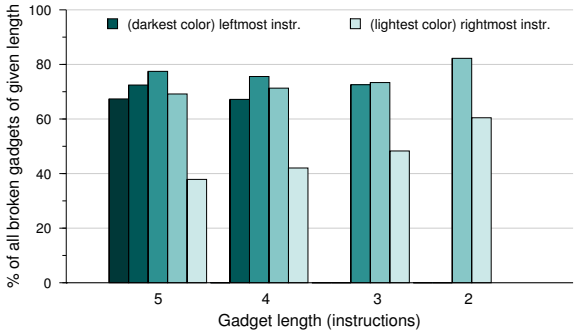

Figure 8. Impact of code randomization on the broken gadgets' instructions according to their location in the gadget. The order of the bars corresponds to the order of the instructions in the gadget. Indicatively, the first (leftmost) instruction of two-instruction gadgets is altered in more than 80% of all broken two-instruction gadgets.

of the unmodified gadgets are located in parts of code that have not been extracted by IDA Pro, and which consequently will never be affected by our transformations. When considering only the gadgets that are contained within the disassembled code regions on which code randomization can be applied, the percentage of affected gadgets slightly increases (dashed line). Given that we do not take into account code blocks that have been identified by IDA Pro using speculative methods, this shows that the use of a more sophisticated code extraction mechanism will increase the number of gadgets that can be modified. Figure 7 shows the total percentage of gadgets modified by each code transformation technique for the same data set. Note that a gadget can be modified by more than one technique. Overall, the total percentage of modifiable gadgets across all PE files is about 76.9%, as shown in Table I.

2) Impact: We identify two qualitatively different ways in which a code transformation can impact a gadget. As discussed in Sec. IV-A, a gadget can be *eliminated*, if any of the applied transformations removes completely its final control transfer instruction. If the final control transfer instruction remains intact, a gadget can then be *broken*, if at least one of its internal instructions is altered, and the CPU and memory state after its execution is different than the original, i.e., the outcome of its computation is not the same. As shown in Table I, in the average case, about 9.5% of *all* gadgets contained in a PE file can be rendered completely unusable. For a vulnerable application, this already removes about one in ten of the available gadgets for the construction of ROP code. Although the rest of the modifiable gadgets (67.4%) is not eliminated, they can be "broken" by probabilistically modifying one or more of their instructions.

In case some of the instructions in a broken gadget can never be altered, it is quite possible that part of its functionality will remain unaffected, and thus an attacker could still use it by relying only on its unmodifiable instructions. Especially for larger gadget sizes, if the possible modifications are clustered only around a certain part of the gadget, e.g., its first instructions, then an attacker could predictably rely on the rest of the gadget. We explore this issue by measuring the number of broken gadgets in which an instruction at a given position can be altered.

Figure 8 shows the impact of code randomization on a broken gadget's instructions, according to their location within the gadget. Each group of bars corresponds to a different gadget length, and in each group, the leftmost bar corresponds to the leftmost instruction of the gadget. For all sizes, the probability that an instruction at a given position will be affected is quite evenly distributed and remains beyond 60%, with the exception of the final (control transfer) instruction. This is expected, since most of the transformations cannot affect the final instruction of intended gadgets (e.g., `ret`). As we observe, the locations of the modified instructions in broken gadgets are almost equally unpredictable.

3) Entropy: Some of the code transformations can perturb a given instruction within a gadget only in a limited number of ways, while others can generate a larger number of permutations. For example, for instructions with only two equivalent forms, atomic instruction substitution can modify a particular location in a gadget only in one way, allowing for two possible states. On the other hand, intra basic block instruction reordering usually results to a large number of possible permutations, especially for larger basic blocks that contain many instructions with no interdependencies.

Usually though, a broken gadget can be modified at multiple locations, and the same location can be altered in multiple ways by more than one code transformations. Consequently, the number of possible randomized states in which a broken gadget can exist, or its randomization *entropy*, corresponds to the product of the number of permutations that each of the different transformations can generate for that gadget. In the worst case, a broken gadget can exist in two possible states: its original form, or its alternative after modification. For example, there are only two possible orderings for the `pop` instructions in an intended gadget of the form `pop reg; pop reg; ret;` given that no other transformation can alter it.

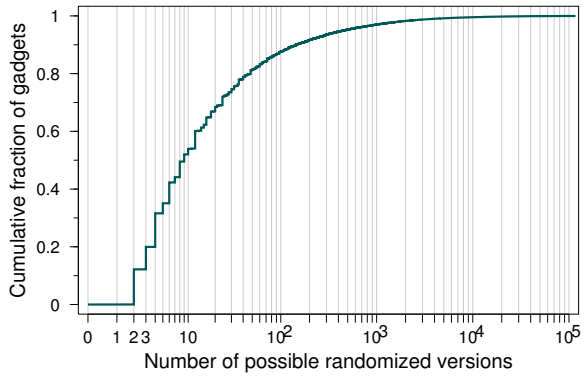

Figure 9. Randomization entropy for broken gadgets.

Figure 9 shows the number of possible randomized versions of each gadget (including its original), as a cumulative fraction of all broken gadgets. As seen in the lower left corner, a small amount of about 12% of the gadgets can be modified only in one way, and thus can exist in two possible states. However, the randomization entropy increases exponentially, and the upper 80% of the gadgets have four or more randomized states. As more of the different transformations are applied on the same gadget, the randomization entropy increases to thousands of possible modified states.

Although for a small amount of gadgets an attacker can have a 50% chance of guessing the actual behavior of a gadget, ROP code relies on a chain of many different gadgets to achieve its purpose (11–18 unique gadgets in the exploits we tested). Even if one of the gadgets behaves in a non-expected way, then the ROP code will fail. Given that code randomization typically breaks (or even eliminates) several of the gadgets used in a ROP exploit, the number of possible randomized states that can prevent the correct execution of the ROP code is usually very high, as demonstrated in Sec. V-C.

B. Correctness and Performance

One of the basic principles of our approach is that the different in-place code randomization techniques should be applied cautiously, without breaking the semantics of the program. A straightforward way to verify the correctness of our code transformations is to apply them on existing code and compare the outcome before and after modification. Simply running a randomized version of a third-party application and verifying that it behaves in the expected way can provide a first indication. However, using this approach, it is hard to exercise a significant part of the code, and potentially incorrect modifications may go unnoticed.

For this purpose, we used the test suite of Wine [61], a compatibility layer that allows Windows applications to run on Unix-like operating systems. Wine provides alternative implementations of the DLLs that comprise the Windows API, and comes with an extensive test suite that covers the implementations of most functions exported by the core Windows DLLs. Each function is executed multiple times using various inputs that test different conditions, and the outcome of each execution is compared against a known, expected result. We ported the test code for about one third of the 109 DLLs included in the test suite of Wine v1.2.2, and used it directly on the actual DLLs from a Windows 7 installation. Using multiple randomized versions of each tested DLL, we verified that in all runs, all tests completed successfully.

We took advantage of the extensive and diverse code execution coverage of this experiment to also evaluate the impact of in-place code randomization to the runtime performance of the modified code. Among the different code transformations, instruction reordering is the only one that could potentially introduce some non-negligible overhead, given that sometimes the chosen ordering may be sub-optimal. We measured the overall CPU user time for the completion of all tests by taking the average time across multiple runs, using both the original and the randomized versions of the DLLs. In all cases, there was no observable difference in the two times, within measurement error.

C. Effectiveness

1) ROP Exploits: We evaluated the effectiveness of in-place code randomization using publicly available ROP exploits against vulnerable Windows applications [53], [62], [63], as well as generic ROP payloads based on commonly used DLLs [64], [65]. These seven different ROP code implementations, listed in Table II, bypass Windows DEP and execute a second-stage shellcode, as described in Sec. II, and work even in the latest version of Windows, with DEP and ASLR enabled. The ROP code used in the three exploits is implemented with gadgets from one or a few DLLs that do not support ASLR, as shown in the second column of Table II. The number of unique gadgets used in each case varies between 10–18, and typically a large part of the gadgets is repeatedly executed at many points throughout the ROP code. When replacing the original non-ASLR DLLs of each application with randomized versions, in all cases the exploits were rendered unsuccessful. Similarly, we used a custom application to test the generic ROP payloads and verified that the ROP code did not succeed when the corresponding DLL was randomized.

The ROP code of the exploit against Acrobat Reader uses just 11 unique gadgets, all coming from a single non-ASLR DLL (icucnv36.dll). From these gadgets, in-place code randomization can alter six of them: one gadget is completely eliminated, while the other five broken gadgets have 2, 2, 3, 4, and 6 possible states, respectively, resulting to a total of 287 randomized states (*in addition* to the always eliminated gadget, which also alone breaks the ROP code). Even if we assume that no elimination were possible, the exploit would still succeed only in one out of the 288 (0.35%) possible instances (including the original) of the given gadget set. Considering that this is a client-side exploit, in which the attacker will probably have only one or a few opportunities for tricking the user to open the malicious PDF file, the achieved randomization entropy is quite high—always assuming that none of the gadgets could have been eliminated. As shown

Table II
ROP EXPLOITS [53], [62], [63] AND GENERIC ROP PAYLOADS [64], [65] TESTED ON WINDOWS 7 SP1.

ROP exploit/payload	non-ASLR DLLs: used for ROP	Gadgets in non-ASLR DLLs	Modifiable (total %: Broken % Eliminated %)	Unique Gadgets Used: Modifiable (Br.,El.)	Combinations
Adobe Reader v9.3.4 [53]	3: 1	36,760	28,637 (77.9: 70.1 7.8)	**11: 6** (5, 1)	287
Integard Pro v2.2.0 [62]	1: 1	5,137	4,027 (78.4: 70.5 7.9)	**16: 10** (9, 1)	322,559
Mplayer Lite r33064 [63]	5: 2	117,822	104,671 (88.8: 70.0 18.8)	**18: 7** (6, 1)	1,128,959
msvcr71.dll [64]	1: 1	10,301	7,129 (69.2: 59.6 9.6)	**14: 9** (8, 1)	3,317,760
msvcr71.dll [65]	1: 1	10,301	7,129 (69.2: 59.6 9.6)	**16: 8** (8, 0)	1,728,000
mscorie.dll [64]	1: 1	1,616	1,304 (80.6: 73.5 7.1)	**10: 4** (4, 0)	25,200
mfc71u.dll [65]	1: 1	86,803	64,053 (73.8: 68.7 5.1)	**11: 6** (6, 0)	170,496

in Table II, the number of possible randomized states in the rest of the cases is several orders of magnitude higher. This is mostly due to the larger number of broken gadgets, as well as due to a few broken gadgets with tens of possible modified states, which both increase the number of states exponentially.

Next, we explored whether the affected gadgets could be directly replaced with unmodifiable gadgets in order to reliably circumvent our technique. Out of the six affected gadgets in the Adobe Reader exploit, only four can be directly replaced, meaning that the exploit cannot be trivially modified to bypass randomization. Furthermore, two of the gadgets have only one replacement each, and both replacements are found in code regions that are not discovered by IDA Pro—both could be randomized using a more precise code extraction method. For the rest of the ROP payloads, there are at least three irreplaceable gadgets in each case.

We should note that the relatively small number of gadgets used in most of these ROP payloads is a worst-case scenario for our technique, which however not only is able to prevent these exploits, but also does not allow the attacker to directly replace all the affected gadgets. Indeed, besides the more complex ROP payloads used in the Integard and Mplayer exploits, the rest of the payloads use API functions that are already imported by a non-ASLR DLL, and simply call them directly using hard-coded addresses. This type of API invocation is much simpler and requires fewer gadgets [26] compared to ROP code like the one used in the Integard and Mplayer exploits (16 and 18 unique gadgets, respectively), which first dynamically locates a pointer to kernel32.dll (always ASLR-enabled in Windows 7) and then gets a handle to `VirtualProtect`.

2) Automated ROP Payload Generation: The fact that some of the randomized gadgets are not directly replaceable does not necessarily mean that the same outcome cannot be achieved using solely unmodifiable gadgets. To assess whether an attacker could construct a ROP payload resistant to in-place code randomization based on gadgets that cannot be randomized, we used Q [26] and Mona [27], two automated ROP code construction tools.

Q is a general-purpose ROP compiler that uses semantic program verification techniques to identify the functionality of gadgets, and provides a custom language, named QooL, for writing input programs. Its current implementation only supports simple QooL programs that call a single function or system call, while passing a single custom argument. In case the function to be called belongs to an ASLR-enabled DLL, Q can compute a handle to it through the import table of a non-ASLR DLL [12], when applicable. We should note that although Q currently compiles only basic QooL programs that call a single API function, this does not limit our evaluation, but on the contrary, stresses even more our technique. The simpler the programs, the fewer the gadgets used, which makes it easier for Q to generate ROP code even when our technique limits the number of available gadgets.

Mona is a plug-in for Immunity Debugger [66] that automates the process of building Windows ROP payloads for bypassing DEP. Given a set of non-ASLR DLLs, Mona searches for available gadgets, categorizes them according to their functionality, and then attempts to automatically generate four alternative ROP payloads for giving execute permission to the embedded shellcode and then invoking it, based on the `VirtualProtect`, `VirtualAlloc`, `NtSetInformationProcess`, and `SetProcessDEPPolicy` API functions (the latter two are not supported in Windows 7).

Considering the functionality of the ROP payloads generated by the two tools, Mona generates slightly more complex payloads, but its gadget composition engine is less sophisticated compared to Q's. Q generates payloads that compute a function address, construct its single argument, and call it. Payloads generated by Mona also call a single memory allocation API function (which though requires the construction of several arguments), copy the shellcode to the newly allocated area, and transfer control to it. Note that the complexity of the ROP code used in the tested exploits is even higher, since they rely on up to four different API functions [53], or "walk up" the stack to discover pointers to non-imported functions from ASLR-enabled DLLs [62], [63].

Table III shows the results of running Q and Mona on the same set of applications and DLLs used in the previous section (for applications, all non-ASLR DLLs are analyzed collectively), for two different cases: when all gadgets are available to the ROP compiler, and when only the non-randomized gadgets are available. The second case aims to build a payload that will be functional even when code randomization is applied. Although both Q and Mona were able to create payloads when applied on the original DLLs in almost all cases, they failed to construct any payload using only non-randomized gadgets in *all* cases.

Although our technique was able to prevent two different

Table III
RESULTS OF RUNNING Q [26] AND MONA [27] ON THE ORIGINAL NON-ASLR DLLS LISTED IN TABLE II, AND THE UNMODIFIED PARTS OF THEIR RANDOMIZED VERSIONS. IN ALL CASES, BOTH TOOLS FAILED TO GENERATE A ROP PAYLOAD USING SOLELY NON-RANDOMIZED GADGETS.

	Q success		Mona success	
Application/DLL	Orig.	Rand.	Orig.	Rand.
Adobe Reader	✔	✘	✔ (VA)	✘
Integard Pro	✔	✘	✘	✘
Mplayer	✔	✘	✔ (VA)	✘
msvcr71.dll	✔	✘	✘	✘
mscorie.dll	✘	✘	✘	✘
mfc71u.dll	✔	✘	✔ (VA,VP)	✘

tools from automatically constructing reliable ROP code, this favorable outcome does not exclude the possibility that a functional payload could still be constructed based solely on non-randomized gadgets, e.g., in a manual way or using an even more sophisticated ROP compiler. However, it clearly demonstrates that in-place code randomization significantly raises the bar for attackers, and makes the construction of reliable ROP code much harder, even in an automated way.

This is reflected on the reduction in the number of available (non-randomized) gadgets after code randomization. Both tools operate in two phases: gadget discovery and code compilation. During the first phase, they search for useful gadgets and categorize them according to their functionality. Tables IV and V show the number of useful gadgets as reported by Q and Mona, respectively, that are available before and after randomization. As shown by the percentage of the remaining gadgets (last column), many gadget types have very few available gadgets or are eliminated completely, which makes the construction of reliable ROP code much harder.

VI. DISCUSSION

In-place code randomization may not always randomize a significant part of the executable address space, and it is hard to give a definitive answer on whether the remaining unmodifiable gadgets would be sufficient for constructing useful ROP code. This depends on the code in the non-ASLR address space of the particular vulnerable process, as well as on the actual operations that need to be achieved using ROP code. Note that Turing-completeness is irrelevant for practical exploitation [26], and none of the gadget sets used in the tested ROP payloads is Turing-complete. For this reason, we emphasize that in-place code randomization should be used as a mitigation technique, in the same fashion as application armoring tools like EMET [51], and not as a complete prevention solution.

As previous studies [2], [5], [26] have shown, though, the feasibility of building a ROP payload is proportional to the size of the non-ASLR code base, and reversely proportional to the complexity of the desired functionality. Our experimental evaluation shows that in all cases, the space of the remaining useful gadgets after randomization is sufficiently small to prevent the automated generation of a ROP payload. At the same time, the tested ROP payloads are far from the complexity of a fully blown ROP-based implementation of the operations required for carrying out an attack, such as dumping a malicious executable on disk and executing it. Currently, this functionality is handled by the embedded shellcode, which in essence allows us to view these ROP payloads as sophisticated versions of return-to-libc. We should stress that the randomization coverage of our prototype implementation is a lower bound for what would be possible using a more sophisticated code extraction method [41], [49]. In our future work, we also plan to relax some of the conservative assumptions that we have made in instruction reordering and register reassignment, using data flow analysis based on constant propagation.

Given its practically zero overhead and direct applicability on third-party executables, in-place code randomization can be readily combined with existing techniques to improve diversity and reduce overheads. For instance, compiler-level techniques against ROP attacks [15], [16] increase significantly the size of the generated code, and also affect the runtime overhead. Incorporating code randomization for eliminating some of the gadgets could offer savings in code expansion and runtime overheads. Our technique is also applicable in conjunction with randomization methods based on code block reordering [17]–[19], to further increase randomization entropy.

In-place code randomization at the binary level is not applicable for software that performs self-checksumming or other runtime code integrity checks. Although not encountered in the tested applications, some third-party programs may use such checks for hindering reverse engineering. Similarly, packed executables cannot be modified directly. However, in most third-party applications, only the setup executable used for software distribution is packed, and after installation all extracted PE files are available for randomization.

VII. RELATED WORK

Almost a decade after the introduction of the return-to-libc technique [28], the wide adoption of non-executable memory page protections in popular OSes sparked a new interest in more advanced forms of code-reuse attacks. The introduction of return-oriented programming [2] and its advancements [3]–[6], [8], [26], [33], [67]–[69] led to its adoption in real-world attacks [10], [11]. ROP exploits are facilitated by the lack of complete address space layout randomization in both Linux [12], and Windows [6], which otherwise would prevent or at least hinder [14] these attacks.

Besides address space randomization, process diversity can also be increased by randomizing the code of each executable segment, e.g., by permuting the order of functions or basic blocks [17]–[19]. However, these techniques are applicable only if the source code or the symbolic debugging information of the application to be protected is available. Our approach is inspired by these works, and attempts to bring the benefits of code randomization on COTS software, for which usually no source code or debugging information is available.

Return-oriented code disrupts the normal control flow of a process by diverting its execution to (potentially unintended) code fragments, most of which otherwise would never be targets of control transfer instructions. Enforcing the integrity of

Table IV
NUMBER OF USEFUL GADGETS IDENTIFIED BY Q [26] IN THE ORIGINAL CODE SEGMENTS / IN THEIR UNMODIFIABLE PARTS AFTER IN-PLACE RANDOMIZATION WAS APPLIED.

Gadget Type	Reader	Integard	Mplayer	msvcr71	mscorie	mfc71u	total	(%)
Pivots	171/27	55/11	156/48	89/18	13/5	65/20	549/129	(23.50)
Storemem	162/11	14/4	105/7	33/6	1/1	69/15	384/44	(11.46)
Move	57/7	25/13	68/35	31/12	7/3	62/60	250/130	(52.00)
ArithmeticStore	89/8	7/3	90/6	31/4	-	16/8	233/29	(12.45)
ArithmeticLoad	587/23	26/8	1194/40	147/24	-	290/104	2244/199	(8.87)
JumpConsts	1/1	1/1	1/1	1/1	1/1	1/1	6/6	(100.00)
SwitchStack	171/27	55/11	156/48	89/18	13/5	65/20	549/129	(23.50)
Loadmem	657/79	18/0	314/129	71/36	-	761/690	1821/934	(51.29)
LoadConst	424/36	121/20	621/138	155/23	14/3	175/67	1510/287	(19.01)
Arithmetic	409/49	59/10	517/66	167/41	8/2	347/190	1507/358	(23.76)

control transfers [20] can effectively protect against code-reuse attacks. Compile-time techniques also prevent the construction of ROP code by generating machine code that does not contain unintended instruction sequences ending with indirect control transfer instructions, and by safeguarding any indirect branches in the actual code using canaries or additional indirection [15], [16]. In contrast to the above approaches, although in-place code randomization does not completely preclude the possibility that working ROP code can be constructed, it can be applied directly on third-party software without access to source code or debugging information.

Another line of defenses are based on runtime solutions that monitor either the frequency of `ret` instructions [22], [23], or the integrity of the stack [21]. Besides the fact these techniques are ineffective against ROP code that uses indirect control transfer instructions other than `ret`, their increased runtime overhead limits their adoption.

VIII. CONCLUSION

The increasing number of exploits against Windows applications that rely on return-oriented programming to bypass exploit mitigations such as DEP and ASLR, necessitates the deployment of additional protection mechanisms that can harden imminently vulnerable third-party applications against these threats. Towards this goal, we have presented in-place code randomization, a technique that offers probabilistic protection against ROP attacks, by randomizing the code of third-party applications using various narrow-scope code transformations.

Our approach is practical: it can be applied directly on third-party executables without relying on debugging information, and does not introduce any runtime overhead. At the same time, it is effective: our experimental evaluation using in-the-wild ROP exploits and two automated ROP code construction toolkits shows that in-place code randomization can thwart ROP attacks against widely used applications, including Adobe Reader on Windows 7, and can prevent the automated generation of ROP code resistant to randomization. Our prototype implementation is publicly available, and as part of our future work, we plan to improve its randomization coverage using more advanced data flow analysis methods, and extend it to support ELF and 64-bit executables.

AVAILABILITY

Our prototype implementation is publicly available at http://nsl.cs.columbia.edu/projects/orp

ACKNOWLEDGEMENTS

We are grateful to the authors of Q for making it available to us, and especially to Edward Schwartz for his assistance. We also thank Úlfar Erlingsson and Periklis Akritidis for their valuable feedback on earlier versions of this paper. This work was supported by DARPA and the US Air Force through Contracts DARPA-FA8750-10-2-0253 and AFRL-FA8650-10-C-7024, respectively, and by the FP7-PEOPLE-2009-IOF project MALCODE, funded by the European Commission under Grant Agreement No. 254116. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, DARPA, or the Air Force.

REFERENCES

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

[2] 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 (CCS)*, 2007.

[3] S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham, "Can DREs provide long-lasting security? the case of return-oriented programming and the AVC advantage," in *Proceedings of the 2009 conference on Electronic Voting Technology/Workshop on Trustworthy Elections (EVT/WOTE)*, 2009.

[4] R. Hund, T. Holz, and F. C. Freiling, "Return-oriented rootkits: bypassing kernel code integrity protection mechanisms," in *Proceedings of the 18th USENIX Security Symposium*, 2009.

[5] T. Dullien, T. Kornau, and R.-P. Weinmann, "A framework for automated architecture-independent gadget search," in *Proceedings of the 4th USENIX Workshop on Offensive Technologies (WOOT)*, 2010.

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

[7] P. Solé, "Hanging on a ROPe," http://www.immunitysec.com/downloads/DEPLIB20_ekoparty.pdf.

[8] D. A. D. Zovi, "Mac OS X return-oriented exploitation." RECON, 2010.

[9] P. Vreugdenhil, "Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit," http://vreugdenhilresearch.nl/Pwn2Own2010-Windows7-InternetExplorer8.pdf.

[10] K. Baumgartner, "The ROP pack," in *Proceedings of the 20th Virus Bulletin International Conference (VB)*, 2010.

[11] M. Parkour, "An overview of exploit packs (update 9) April 5 2011," http://contagiodump.blogspot.com/2010/06/overview-of-exploit-packs-update.html.

[12] 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.

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

[14] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and Communications Security (CCS)*, 2004.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and Communications Security (CCS)*, 2005.

[21] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: A practical protection tool to protect against return-oriented programming," in *Proceedings of the 6th Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[22] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "DROP: Detecting return-oriented programming malicious code," in *Proceedings of the 5th International Conference on Information Systems Security (ICISS)*, 2009.

[23] L. Davi, A.-R. Sadeghi, and M. Winandy, "Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks," in *Proceedings of the 2009 ACM workshop on Scalable Trusted Computing (STC)*, 2009.

[24] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proceedings of the 10th ACM conference on Computer and Communications Security (CCS)*, 2003.

[25] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," in *Proceedings of the 10th ACM conference on Computer and Communications Security (CCS)*, 2003.

[26] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy," in *Proceedings of the 20th USENIX Security Symposium*, 2011.

[27] Corelan Team, "Mona," http://redmine.corelan.be/projects/mona.

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

[29] T. Newsham, "Non-exec stack," 2000, http://seclists.org/bugtraq/2000/May/90.

[30] Nergal, "The advanced return-into-lib(c) exploits: PaX case study," *Phrack*, vol. 11, no. 58, Dec. 2001.

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

[32] Ú. Erlingsson, "Low-level software security: Attack and defenses," Microsoft Research, Tech. Rep. MSR-TR-07-153, 2007, http://research.microsoft.com/pubs/64363/tr-2007-153.pdf.

[33] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and Communications Security (CCS)*, 2010.

[34] F. B. Cohen, "Operating system protection through program evolution," *Computers and Security*, vol. 12, pp. 565–584, Oct. 1993.

[35] P. Ször, *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, February 2005.

[36] 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.

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

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

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

[40] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *Proceedings of the 13th USENIX Security Symposium*, 2004.

[41] M. Smithson, K. Anand, A. Kotha, K. Elwazeer, N. Giles, and R. Barua, "Binary rewriting without relocation information," University of Maryland, Tech. Rep., 2010, http://www.ece.umd.edu/~barua/without- relocation-technical-report10.pdf.

[42] P. Saxena, R. Sekar, and V. Puranik, "Efficient fine-grained binary instrumentation with applications to taint-tracking," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code Generation and Optimization (CGO)*, 2008.

[43] Skape, "Locreate: An anagram for relocate," *Uninformed*, vol. 6, 2007.

[44] M. Pietrek, "An in-depth look into the Win32 portable executable file format, part 2," http://msdn.microsoft.com/en-us/magazine/cc301808.aspx.

[45] I. Guilfanov, "Jump tables," http://www.hexblog.com/?p=68.

[46] ——, "Decompilers and beyond." Black Hat USA, 2008.

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

[48] 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.

[49] S. Nanda, W. Li, L.-C. Lam, and T.-c. Chiueh, "Bird: Binary interpretation using runtime disassembly," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2006.

[50] L. C. Harris and B. P. Miller, "Practical analysis of stripped binary code," *SIGARCH Comput. Archit. News*, vol. 33, pp. 63–68, December 2005.

[51] Microsoft, "Enhanced Mitigation Experience Toolkit v2.1," http://www.microsoft.com/download/en/details.aspx?id=1677.

[52] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[53] "Adobe CoolType SING Table "uniqueName" Stack Buffer Overflow," http://www.exploit-db.com/exploits/16619/.

[54] R. El-Khalil and A. D. Keromytis, "Hydan: Hiding information in program binaries," in *Proceedings of the International Conference on Information and Communications Security, (ICICS)*, 2004.

[55] *Intel 64 and IA-32 Architectures Software Developer's Manual*, ser. Volume 2 (2A & 2B): Instruction Set Reference, A-Z, 2011, http://www.intel.com/Assets/PDF/manual/325383.pdf.

[56] S. S. Muchnick, *Advanced compiler design and implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[57] Y. L. Varol and D. Rotem, "An algorithm to generate all topological sorting arrangements," *Comput. J.*, vol. 24, no. 1, pp. 83–84, 1981.

[58] A. Fog, "Calling conventions for different C++ compilers and operating systems," http://agner.org/optimize/calling_conventions.pdf.

[59] Skape and Skywing, "Bypassing Windows hardware-enforced DEP," *Uninformed*, vol. 2, Sep. 2005.

[60] F. Bouchez, "A study of spilling and coalescing in register allocation as two separate phases," Ph.D. dissertation, École normale supérieure de Lyon, April 2009.

[61] "Wine," http://www.winehq.org.

[62] "Integard Pro 2.2.0.9026 (Win7 ROP-Code Metasploit Module)," http://www.exploit-db.com/exploits/15016/.

[63] "MPlayer (r33064 Lite) Buffer Overflow + ROP exploit," http://www.exploit-db.com/exploits/17124/.

[64] "White Phosphorus Exploit Pack," http://www.whitephosphorus.org/.

[65] Corelan Team, "Corelan ROPdb," https://www.corelan.be/index.php/security/corelan-ropdb/.

[66] "Immunity Debugger," http://www.immunityinc.com/products-immdbg.shtml.

[67] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: generalizing return-oriented programming to RISC," in *Proceedings of the 15th ACM conference on Computer and Communications Security (CCS)*, 2008.

[68] T. Bletsch, X. Jiang, V. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proceedings of the 6th Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[69] P. Solé, "Defeating DEP, the Immunitiy Debugger way," http://www.immunitysec.com/downloads/DEPLIB.pdf.

Table V

NUMBER OF USEFUL GADGETS IDENTIFIED BY MONA [27] IN THE ORIGINAL CODE SEGMENTS / IN THEIR UNMODIFIABLE PARTS AFTER IN-PLACE RANDOMIZATION WAS APPLIED.

Gadget Type	Reader	Integard	Mplayer	msvcr71	mscorie	mfc71u	total	(%)
add eax -> ebx	-	-	-	3/0	-	-	3/0	(0.00)
add ebp -> eax	1/0	-	-	-	-	1/0	2/0	(0.00)
add ebp -> ebx	-	-	-	-	-	1/1	1/1	(100.00)
add ebp -> edi	-	-	-	-	-	1/1	1/1	(100.00)
add ebp -> edx	-	-	2/0	-	-	-	2/0	(0.00)
add ebx -> eax	1/0	-	4/0	-	-	-	5/0	(0.00)
add ebx -> ecx	-	-	1/0	-	-	-	1/0	(0.00)
add ebx -> edx	1/0	1/1	1/0	1/0	1/1	1/0	6/2	(33.33)
add ecx -> eax	5/0	-	4/0	2/0	-	-	11/0	(0.00)
add ecx -> ebp	-	-	3/1	-	-	-	3/1	(33.33)
add edi -> eax	4/0	-	3/0	-	-	1/0	8/0	(0.00)
add edi -> ecx	-	-	8/0	-	-	-	8/0	(0.00)
add edi -> edx	-	-	4/0	-	-	-	4/0	(0.00)
add edx -> eax	3/0	-	5/0	-	-	-	8/0	(0.00)
add esi -> eax	9/0	-	5/0	2/0	-	-	16/0	(0.00)
add esi -> ecx	-	-	16/0	-	-	-	16/0	(0.00)
add esi -> edi	-	-	3/0	-	-	4/4	7/4	(57.14)
add value to eax	3/2	2/1	2/2	2/2	2/1	4/1	15/9	(60.00)
add value to ebx	1/0	-	-	-	-	-	1/0	(0.00)
add value to edi	-	-	-	-	-	1/0	1/0	(0.00)
add value to edx	-	-	1/0	-	-	-	1/0	(0.00)
add value to esi	-	-	-	-	-	1/0	1/0	(0.00)
dec eax	24/9	-	84/24	22/4	1/1	33/11	164/49	(29.88)
dec ebp	-	-	3/0	-	-	1/1	4/1	(25.00)
dec ebx	-	-	4/3	-	-	-	4/3	(75.00)
dec ecx	2/0	5/5	18/12	63/59	1/1	186/177	275/254	(92.36)
dec edi	2/0	-	2/0	-	-	2/2	6/2	(33.33)
dec edx	111/87	-	3/2	1/1	-	3/2	118/92	(77.97)
dec esi	1/0	-	5/3	1/0	-	1/0	8/3	(37.50)
empty eax	156/2	1/0	133/0	89/0	5/0	196/4	580/6	(1.03)
empty edi	-	-	-	1/0	-	-	1/0	(0.00)
empty edx	2/0	-	2/0	-	-	5/0	9/0	(0.00)
inc eax	51/19	2/0	53/6	108/95	9/2	281/141	504/263	(52.18)
inc ebp	-	-	134/0	1/1	-	2/1	137/2	(1.46)
inc ebx	6/2	-	9/3	12/1	-	6/1	33/7	(21.21)
inc ecx	0/5	-	9/0	1/0	-	12/8	22/13	(59.09)
inc edi	3/0	-	1/1	2/0	-	7/0	13/1	(7.69)
inc edx	3/0	-	37/1	1/0	-	1/1	42/2	(4.76)
inc esi	14/1	1/0	2/0	3/0	-	11/1	31/2	(6.45)
move eax -> ebp	3/0	-	23/2	-	-	5/1	31/3	(9.68)
move eax -> ebx	-	-	52/0	3/0	-	2/1	57/1	(1.75)
move eax -> ecx	1/0	1/1	7/1	-	-	-	9/2	(22.22)
move eax -> edi	4/0	-	7/0	-	-	17/1	28/1	(3.57)
move eax -> edx	-	-	10/1	-	-	-	10/1	(10.00)
move eax -> esi	2/0	-	19/0	-	-	13/1	34/1	(2.94)
move eax -> esp	11/2	-	30/3	1/0	-	43/23	85/28	(32.94)
move ebp -> eax	34/0	-	80/2	2/0	-	17/1	133/3	(2.26)
move ebp -> ebx	-	-	2/0	-	-	1/1	3/1	(33.33)
move ebp -> edi	5/0	-	2/0	-	-	2/1	9/1	(11.11)
move ebp -> edx	-	-	6/0	-	-	-	6/0	(0.00)
move ebx -> eax	96/0	2/0	151/0	8/0	1/0	37/1	295/1	(0.34)
move ebx -> ecx	-	-	1/0	-	-	-	1/0	(0.00)
move ebx -> edi	1/0	-	-	-	-	3/0	4/0	(0.00)
move ebx -> edx	1/0	1/1	1/0	1/0	1/1	1/0	6/2	(33.33)
move ebx -> esp	4/0	-	2/0	2/1	-	-	8/1	(12.50)
move ecx -> eax	26/1	3/2	46/1	10/4	1/0	41/2	127/10	(7.87)
move ecx -> ebp	-	-	3/1	-	1/0	3/1	7/2	(28.57)
move ecx -> ebx	-	-	4/0	-	-	-	4/0	(0.00)
move ecx -> edi	-	-	1/0	-	-	6/0	7/0	(0.00)
move ecx -> edx	2/0	-	-	-	-	2/0	4/0	(0.00)
move ecx -> esi	1/0	-	-	-	-	5/0	6/0	(0.00)
move ecx -> esp	-	-	-	-	-	2/0	2/0	(0.00)
move edi -> eax	125/0	-	92/8	15/0	6/0	96/1	334/9	(2.69)
move edi -> ebp	1/0	-	-	-	-	-	1/0	(0.00)
move edi -> ebx	-	-	1/0	-	-	-	1/0	(0.00)
move edi -> ecx	1/0	-	8/0	-	-	-	9/0	(0.00)
move edi -> edx	-	-	19/0	-	-	-	19/0	(0.00)
move edi -> esi	-	-	3/0	-	-	5/5	8/5	(62.50)
move edi -> esp	-	-	19/0	-	-	-	19/0	(0.00)
move edx -> eax	17/1	-	92/1	1/0	1/1	6/0	117/3	(2.56)
move edx -> ebx	1/0	-	3/0	-	-	-	4/0	(0.00)
move edx -> ecx	-	-	-	1/0	-	-	1/0	(0.00)
move edx -> edi	1/0	-	-	-	-	1/0	2/0	(0.00)
move edx -> esi	1/0	-	-	-	-	1/0	2/0	(0.00)
move esi -> eax	488/0	2/0	136/0	58/0	12/1	513/2	1209/3	(0.25)
move esi -> ebx	-	-	2/0	-	-	-	2/0	(0.00)
move esi -> ecx	2/0	-	16/0	-	-	-	18/0	(0.00)
move esi -> edi	-	-	3/0	-	-	4/4	7/4	(57.14)
move esi -> edx	-	-	8/0	-	-	-	8/0	(0.00)
move esi -> esp	1/0	-	17/0	-	-	-	18/0	(0.00)
move esp -> eax	1/0	-	1/0	-	-	-	2/0	(0.00)
move esp -> ebp	-	-	1/0	-	-	-	1/0	(0.00)
move esp -> ebx	5/0	-	85/0	-	-	-	90/0	(0.00)
move esp -> ecx	8/0	-	-	1/0	-	-	9/0	(0.00)
move esp -> edi	37/0	-	10/0	-	-	2/0	49/0	(0.00)
move esp -> esi	20/0	-	4/0	2/0	-	5/0	31/0	(0.00)
neg eax	3/1	-	1/1	7/0	-	9/8	20/10	(50.00)
neg edx	-	-	-	1/0	-	-	1/0	(0.00)
pickup pointer into eax	15/10	1/0	12/3	2/1	-	23/15	53/29	(54.72)
pickup pointer into ecx	-	-	2/0	-	-	-	2/0	(0.00)
pushad	7/0	-	26/4	1/0	-	17/12	51/16	(31.37)
xor ebp -> eax	1/0	-	-	-	-	-	1/0	(0.00)
xor edx -> eax	-	-	1/0	-	-	-	1/0	(0.00)
xor esi -> eax	1/0	-	-	-	-	-	1/0	(0.00)