Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code

Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, Zhiqiang Lin
Department of Computer Science, The University of Texas at Dallas
800 W. Campbell Rd, Richardson, TX, 75252
{richard.wartell, vishwath.mohan, hamlen, zhiqiang.lin}@utdallas.edu

Abstract

Unlike library code, whose instruction addresses can be randomized by *address space layout randomization* (ASLR), application binary code often has static instruction addresses. Attackers can exploit this limitation to craft robust shell codes for such applications, as demonstrated by a recent attack that reuses instruction *gadgets* from the static binary code of victim applications.

This paper introduces binary *stirring*, a new technique that imbues x86 native code with the ability to self-randomize its instruction addresses each time it is launched. The input to STIR is only the application binary code without any source code, debug symbols, or relocation information. The output is a new binary whose basic block addresses are dynamically determined at load-time. Therefore, even if an attacker can find code gadgets in one instance of the binary, the instruction addresses in other instances are unpredictable. An array of binary transformation techniques enable STIR to transparently protect large, realistic applications that cannot be perfectly disassembled due to computed jumps, code-data interleaving, OS callbacks, dynamic linking and a variety of other difficult binary features. Evaluation of STIR for both Windows and Linux platforms shows that stirring introduces about 1.6% overhead on average to application runtimes.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; D.3.4 [**Software**]: Processors—*Code generation; Translator writing systems and compiler generators*

Keywords

obfuscation, randomization, return-oriented programming, software security

1. INTRODUCTION

Subverting control-flows of vulnerable programs by hijacking function pointers (e.g., return addresses) and redirecting them to shell code has long been a dream goal of attackers. For such an attack to succeed, there are two conditions: (1) the targeted software is vulnerable to redirection, and (2) the attacker-supplied shell code is executable. Consequently, to stop these attacks, a great deal of research has focused on identifying and eliminating software vulnerabilities, either through static analysis of program source code (e.g., [37]) or through dynamic analysis or symbolic execution of program binary code (e.g., [13, 27]).

Meanwhile, there is also a significant amount of work focusing on how to prevent the execution of shell code based on its origin or location. Initially, attackers directly injected malicious machine code into vulnerable programs, prompting the development of W$\oplus$X (*write-xor-execute*) protections such as DEP [4] and ExecShield [57] to block execution of the injected payloads. In response, attackers began to redirect control flows directly to potentially dangerous code already present in victim process address spaces (e.g., in standard libraries), bypassing W$\oplus$X. Return-into-libc attacks [55] and return oriented programming (ROP) [12, 15, 51] are two major categories of such attacks. As a result, address space layout randomization (ASLR) [10, 45] was invented to frustrate attacks that bypass W$\oplus$X.

ASLR has significantly raised the bar for standard library-based shell code because attackers cannot predict the addresses of dangerous instructions to which they wish to transfer control. However, a recent attack from Q [50] has demonstrated that attackers can alternatively redirect control to shell code constructed from *gadgets* (i.e., short instruction sequences) already present in the application binary code. Such an attack is extremely dangerous since instruction addresses in most application binaries are fixed (i.e., static) once compiled (except for position independent code). This allows attackers to create robust shell code for many binaries [50].

Recent attempts to solve this issue have employed both static and dynamic techniques. In-place-randomization (IPR) [44] statically smashes unwanted gadgets by changing their semantics or reordering their constituent instructions without perturbing the rest of the binary. Alternatively, ILR [29] dynamically eliminates gadgets by randomizing all instruction addresses and using a fall-through map to dynamically guide execution through the reordered instructions. While these two approaches are valuable first steps, IPR suffers from deployment issues (since millions of separately shipped, randomized copies are required to obtain a sufficiently diverse field of application instances), and ILR suffers from high performance overhead (because of its highly dynamic, VM-based approach).

This paper introduces a new technique, *Self-Transforming Instruction Relocation (STIR)*, that transforms legacy application binary code into *self-randomizing* code that statically re-randomizes itself each time it is loaded. The capacity to re-randomize legacy code (i.e., code without debug symbols or relocation information) at load-time greatly eases deployment, and its static code transformation approach yields significantly reduced performance overheads. Moreover, randomizing at basic block granularity achieves higher entropy

CCS'12, October 16–18, 2012, Raleigh, North Carolina, USA.
Copyright 2012 ACM 978-1-4503-1651-4/12/10 ...$15.00.

than ASLR, which only randomizes section base addresses, and can therefore be susceptible to derandomization attacks [46, 52].

STIR is a fully automatic, binary-centric solution that does not require any source code or symbolic information for the target binary program. STIR-enabled code randomly reorders the basic blocks in each binary code section each time it is launched, frustrating attempts to predict the locations of gadgets. It is therefore fully transparent, and there is no modification to the OS or compiler. This makes it easily deployable; software vendors or end users need only apply STIR to their binaries to generate one self-randomizing copy, and can thereafter distribute the binary code normally.

Randomizing legacy CISC code for real-world OS's (Microsoft Windows and Linux) without compiler support raises many challenges, including semantic preservation of dynamically computed jumps, code interleaved with data, function callbacks, and imperfect disassembly information. These challenges are detailed further in §2. In this paper, we develop a suite of novel techniques, including conservative disassembly, jump table recovery, and dynamic dispatch, to address these challenges. Central to our approach is a binary transformation strategy that expects and tolerates many forms of disassembly errors by conservatively treating every byte in target code sections as both a potential instruction starting point and static data. This obviates the need for perfect disassemblies, which are seldom realizable in practice without source code.

We have implemented STIR and evaluated it on both Windows and Linux platforms with a large number of legacy binary programs. Our experimental results show that STIR can successfully transform application binaries with self-randomized instruction addresses, and that doing so introduces about 1.6% overhead (significantly better than ILR's 16%) on average at runtime to the applications.

In summary, this paper makes the following contributions.

- We present a mostly static, instruction address randomization technique to counter ROP shell-code that reuses gadgets in legacy binaries. Randomization is achieved via a novel static binary rewriting that first transforms the binary into a self-randomizable representation, followed by a load-time phase that *stirs* the binary at program start, yielding a different ordering of the instruction addresses each time the program is launched.

- To enable static rewriting of legacy code, we have developed a number of new techniques in support of native x86 code for computed jumps, code-data interleaving, OS callbacks, dynamic linking, and disassembly error tolerance. We believe these techniques constitute a significant advancement in static binary writing.

- We have implemented our entire system on two mainstream OS architectures (Windows and Linux) and tested it with a large number of application binaries. The implementation consists of a conservative, static *disassembler* and a load-time *reassembler*. Our empirical evaluation shows that STIR is a promising approach for defending real-world legacy binaries against ROP shell code.

Section 2 begins with a description of challenges that a legacy native code rewriter must overcome, and an overview of our approach. Section 3 elaborates by presenting a detailed design of STIR, including its static disassembler and load-time reassembler. Section 4 reports the results of our evaluation of STIR on over 100 Windows and Linux binaries. Section 5 discusses known limitations of the system, and Section 6 provides direct comparisons to related work. Finally, Section 7 concludes.

2. SYSTEM OVERVIEW

Users of our system submit legacy x86 COTS binaries (PE files for Windows, or ELF files for Linux) to a rewriter. The rewriter disassembles, transforms, and reassembles the target code into a new, self-randomizing binary that reorders its own basic blocks each time it is executed. Instruction address randomization is achieved by randomizing the basic blocks. No source code, relocation information, debug sections, symbol stores (e.g., PDB files), or hints are required. Expert users may guide the disassembly process through the use of an interactive disassembler (e.g., IDA Pro [28]), but our system is designed to make such guidance unnecessary in the vast majority of cases.

2.1 Challenges

Achieving instruction-level randomization of legacy, x86 binaries without source-level information introduces numerous challenges:

Computed jumps: Native x86 code often dynamically computes jump destinations from data values at runtime. Such operations pervade almost all x86 binaries; for example, binaries compiled from object-oriented languages typically draw code pointers from data in method dispatch tables. These pointers can undergo arbitrary binary arithmetic before they are used, such as logic that decodes them from an obfuscated representation intended to thwart buffer overrun attacks.

Preserving the semantics of computed jumps after stirring requires an efficient means of dynamically identifying and re-pointing all code pointers to the relocated instruction addresses. Prior work, such as static binary rewriters for software fault isolation (e.g., [25, 39, 53, 58]), relies upon compile-time support to handle this. However, randomizing legacy code for which there is no source-level relocation or debug information requires a new solution.

Code interleaved with data: Modern compilers aggressively interleave static data within code sections in both PE and ELF binaries for performance reasons. In the compiled binaries there is generally no means of distinguishing the data bytes from the code. Inadvertently randomizing the data along with the code breaks the binary, introducing difficulties for instruction-level randomizers. Viable solutions must somehow preserve the data whilst randomizing all the reachable code.

Disassembly undecidability: It is not possible in general to fully disassemble arbitrary x86 binaries purely statically. All static disassemblers rely on heuristics to find the reachable code amidst the data, and even the best disassemblers frequently guess incorrectly even for non-malicious, non-obfuscated binaries [59]. Solutions that assume fully correct disassemblies are therefore impractical for real-world, legacy, COTS binaries.

Callbacks: A callback occurs when the OS uses a code pointer previously passed from the program as a computed jump destination. Such callbacks are a mainstay of event-driven applications. Unlike typical computed jumps, callback pointers are not used as jump targets by any instruction visible to the randomizer. The only instructions that use them as jump targets are within the OS. This makes these code pointers especially difficult to identify and re-point correctly.

Position-dependent instructions: Instructions whose behavior will break if they are relocated within the section that contains them are said to be *position-dependent*. Ironically, position-dependent instructions are typically found within blocks of *position independent code* (PIC)—code sections designed to be relocatable *as a group*

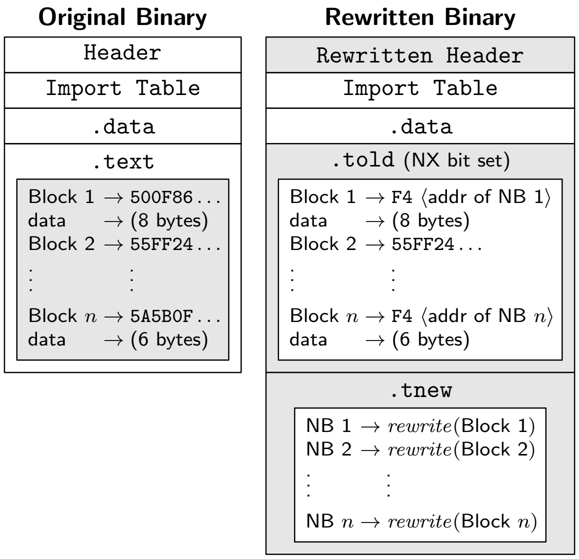

Figure 1: Static binary rewriting phase

at load-time or runtime [42]. The position independence of such code is typically achieved via instructions that dynamically compute their own addresses and expect to find the other instructions of the section at known offsets relative to themselves. Such instructions break if relocated within the section, introducing difficulties for more fine-grained, instruction-level randomization.

2.2 Approach Overview

Our system architecture addresses these challenges in two phases: (1) a static phase, depicted by Fig. 1, that transforms the binary into a more easily randomizable form, and (2) a load-time phase that stirs the binary by randomly reordering its instructions each time it starts. Code/data interleaving, imperfect disassembly, and position independent instructions are addressed by the static phase. Computed jumps and callbacks are further assisted by the load-time phase.

To solve the code/data interleaving and uncomputable disassembly problems, our static phase adopts a novel approach that treats all bytes as both data *and* code. To do so, it doubles each code segment into two separate segments—one in which all bytes are treated as data, and another in which all bytes are treated as code. This approach of keeping a copy of original binary code is partially inspired by DynInst [31] (although DynInst keeps only one copy, whereas STIR keeps both).

In the data-only copy (`.told`), all bytes are preserved at their original addresses, but the section is set non-executable (NX), safely deactivating any original, unrandomized code (including any gadgets) that it may contain. This safely preserves all the static data at its original addresses without the need to statically identify which bytes are data and which are code.

In the code-only copy (`.tnew`), all bytes are disassembled into code blocks that can be randomly stirred into a new layout each time the program starts. Any data bytes simply become harmless, unreachable code in the new binary's code section. Thus, there is no need to statically predict which bytes are part of reachable control-flows and which are not.

Random stirring of the code-only section is performed during the second stage by a trusted library statically linked into the new binary. The library randomly reorders all basic blocks in the new code section each time the program starts. The load order of the system guarantees that this library initializer code always runs before the target code it stirs.

```
Original:
.text:0040CC9B  FF D0        call eax
```

```
Rewritten:
.tnew:00436EDA  80 38 F4     cmp byte ptr [eax], F4h
.tnew:00436EDD  0F 44 40 01  cmovz eax, [eax+1]
.tnew:00436EE1  FF D0        call eax
```

Figure 2: Semantic preservation of computed jumps

Afterwards, some code pointers (e.g., immediate operands and those pushed onto the stack by `call` instructions) have been repointed to correct addresses, but others (e.g., those in method dispatch tables) continue to point into the data-only segment. Recall that the data-only segment is non-executable, so attempting to use one of these *stale pointers* as a computed jump target results in an exception (usually a crash). To solve this computed jump problem, our static phase additionally translates all computed jump instructions from the original code into a short alternative sequence in the new code that dynamically detects and re-points old pointers to new addresses at runtime. Computed jumps are extremely common, so keeping the replacement sequence small and efficient is crucial for maintaining good performance.

We discovered a 2-instruction sequence, shown in Fig. 2, that reliably re-points such code pointers with low overhead. Conceptually, our solution repurposes part of the data-only segment as a lookup table that maps old code addresses to the new one. Instructions at likely computed jump targets in the old code are overwritten during the static and load-time phases with pointers to their corresponding locations in the new code segment. Even though static disassemblers cannot reliably distinguish code from data or anticipate the destinations of specific computed jump instructions, they can identify likely computed jump targets, such as function prologues, with high accuracy. Such heuristics suffice to identify a superset of all computed jump targets, although which computed jumps go where remains unknown.

The instruction sequence in Fig. 2 leverages this information by conditionally replacing each code pointer with the correct pointer stored at the location to which it points, efficiently patching the computed control-flows at runtime. To easily distinguish old code pointers from new, our lookup table prefixes each pointer entry with tag byte `0xF4`, which encodes an illegal x86 instruction. Thus, pointers whose destinations start with the tag are stale and need to be updated, while those that do not are already correct.

Callbacks are facilitated by a similar process that involves rerouting control-flows that cross the user-code/OS boundary through the trusted library, which re-points callback pointers. The details of this process and the preservation of position-dependent instructions are discussed in greater detail the in the next section.

3. DETAILED DESIGN

The architecture of STIR is shown in Fig. 3. It includes three main components: (1) a conservative disassembler, (2) a *lookup table generator*, and (3) a load-time reassembler. At a high level, our disassembler takes a target binary and transforms it to a randomizable representation. An address map of the randomizable representation is encoded into the new binary by the lookup table generator. This is used by the load-time reassembler to efficiently randomize the new binary's code section each time it is launched.

This section presents a detailed design of each component. We first outline the static phase of our algorithm (our conservative disassembler and lookup table generator) in §3.1, followed by the

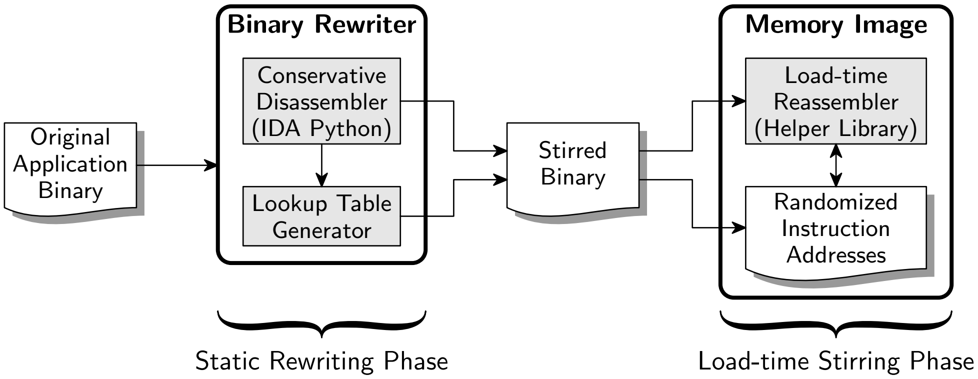
Original Application Binary
Binary Rewriter
Conservative Disassembler (IDA Python)
Lookup Table Generator
Stirred Binary
Memory Image
Load-time Reassembler (Helper Library)
Randomized Instruction Addresses
Static Rewriting Phase
Load-time Stirring Phase

Figure 3: System architecture

```
Algorithm 1 $Trans(\alpha, c)$: Translate one instruction
Input: address mapping $\alpha : \mathbb{Z} \rightharpoonup \mathbb{Z}$ and instruction $c$
Output: translated instruction(s)
  if $IsComputedJump(c)$ then
    $op \leftarrow Operand(c)$
    if $IsRegister(op)$ then
      return [ cmp $op$, F4h;
               cmovz $op$, [$op$+1]; $c$ ]
    else if $IsMemory(op)$ then
      $Operand(c) \leftarrow$ eax
      return [ mov eax, $op$;
               cmp [eax], F4h;
               cmovz eax, [eax+1]; $c$ ]
    end if
  else if $IsDirectJump(c)$ then
    $t \leftarrow OffsetOperand(c)$
    return $c$ with operand changed to $\alpha(t)$
  else
    return $c$
  end if
```

```
Algorithm 2 Translate all instructions
Input: instruction list $C$
Output: rewritten block list $B$
  $B \leftarrow [\,]$
  $\alpha \leftarrow \emptyset$
  $t \leftarrow$ base address of .told section
  $t' \leftarrow$ base address of .tnew section
  for all $c \in C$ do
    if $IsCode(c)$ then
      $\alpha \leftarrow \alpha \cup \{(t, t')\}$
      $t' \leftarrow t' + |Trans(\iota, c)|$
    end if
    $t \leftarrow t + |c|$
  end for
  for all $c \in C$ do
    if $IsCode(c)$ then
      append $Trans(\alpha, c)$ to $B$
    end if
  end for
  return $B$
```

load-time phase (our reassembler) in §3.2. Section 3.3 walks through an example. Finally, §3.4 addresses practical compatibility issues.

3.1 Static Rewriting Phase

Target binaries are first disassembled to assembly code. We use the IDA Pro disassembler from Hex-rays for this purpose [28], though any disassembler capable of accurately identifying likely computed jump targets could be substituted.

The resulting disassembly may contain harmless errors that misidentify data bytes as code, or that misidentify some code addresses as possible computed jump targets; but errors that omit code or misidentify data as computed jump targets can lead to non-functional rewritten code. We therefore use settings that encourage the disassembler to interpret all bytes that constitute valid instruction encodings as code, and that identify all instructions that implement prologues for known calling conventions as possible computed jump targets. These settings suffice to avoid all harmful disassembly errors in our experiments (see §4).

The assembly code is next partitioned into basic blocks, where a basic block can be any contiguous sequence of instructions with a single entry point. Each block must also end with an unconditional jump, but STIR can meet this requirement by inserting `jmp 0` instructions (a semantic no-op) to partition the code into arbitrarily small blocks during rewriting. The resulting blocks are copied and translated into a new binary section according to Algorithms 1–2, which we implemented as an IDAPython script.

Algorithm 1 translates a single instruction into its replacement in the new code section. Most instructions are left unchanged, but computed jumps are replaced with the lookup table code described in §2.2, and direct branches are re-pointed according to address mapping α.

Algorithm 2 calls Algorithm 1 as a subroutine to translate all the instructions. Its initial pass first computes mapping α by using identity function ι as the address mapping.[1] The second pass uses the resulting α to generate the final new code section with direct branches re-targeted.

Once the new code section has been generated, the lookup table generator overwrites all potential computed jump targets t in the original code section with a tag byte `0xF4` followed by 4-byte pointer $\alpha(t)$. This implements the lookup table described in §2.2.

It may seem more natural to implement range checks to identify stale pointers rather than using tag bytes. However, in general a stirred binary may consist of many separate modules, each of which has undergone separate stirring, and which freely exchange stale

[1] Some x86 instructions' lengths can change when ι is replaced by α. Our rewriter conservatively translates these to their longest encodings during the first pass to avoid such changes, but a more optimal rewriter could use multiple passes to generate smaller code.

code pointers at runtime. Since each module loads into a contiguous virtual address space, it is not possible to place all the old code sections within a single virtual address range. Thus, implementing pointer range checks properly would require many nested conditionals, impairing performance. Our use of tag bytes reduces this to a single conditional move instruction and no conditional branches.

The resulting binary is finalized by editing its binary header to import the library that performs binary stirring at program start. PE and ELF headers cannot be safely lengthened without potentially moving the sections that follow, introducing a host of data relocation problems. To avoid this, we simply substitute the import table entry for a standard system library (`kernel32.dll` on Windows) with an equal-length entry for our library. Our library exports all symbols of the system library as forwards to the real system library, allowing it to be transparently used as its replacement. This keeps the header length invariant while importing all the new code necessary for stirring.

3.2 Load-time Stirring Phase

When the rewritten program is launched, the STIR library's initializer code runs to completion before any code in STIR-enabled modules that link to it. On Windows this is achieved by the system load order, which guarantees that statically linked libraries initialize before modules that link to them. On Linux, the library is implemented as a shared object (SO) that is injected into the address space of STIR-enabled processes using the LD_PRELOAD environment variable. When this variable is set to the path of a shared object, the system loader ensures that the shared object is loaded first, before any of the other libraries that a binary may need.

The library initializer performs two main tasks at program start:

- All basic blocks in the linking module's `.tnew` section are randomly reordered. During this stirring, direct branch operands are repointed according to address mapping α, computed during the static phase.
- The lookup table in the linking module's `.told` section is updated according to α to point to the new basic block locations.

Once the initialization is complete, the `.tnew` section is assigned the same access permissions as the original program's `.text` section. This preserves non-writability of code employing W$\oplus$X protections.

To further minimize the attack surface, the library is designed to have as few return instructions as possible. The majority of the library that implements stirring is loaded dynamically into the address space at library initialization and then unloaded before the stirred binary runs. Thus, it is completely unavailable to attackers. The remainder of the library that stays resident performs small bookkeeping operations, such as callback support (see §3.4). It contains less than 5 return instructions total.

3.3 An Example

To illustrate our technique, Fig. 4 shows the disassembly of a part of the original binary's `.text` section and its counterparts in the rewritten binary's `.told` and `.tnew` sections after passing through the static and load-time phases described above.

The disassembly of the `.text` section shows two potential computed jump targets, at addresses `0x404B00` and `0x404B18`, that each correspond to a basic block entry point. In the rewritten `.told` section, the underlined values show how each is overwritten with the tag byte `0xF4` followed by the 4-byte pointer $\alpha(t)$ that represents its new location in the `.tnew` section.[2] All remaining bytes from the original code section are left unchanged (though the section is set non-executable) to ensure that any data misclassified as code is still accessible to instructions that may refer to it.

Original .text:

```
.text:00404AF0 00 4B 40 00          .dword 00404B00h
.text:00404AF4 18 4B 40 00          .dword 00404B18h
.text:00404AF8 CC (×8)              .align 16
.text:00404B00 8B 04 85 F0 4A 40 00 mov eax,[eax*4+404AF0h]
.text:00404B07 FF E1                jmp eax
.text:00404B09 CC CC CC CC CC CC CC .align 16
.text:00404B10 55                   push ebp
.text:00404B11 8B E5                mov esp, ebp
.text:00404B13 C3                   retn
.text:00404B14 CC CC CC CC          .align 8
.text:00404B18 55                   push ebp
.text:00404B19 83 F8 01             cmp eax, 1
.text:00404B1C 7D 02                jge 404B20h
.text:00404B1E 33 C0                xor eax, eax
.text:00404B20 8B C1                mov eax, ecx
.text:00404B22 E8 D9 FF FF FF       call 404B00h
```

STIRred .told (Jump Table):

```
.told:00404AF0 00 4B 40 00 18 4B 40 00
.told:00404AF8 CC CC CC CC CC CC CC CC
.told:00404B00 F4 4C 23 51 00 40 00 FF
.told:00404B08 E1 CC CC CC CC CC CC CC
.told:00404B10 55 8B E5 C3 CC CC CC CC
.told:00404B18 F4 12 5B 52 00 02 33 C0
.told:00404B20 8B C1 E8 D9 FF FF FF
```

STIRred .tnew:

```
.tnew:0051234C 8B 04 85 F0 4A 40 00 mov eax,[eax*4+404AF0h]
.tnew:00512353 80 38 F4             cmp F4h, [eax]
.tnew:00512356 0F 44 40 01          cmov eax, [eax+1]
.tnew:0051235A FF E1                jmp eax
                    ...(other basic blocks)...
.tnew:00525B12 55                   push ebp
.tnew:00525B13 83 F8 01             cmp eax, 1
.tnew:00525B16 0F 8D 00 00 00 02    jge 525B1Eh
.tnew:00525B1C 33 C0                xor eax, eax
.tnew:00525B1E 8B C1                mov eax, ecx
.tnew:00525B20 E8 27 C8 FE FF       call 51234C
                    ...(other basic blocks)...
.tnew:0053AF21 55                   push ebp
.tnew:0053AF22 8B E5                mov esp, ebp
.tnew:0053AF24 C3                   retn
```

Figure 4: A stirring example

The `.tnew` section contains the duplicated code after stirring. Basic blocks `0x404B00`, `0x404B10` and `0x404B18`, which were previously adjacent, are relocated to randomly chosen positions `0x51234C`, `0x53AF21` and `0x525B12` (respectively) within the new code section. Non-branch instructions are duplicated as is, but static branches are re-targeted to the new locations of their destinations. Additionally, as address `0x525B16` shows, branch instructions are conservatively translated to their longest encodings to accommodate their more distant new targets.

3.4 Special Cases

Real-world x86 COTS binaries generated by arbitrary compilers have some obscure features, some of which required us to implement special extensions to our framework to support them. In this section we describe the major ones and our solutions.

3.4.1 Callbacks

Real-world OS's—especially Windows—make copious use of callbacks for event-driven programming. User code solicits callbacks by passing code pointers to a *callback registration function* exported by the system. The supplied pointers are later invoked by the OS in response to events of interest, such as mouse clicks or timer interrupts. Our approach of dynamically re-pointing stale

[2] This value changes during each load-time stirring.

```
Original:
.text:0804894B  E8 00 00 00 00    call 08048950h
.text:08048950  5B                pop ebx
.text:08048951  81 C3 A4 56 00 00 add ebx, 56A4h
.text:08048957  8B 93 F8 FF FF FF mov edx, [ebx-8]
```

```
Rewritten:
.tnew:0804F007  E8 00 00 00 00    call 0804F00Ch
.tnew:0804F00C  5B                pop ebx
.tnew:0804F00D  BB F4 DF 04 08    mov ebx, 0804DFF4h
.tnew:0804F012  90                nop
.tnew:0804F013  8B 93 F8 FF FF FF mov edx, [ebx-8]
```

Figure 5: Position-independent code

pointers at the sites of dynamic calls does not work when the call site is located within an unstirred binary, such as an OS kernel module.

To compensate, our helper library hooks [30] all import address table entries of known callback registration functions exported by unstirred modules. The hooks re-point all calls to these functions to a helper library that first identifies and corrects any stale pointer arguments before passing control on to the system function. This interposition ensures that the OS receives correct pointer arguments that do not point into the old code section.

3.4.2 *Position Independent Code*

PIC instructions compute their own address at runtime and perform pointer arithmetic to locate other instructions and data tables within the section. An underlying assumption behind this implementation is that even though the absolute positions of these instructions in the virtual address space may change, their position relative to one another does not. This assumption is violated by stirring, necessitating a specialized solution.

All PIC that we encountered in our experiments had the form shown in the first half of Fig. 5. The call instruction has the effect of pushing the address of the following instruction onto the stack and falling through to it. The following instruction pops this address into a register, thereby computing its own address. Later this address flows into a computation that uses it to find the base address of a global offset table at the end of the section. In the example, constant `56A4h` is the compile-time distance from the beginning of the pop instruction to the start of the global offset table.

To support PIC, our rewriter identifies call instructions with operands of 0 and performs a simple data-flow analysis to identify instructions that use the pushed address in an arithmetic computation. It then replaces the computation with an instruction sequence of the same length that loads the desired address from the STIR system tables. This allows the STIR system to maintain position independence of the code across stirring. In Fig. 5, the `nop` instruction is added to ensure that the length matches that of the replaced computation.

Our analysis is not guaranteed to find all possible forms of PIC. For example, PIC that uses some other instruction to compute its address, or that allows the resulting address to flow through the heap before use, would defeat our analysis, causing the rewritten binary to crash at runtime. However, our analysis sufficed to support all PIC instances that we encountered, and compiler documentation of PIC standards indicates that there is only a very limited range of PIC implementations that needs to be supported [42].

3.4.3 *Statically Computed Returns*

Although returns are technically computed jumps (because they draw their destinations from the stack), our rewriting algorithm does not guard them with checks for stale pointers. This is a performance optimization that assumes that all return addresses are pushed onto the stack by calls; thus, no return addresses are stale.

Original .text: | **Jump table .told:**

```
2 bytes lost
func_1:                                  func_1:
.text:40EAA9 33 C0  xor eax, eax         .text:40EAA9 F4 2E
.text:40EAAB C3     retn                 .text:40EAAB 04
func_2:                                  func_2:
.text:40EAAC 50     push eax             .text:40EAAC F4
```

```
0 bytes lost
func_1:                                  func_1:
.text:40EAA9 33 C0  xor eax, eax         .text:40EAA9 F4 2E
.text:40EAAB 5B     pop ebx              .text:40EAAB 25
.text:40EAAC 5E     pop esi              .text:40EAAC 42
.text:40EAAD C3     retn                 .text:40EAAD 00
func_2:                                  func_2:
.text:40EAAE 50     push eax             .text:40EAAE F4
```

Figure 6: Overlapping function pointers

This assumption was met by all binaries we studied except for a certain pattern of initializer code generated by GNU Compilers. The code sequence in question pushes three immediate operands onto the stack, which later flow to returns. We supported this by treating those three instructions as a special case, augmenting them with stale pointer checks that correct them at the time they are pushed instead of at the time they are used. A more general solution could rewrite all return instructions with stale pointer guards, probably at the cost of performance.

3.4.4 *Short Functions*

Our jump table implementation overwrites each computed jump target with a 5-byte tagged pointer. This design assumes that nearby computed jump targets are at least 5 bytes apart; otherwise the two pointers must overlap. An example of this type of jump table collision is shown in Fig. 6, where the first row has two jump table destinations overlapping two bytes of each other, and the second row does not overlap at all. Such closely packed destinations are rare, since most computed jump destinations are already 16-byte aligned for performance reasons, and since all binaries compatible with *hot-patching* technology have at least 5 bytes of padding between consecutive function entry points (enough to encode a long jump instruction) [32].

In the rare case that two computed jump targets are closer, the rewriter strategically chooses stirred block addresses within the new code section whose pointer representations can safely overlap. For example, if the `.tnew` section is based at address `0x04000000`, the byte sequence `F4 00 F4 00 04 00 04` encodes two overlapping, little-endian, tagged pointers to basic block addresses `0x0400F400` and `0x04000400`, respectively. This strategy suffices to support at least 135 two-pointer collisions and 9 three-pointer collisions per rewritten code page—far more than we saw in any binary we studied.

4. EMPIRICAL EVALUATION

4.1 Effectiveness

4.1.1 *Rewriting Time and Space Overheads*

To evaluate the effectiveness of our system, we tested both the Windows and Linux versions of STIR with a variety of COTS and benchmark binaries. Both Windows and Linux tests were carried out on Windows 7 and Ubuntu 12 running on an Intel Core i5 dual core, 2.67GHz laptop with 4GB of physical RAM.

On Windows, we tested STIR against the SPEC CPU 2000 benchmark suite as well as popular applications like Notepad++ and

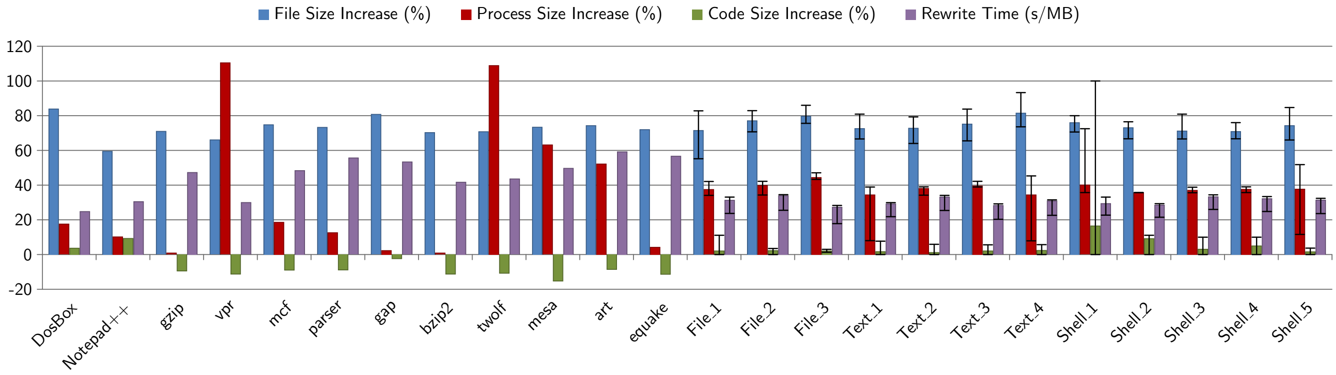

Figure 7: Static rewriting times and size increases

Table 1: Linux test programs grouped by type and size

Group	**Sizes** (KB)	**Programs**
File_1	17–37	dircolors, ln, mkdir, mkfifo, mknod, mktemp, rmdir, sync
File_2	41–45	chgrp, chmod, chown, dd, rm, shred, touch, truncate
File_3	49–97	chcon, cp, df, dir, install, ls, mv, vdir
Text_1	21–25	base64, cksum, comm, expand, fmt, fold, paste, unexpand
Text_2	25–29	cut, join, md5sum, nl, sha1sum, shuf, tac, tsort
Text_3	29–37	cat, csplit, head, sha224sum, sum, tr, uniq, wc
Text_4	37–89	od, pr, ptx, sha256sum, sha384sum, sha512sum, sort, split, tail
Shell_1	5–17	basename, dirname, env, false, hostid, link, logname, uptime
Shell_2	17–21	arch, echo, printenv, true, tty, unlink, whoami, yes
Shell_3	21	group, id, nice, noshup, pathchk, pwd, runcon, sleep
Shell_4	21–29	chroot, expr, factor, pinky, readlink, tee, test, uname, users
Shell_5	30–85	date, du, printf, seq, stat, stty, su, timeout, who

DosBox. For the Linux version, we evaluated our system against the 99 binaries in the coreutils toolchain (v7.0) for the Linux version. Due to space limitations, figures only present Windows binaries and a selection of 10 Linux binaries. In all of our tests, stirred binaries exhibited the same behavior and output as their original counterparts. Average overheads only cover binaries that run for more than 500ms.

Figure 7 shows how the rewriting phase affects the file size and code section sizes of each binary, which increase on average by 73% and 3% respectively. However, runtime process sizes increase by only 37% on average, with the majority of the increase due to the additional library that is loaded into memory. Our current helper library implementation makes no attempt to conserve its virtual memory allocations, so we believe that process sizes can be further reduced in future development. Occasionally our disassembler is able to safely exclude large sections of static data from rewritten code sections, leading to decreased code sizes. For example, `mesa`'s code section decreases by more than 15%. On average, static rewriting of Windows binaries requires 45 seconds per megabyte of code sections, whereas Linux binaries require 31 seconds per megabyte.

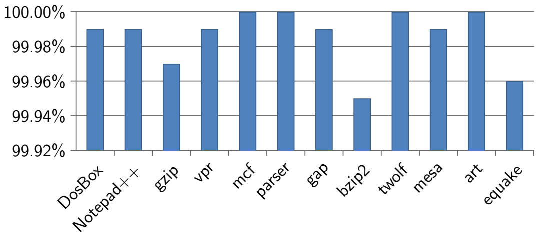

Figure 8: Gadget reduction for Windows binaries

Linux filenames in Fig. 7 are grouped by type (File, Text, and Shell) and by program size due to the large number of programs. Table 1 lists the programs in each group.

4.1.2 Gadget Elimination

One means of evaluating ROP attack protection is to count the number of gadgets that remain after securing each binary. There are several tools available for such evaluation, including Mona [20] on Windows and RoPGadget [48] on Linux. We used Mona to evaluate the stirred Windows SPEC2000 benchmark programs. Mona reports the number of gadgets the binary contains after the load-time phase is complete. We define a gadget as *unusable* if it is no longer at the same virtual address after basic block randomization. Figure 8 shows that on average STIR causes 99.99% of gadgets to become unusable. The only gadgets that remain after randomization of the test programs consist of a `pop` and a `retn` instruction that happened to fall onto the same address. Most malware payloads are not expressible with such primitive gadgets to our knowledge.

We also applied the Q exploit hardening system [50] to evaluate the effectiveness of our system. Since Q is a purely static gadget detection and attack payload generation tool, running Q dynamically after a binary has been stirred is not possible. Instead, we ran Q on a number of Linux binaries (*viz.*, rsync, opendchub, gv, and proftpd) to generate a payload, and then ran a script that began execution of the stirred binary, testing each of the gadgets Q selected for its payload after randomization. Attacks whose gadgets all remained usable after stirring were deemed successful; otherwise, Q's payload fails. In our experiments, no payload generated by Q was able to succeed against STIR.

4.2 Performance Overhead

Runtime performance statistics for Windows and Linux binaries are shown in Figs. 9 and 10, respectively, with each bar reflecting the application's median overhead over 20 trials. The median overhead

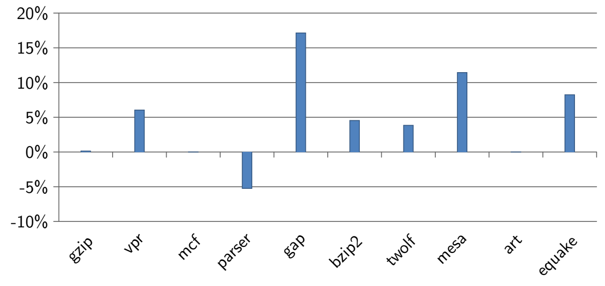

Figure 9: Runtime overheads for Windows binaries

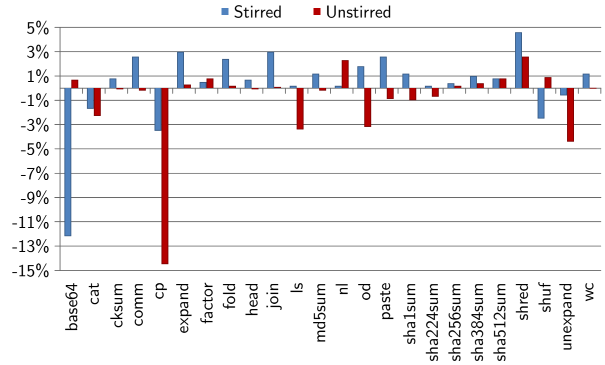

Figure 10: Runtime overheads for Linux binaries

is 4.6% for Windows applications, 0.3% for Linux applications, and 1.6% overall.

To isolate the effects of caching, Fig. 10 additionally reports the runtime overhead (discounting startup and initialization time) of *unstirred* Linux binaries, in which the load-time stirring phase was replaced by a loop that touches each original code byte without rewriting it, and then runs this unmodified, original code. This potentially has the effect of pre-fetching some or all of the code into the cache, decreasing some runtimes (although, as the figure shows, in practice the results are not consistent). Stirred binaries exhibit a median overhead of 1.2% over unstirred ones.

Amongst the Windows binaries, the `gap` SPEC2000 benchmark program consistently returns the worst overhead of 35%. This may be due to excessive numbers of callback functions or computed jumps. In contrast, the `parser` benchmark actually increases in speed by 5%. We speculate that this is due to improved locality resulting from separation of static data from the code (at the expense of increased process size). On average, the SPEC2000 benchmarks exhibit an overhead increase of 6.6%.

We do not present any runtime information for DosBox and Notepad++, since both are user-interactive. We did, however, manually confirm that all program features remain functional after transformation, and no performance degradation is observable.

To separate the load-time overhead of the stirring phase from the rest of the runtime overhead, Fig. 11 plots the stirring time against the code size. As expected, the graph shows that the increase in load-times is roughly linear with respect to code sizes, requiring 1.37ms of load-time stirring per KB of code on average.

For the most part, none of our tests require manual intervention by the user; all are fully automatic. The only exception to this is that IDA Pro's disassembly of each SPEC2000 benchmark program contained exactly two identical errors due to a known bug in its control-flow analysis heuristic. We manually corrected these two errors in each case before proceeding with static rewriting.

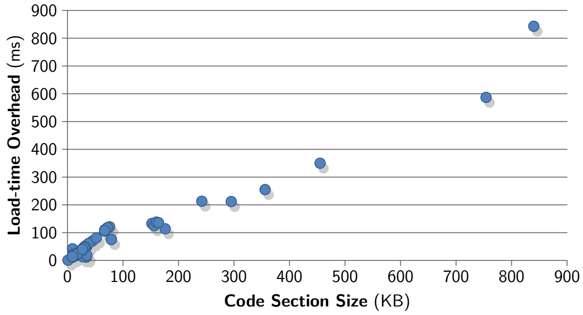

Figure 11: Load-time overhead vs. code size

5. DISCUSSION

In this section we first examine the entropy of our randomization, and then discuss limitations of STIR and propose future work.

5.1 Randomization Entropy

Our threat model assumes that attackers do not have unrestricted read access to process address spaces. (Such powerful access renders address randomization defenses ineffective.) Under this assumption, the search space that an attacker must explore to defeat STIR is much larger than for ASLR (including 64-bit ASLR and PIE [24]) because even the relative locations of gadgets within sections are randomized by STIR.

For example, past work concludes that a brute force attack against an exploitable Apache web server equipped with standard (fixed) PaX ASLR requires an expected $2^n/2 = 2^{n-1}$ probes for success, where n is the number of bits of randomness in the address space [52]. Re-randomizing ASLR doubles this to 2^n expected probes. However, STIR increases this to an expected $(2^n)!/(2(2^n - g)!)$ probes even without re-randomization, where g is the number of gadgets in the payload, since each probe must independently guess the positions of all g gadgets. On a 64-bit architecture with 14-bit aligned pages and 1 bit reserved for the kernel (i.e., $n = 50$), the expected number of probes for a g=3-gadget attack is therefore over 7.92×10^{28} ($\approx 2^{49}2^{48}/2$) times greater with STIR than with re-randomizing ASLR.

Brute force attacks that defeat ASLR by randomly guessing gadget locations over many trials [52] or using the location of one binary feature to infer the locations of others [46] therefore fail when applied to STIR. Runtime re-stirring could enlarge the search space even further; this is an avenue of future work (see §5.2).

The entropy of our randomization procedure is a function of the average basic block size of each binary relative to the size of the code section in which it resides. STIR also chooses random, independent base addresses for the new code section(s) and lookup table(s). This level of entropy is sufficient to render almost all gadgets unusable in our tests, but more cautious users may wish to increase the entropy further. To do so, STIR could be extended to pad rewritten code sections with random, unreachable instruction sequences that do not contain returns or jumps. This extra information would increase the number of basic blocks and decrease their relative size, thus increasing the entropy of the system.

5.2 Limitations and Future Work

STIR currently only randomizes code available at load time; code generated at runtime is not stirred. Thus, obfuscated code that unpacks itself at runtime derives little benefit from STIR, since the unpacked code is never randomized. Similarly, JIT-compiled code that is generated at runtime remains unrandomized (although the static JIT compiler is randomized by STIR). Periodically re-stirring stir-enabled binaries at runtime could help protect runtime-generated code, but is difficult to realize in practice because it introduces new, more complex varieties of stale pointers for each round of stirring. Implementing a runtime re-stirring system is therefore reserved as a subject of future work.

Our present work focuses on randomizing application main modules rather than libraries because library randomization is a less pressing concern (since libraries already benefit from ASLR randomization) and is significantly easier to realize (since almost all libraries support rebasing, and therefore contain relocation information). Our current implementation therefore only randomizes application main modules, not Windows DLLs or Linux shared objects (SOs). However, library stirring is not difficult to add. It merely requires the implementation of a custom loader that adjusts export address tables of stirred libraries after stirring but before dynamic linking. This is a conceptually easy extension that we intend to undertake in the future.

The experiments reported in §4 divide basic blocks at unconditional jump instructions already present in the original code. However, STIR can divide basic blocks at any instruction boundary by inserting jumps that explicitly fall through (i.e., `jmp 0`). This increases the entropy by partitioning the code into smaller, more numerous basic blocks. It also has the benefit of breaking (rather than merely relocating) ROP gadgets that rely on aliased instruction sequences whose encodings span the encodings of adjacent instructions in the application programming. Relocating the instructions so that they are non-adjacent breaks such a gadget. The tradeoff is increased code size (due to the new instructions) and increased runtime overhead (due to the larger number of jumps). Whether this tradeoff is worthwhile is an open question that should be investigated by future work.

Although our randomization strategy defeats typical ROP attacks that chain gadgets, it does not protect against control-flow hijacking attacks that simply call a legitimate computed jump target (e.g., an original method) with corrupted arguments. STIR's address translation logic permits such jumps because it detects and re-points stale pointers to legitimate computed jump targets at runtime. At the binary level, there is little that distinguishes a non-corrupted but stale pointer to such a method (e.g., one drawn from a method dispatch table) from one created by an attacker. Blocking these attacks requires a more refined control-flow integrity policy that dictates exactly which computed jumps may target which methods (cf., [1]). Reliably extracting such information from legacy binaries is a difficult open problem.

As mentioned in §3.4.1, our system requires a list of all callback registration functions in unstirred libraries. While the callback registration functions exported by system libraries are theoretically well documented parts of the public system API, in practice we have found that some are less than well documented. For example, some compilers generate calls to internal Windows `libc` functions for which we could find no documentation in any reference manual. To determine the signatures of these callees we were forced to disassemble and reverse-engineer the system modules that contain them. Maintaining a complete list of callback registration functions for a large OS can therefore be challenging when the system API documentation is incorrect or incomplete.

6. RELATED WORK

6.1 Security through Randomization

Forrest et al. [26] have suggested that monoculture is one of the main reasons why computers are vulnerable to large-scale, repeatable attacks including the most recent robust ROP shell codes [50]. As such, randomization has been introduced to increase the diversity of software. This strategy has been widely instantiated in existing works, such as ASLR [9, 10, 45], instruction set randomization (ISR) [5], data randomization [8, 14, 21], OS interface randomization [19], and multi-variant systems [11, 22, 47].

Address Space Layout Randomization: ASLR is a practical technique that has been adopted by many modern OSes such as Windows and Linux. The goal of ASLR is to obscure the location of code and data objects that are resident in memory, including the addresses of the program stack, heap, and shared library code [9, 10, 45, 60]. ASLR is currently implemented through modifying the OS kernel [45], system loader [60], and application source or binary code [9, 10]. However, all of these approaches require source code information (e.g., debug symbols or relocation data) in order to randomize the instruction addresses of most main modules. This motivates our work, which extends instruction address randomization to the majority of legacy main modules that lack such information.

In addition, existing source-agnostic ASLR approaches are limited to randomizations of relatively low granularity, leaving them vulnerable to derandomization attacks that can succeed even when the address space is large [46]. For example, ASLR can relocate and reorder some sections as wholes, but not the relative positions of the binary features within the sections. This leaves them vulnerable to attacks that reliably infer the relative positions of vulnerable code features irrespective of the size of the address space. In contrast, STIR randomizes the relative positions of such features, defeating such attacks.

Instruction Set Randomization: ISR is an approach to prevent code injection attacks by randomizing the underlying system instructions [5, 34]. In this approach, instructions are encrypted with a set of random keys and then decrypted before being fetched and executed by the CPU. ISR is effective for preventing code injections but cannot prevent ROP attacks. The technique is also hard to deploy in practice, requiring encryption of any supported software.

Data Randomization: As a dual to ISR, program data can also be encrypted and decrypted. PointGuard [21] encrypts all pointers while they reside in memory and decrypts them only before they are loaded into CPU registers. Recent work has presented a new data randomization technique that provides probabilistic protection against memory exploits by XORing data with random masks [8, 14]. DSR can help to prevent ROP attacks by decrypting attacker-injected code pointers to random addresses. However, data randomization requires recompilation of programs, which hinders its practicality.

OS Interface Randomization: System call mappings, global library entry points, and stack placement can all be randomized to mitigate buffer overflow attacks by increasing the heterogeneity of computer systems [19]. Similarly, RandSys [33] combines ASLR and ISR to randomize the system service interface when loading a program, and de-randomizes the instrumented interface for the correct execution at runtime. Similar to ISR, OS interface randomization cannot prevent ROP attacks in which all the attack code is drawn from the existing content of the victim address space.

Multi-variant Systems: Our work is also related to N-variant systems [11, 22, 47], which likewise leverage diversification to improve security. N-variant is an architectural framework that employs a set of automatically diversified variants to execute a common task. Any divergence among the outputs raises an alarm and can hence detect the attack. DieHard [7] is a simplified multi-variant framework that uses heap object randomization to make the variants generate different outputs in case of error or attack. Exterminator [40] extends this idea to derive runtime patches and automatically fix program bugs. Multi-variant systems frustrate ROP attacks by forcing the attacker to simultaneously subvert all the running variants, but require source code information in order to successfully apply comprehensive, semantics-preserving diversification of large applications.

6.2 ROP Defenses

In addition to diversification defenses, there are other techniques that specifically target ROP attacks. DROP [17] instruments program binary code and monitors the frequency of return instructions, which tend to rise during ROP attacks that rely heavily upon stack pointers to hijack control-flows. While DROP has been shown to be effective for ROP shell code detection, it suffers up to 5x performance overhead on average.

ROPdefender [23] is another binary instrumentation-based technique, which duplicates return addresses on a shadow stack and further evaluates each return instruction during program execution to detect mismatched calls and returns. ROPdefender is quite effective, and unlike DROP it only introduces 2x performance overhead. Other defenses, such as return-less kernels [38] and gadget-less binaries [41], eliminate return instructions during compilation.

There are also two compiler-based approaches to defeating ROP attacks. G-Free [41] removes gadgets from program binaries at compile time by eliminating all unaligned free-branch instructions and protecting the remaining aligned free-branch instructions. The other compiler approach generates return-less code by removing the `ret` opcode to prevent gadget formation [38].

Most recently, IPR [44] and ILR [29] have been proposed to alleviate the problem of ROP attacks. IPR uses in-place code randomization and instruction replacement to eliminate gadgets. Since its transformations are strictly in-place, it can only eliminate gadgets for which a semantics-preserving, size-invariant code transformation can be found. On average, 77% of gadgets meet this requirement, and can therefore be eliminated or broken by the technique. However, with large binaries that contain more than 100K gadgets, this is not enough to ensure secure execution. Also, IPR is difficult to deploy since each randomized application copy must be separately distributed.

ILR adopts a highly dynamic approach that first statically randomizes most instruction addresses and then dynamically guides control-flows through the randomized layout at runtime using a VM equipped with a fall-through map. However, like IPR, it is unable to move all instructions (mainly due to indirect branches), and therefore consistently preserves the locations of some gadgets. In addition, its dependence on a VM inevitably leads to higher performance overheads than purely static approaches.

STIR is orthogonal and complementary to these techniques in that it can be transparently applied to legacy code without code-producer cooperation.

6.3 Binary Rewriting

Static Approaches: Static binary rewriting is a core technology for many applications, such as software fault isolation [58], static instrumentation, in-lined reference monitoring [49], and tamper-proofing [3]. There are a large body of rewriting techniques, including PittsField [39], Google Native Client [61], and Diablo [56]. Their rewriting techniques typically target the assembly code yielded by a specific compiler with specific compiler options (e.g., [25, 39, 53, 58, 61]) or type-safe byte-code languages (e.g., [2, 6, 16]).

However, most static binary rewriters make strong assumptions about target binaries in order to successfully preserve their behavior. In contrast, STIR is compiler-agnostic, requires no relocation or debugging information, and has no reliance on symbol stores. This is important for practical applicability since most COTS legacy binaries lack this information.

SecondWrite [54] is the only other static binary rewriting system to our knowledge that targets COTS native code without relocation information. However, it is unsuitable for ROP protection because its rewriting algorithm must retain an executable copy of the original code in every rewritten binary, preserving gadgets. In addition, it is not yet mature enough to rewrite large real-world commercial applications; published experiments are currently limited to small, gcc-compiled programs [43].

Dynamic Approaches: Dynamic binary instrumentation is another approach to mitigating ROP attacks. Systems such as DynInst [31] and program shepherding [35] have the potential to intercept, analyze, and modify instructions at runtime to remove or modify the locations of gadgets, if extended with some form of basic block randomization.

However, as demonstrated by DROP [17] and ROPdefender [23], these systems tend to exhibit significantly higher overheads. STIR inlines the analysis necessary for basic block randomization, whereas dynamic instrumentation defers much of this analysis to computationally expensive runtime context switches between the application and the VM. For example, DynInst and DynamoRIO [18] exhibit 10–160x overhead and 30% overhead, respectively [18, 36].

7. CONCLUSION

We have presented STIR, a system that imbues legacy x86 binaries with self-randomizing instruction addresses. The system supports COTS binaries for both Windows and Linux, including those with dynamically computed jumps, code-data interleaving, OS callbacks, dynamic linking, and imperfect disassemblies. The system is compiler-agnostic and requires no form of code-producer cooperation. We have devised and implemented an array of novel techniques to address these challenges. Our evaluation shows that STIR can randomize a large body of large-scale, legacy x86 code, introducing about 1.6% runtime overhead on average to randomized applications.

8. ACKNOWLEDGMENTS

We thank David Brumley and Edward Schwartz for sharing Q [50] to better evaluate our system. We are also grateful to our shepherd and the anonymous reviewers for their insightful comments, which significantly improved the paper.

This research was supported in part by AFOSR grants FA9550-08-1-0044 and FA9550-10-1-0088, NSF grant #1054629, and DARPA grant #12011593. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily of the AFOSR, NSF, or DARPA.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Information and System Security*, 13(1), 2009.

[2] I. Aktug and K. Naliuka. ConSpec – a formal language for policy specification. In *Proc. Workshop on Run Time Enforcement for Mobile and Distributed Systems*, pages 45–58, 2007.

[3] B. Anckaert, M. Jakubowski, R. Venkatesan, and K. D. Bosschere. Run-time randomization to mitigate tampering. In *Proc. 2nd Int. Conf. on Advances in Information and Computer Security*, pages 153–168, 2007.

[4] S. Andersen. Part 3: Memory protection technologies. In V. Abella, editor, *Changes in Functionality in Windows XP Service Pack 2*. Microsoft TechNet, 2004. http://technet.microsoft.com/en-us/library/bb457155.aspx.

[5] 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 *Proc. 10th ACM Conf. on Computer and Communications Security*, pages 281–289, 2003.

[6] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 305–314, 2005.

[7] E. D. Berger and B. G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 158–168, 2006.

[8] S. Bhatkar and R. Sekar. Data space randomization. In *Proc. Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–22, 2008.

[9] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proc. 12th USENIX Security Symposium*, pages 105–120, 2003.

[10] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proc. 14th USENIX Security Symposium*, pages 255–270, 2005.

[11] D. Bruschi, L. Cavallaro, and A. Lanzi. Diversified process replicæ for defeating memory error exploits. In *Proc. IEEE Int. Performance, Computing, and Communications Conf.*, pages 434–441, 2007.

[12] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proc. 15th ACM Conf. on Computer and Communications Security*, pages 27–38, 2008.

[13] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Proc. 13th ACM Conf. on Computer and Communications Security*, pages 322–335, 2006.

[14] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro. Data randomization. Technical Report MSR-TR-2008-120, Microsoft Research, 2008.

[15] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proc. 17th ACM Conf. on Computer and Communications Security*, pages 559–572, 2010.

[16] F. Chen and G. Roşu. Java-MOP: A monitoring oriented programming environment for Java. In *Proc. 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 546–550, 2005.

[17] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. DROP: Detecting return-oriented programming malicious code. In *Proc. 5th Int. Conf. on Information Systems Security*, pages 163–177, 2009.

[18] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. TaintTrace: Efficient flow tracing with dynamic binary rewriting. In *Proc. 11th IEEE Symposium on Computers and Communications*, pages 749–754, 2006.

[19] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Carnegie Mellon University, 2002.

[20] Corelan Team. Mona, 2012. http://redmine.corelan.be/projects/mona.

[21] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard$^{\text{TM}}$: Protecting pointers from buffer overflow vulnerabilities. In *Proc. 12th USENIX Security Symposium*, pages 91–104, 2003.

[22] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-Variant Systems: A secretless framework for security through diversity. In *Proc. 15th USENIX Security Symposium*, 2006.

[23] L. Davi, A.-R. Sadeghi, and M. Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proc. 6th ACM Symposium on Information, Computer and Communications Security*, pages 40–51, 2011.

[24] U. Drepper. Security enhancements in Red Hat Enterprise Linux (beside SELinux), version 1.6, section 5: Position independent executables, December 2005. http://www.akkadia.org/drepper/nonselsec.pdf.

[25] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proc. New Security Paradigms Workshop*, 1999.

[26] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Proc. 6th Workshop on Hot Topics in Operating Systems*, page 67, 1997.

[27] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Proc. 15th Annual Network and Distributed System Security Symposium*, 2008.

[28] Hex-Rays. The IDA Pro disassembler and debugger, 2012. http://www.hex-rays.com/idapro.

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

[30] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*, chapter 4: The Age-Old Art of Hooking, pages 73–74. Pearson Education, Inc., 2006.

[31] J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. In *Proc. Scalable High Performance Computing Conf.*, pages 841–850, 1994.

[32] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *Proc. 3rd USENIX Windows NT Symposium*, 1999.

[33] X. Jiang, H. J. Wang, D. Xu, and Y.-M. Wang. RandSys: Thwarting code injection attacks with system service interface randomization. In *Proc. 26th IEEE Int. Symposium on Reliable Distributed Systems*, pages 209–218, 2007.

[34] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proc. 10th ACM Conf. on Computer and Communications Security*, pages 272–280, 2003.

[35] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure

execution via program shepherding. In *Proc. 11th USENIX Security Symposium*, 2002.

[36] M. O. Lam, J. K. Hollingsworth, and G. W. Stewart. Dynamic floating-point cancellation detection. In *Proc. 1st Int. Workshop on High-performance Infrastructure for Scalable Tools*, 2011.

[37] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proc. 10th USENIX Security Symposium*, 2001.

[38] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with "Return-less" kernels. In *Proc. 5th European Conf. on Computer Systems*, pages 195–208, 2010.

[39] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Proc. 15th USENIX Security Symposium*, pages 209–224, 2006.

[40] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically correcting memory errors with high probability. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 1–11, 2007.

[41] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-Free: Defeating return-oriented programming through gadget-less binaries. In *Proc. 26th Annual Computer Security Applications Conf.*, pages 49–58, 2010.

[42] Oracle Corporation. Position-independent code. In *Linker and Libraries Guide*. 2010. http://docs.oracle.com/cd/E19082-01/819-0690/chapter4-29405/index.html.

[43] P. O'Sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. D. Keromytis. Retrofitting security in COTS software with binary rewriting. In *Proc. Int. Information Security Conf.*, pages 154–172, 2011.

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

[45] PaX Team. PaX address space layout randomization (ASLR), 2003. http://pax.grsecurity.net/docs/aslr.txt.

[46] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *Proc. 25th Annual Computer Security Applications Conf.*, pages 60–69, 2009.

[47] B. Salamat, T. Jackson, A. Gal, and M. Franz. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proc. 4th ACM European Conf. on Computer Systems*, pages 33–46, 2009.

[48] J. Salwan. ROPgadget, 2012. http://shell-storm.org/project/ROPgadget.

[49] F. B. Schneider. Enforceable security policies. *ACM Trans. Information and Systems Security*, 3(1):30–50, 2000.

[50] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *Proc. 20th USENIX Security Symposium*, 2011.

[51] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proc. 14th ACM Conf. on Computer and Communications Security*, pages 552–561, 2007.

[52] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proc. 11th ACM Conf. on Computer and Communications Security*, pages 298–307, 2004.

[53] C. Small and M. I. Seltzer. A comparison of OS extension technologies. In *Proc. USENIX Annual Technical Conf.*, pages 41–54, 1996.

[54] M. Smithson, K. Anand, A. Kotha, K. Elwazeer, N. Giles, and R. Barua. Binary rewriting without relocation information. Technical report, U. Maryland, November 2010.

[55] Solar Designer. "return-to-libc" attack. Bugtraq, August 1997.

[56] B. D. Sutter, B. D. Bus, and K. D. Bosschere. Link-time binary rewriting techniques for program compaction. *ACM Trans. Programming Languages and Systems*, 27(5):882–945, 2005.

[57] A. van de Ven. New security enhancements in Red Hat Enterprise Linux v.3, update 3. Whitepaper WHP0006US, Red Hat, 2004. http://people.redhat.com/mingo/exec-shield/docs/WHP0006US_Execshield.pdf.

[58] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. 14th ACM Symposium on Operating Systems Principles*, pages 203–216, 1993.

[59] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham. Differentiating code from data in x86 binaries. In *Proc. European Conf. on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, volume 3, pages 522–536, 2011.

[60] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Proc. 22nd Int. Symposium on Reliable Distributed Systems*, pages 260–269, 2003.

[61] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *Proc. 30th IEEE Symposium on Security and Privacy*, pages 79–93, 2009.