ILR: Where'd My Gadgets Go?

Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, Jack W. Davidson
University of Virginia, Department of Computer Science
Charlottesville, VA
{*hiser,an7s,mc2zk,mh,jwd*}*@virginia.edu*

***Abstract*—Through randomization of the memory space and the confinement of code to non-data pages, computer security researchers have made a wide range of attacks against program binaries more difficult. However, attacks have evolved to exploit weaknesses in these defenses.**

To thwart these attacks, we introduce a novel technique called Instruction Location Randomization (ILR). Conceptually, ILR randomizes the location of *every* instruction in a program, thwarting an attacker's ability to re-use program functionality (e.g., arc-injection attacks and return-oriented programming attacks).

ILR operates on arbitrary executable programs, requires no compiler support, and requires no user interaction. Thus, it can be automatically applied post-deployment, allowing easy and frequent re-randomization.

Our preliminary prototype, working on 32-bit x86 Linux ELF binaries, provides a high degree of entropy. Individual instructions are randomly placed within a 31-bit address space. Thus, attacks that rely on *a priori* knowledge of the location of code or derandomization are not feasible. We demonstrated ILR's defensive capabilities by defeating attacks against programs with vulnerabilities, including Adobe's PDF viewer, `acroread`, which had an in-the-wild vulnerability. Additionally, using an industry-standard CPU performance benchmark suite, we compared the run time of prototype ILR-protected executables to that of native executables. The average run-time overhead of ILR was 13% with more than half the programs having effectively no overhead (15 out of 29), indicating that ILR is a realistic and cost-effective mitigation technique.

***Keywords*-Randomization; Exploit prevention; Diversity; ASLR; Return-oriented-programming, arc-injection;**

I. Introduction

Computer software controls many major aspects of modern life, including air travel, power distribution, banking, medical treatment, traffic control, and a myriad of other essential infrastructures. Unfortunately, weaknesses in software code (such as memory corruption, fixed-width integer computation errors, input validation oversights, and format string vulnerabilities) remain common. Via these weaknesses, attackers are able to hijack an application's intended control flow to violate security policies (exfiltrating secret data, allowing remote access, bypassing authentication, or eliminating services) [1–4].

Unfortunately, modern deployed defenses fail to thoroughly mitigate these threats, even when composed. Perhaps the most commonly deployed defenses are Address Space Layout Randomization (ASLR) [5] and W⊕X [5, 6]. In theory, ASLR randomizes the addresses used in a program. Unfortunately, only some addresses are randomized in modern implementations. For example, the main program text is not randomized on Linux implementations since programs do not have enough information to safely relocate this portion of code. Further, ASLR only randomizes the base address of loaded modules, not each address within the module. Thus, ASLR is vulnerable to information-leakage and entropy-exhausting attacks [7, 8]. W⊕X seeks to delineate code from data to prevent code-injection attacks. However, arc-injection attacks and various forms of return-oriented programming (ROP) attacks bypass W⊕X through reuse of code already embedded in the program [2, 8–10].

In this paper we describe a novel technique, called Instruction Location Randomization (ILR), that conceptually randomizes the location of *every* instruction in a program. ILR can use the full address space of the process (e.g., 32-bits on 32-bit processors such as the x86). Information leakage attacks that discover information about the location of a code block (e.g., the randomized base address of a dynamically loaded module or the start of a function) are infeasible for two reasons: 1) the randomized code addresses are protected from leakage and 2) a leak provides no information about the location of other code blocks.

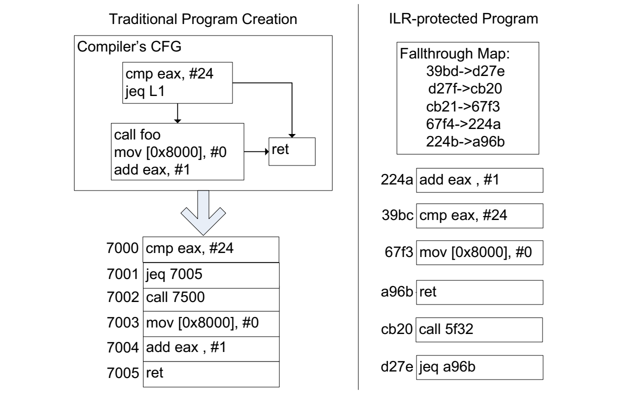
ILR changes a fundamental characteristic typically used by attackers—predictable code layout. For example, programs are arranged sequentially in memory starting at a base address, as shown in the left of Figure 1.[1]

In this example, the address used to return from function `foo` (7003) might be leaked if there is a vulnerability in the function. An attacker that learns this information can easily determine the location of all other instructions. Attackers routinely rely on the fundamental assumption of predictable code layout to craft attacks such as arc-injection and the various forms of return-oriented programming. In the example, an attacker might use the address of the `add` instruction to mount an ROP attack using `add eax, #1;ret` as an ROP gadget.[2] For a detailed explanation of

[1] For simplicity, the figure and discussion assume all instructions are one byte. Our general approach, prototype implementation, and security discussion do not rely on this fact.

[2] ROP gadgets are short sequences of code, typically ending in a return instruction, that perform some small portion of the attack.

DOI 10.1109/SP.2012.39



```
cmp eax, #24
jeq L1

call foo
mov [0x8000], #0
add eax, #1

ret
```

```
7000 cmp eax, #24
7001 jeq 7005
7002 call 7500
7003 mov [0x8000], #0
7004 add eax , #1
7005 ret
```

```
Fallthrough Map:
  39bd->d27e
  d27f->cb20
  cb21->67f3
  67f4->224a
  224b->a96b

224a add eax , #1
39bc cmp eax, #24
67f3 mov [0x8000], #0
a96b ret
cb20 call 5f32
d27e jeq a96b
```

Figure 1. Traditional program creation versus an ILR-protected program. In a traditional program, instructions are arranged sequentially and predictably, allowing an attack. With an ILR-protected program, instructions are distributed across memory randomly, preventing attack.

ROP gadgets and how they are combined to form an attack, please see Shacham's prior work [2].

ILR adopts an execution model where each instruction has an explicitly specified successor. Thus, each instruction's successor is independent of its location. This model of execution allows instructions to be randomly scattered throughout the memory space. Hiding the explicit successor information prevents an attacker from predicting the location of an instruction based on the location of another instruction.

ILR's "non-sequential" execution model is provided through the use of a process-level virtual machine (PVM) based on highly efficient software dynamic translation technology [11–13]. The PVM handles executing the non-sequential, randomized code on the host machine.

We have implemented a prototype ILR implementation for Linux on the x86 and Section III provides complete implementation details. In short, ILR operates on arbitrary executables, requires no compiler support, and no user interaction. Using a set of vulnerable programs (including a binary distributed by Adobe to read PDF files) and ASLR- and W$\oplus$X-defeating exploits, we demonstrate that ILR detects and thwarts these attacks. An important consideration of any mitigation technique is the run-time overhead. Many proposed mitigation techniques incur high overheads—as much as 90% to 2000% [14, 15]. Using a large industry-standard CPU performance benchmark suite [16], we compared the run time of ILR-protected executables to that of native executables. The average run-time overhead of ILR was 13% with over half of all programs having effectively no overhead (less than 3%) indicating that ILR is a realistic and cost-effective mitigation technique.

This paper makes several contributions. It:

- presents Instruction Location Randomization (ILR), a technique that provides high-entropy diversity for relocating instructions with low run-time overhead,
- demonstrates that ILR defeats arc-injection and ROP attacks on arbitrary binaries without need for compiler, linker, operating system or hypervisor support,
- provides a complete description of how ILR can achieve its goals despite inherent uncertainty about a program's structure, such as where code and data reside, and
- thoroughly analyzes the security, effectiveness, and performance of ILR in a prototype system on large, real-world benchmarks.

The remainder of the paper is organized as follows: Section II first discusses the threat model within which ILR operates. Section III describes the details of ILR. Sections IV and V provide an evaluation and security discussion of the proposed techniques. Section VI compares our work to related work in the field. Finally, Section VII summarizes our findings.

II. Threat Model

We assume that the unprotected program is created and distributed to an end user (and possibly the attacker) in binary form. The program has been tested, but not guaranteed to be free from programmatic errors that might allow malicious exploit, such as memory errors. The program is assumed to be free from intentionally planted back doors, trojans, etc. Furthermore, the program is to be protected and deployed in a setting where the other software on the system is believed to be operating correctly, and the system administrator is trusted. An attacker does not have direct access to the system or the protected program. However, the attacker understands the protection methodology and may have access to tools for applying ILR protections. The attacker also has access to the unprotected version of the program, and can specify malicious input to the protected program.

In particular, ILR focuses on preventing attacks which rely on code being located predictably. This threat model includes a large range of possible attacks against a program. For example, many attacks against client and server software fit this model. Document viewers/editors (Adobe PDF viewer, Microsoft Word), e-mail clients (Microsoft Outlook, Mozilla Thunderbird), and web browsers (Mozilla Firefox, Microsoft Internet Explorer, Google Chrome) need to be protected from these types of threats anytime a user requests the program to examine data from an untrusted source.

III. Instruction Location Randomization

ILR's goals are to achieve high randomization and low run-time overhead. Figure 1 conceptually illustrates the effect of ILR and how it mitigates malicious attacks. The top left of the figure shows the control-flow graph of a particular program segment. The compiler and the linker collaborate to produce an executable file where instructions are laid out so they can be loaded into memory when the program is executed. A typical layout of code is shown at the bottom left of the figure.

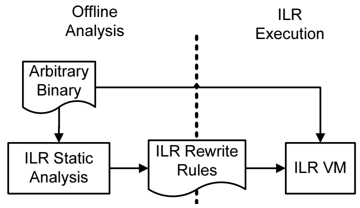


Figure 2. High-level overview of ILR architecture.

An attacker, through knowledge of the instruction-set architecture and the executable format, can easily locate portions of code that may be useful in crafting an attack. For example, the attacker may identify the instruction sequence at locations 7004 and 7005 as being a gadget useful in crafting an ROP attack. This particular gadget adds one to register `eax`. By identifying a set of gadgets and exploiting a vulnerability, an attacker can cause a set of gadgets to be executed that effect the attack.

The right side of the figure shows the layout of the code when ILR is applied. The program instructions are randomly scattered through memory. With an address space of 32 bits, it is infeasible for an attacker to locate a set of gadgets that could be used to craft an attack.

To execute the randomized program, we employ a highly efficient PVM that fetches and executes the instructions in the proper order even though they are randomly scattered throughout memory. This process is accomplished via a specification that describes the execution successor of each instruction in the program. This specification, called a fallthrough map, is shown at the top right of Figure 1. The PVM interprets the fallthrough map to fetch and execute instructions on the host hardware. The following subsections describe the process of automatically producing an ILR-protected executable and its execution.

A. ILR Architecture

Figure 2 shows the high-level architecture of the ILR process. ILR has an offline analysis phase to relocate instructions in the binary and generate a set of rewriting rules that describe how and where the newly located instructions are to be executed, and how control should flow between them, (shown as the fallthrough map in Figure 1). The randomized program is executed on the native hardware by a PVM that uses the fallthrough map to guide execution.

The rewriting rules come in two forms. The first form, the instruction definition form, indicates that there is an instruction at a particular location. The first line of Figure 3 gives an example. In this example, address $0x39bc$ has the instruction `cmp eax, #24`. Note that the rule indicates that an instruction fetched from address $0x39bc$ should be the `cmp` instruction. However, data fetches from address $0x39bc$ are unaffected. This distinction allows ILR to relocate instructions even if instructions and data are overlapped.

```
39bc ** cmp eax, #24
39bd -> d27e
d27e ** jeq a96b
d27f -> cb20
cb20 ** call 5f32
cb21 -> 67f3
67f3 ** mov [0x8000], 0
67f4 -> a96b
224a ** add eax, #1
224b -> 67f3
a96b ** ret
```

Figure 3. ILR rewrite rules corresponding to the example in Figure 1.

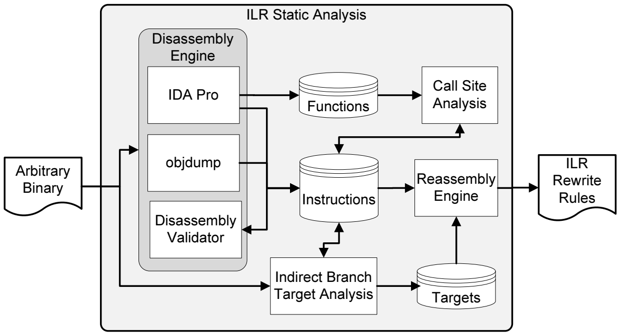


Figure 4. High-level overview of the static analysis engine used in ILR.

An example of the second form of an ILR rewrite rule, the redirect form, is shown in the second line of Figure 3. This line specifies the fallthrough instruction for the `cmp` at location $0x39bc$. A normal processor would immediately fetch from the location $0x39bd$ after fetching the `cmp` instruction. Instead, ILR execution checks for a redirection of the fallthrough. In this case, the fallthrough instruction is at $0xd27e$. The remaining lines show the full set of rewrite rules for the example in Figure 1.

The ILR architecture fetches, decodes and executes instructions in the traditional style, but checks for rewriting rules before fetching an instruction or calculating an instruction's fallthrough address.

B. Offline Analysis

The static analysis phase creates an ILR-protected program with random placement of every instruction in the program. For such randomization, the static analysis locates instructions, indirect branch targets, and identifies call sites for additional analysis. Figure 4 shows the organization of the static analysis used for ILR.

1) Disassembly Engine: The goal of the ILR disassembly engine is to locate any byte that might be the start of an instruction. We use a recursive descent disassembler (IDA Pro) and a linear scan disassembler (`objdump`) [17]. To ensure that all instructions are identified, we added the disassembly validator module. The disassembly validator iterates over every instruction found by either IDA Pro and `objdump`, and verifies that both the fallthrough and (direct) target instructions are inserted into the instruction database.

Since exact instruction start locations in the executable segment are not known, some of the instructions in the instruction database may not represent instructions that were intended by the program's original assembly code. We make no attempt to determine which are the intended instructions, and which are not. We simply choose to relocate all of them. Any data address that is mis-identified as a code address will not be executed, therefore the corresponding rewrite rules will simply never be accessed.

One last responsibility of the Disassembly Engine is to record the functions that IDA Pro detects. We record each function as a set of instructions.

2) Indirect Branch Target Analysis: The goal of the indirect branch target analysis phase is to detect any location in the program that might be the target of an Indirect Branch (IB). IBs create a distinct problem for ILR. Indirect Branch Targets (IBTs) may be encoded in the instructions or data of a program, and it is challenging to determine which program bytes represent an IBT and which do not. Since we wish to randomize any arbitrary binary, our technique must tolerate imprecision in detecting which constants are an IBT in the program and which are not. Our solution is to perform a byte-by-byte scan of the program's data, and further scan the disassembled code to determine any pointer-sized constant which could feasibly be an indirect branch target.

We find that in most programs, this simple heuristic is sufficient (see Section IV-D3 for details). However, when C++ programs use exception handling (`try`/`catch` blocks), the compiler uses location-relative addressing to encode IBTs for properly unwinding the stack, and invoking exception handlers. Our technique parses the portions of the ELF file that contain the tables used to drive the unwinding and exception throwing process, and records IBTs appropriately.

Rewriting the bytes in the program that encode an IBT might induce an error in the program if those bytes are used for something besides jumping to an instruction. To avoid breaking the program when the analysis is wrong, we choose to leave those program bytes unmodified. Unfortunately, not rewriting the IBTs encoded in the program means that the program might jump to the address of an original program (and hence unrandomized) instruction.

To accommodate indirect branches jumping to unrandomized addresses, each instruction that might be an IBT generates an additional ILR rule in the program. The additional rule uses the redirect form to map the unrandomized address to the new, randomized address. Thus, any indirect branch that targets an unrandomized address, correctly continues execution at the randomized address.

Unfortunately, attackers may know the unrandomized addresses in a program, and if they can inject a control transfer to one of these addresses, they might be able to successfully perform an attack. The evaluation in Section IV-D3 shows the number of IBTs detected in most programs is very limited, and restricting attacks to only these targets significantly reduces the attack surface.

3) Call Site Analysis: Since unrandomized instructions may allow attacks, we wish to randomize the return address for function calls. The call site analysis phase analyzes the call instructions in a program to determine if the return address can be randomized. Typically, a `call` instruction stores a return address, and when execution of the function completes, a `ret` instruction jumps to the address that was stored. Most functions obey these semantics. Unfortunately, call instructions can be used for other purposes, such as obtaining the current program counter when position-independent code or data is found in a library. Such a call instruction is often called a *thunk*. Numerous other uses of return addresses are possible.

The analysis proceeds as follows. If the call instruction is to a known location that starts a function, we analyze the function further. If the function can be analyzed as having only standard function exits (using the return instruction), having only entrances via the function's entry instruction, and having no direct accesses to the return value (such as with a `mov eax, [ebp+4]` instruction), then ILR declares that it is safe to rewrite the call instruction to store a randomized return address.

Our heuristic makes the assumption that indirect memory accesses should not access the return address. While not strictly true for all programs, we find that the heuristic generally holds for programs compiled from high-level languages. One exception to our heuristic is again the C++ exception handling routines that "walk the stack." The routines use the return address to locate the appropriate unwinding, cleanup, and exception handling codes to invoke. Like with the IBT analysis, we adjust the call site analysis to take into account the exception handling tables, so that call sites with exception handling cannot push a randomized return address.

Once the analysis is complete, the ILR rules for calls are emitted. If the call site analysis determines that the call can randomize the return address, no additional rules are required, and the call instruction's location is randomized by simply emitting the standard rewrite rules. If, however, the non-randomized return address must be stored, we have two choices: 1) we could choose to pin the call instruction to its original location, so that the nonrandomized return address is stored, or 2) rewrite the call (using ILR rewrite rules) into a sequence of instructions that stores the unrandomized return address and transfers control appropriately. Since pinning instructions leads to a decrease in randomization, we choose the second option. Most machines can efficiently store the return address and perform the control transfer necessary to mimic a call instruction, typically using only 2-3 instructions. For example, on the IA32 instruction set architecture, a `call foo` instruction can be replaced with two instructions, `push <unrandomized address>; jmp foo`, resulting in only one extra instruction. This transformation is exactly what is performed by our call site

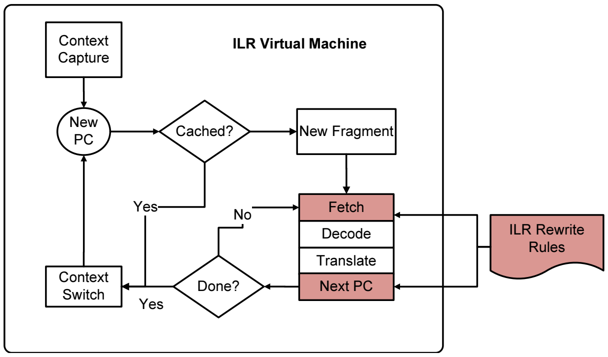


Figure 5. Details of the ILR Virtual Machine.

analysis when we detect that a call instruction cannot push a randomized return address. Furthermore, the unrandomized return address is marked as a possible indirect branch target, since we are not sure how the return address will be used.

4) Reassembly Engine: After completely analyzing the program's instructions, IBTs, and call sites, the reassembly engine gets invoked. The reassembly engine's purpose is to create the rewrite rules necessary to create the randomized program. For each instruction in the database, the engine emits a set of rewrite rules. First, it emits the rules necessary to relocate the instruction. Note that if the instruction has a direct branch target encoded in it (such as a `jmp L1`), that branch target is rewritten to the randomized address of the branch target. Then, the reassembly engine emits the rule to map the instruction's fallthrough address to the randomized location for the fallthrough instruction.

As a post-processing step, each byte of the original executable text gets an additional rule. If the address of the program text is marked as a possible IBT, the reassembly engine adds a rule to redirect that address to the randomized address for that instruction, effectively pinning the instruction. Any other byte of the executable code segment gets a rule to map its address to a handler that prints an error message and exits in a controlled manner. Thus, any possible arc-injection or ROP attacks must jump to the start of an instruction, and not bytes located within an instruction.

C. Running an ILR-protected Program

To apply the rewrite rules generated by the static analysis steps, ILR uses a specific ILR VM. We believe that a per-process virtual machine (PVM) is the best choice for the ILR VM since it can be easily deployed and has low performance and runtime overheads [11, 18, 19]. Figure 5 shows a typical PVM augmented with ILR extensions. The following paragraphs provide a brief introduction to typical PVM operation, and describe those extensions.

PVMs dynamically load an application and mediate application execution by examining and translating an application's instructions before they execute on the host CPU. Most PVMs operate as co-routines with the application that they are protecting. Translated application instructions are held in a PVM-managed cache called a fragment cache. The PVM is first entered by capturing and saving the application context (e.g., program counter (PC), condition codes, registers, etc.) Following context capture, the PVM processes the next application instruction. If a translation for this instruction has been previously cached, the PVM transfers control to the cached translated instructions.

If there is no cached translation for the next application instruction, the PVM allocates storage in the fragment cache for a new fragment of translated instructions. The PVM then populates the fragment by fetching, decoding, and translating application instructions one-by-one until an end-of-fragment condition is met. As the application executes under the PVM's control, more and more of the application's working set of instructions materialize in the fragment cache.

Implementation of ILR within a PVM requires several simple extensions to a typical PVM. First, we must modify the PVM startup code to read the ILR rewrite rules (not pictured). Next, we need to override the PVM's instruction fetching mechanism to first check, then read from ILR rewrite rules as appropriate. Lastly, we need to modify the next-PC operation to obey the fallthrough map that ILR provides in the rewrite rules.

One further extension is necessary for security. The PVM must take steps to protect itself and its code cache from being compromised by a program that an attacker is attempting to control. Since the PVM typically shares an address space with the program, the PVM must take care not to allow the program to attempt to jump into the PVM's code. Further, the PVM should prevent the randomized instruction addresses from being leaked to the user. Such protections can be accomplished by making the PVM's code and data unaccessible via standard memory protection mechanisms whenever the untrusted application code is executing. We discuss the technical details of one mechanism in Section V-A.

IV. Evaluation

A. Prototype Implementation

Our development and evaluation system were based on a Linux kernel version 2.6.32-34-generic as part of our Ubuntu 10.04.03 LTS release configured with gcc 4.4.3. We used IDA Pro version 6.1, and `objdump` version `2.20.1-system.20100303` [17].

As the static analyzer components need to store instructions, functions, and indirect branch targets, we used a Postgres database. This choice turned out to be wise considering some programs we evaluated contained almost half a million instructions. Each instruction is marked as being part of a function, and whether is has been detected as a possible indirect branch target.

We implemented the disassembly validator, call site analysis, indirect branch target analysis and reassembly engine to access the database and deposit their information back to the database. This modular design turned out to be useful for implementing, debugging, and deploying the system.

Our implementation of the reassembly engine is split into two phases. The first phase reads the database and emits a symbolic, relocatable, assembly version of the ILR rewrite rules to a file on the file system. The second step performs the randomization, and binds the assembly version of instructions to a machine code form. Splitting the tool into two portions aids in re-randomization (as the full database of instructions is no longer necessary) and run time (as the database need not be accessed at runtime).

Our ILR VM is based on Strata [11]. The modifications for ILR required about only 1K lines of code.

While our prototype implementation is based on the tools and operating system above, we believe our techniques are general, and can be easily applied to any hardware, operating system, PVM, or executable format.

B. Experimental Setup

We evaluated the effectiveness and performance of the ILR prototype using the SPEC CPU2006 benchmark suite [16]. These benchmarks are state-of-the-art, industry-standardized benchmarks designed to stress a system. The benchmarks are processor, memory and compiler stressing. The benchmarks are provided as source, and we compiled them with gcc, g++, or gfortran (as dictated by the program's source code) version 4.4.3 before applying our ILR technique. The benchmarks are compiled at optimization level -O2, and use static linking. We used static linking to thoroughly demonstrate the effectiveness of our system at randomizing large bodies of code, and to fully test the system using all the odd, compiler-specific, language-specific, hand-coded, or otherwise abnormal code that is often found in libraries. Furthermore, having all the code packaged into one executable increases the attack surface making it easier to locate an ROP gadget. Thus, we believe our evaluation is a worst-case analysis for these benchmarks.

We run our experiments on a system with a 4-core, AMD Phenom II B55 processor, running at 3.2 GHz. The machine has 512KB of L1 cache, 2MB of L2 cache, 6MB of L3 cache, and 4GB of main memory. Performance numbers are gathered by averaging 3 runs of each benchmark. Unless otherwise noted, the performance of a protected binary is reported by normalizing its run time to the run time of the corresponding original binary produced by the compiler.

C. Security-Related Experiments

To verify that our technique stops attacks that are successful against ASLR and W$\oplus$X protected systems, we performed a number of tests on vulnerable programs. For each test, ASLR and W$\oplus$X were enabled.

In the first test, we used a small program (44 lines of code) that had a simple stack-based buffer overflow. The program assigns grades to students based on the program's input, the student's name. A malicious input can cause a buffer overflow enabling an attack.

We created a simple arc-injection attack which causes the program to print out a grade of B when the student should receive a D. It was trivial to perform the arc-injection. ASLR was ineffective because no randomized addresses were used—only the unrandomized addresses in the main program. Similarly, W$\oplus$X was ineffective because the attack only relied on instructions that were already part of the program. We also used a tool called ROPgadget [20] to craft an ROP attack that causes the program to start a shell which can execute an arbitrary command. Again, ASLR and W$\oplus$X were ineffective. ILR, however, thwarted the attack.

We next verified our technique against a vulnerability in a realistic program: a Linux PDF viewer, `xpdf`. We seeded a vulnerability in the input processing routines. An appropriately long input can trigger a stack overflow. In this case, we were able to use ROPgadget to craft an attack to create a shell. ILR was again able to prevent the attack.

Lastly, we used version 9.3.0 of Adobe's PDF viewer, `acroread`, that we downloaded from Adobe's website in binary form. The program has a well-documented vulnerability when parsing image files (see CVE-2006-3459) that allows arc-injection and ROP attacks [21]. Again, we used ROPgadget to craft an ROP attack payload for this vulnerability to start a shell program. Because exploiting the vulnerability is more complicated, it took additional effort to adapt the attack. Using information from Security Focus's website, we were able to create a malicious PDF file that effected the ROP attack [21]. ILR successfully processes and randomizes the 24MB executable, and thwarts the attack.

Section IV-E discusses ILR's effect on the use of such tools as ROPgadget, and Section V-B describes how randomized addresses needed for the attack are protected from exfiltration by the ILR VM. Consequently, we believe attacks using programs such as ROPgadget are not possible with ILR.

D. Effectiveness of ILR Components

1) Disassembly Engine: The goal of the Disassembly Engine is to locate any instruction which might be executed, so that the instruction can be relocated later. For our benchmarks, we found that the disassembly engine successfully located 100% of the executed instructions for all benchmarks. The Disassembly Engine has met its first goal. We omit further discussion on disassembly as such techniques are well studied [22–24].

A secondary goal of the Disassembly Engine is to introduce few conflicting facts about instruction locations into the database. We measured the fraction of bytes in the executable segments that belonged to more than one instruction.

On average, only 0.005% of bytes were represented as part of more than one instruction with the worst-case having only 0.012% of bytes in conflict. Thus, we believe that the disassembly engine has met its second goal.

2) Call Site Analysis: Figure 6 shows the percentage of call sites marked as safe to randomize their return addresses. The first bar shows that our technique works well for some benchmarks. `403.gcc`, for example, has 91% of the return addresses randomized while `416.gamess` reaches 97%. Other benchmarks do not perform as well; `447.dealII` and `483.xalancbmk` only manage to identify 5% and 3% of return addresses as randomizable. The C++ benchmarks (`447.dealII`, `450.soplex`, `453.povray`, `470.lbm`, and `471.omnetpp`) do especially poorly. Only 10% of calls can use a randomized return address.

To understand why the call site analysis phase was less effective on some benchmarks, we examined the reasons that the call site analysis indicated that a randomized return address could not be used. Figure 7 shows the results as a fraction of all call instructions. We find that indirect calls (which cannot use a randomized return address because our analysis does not attempt to determine possible targets) result in a small fraction of unrandomized return addresses, resulting in 5% of calls on average. Possible non-standard uses of the return address, such as thunks, result in only 7.6% of return addresses. Interestingly, we find that direct call instructions to targets that we were not able to include in our disassembly result in 1.2% of the total call instructions. Closer inspection indicates that the compiler is actually emitting a `call 0x0` instruction in many library functions. If this type of call instruction were to ever execute, it would cause a fault in the program, but the call instruction is (dynamically) unreachable code. The compiler cannot detect this fact, and so cannot eliminate the call. A minor improvement would randomize the return address for this type of call, knowing that the return address cannot be used if the call instruction causes a fault. Together, these causes represent only 21% of all unrandomized call instructions.

The top bar in the figure shows the real cause of the poor performance, especially in C++ programs. More than 32% of call instructions are marked as not being able to randomize the return address because of the exception handling tables used in the ELF file. In the C++ programs, this number jumps up to an average of 79%! In C++ programs, the compiler typically cannot calculate when a function, f, makes a call, whether the called function will throw an exception and need to clean up f's stack. Consequently, the C++ compiler emits cleanup code into f, and adds to the `.eh_frame` and `.gcc_except_table` ELF sections to drive the exception handling routines. Because most functions with a call site fit this form, most call instructions cannot have a randomized return instruction.

It is interesting that even the C and Fortran benchmarks use the exception handling table. The C/Fortran benchmarks' application code does not seem to directly add to these tables. Instead, the table entries come from library routines that are compiled to work with C++ source.

We believe that modifying the ILR toolchain to edit the exception handling tables to reflect the randomization would be feasible. The tables are in a fixed, known format and can easily be rewritten with randomized addresses. Other solutions are possible as well. For example, detecting if C++ exception handling is actually used in the program or a portion of the program would allow return address randomization to be selectively applied. While fully exploring this idea is beyond the scope of this paper, we were able to modify our ILR toolchain to ignore the exception handling tables when calculating safe calls. We term the ILR toolchain with this modifications ILR+. ILR+ represents a very close approximation to a system that could easily be achieved by rewriting the exception handling tables in a binary.

With ILR+, the call site analysis performs well across all benchmarks. As Figure 6 shows, 93% of all calls are marked as using a randomized return address.

3) Indirect Branch Target Analysis: We continue our evaluation of ILR by measuring the effectiveness of the analysis of indirect branch targets (including return addresses). Figure 8 shows the fraction of instructions detected as possible indirect branch targets. On average, only 2.2% and 0.60% of the instructions are marked as indirect branch targets for ILR and ILR+, respectively. Consequently, we believe our scheme for detecting possible IBTs is not too aggressive in marking instructions as possible indirect branch targets.

4) Moved Instructions: Because we emit rewrites for every byte of the executable segment, technically all instructions are moved. However, IBTs get a rule that maps the unrandomized address to the relocated instruction. Despite technically being moved, we consider this an unmoved (or pinned) instruction because if an attacker were to inject an arc or locate an ROP gadget at the unrandomized address, they could still exploit that information in the randomized program.

Figure 9 shows the percentage of instructions moved for our benchmarks. The first bar shows the effectiveness of ILR without call site analysis; approximately 95.0% of instructions were successfully and safely located at randomized addresses. The second bar shows call site analysis for standard ILR; 97.4% of instructions are moved. The last bar shows the results for ILR+, almost all instructions (99.1%) are assigned to a randomized location in memory. This randomization represents a two order of magnitude reduction in the attack surface for arc-injection and ROP attacks.

E. ILR Security

To assess the security of ILR, we first note that up to 99.7% of the instructions can be randomized. Furthermore, all of the executable bytes of a program that do not make

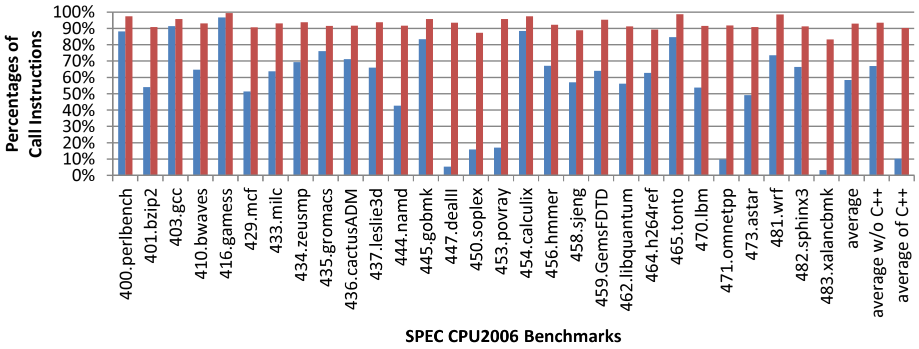


Figure 6. Percent of call instructions which ILR and ILR+'s call site analysis deemed safe for using a randomized return address. On average, only 58% of call instructions were identified as safe to use a randomized return address.

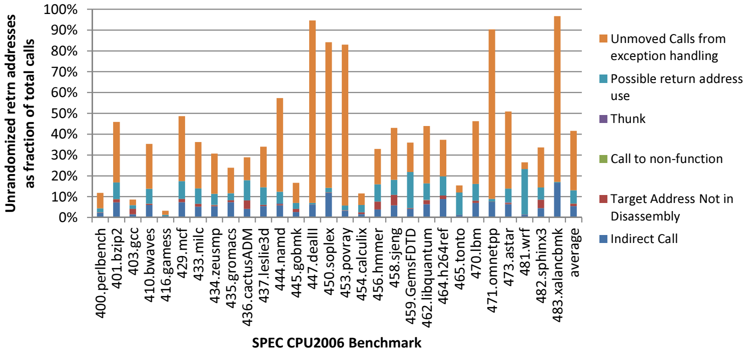


Figure 7. Breakdown of call instructions marked as unsafe for using a randomized return address. C++'s exception handling mechanism results in a severe reduction in return address randomization.

up a compiler-intended instruction sequence are marked as invalid for execution. These features of ILR reduce the attack surface for arc-injection by over two orders of magnitude. We believe it would be very difficult for an attacker to inject even one control-flow arc that achieves a meaningful result.

However, it has recently been shown that even small programs (with at least 20KB of program text) contain enough executable bytes to successfully produce an ROP attack [25]. The basic ILR algorithm reduces the unrandomized program text to less than 20KB for 26 of the 29 SPEC2006 benchmarks, while ILR+ reduces the attack surface to below 20KB for 28 of 29 benchmarks. On average, ILR+ reduces the attack surface to just 3KB! Thus, even state-of-the-art gadget compilers likely can not detect enough gadgets to mount an ROP attack in an ILR+-protected program.

To more directly validate that ILR successfully randomizes enough gadget locations to make ROP attacks infeasible, we further examine the SPEC benchmarks. While we know of no vulnerabilities in these benchmarks, they, like all large pieces of software, may in fact have an error that might allow an ROP attack. We study the feasibility of such an attack on these large applications if an appropriate vulnerability were to be found or seeded.

To search for gadgets in these benchmarks, we use a tool available online, ROPgadget [20]. The tool contains a database of gadget patterns and scans binary programs to identify specific gadgets within an executable. For example, one of the gadget patterns is `mov e?x, e?x;ret`, which identifies gadgets that move one register to another. We experiment with two versions of the tool, version 2.3 and 3.1. Version 2.3's database contains 60 gadget patterns, while version 3.1 has significantly more: 185 gadget patterns. Version 3.1 also contains a simple gadget compiler that matches gadgets with an attack template to form a complete attack payload. While these payloads do not automatically exploit a vulnerability in a program, they represent a significant portion of the attack. Converting an attack payload into an actual attack is dependent on the exact vulnerability, and is not automated. However, if ROPgadget cannot assemble the attack payload from the attack template, this failure indicates that the templated ROP attack could not proceed, even with a suitable vulnerability. ROPgadget 3.1 comes with two simple attack templates.

For the experiment, we modified both versions of ROPgadget to ignore randomized addresses, so that the tool can only locate gadgets at the unrandomized code addresses. This modification mimics an attacker's abilities via a remote

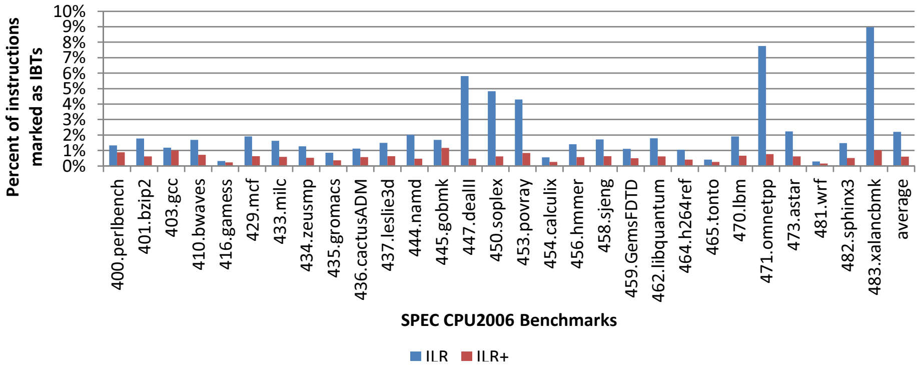


Figure 8. Percent of instructions marked as possible indirect branch targets. Only 2.2% and 0.60% of instructions are marked on average for the two techniques, indicating that ILR's IBT analysis is effective.

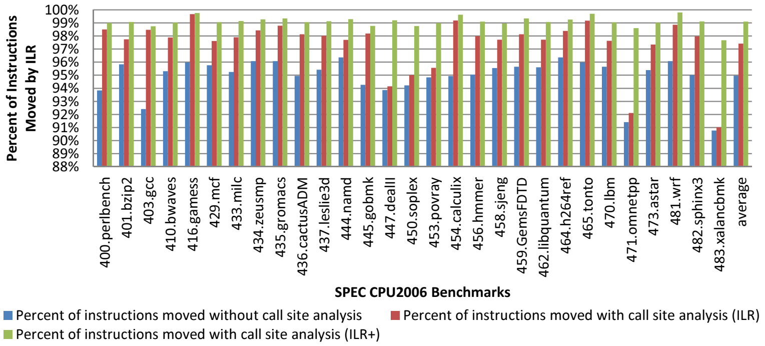


Figure 9. Percent of instructions moved using ILR. Results demonstrate that ILR can randomize the location of almost all instructions within an arbitrary binary program.

attack. Figure 10 shows how ILR affects an attacker's ability to mount an ROP attack. The first bar shows the percentage of unique gadgets that have been moved by ILR. We count unique gadgets because typically an attacker could re-use a gadget if needed, and any particular instance of a gadget is likely sufficient to mount an attack which used that gadget. Over 94% are moved on average, with `483.xalancbmk` being the worst performing at only 87%. The second bar shows the results for ROPgadget version 3.1. Even more of the gadgets appear to be hidden; over 90% in all cases, and 96% on average. What the figure does not show, however, is that version 3.1 located slightly more gadgets in the ILR-protected version, but found many more gadgets in the unprotected version, thus the overall ratio has improved, indicating that ILR is effective at hiding most gadgets in a program, even in the face of a better gadget identification framework. This result is quantified in the last bar of the figure where we count not unique gadgets, but all gadgets (including duplicates). On average, 99.96% of the total gadgets have had their location randomized.

On average, only 2.48 gadgets remain in the program. The worst performing benchmark, `483.xalancbmk`, has 6 unique gadgets, versus 67 for the unprotected program. Six gadgets is not enough to mount an attack in most cases. Even the two very simple attack templates included with ROPgadget require 8 and 9 gadgets. We note that on an unprotected application, the gadget compiler can successfully generate an attack payload for every program. In fact, both attacks are automatically detected as possible on 9 of the benchmarks. On the protected program, no attack payloads are ever successfully generated.

With ILR+ (results not shown) the probability of mounting an attack is further reduced. Most ILR+ protected applications have only one gadget (21 of 29 benchmarks). In every case, this lone gadget is an `int 0x80` sequence. Used alone, this gadget cannot mount an attack. On average, only 1.5 gadgets remain available with ILR+.

F. Performance Metrics

1) Run-time Overhead: Figure 11 shows the performance overhead of the base VM (Strata), as well as the overhead of ILR and ILR+. We see that Strata adds much of the overhead for the applications, and applying the randomization costs little additional overhead. On average, Strata adds only 8% overhead, with an additional 8% used for ILR. This extra overhead occurs in the short-running, but large code size benchmarks, for example, `400.perlbench`, `403.gcc`,

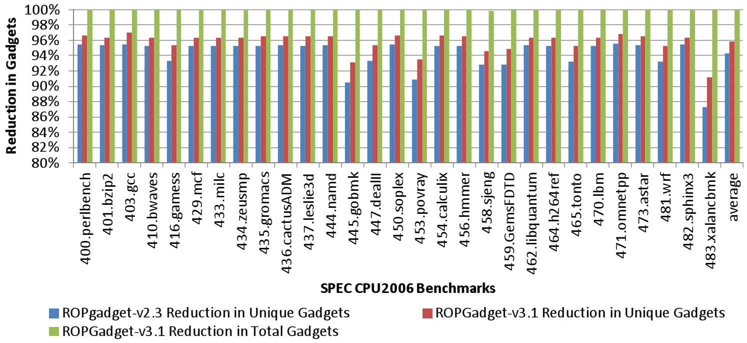


Figure 10. Reduction of number of gadgets found using ILR. Almost all gadgets are successfully randomized, and consequently unavailable for use in an attack.

and `416.gamess`). The overhead added is mostly due to the startup overhead of reading the rewrite rules. In `481.wrf` benchmark, for example, we note that reading the rewrite rules takes about 45 seconds, and that the 7% overhead difference between basic virtualization and ILR also corresponds to about 46 seconds. We believe that this startup overhead could be greatly reduced by a better rewrite rule format than ASCII. Section IV-F2 discusses optimizing the rewrite rules in more detail.

ILR+ actually reduces the overhead (by 3% to only 13%) compared to ILR. This reduction is due to more call sites being randomized. As mentioned in Section III-B3, storing an unrandomized return address takes one extra instruction. With more return addresses randomized, the instruction count is reduced. Because ILR+ has the largest effects on the C++ benchmarks, we see this difference most in the C++ benchmarks that are ILR+ compatible (`447.dealII`, `450.soplex`, and `483.xalancbmk`).

Taken together, we believe there is strong evidence that ILR can be implemented efficiently, perhaps as low as the basic virtualization overhead of only 8%. Even our prototype implementation, which has overheads of 13%-16% on average could be used to protect many applications.

2) Space Overhead: Our prototype implementation has memory overhead from two sources. The first is from the PVM we used to implement the ILR VM. Such overheads are well studied, and not particularly significant for modern systems [26, 27].

The second source of overhead is the handling of the ILR rewrite rules. In our prototype implementation, we made the design choice to use ASCII for the ILR rewrite rules. Our choice makes sense for an evaluation prototype: we favored human readability and ease of debugging over raw performance or storage efficiency. Consequently, we note that the on-disk size of the rewrite rules can be quite large. For example, the largest benchmark, `481.wrf`, has 264MB of rewrite rules. The in-memory size is even worse, 345MB. This overhead is largely due to our hashtable implementation that stores each byte of an instruction in a separate hash bucket, which allocates many words of data for each byte stored in an ILR rewrite rule. However, `481.wrf` is clearly a worst-case for our benchmarks. The average size of the rewrite rules (104MB) is less than half that for `481.wrf`.

While our prototype implementation is currently inefficient, we do not believe the rewrite rules are an inherent limitation of ILR. Many techniques exist for minimizing this overhead. For example, we used the gzip compression utility to compress the rewrite rules, and obtained an average size of 14MB. We believe that a binary encoding of the rewrite rules and an efficient memory storage technique could easily reduce the memory used to well under 14MB. On today's systems with multiple gigabytes of main memory, such space overhead should be easily tolerated.

3) Analysis Time: We measured the analysis time of the ILR technique. We were able to process the SPEC benchmarks in an average of 23 minutes each. Only the last step of the process creates any randomization, so most of that processing time can be re-used if one wanted to re-randomize. The randomization step itself took only 36 seconds, indicating that re-randomization once analysis is complete could proceed very quickly.

V. Security Discussion

A. Protecting the ILR VM

This section discusses several issues related to the security of the VM used to implement ILR.

The first issue that arises is the VM's potential for being vulnerable to an ROP or arc-injection attack. First, we note that the input to the VM is actually the program's instructions and the ILR rewrite rules, which we assume to be benign. Malicious programs or malicious rewrite rules are beyond the scope of our remote-attacker threat model. Benign programs and rewrite rules help, as that is the majority of input for the VM, but does not absolutely preclude an attacker from providing input to the program that somehow exercises a vulnerability in the VM. Still, we feel

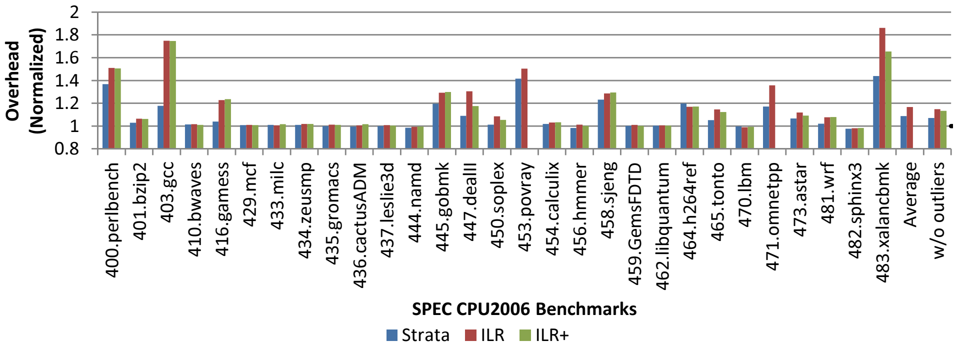


Figure 11. Performance overhead of ILR and ILR+, along with the average overhead and the average without the `453.povray` and `481.omnetpp` benchmarks. With an average overhead of only 16% and 13%, most applications could be reasonably protected by our ILR or ILR+ prototypes. Further, ILR overhead could be reduced to that of basic virtualization, at only 8%.

this threat is minimal, and could be addressed via a variety of techniques. We believe that formal verification should be possible since the VM's code is typically quite small. (Strata's fully-featured IA32 implementation is only 18K lines of code.) Much of the code is related to the decoder for the machine's ISA, which might be automatically verified or generated from an ISA description. Even without formal verification, bugs within a VM can largely be addressed via iterative refinement, code-review, static analysis, and compiler-based protection techniques. The last item has significant potential for protecting the VM in this case. If randomization (stack, heap, instruction-location, etc.) could be used on the VM at the deployed location, most attacks directly on the VM could be mitigated.

The more significant threat to the VM is that a vulnerability in the application allows the application's code to overwrite some portion of the VM, or to have the VM start interpreting some portion of itself. Since a process-level VM typically resides in the process' address space, we need to guard against these threats directly.

We do so by augmenting the VM to verify any instruction before it is fetched for analysis. The VM ensures that the instruction originates from allowable portions of the application text (for pinned instructions) or an ILR rewrite rule. The VM is prohibited from translating itself or its generated code, and consequently the VM's code cannot be used for arc-injection or ROP attacks. Our prototype implementation includes these protections.

To prevent a compromised application from overwriting the VM's code or data, we use standard hardware memory protection mechanisms. When executing the untrusted application code, the VM turns off read, write, and execute permission on memory used by the VM, leaving only execute (but not write) permission on the code cache. The VM also watches for attempts by the application to change these permissions. Previous work shows this technique to be effective and cost very little [28].

Together, we believe that good coding practices, verification, randomization, and actively protecting the VM from a compromised application can result in a safe VM.

B. *Entropy Exhausting Attacks*

The entropy of the ILR technique can be quite high. Since the ILR technique separates data and instruction memory, randomized instructions can be located anywhere in memory, even at the same addresses as program data, VM code or data. Many operating systems reserve some pages of memory specifically for code to interface with the operating system, so those pages could not be used for randomized addresses. Further, any unrandomized instructions restrict the entropy of the remaining instructions. Since there are very few unmoved instructions, and almost all other addresses are available for randomization, we believe that it would be easy to produce a system that has at least 31-bits of entropy on a 32-bit address system and at least 63-bits of entropy on a 64-bit system. Thus, randomly attempting to guess gadget addresses is completely infeasible and ILR can evade attacks which attempt to reduce the entropy of a system.

C. *Information Leakage Attacks*

A more likely attack scenario is that an attacker is able to leak information about randomized addresses. Fortunately, the memory-page protection techniques mentioned in Section V-A prevent leaking of information about most randomized addresses. The only randomized addresses that might be leaked are those that potentially end up in the application's visible data. For ILR, that is the randomized return addresses that might be stored on the application's stack. For a complete ILR+ implementation, it also includes any randomized addresses that are written into the application's exception handling tables.

In theory, all of these addresses might be leaked to an attacker. However, revisiting Figure 9, we see that on average only 5% of addresses in the total program could be known by the user. In practice, only a few randomized return addresses

```
lea  eax, [eax+eax*8+0x80b4545]
jmp  eax
```

Figure 12. An example of a calculated branch target from gcc's library for arbitrary precision arithmetic.

are available in the application at any instance, and most return addresses could not actually be leaked. If it were possible for the entire exception handling table to be leaked, the number of available addresses would likely be very close to the ILR results, and no ROP attacks are available against ILR in our benchmarks, as seen in Section IV-E.

Furthermore, since our ILR technique is designed to be applied to arbitrary executables, re-randomization could occur regularly with little overhead. Regular re-randomization of high-entropy systems has been shown to be effective in the context of information leakage [29].

Thus, information leakage is not a problem for ILR.

D. False Detections

A false detection occurs when the program performs an operation that is detected as illegal, when there is no attack underway. On our benchmark suite, we found that there were no false detections with ILR. Since our implementation of ILR+ is incomplete, we did observe two false detections. Both `453.povray` and `471.omnetpp` resulted in incorrect output (from faulting the program) when attempting to throw an exception. A complete implementation of ILR+ would not demonstrate this problem. We believe this indicates that false detections would be rare in real programs. Nonetheless, we discuss some possible mechanisms by which false detections might occur.

False detections might occur if a program calculates an indirect branch target, instead of simply storing the target in data memory as is most common. We found one example of this type of code in `gcc`'s library for doing arbitrary precision arithmetic. The example, shown in Figure 12 and originally written in assembly, is used to dispatch into a switch-style table of code blocks. Each block in the table is 9 bytes long. The assembly multiplies register `eax` by 9 (`eax+eax*8`), then adds the the base of the first code block before finally jumping to that address. A similar construct might be generated by a compiler, but we know of no compilers which generate this type of code for a switch statement. Other constructs exist that might hide code addresses. For example, a function pointer might be calculated for some reason, such as for obfuscation techniques.

A more common compiler construct that might calculate an indirect branch target is position independent code (PIC). In PIC mode, the compiler will often generate a code address by emitting a sequence of instructions that adds the current PC and a constant offset, knowing that the desired code address is a fixed distance from the current PC. PIC code is not standard due to its performance overhead.

In most of these cases, we believe that a more advanced indirect branch analysis would solve the problem. For example, the code in Figure 12 is prefixed by code to verify that register `eax` is in proper bounds. A simple range analysis on the values that can reach the `jmp` instruction would reveal the possible indirect branch targets.

Furthermore, our experience indicates that the ILR technique can easily print the address of an indirect branch target if a false detection is encountered. A profile-based or feedback-based mechanism that incorporates newly discovered IBTs would be easy to implement to reduce false detections over time if the IBT can be detected as derived only from safe sources.

E. Shared Libraries

Modern computer systems are built using libraries that are loaded on demand, and possibly shared among many processes. Linux uses the `.so` (Shared Object) format, while Windows uses the `.dll` (Dynamically Linked Library) model. Our system is capable of processing and randomizing a program that uses dynamic linking. Generally, analysis of these types of programs is easier for our system. Since the code is divided into libraries, we know that if a library contains a constant, the constant can only be an IBT in the library being considered, not to other libraries. Thus, this separation dramatically reduces the number of potential IBTs for a library. Furthermore, externally visible functions and symbols need to be referenced by a handle that is given in the library's headers. Extracting these types of indirect branch targets is trivial.

While our prototype can process and effectively randomize programs that require shared libraries, it does not actually randomize the libraries. Both Linux and Windows support some form of ASLR which provides coarse-granularity randomization of shared libraries. We believe our technique could easily be extended to include full randomization of shared libraries, but it is not clear that doing so would always be the best solution. When feasible, it seems better to provide randomization within the library itself. On Linux, this randomization could be accomplished by using a randomizing compiler to generate a per-system version of the libraries. When library source code is not available, such as on Windows-based systems, ILR-based randomization would be important. To achieve this, ILR-rewrite rules for a library would have to be loaded and symbolic addresses resolved whenever a new library entered the system. Such a mechanism could be easily included in a dynamic loader, or by having the ILR VM watch for library loading events.

F. Self-modifying Code

Our ILR implementation does not currently support self-modifing, dynamically generated, or just-in-time compiled (JITted) code because our underlying VM does not support such constructs. However, the ILR mechanism itself should operate properly with dynamically generated and JITted code, which is significantly more common than self-modifing code. ILR would not randomize the generated code, but we believe that to be an easy task for the JITter. A security-minded JITter would perform this simple operation.

VI. Related Work

A. ROP Defenses

The original authors of ROP have described ROP's salient feature as "Turing completeness without code injection" [9]. ROP invalidates the assumption that attack payloads are intrinsically external by nature as ROP re-uses code fragments already present in a target program. Defensive techniques such as various forms of instruction-set randomization that target code injection attacks directly are completely circumvented by arc-injection attacks in general [28, 30, 31], of which ROP, return-to-libc [32, 33], and partial overwriting attacks of return addresses [10] are special cases. W$\oplus$X is also circumvented as it implicitly assumes that external code will be executed from data pages [5].

Since the original seminal work on ROP [2], several defensive techniques have been proposed. Early defenses targeted what would emerge to be non-essential features of ROP attacks. For example, DROP [14] instruments binaries searching for short consecutive sequences of instructions ending in a return instruction. Li et al. and Onarlioglu et al. avoid gadget-like instruction sequences altogether when generating code [34, 35]. Kil et al. permute function locations to randomize gadget locations, but require additional compile-time information [36]. ROPDefender [15] and TRUSS [37] look for mismatched calls and returns essentially using a shadow stack.

Checkoway et al. showed that the use of the return instruction is not a necessary condition in building ROP gadgets, thereby bypassing such ad-hoc defenses [9]. The balance against ad hoc defenses is further tilted by recent works that have automated the process of gadget discovery [20, 38, 39] and ROP exploit compilation and hardening [25].

TRUSS [37], ROPDefender [15], DROP [14], and TaintCheck [40] use software dynamic translation frameworks for instrumenting code and implementing their respective defenses. TaintCheck uses dynamic taint analysis and provides a comprehensive approach to thwarting ROP attacks by detecting attempts at control-flow hijacking, though it suffers from high overhead (over 20X). Performance overhead for ROPDefender is approximately 2X overhead on the SPEC2006 benchmarks, while preliminary performances measurements for DROP range from 1.9X to 21X. While not directly comparable, ILR achieves average performance overhead of only 13-16%, which makes it practical for deployment.

B. Defenses based on randomization

In contrast to approaches that look for specific ROP patterns, ILR provides a comprehensive defense based on high-entropy diversification to thwart attacks. ILR provides 31 bits of entropy (out of a maximum of 32 for our experimental prototype) which makes derandomizing attacks impractical. ASLR on a 32-bit architecture only provides 16 bits of entropy and is susceptible to brute-force attacks [7]. Even on 64-bit architectures, there would be two potential problems. The first is that ASLR is not applied universally throughout the address space. Even when using dynamically-linked libraries, it is common for the main program text to start at a known fixed location. Red Hat developed Position Independent Executable to remedy this situation [41]. However, PIE requires recompilation. The second problem is that ASLR and other coarse-grained technology such as PIE do not perform intra-library randomization. Any information leaked as to the location of one function, or even one address, could be used to infer the complete layout of a library. Roglia et al. demonstrated a single-shot return-to-libc attack that used ROP gadgets to leak information about the base address of libc, and bootstrapped this information into all other libc functions [8]. Their proposed remedy of encrypting the Global Offset Table was specific to their attacks and leaves open the possibility of other leakage attacks.

Bhatkar et al. use source-to-source transformation techniques to produce self-randomizing programs (SRP) to combat memory error exploits [42]. Unlike other compiler-based randomization techniques [43], SRP produces a single program image, which makes it more practical for deployment. SRP randomizes code at the granularity of individual functions and therefore retains a larger attack surface than the ILR approach of randomizing at the instruction level.

Instruction Set Randomization (ISR) helps defeat code-injection attacks, but provides no protection against arc-injection and ROP attacks [28].

C. Control Flow Integrity

Control flow integrity (CFI) is designed to ensure the control flow of a program is not hijacked [44]. CFI relies on the Vulcan instrumentation system. The Vulcan system allows instruction discovery, static analysis, and binary rewriting.

Figure 13 shows an example program. In the figure, CFI enforces that the return instruction (in function `log`) can only jump to the instruction after a call to the `log` function. In this case, this policy allows an arc-injection attack if the `log` function is vulnerable. An attacker might be able to overwrite the return address to erroneously jump to L2, thereby granting additional access. Even the best static analysis cannot mitigate these threats using CFI.

Further, a partial overwrite attack might defeat ASLR in this example, since the distance between the two return sites

```
        call log
L1:     cmp [isRoot], #1
        jeq L3
        ...
        call log
L2:     call grantAccess
L3:     ...
log:    ...
        ret
```

Figure 13. Example demonstrating CFI's weakness. The return instruction jump either `L1` and `L2`, possibly allowing additional access if the `log` function is vulnerable.

is fixed. Since ILR randomizes this distance, ILR can defeat partial-overwrite attacks.

VII. Conclusions

This paper presents instruction location randomization (ILR), a high-entropy technique for relocating instructions within an arbitrary binary. ILR is shown to effectively hide 99.96% of ROP gadgets from an attacker, a 3.5 order of magnitude reduction in attack surface.

This work describes the general technique, as well as evaluates two versions of an ILR prototype. It further discusses the security implications of ILR. We find that ILR can be applied to a wide range of binary programs compiled from C, Fortran, and C++. Performance overhead is shown to be as low as 13% across the 29 SPEC CPU2006 industry-standard benchmarks [16].

This work surpasses state-of-the-art techniques for defeating attacks in a variety of ways. In particular, the technique:

- can be easily and efficiently applied to binary programs,
- provides up to 31 bits of entropy for instruction locations on 32-bit systems,
- can regularly re-randomize a program to thwart entropy-exhausting or information-leakage attacks,
- provides low execution overhead,
- randomizes statically and dynamically linked programs, and
- defeats attacks against large, real-world programs including the Linux PDF viewer, `xpdf`, and Adobe's PDF viewer, `acroread`.

Taken together, these results demonstrate that ILR can be used in a wide variety of real-world situations to provide strong protection against attacks.

Acknowledgment

This research is supported by National Science Foundation (NSF) grant CNS-0716446, the Army Research Office (ARO) grant W911-10-0131, the Air Force Research Laboratory (AFRL) contract FA8650-10-C-7025, and DoD AFOSR MURI grant FA9550-07-1-0532. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the NSF, AFRL, ARO, DoD, or the U.S. Government.

References

[1] J. Pincus and B. Baker, "Beyond stack smashing: Recent advances in exploiting buffer overruns," *IEEE Security & Privacy*, vol. 2, no. 4, pp. 20–27, Jul/Aug 2004.

[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*. ACM, 2007, pp. 552–561.

[3] 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*. ACM, 2008, pp. 27–38.

[4] D. Dai Zovi, "Practical return-oriented programming," *SOURCE Boston*, 2010.

[5] The PAX Team, http://pax.grsecurity.net.

[6] M. Howard and M. Thomlinson, "Windows vista ISV security," *Microsoft Corporation, April*, vol. 6, 2007.

[7] H. Shacham, M. Page, B. Pfaff, E. 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*. ACM, 2004, pp. 298–307.

[8] G. Roglia, L. Martignoni, R. Paleari, and D. Bruschi, "Surgically returning to randomized lib (c)," in *2009 Annual Computer Security Applications Conference*. IEEE, 2009, pp. 60–69.

[9] S. Checkoway, L. Davi, A. Dmitrienko, A. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*. ACM, 2010, pp. 559–572.

[10] T. Durden, "Bypassing PaX ASLR protection," *Phrack Magazine*, vol. 0x0b, no. 0x3b, 2002. [Online]. Available: http://www.phrack.org/issues.html?issue=59&id=9

[11] K. Scott, N. Kumar, S. Velusamy, B. R. Childers, J. W. Davidson, and M. L. Soffa, "Retargetable and reconfigurable software dynamic translation," in *International Symposium on Code Generation and Optimization*. San Francisco, CA: IEEE Computer Society, Mar. 2003, pp. 36–47.

[12] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system," in *SIGPLAN '00 Conference on Programming Language Design and Implementation*, 2000, pp. 1–12.

[13] M. Payer and T. Gross, "Fine-grained user-space security through virtualization," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, 2011, pp. 157–168.

[14] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "DROP: Detecting return-oriented programming malicious code," *Information Systems Security*, pp. 163–177, 2009.

[15] L. Davi, A. Sadeghi, and M. Winandy, "ROPdefender: A detection tool to defend against return-oriented programming

attacks," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 2011, pp. 40–51.

[16] Standard Performance Evaluation Corporation, "SPEC CPU2006 Benchmarks," http://www.spec.org/osg/cpu2006.

[17] (2011, November) Hex-rays website. [Online]. Available: http://www.hex-rays.com/products/ida/index.shtml

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM Press, 2005, pp. 190–200.

[19] M. Voss and R. Eigenmann, "A framework for remote dynamic program optimization," in *Proceedings of the ACM Workshop on Dynamic Optimization Dynamo '00*, 2000.

[20] "Shell storm website," http://www.shell-sorm.org/project/ROPgadget/.

[21] (2008) Libtiff tifffetchshortpair remote buffer overflow vulnerability. [Online]. Available: http://www.securityfocus.com/bid/19283

[22] A. Kapoor, "An approach towards disassembly of malicious binary executables," Ph.D. dissertation, University of Louisiana, 2004.

[23] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *Proceedings of the 13th USENIX Security Symposium*, 2004, pp. 255–270.

[24] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited," in *Proceedings of the 9th Working Conference on Reverse Engineering*. IEEE, 2002, pp. 45–54.

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

[26] J. Hiser, D. Williams, W. Hu, J. Davidson, J. Mars, and B. Childers, "Evaluating indirect branch handling mechanisms in software dynamic translation systems," in *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2007, pp. 61–73.

[27] A. Guha, K. Hazelwood, and M. Soffa, "Reducing exit stub memory consumption in code caches," *High Performance Embedded Architectures and Compilers*, pp. 87–101, 2007.

[28] W. Hu, J. Hiser, D. Williams, A. Filipi, J. Davidson, D. Evans, J. Knight, A. Nguyen-Tuong, and J. Rowanhill, "Secure and practical defense against code-injection attacks using software dynamic translation," in *Proceedings of the 2nd International Conference on Virtual Execution Environments*. ACM, 2006, pp. 2–12.

[29] A. Nguyen-Tuong, A. Wang, J. Hiser, J. Knight, and J. Davidson, "On the effectiveness of the metamorphic shield," in *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*. ACM, 2010, pp. 170–174.

[30] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanovic, "Randomized instruction set emulation," *ACM Transactions on Information System Security.*, vol. 8, no. 1, pp. 3–40, 2005.

[31] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *CCS '03: Proceedings of the 10th ACM Conference on Computer and Communications Security*. New York, NY, USA: ACM Press, 2003, pp. 272–280.

[32] S. Designer, ""return-to-libc" attack," *Bugtraq, Aug*, 1997.

[33] Nergal, "The advanced return-into-lib(c) exploits (PaX case study)." *Phrack Magazine, 58(4)*, December 2001.

[34] 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*, ser. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 195–208.

[35] 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*. ACM, 2010, pp. 49–58.

[36] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software," in *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. Ieee, 2006, pp. 339–348.

[37] S. Sinnadurai, Q. Zhao, and W. fai Wong, "Transparent runtime shadow stack: Protection against malicious return address modifications," 2008.

[38] T. Dullien and T. Kornau, "A framework for automated architecture-independent gadget search," in *4th USENIX Workshop on Offensive Technologies*, 2010.

[39] R. G. Roemer, "Finding the bad in good code: Automated return-oriented programming exploit discovery," 2009.

[40] D. S. James Newsome, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proceedings of the Network and Distributed System Security Symposium*, 2005.

[41] A. van de Ven, "New security enhancements in red hat enterprise linux v.3, update 3." Red Hat, Inc., 2004.

[42] S. Bhatkar, R. Sekar, and D. C. DuVarney, "Efficient techniques for comprehensive protection from memory exploits," in *Proceedings of the 14th Conference on USENIX Security Symposium*. USENIX Association, 2005.

[43] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Warner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz, "Compiler-generated software diversity," 2011.

[44] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*. ACM, 2005, pp. 340–353.