Leakage-Resilient Layout Randomization for Mobile Devices

Kjell Braden[†§], Stephen Crane[‡], Lucas Davi[†], Michael Franz[*] Per Larsen[*‡],
Christopher Liebchen[†], Ahmad-Reza Sadeghi[†],
[†]CASED/Technische Universität Darmstadt, Germany.
{lucas.davi,christopher.liebchen,ahmad.sadeghi}@trust.cased.de
[§]EURECOM, France. kjell.braden@eurecom.fr
[‡]Immunant, Inc. sjc@immunant.com
[*]University of California, Irvine. {perl,franz}@uci.edu

***Abstract*—Attack techniques based on code reuse continue to enable real-world exploits bypassing all current mitigations. Code randomization defenses greatly improve resilience against code reuse. Unfortunately, sophisticated modern attacks such as JIT-ROP can circumvent randomization by discovering the actual code layout on the target and relocating the attack payload on the fly. Hence, effective code randomization additionally requires that the code layout cannot be leaked to adversaries.**

Previous approaches to leakage-resilient diversity have either relied on hardware features that are not available in all processors, particularly resource-limited processors commonly found in mobile devices, or they have had high memory overheads. We introduce a code randomization technique that avoids these limitations and scales down to mobile and embedded devices: Leakage-Resilient Layout Randomization (LR^2).

Whereas previous solutions have relied on virtualization, x86 segmentation, or virtual memory support, LR^2 merely requires the underlying processor to enforce a W$\oplus$X policy—a feature that is virtually ubiquitous in modern processors, including mobile and embedded variants. Our evaluation shows that LR^2 provides the same security as existing virtualization-based solutions while avoiding design decisions that would prevent deployment on less capable yet equally vulnerable systems. Although we enforce execute-only permissions in software, LR^2 is as efficient as the best-in-class virtualization-based solution.

I. Motivation

The recent "Stagefright" vulnerability exposed an estimated 950 million Android systems to remote exploitation [21]. Similarly, the "One Class to Rule them All" [40] zero-day vulnerability affected 55% of all Android devices. These are just the most recent incidents in a long series of vulnerabilities that enable attackers to mount code-reuse attacks [37, 43] against mobile devices. Moreover, because these devices run scripting capable web browsers, they are also exposed to sophisticated code-reuse attacks that can bypass ASLR and even fine-grained code randomization by exploiting information-leakage vulnerabilities [11, 20, 48, 50]. Just-in-time attacks (JIT-ROP) [50] are particularly challenging because they misuse run-time scripting to analyze the target memory layout after randomization and relocate a return-oriented programming (ROP) payload accordingly.

There are several alternatives to code randomization aimed to defend against code-reuse attacks, including control-flow integrity (CFI) [1] and code-pointer integrity (CPI) [28]. However, these defenses come with their own set of challenges and tend to have high worst-case performance overheads. We focus on code randomization techniques since they are known to be efficient [18, 25] and scalable to complex, real-world applications such as web browsers, language runtimes, and operating system kernels without the need to perform elaborate static program analysis during compilation.

Recent code randomization defenses offer varying degrees of resilience to JIT-ROP attacks [4, 6, 14, 15, 20, 22, 31, 35]. However, all of these approaches target x86 systems and are, for one reason or another, unfit for use on mobile and embedded devices, a segment which is currently dominated by ARM processors. This motivates our search for randomization frameworks that offer the same security properties as the state-of-the-art solutions for x86 systems while removing the limitations, such as dependence on expensive hardware features, that make them unsuitable for mobile and embedded devices.

The capabilities of mobile and embedded processors vary widely. For instance, many micro-processors do not have a full memory management unit (MMU) with virtual memory support. Instead they use a memory protection unit (MPU) which saves space and facilitates real-time operation[1]. Processors without an MMU can therefore not support defenses that require virtual memory support [4, 14, 15, 22]. High-end ARM processors contain MMUs and therefore offer full virtual memory support. However, current ARM processors do not support[2] execute-only memory (XoM) [2] which is a fundamental requirement for randomization-based defenses offering comprehensive resilience to memory disclosure [14, 15].

Therefore, our goal is to design a leakage-resilient layout randomization approach, dubbed LR^2, that enforces XoM purely in *software* making our technique applicable to MMU-less hardware as well. Inspired by software-fault isolation techniques (SFI) [45, 47, 53], we enforce XoM by masking load addresses to prevent the program from reading from any code addresses. However, software-enforced XoM is fundamentally different from SFI: First, XoM protects trusted code

[1]MPUs can still enforce W$\oplus$X policies for a given address range.

[2]Firmware executed from non-volatile storage can be marked as execute-only. Code executing out of RAM cannot be marked execute-only on current processors.

NDSS '16, 21-24 February 2016, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2016.23364

that is executing as intended whereas SFI constrains untrusted code that may use return-oriented programming techniques to execute instruction sequences in an unforeseen manner to break isolation of the security sandbox. We take advantage of these completely different threat models to enforce XoM in software using far fewer load-masking instructions than any SFI implementation would require; Section IV-B provides a detailed comparison. A second key difference between SFI approaches and LR^2 is that we hide code pointers because they can otherwise lead to indirect leakage of the randomized code layout. Code pointers reveal where functions begin and return addresses reveal the location of call-preceded gadgets [19, 24]. We protect pointers to functions and methods (forward pointers) by replacing them with pointers to trampolines (direct jumps) stored in XoM [14]. We protect return addresses (backward pointers) using an optimized pointer encryption scheme that hides per-function encryption keys on XoM pages.

Thanks to software-enforced XoM, LR^2 only requires that the underlying hardware provides code integrity by enforcing a writable XOR executable (W⊕X) policy. This requirement is met by all recent ARM processors whether they have a basic MPU or a full MMU. Support for W⊕X policies is similarly commonplace in recent MIPS processors.

In summary, our paper contributes:

- LR^2, the first leakage-resilient layout randomization defense that offers the full benefits of execute-only memory (XoM) without any of the limitations making previous solutions bypassable or unsuitable for mobile devices. LR^2 prevents *direct* disclosure by ensuring that adversaries cannot use load instructions to access code pages and prevents *indirect* disclosure by hiding return addresses and other pointers to code.
- An efficient return address hiding technique that leverages a combination of XoM, code randomization, XOR encryption, and the fact that ARM and MIPS processors store return addresses in a *link register* rather than directly to the stack.
- A fully-fledged prototype implementation of our techniques capable of protecting Linux applications running atop ARM processors.
- A detailed and careful evaluation showing that LR^2 defeats a real-world JIT-ROP attack against the Chromium web browser. Our SPEC CPU2006 measurements shows an average overhead of 6.6% which matches the the 6.4% overhead for a comparable virtualization-based x86 solution [14].

II. Background

After W⊕X policies became commonplace, code reuse replaced code injection as the key exploitation technique. At first, attackers reused whole functions in dynamically linked libraries [37] but later switched to return-oriented programming (ROP) [43] that reuses short instruction sequences ending in returns (gadgets). Most recently, Schuster et al. [46] introduced counterfeit object-oriented programming (COOP), a technique that reuses C++ virtual methods to bypass many control-flow integrity (CFI) [1] and code randomization defenses [29].

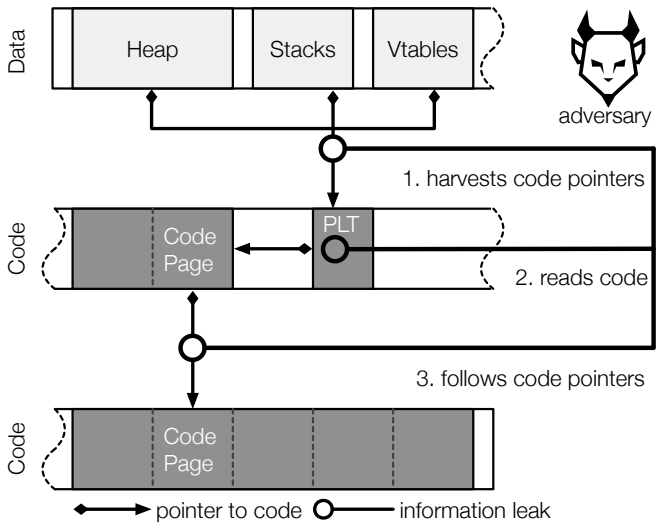

Figure 1: Memory regions and pointers between them. Generalized JIT-ROP harvests pointers from the heap, stack, or other data pages to code pages (step 1). The original JIT-ROP attack [50] recursively reads and disassembles code pages (steps 2-3). Indirect JIT-ROP attacks [11, 20] omits these steps.

ASLR, address space layout permutation (ASLP) [27], and other types of code randomization greatly increase resilience to code reuse by hiding the address space layout from adversaries. However, the results of randomization can be disclosed using information-leakage vulnerabilities [7, 20, 48, 49, 50, 52]. The just-in-time code-reuse (JIT-ROP) techniques [20, 50] are particularly powerful as they use malicious JavaScript to overflow a buffer, access arbitrary memory, and analyze the randomized layout of the victim browser process. Figure 1 illustrates ways that adversaries disclose and analyze memory contents by constructing a read primitive out of a corrupted array object. In step one, pointers to code are harvested from the heap, stacks, virtual method tables (vtables), and any other data that can be located by the adversary. Heaps, for instance, contain function pointers and C++ objects that point to vtables. These in turn point to C++ virtual methods. Stacks predominantly contain pointers to call-preceded locations inside functions. In step two, the adversary uses these pointers to locate and read code pages directly and, in step three, follows references to other code pages recursively until all necessary gadgets have been located. Early defenses against JIT-ROP made references between code pages opaque [3] or emulated execute-only memory [4, 22]. However, the *indirect* JIT-ROP attack [20] shows that the initial pointer harvesting step is sufficient to launch JIT-ROP attacks against code randomized at the level of functions [27] or code pages [3]. Building on these lessons, the Readactor [14] approach prevents all three memory leakage steps in Figure 1 by combining XoM with a pointer indirection mechanism known as code-pointer hiding (CPH). With CPH, all code pointers in readable memory are replaced with *trampoline pointers* that point into an array of direct jumps (*trampolines*) to functions and return sites. Trampolines cannot be used to indirectly disclose the code layout because trampolines are randomized and stored in XoM.

While the Readactor approach offers leakage resilience, it targets high-end x86 systems that can support XoM natively which precludes deployment on mobile devices. Moreover,

their code-pointer hiding component requires additional computational and storage resources. Our LR^2 solution, described in Section IV, provides the same security but significantly reduces the associated resource and hardware requirements.

III. Adversary Model

We use the following adversary model:

- The adversary cannot compromise the protected program at compile or load-time. Therefore the adversary has no a priori knowledge of the code layout.
- The underlying hardware enforces a W⊕X policy which prevents code injection. Note that even low-end devices that have an MPU (rather than an MMU) are able to meet this requirement.
- At run time, the attacker can read and write data memory such as the stack, heap and global variables. This models the presence of memory corruption errors that allow control-flow hijacking and information leakage.
- Attacks against the underlying hardware or operating system fall outside the scope of this paper. This includes any attack that uses timing, cache, virtual machine, or fault side channels to disclose the code layout.

Our adversary model is consistent with prior research on leakage-resilient layout randomization [4, 6, 14, 15, 20, 22].

IV. LR^2

Like similar defenses, LR^2 consists of a series of code transformations. We prototype these transformations as compiler passes operating on source code. Compile-time transformation is not fundamental to our solution. The same approach could be applied by rewriting the program on disk or as it is being loaded into memory.

We perform the following transformations:

- **Load masking** to enforce XoM in software (Section IV-A). XoM prevents direct disclosure of the code layout and forms the basis for the following transformations. We describe conventional and novel optimizations for efficient instrumentation in Section IV-B.
- **Forward-pointer hiding** (Section IV-C). We replace forward pointers to functions and virtual methods with pointers into an array of trampolines, i.e., direct jumps to the original pointer address, stored in XoM to prevent *indirect* disclosure similar to Crane et al. [14].
- **Return-address hiding** (Section IV-D). While we could have hidden return addresses in the same way as we hide forward code pointers, this approach is sub-optimal. First, the return address trampolines (a call and a jump) take up more space than trampolines for forward code pointers (a single jump). Second, this naive approach would require a trampoline between each caller and callee which further increases the memory overhead.
- **Fine-grained code randomization** (Section IV-E). The preceding techniques prevent disclosure of the code layout, so we must evaluate our system in conjunction with fine-grained diversity techniques.

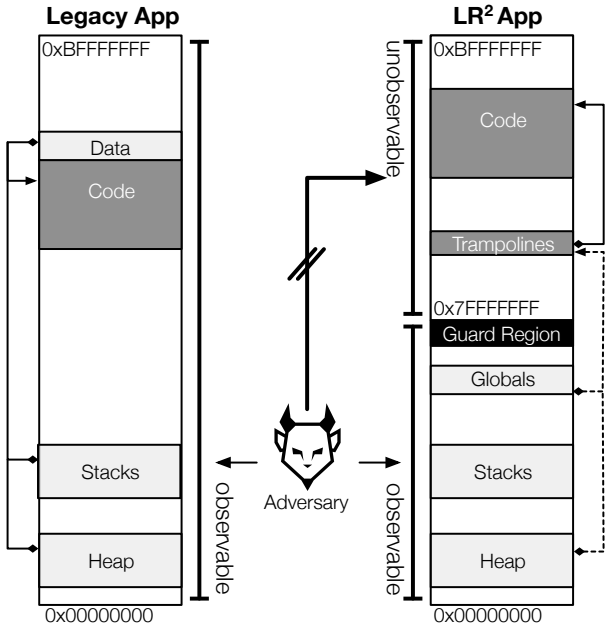

Figure 2: Left: In legacy applications, all pages are observable by attackers. The stack, heap and global areas contain pointers that disclose the location of code pages. Right: In LR^2 applications, attackers are prevented from observing the upper half of the address space which contains all code. Moreover, attacker observable memory only contains trampoline pointers (dotted arrows) that do not disclose code locations. Finally, return addresses on the stack are encrypted (not shown).

We describe each of these components in detail in the following subsections, along with our prototype LLVM-based toolchain, including dynamic loading and full protection of system libraries.

A. Software-Enforced XoM

On ARM and other RISC instruction sets, all reads from memory use a load instruction (`ldr` on ARM). To enforce XoM purely in software (to avoid reliance on MMU features), we prevent all memory loads from reading program code. We enforce this property by 1) splitting the program code and data memory into separate memory regions and 2) by ensuring that no load instruction can ever access the code region. We mask every attacker-controlled address that may be used by a load instruction to prevent it from addressing a code page.

We split the virtual memory address space into two halves to simplify load address masking; data resides in the lower half of the address space and code in the upper half (see the right side of Figure 2). Note that we include a *guard region* which consists of 2 memory pages marked as non-accessible. The guard region allows us to optimize loads that add a small constant offset to a base address. With this split, our run-time instrumentation simply checks the most significant bit (MSB) of the address to determine whether it points to data or code. All valid data addresses (and thus all safe memory loads) must have a zero MSB.

Since we enforce a memory-access policy rather than program integrity in the face of memory corruption, we can optimize our checks to fail safely if the program attempts to read a code address. The ARM instruction set has two options we can use to enforce efficient address checks: the bit clear instruction (`bic`) or a test instruction (`tst`) followed by a predicated load. Either clearing or checking the MSB of the address before a load ensures that the load will never read from the code section. The program may still behave incorrectly if the attacker overwrites an address, but the attacker cannot read any execute-only memory.

The following code uses `bic` masking instrumentation which clears the MSB of the address before accessing memory. This instrumentation is applicable to all load instructions.

```
bic     r0, r0, #0x80000000
ldr     r1, [r0]
```

Listing 1: `bic` masking example

The `tst` masking shown below instead avoids a data dependency between the masking instruction and the load by predicating the load based on a test of the MSB of the address. If an attacker has corrupted the address to point into the code section, the load will not be executed at all since the test will fail. The `tst` masking has the added benefit that we can handle failure gracefully by inserting instrumentation which jumps to an address violation handler in case of failure. However, `tst` is not applicable to loads which are already predicated on an existing condition. In addition, we found that the `bic` masking is up to twice as efficient as `tst` masking on our test hardware, even with the data dependency. One possible reason for this is that the predicated instruction will be speculatively executed according to the branch predictor, causing a pipeline discard in the case of a misprediction. At the same time, `bic` masking benefits greatly from out-of-order execution if the load result is not immediately required.

```
tst     r0, #0x80000000
ldreq   r1, [r0]
```

Listing 2: `tst` masking example

B. Optimized Load Masking

Masking addresses before every load instruction is both redundant and inefficient as many loads are provably safe. To optimize our instrumentation, we omit checks for loads that we can guarantee will never read an unconstrained code address. We start with similar optimizations to previous work, including optimizations adapted specifically for ARM, and then discuss a novel optimization opportunity that is not applicable to any SFI technique.

a) SFI-Inspired Optimizations: We perform several optimizations mentioned by Wahbe et al. [53] in their seminal work on SFI. We allow base register plus small constant addressing by masking only the base register, avoiding the need for an additional address computation `add` instruction. We also allow constant offset stack accesses without needing checks by ensuring that the stack pointer always points to a valid address in the data section. All stack pointer modifications with a non-constant operand are checked to enforce this property.

Additionally, we do not constrain program counter relative loads with constant offsets. ARM does not allow for 32-bit immediate instructions operands, and therefore large constants are stored in a constant pool allocated after each function. These constant pools are necessarily readable data in the code section, but access to the constant pool is highly constrained. All constant pool loads use a constant offset from the current program counter and therefore cannot be used by attackers to access the surrounding code.

b) XoM-Specific Optimizations: Although software XoM is inspired by SFI, the two techniques solve fundamentally different problems. SFI isolates potentially malicious code whereas software XoM constrains benign code operating on potentially malicious inputs. In other words, SFI must operate on the assumption that the adversary is already executing untrusted code in arbitrary order whereas software XoM trusts the code it instruments and therefore assumes that the control-flow has not yet been hijacked.

Since we trust the executing code, we can make optimizations to our software XoM implementation that are not applicable when performing traditional SFI load masking. Specifically, we do not need to mask load addresses directly before the load instruction. Instead, we insert the masking operation directly after the instructions that compute the load address. In many cases, a single masking operation suffices to protect multiple loads from the same base address. Registers holding the masked address may be spilled to the stack by the register allocator. Since the stack contents are assumed to be under attacker control (Conti et al. [11] recently demonstrated such an attack), we re-mask any addresses that are loaded from the stack. In contrast, SFI requires that address checks remain in the same instruction bundle as their use, so that a malicious program may not jump between the check and its use. In our experiments, the ability to hoist masking operations allows us to insert 43% fewer masking operations relative to SFI policies that must mask each potentially unsafe load in untrusted code. Figure 3 shows an example in which we are able to remove a masking operations in a loop which substantially reduces the number of `bic` instructions executed from $2n + 1$ to $n + 1$ where n is the number of loop iterations.

C. Forward-Pointer Hiding

As explained in Section II, adversaries can scan the stack, heap, and static data areas for code pointers that indirectly disclose the code layout. We therefore seek ways to identify functions and return sites without revealing their location. The first major category of code pointers are function pointers, used by the program for indirect function calls. Closely related are basic block addresses used in situations such as switch case tables. We handle all forward code pointers in the same manner but use a special, optimized scheme for return addresses as explained in the following section.

We protect against an attacker using forward code pointers to disclose code layout by indirecting all code pointers through a randomized trampoline table, as proposed by Crane et al. [14]. For each code location referenced by a readable code pointer, we create a trampoline consisting of a direct jump to the target address. We then rewrite all references to the original address to refer instead to the trampoline. Thus, the trampoline

```
1  ; calculate address
2  add r0, r0, r8
3  ; store address on stack
4  str r0, [sp+#12]
5
6  loop:
7  bic r0, r0, #0x80000000
8  ; load address
9  ldr r1, [r0]
10 bic r0, r0, #0x80000000
11 ; load + constant offset
12 ldr r2, [r0+#4]
13 add r0, r0, #8
14 ; check loop condition
15 cmp r0, r3
16 bne loop
17
18 loopend:
19 ; restore address from
20 ;  stack, now unsafe
21 ldr r0, [sp+#12]
22 bic r0, r0, #0x80000000
23 ; load address
24 ldr r2, [r0]
```

Software-Fault Isolation

```
1  ; calculate address
2  add r0, r0, r8
3  ; store address on stack
4  str r0, [sp+#12]
5
6  loop:
7  bic r0, r0, #0x80000000
8  ; load address
9  ldr r1, [r0]
10
11 ; load + constant offset
12 ldr r2, [r0+#4]
13 add r0, r0, #8
14 ; check loop condition
15 cmp r0, r3
16 bne loop
17
18 loopend:
19 ; restore address from
20 ;  stack, now unsafe
21 ldr r0, [sp+#12]
22 bic r0, r0, #0x80000000
23 ; load address
24 ldr r2, [r0]
```

Software XoM

Figure 3: Differences between load-masking for software-fault isolation (left) and software-enforcement of XoM (right). Because SFI must consider existing code malicious, it must mask load addresses directly before every use. In contrast, software XoM is protecting trusted code executing legitimate control-flow paths, and can therefore use a single masking operation to protect multiple uses.

address, rather than the function address, is stored in readable memory. We randomize trampoline ordering to remove any correlation between the address of the trampoline (potentially available to the attacker) and the actual code address of the target. Hence, even if an attacker leaks the address of a trampoline, it does not reveal anything about the code layout.

D. Return-Address Hiding

In principle, we could hide return addresses using the same trampoline mechanism that we use to protect forward pointers. However, the return address trampolines used by Crane et al. [14] require two instructions rather than the single direct jump we use for forward pointers. At every call site, the caller jumps to a trampoline containing 1) the original call instruction, and 2) a direct jump back to the caller. This way, the return address that is pushed on the stack points into a trampoline rather than a function. However, due to the direct jump following the call, every call site must use a unique return address trampoline.

Return addresses are extremely common. Thus, the extra trampoline indirections add non-trivial performance overhead. Additionally, code size is critical on mobile devices. For these reasons, we take an alternative approach. Due to the way ARM and other RISC instruction sets perform calls and returns, we can provide significantly stronger protection than the return address trampolines of Crane et al. without expensive trampolines for each call site. We build upon the foundation of XoM to safely secure an unreadable, per-function key to encrypt every return address stored on the stack.

While x86 call instructions push the return address directly onto the stack, the branch and link instruction (`bl`) on ARM and other RISC processors instead places the return address in a link register. This gives us an opportunity to encrypt the return address when it is spilled onto the stack[3]. We XOR all return addresses (stored in the link register) before they are pushed on the stack similarly to the PointGuard approach by Cowan et al. [13]. PointGuard, however, uses a much weaker threat model. It assumed that the adversary cannot read arbitrary memory. In our stronger attacker model (see Section III), we must prevent the adversary from disclosing or deriving the stored XOR keys. We therefore use a per-function key embedded as a constant in the code which, thanks to XoM, is inaccessible to adversaries at run time. In our current implementation, these keys are embedded at compile time. As this might be vulnerable to offline analysis, we are currently working on extending LR^2 to randomize the keys at load time.

Listing 3 shows an example of our return-address hiding technique. Line 2 loads the per-function key for the current function, and on line 3 it is XORed into the current return address before this address is spilled to the stack in line 4. Lines 8-11 replace the normal `pop {pc}` instruction used to pop the saved return address directly into the program counter. On lines 8-10, the encrypted return address is popped off the stack and decrypted, and on line 11 the program branches to the decrypted return address.

Considering the advantages of protecting return addresses using XOR encryption, the question arises whether forward pointers can be protected with the same technique. An important difference between forward pointers and return addresses is that the former may cross module boundaries. For instance, an application protected by LR^2 may pass a pointer to an unprotected library or the OS kernel to receive callbacks. The

[3]Leaf functions do not need to spill the return address onto the stack.

```
1  function:
2   ldr     r12, .FUNCTION_KEY
3   eor     lr, lr, r12
4   push    {lr}
5
6  [ function contents here ]
7
8   pop     {lr}
9   ldr     r3, .FUNCTION_KEY
10  eor     lr, lr, r3
11  bx      lr
12
13 .FUNCTION_KEY:   ; constant pool entry, embedded
14 .long            ; in non-readable memory
15  0xeb6379b3
```

Listing 3: Return-address hiding example. Note that constant pool entries are embedded in non-readable memory, as described in Section IV-B.

trampoline mechanism used for forward pointers ensures transparent interoperability with unprotected code while XOR encryption does not without further instrumentation, since legacy code would not know that forward pointers are encrypted. In practice, function calls and returns occur more frequently than forward pointer dispatches, so optimizing return address protection is far more important.

1) Exception Handling: Itanium ABI exception handling uses stack unwinding and matches call sites to exception index tables. Since our return-address hiding scheme encrypts call site addresses on the stack, stack unwinding will fail and break exception handling. All indirect disclosure protections which hide return addresses from an attacker will be similarly incompatible with stack unwinding, which depends on correctly mapping return addresses to stack frame layout information.

We modified LLVM's stack unwinding library implementation `libunwind` to handle encrypted return addresses. Since the first return address is stored in the link register, the stack unwinder can determine the first call site. From the call site, the stack unwinder is able to determine the function and read the XOR key that was used to encrypt the next return address using a whitelisted memory load. By recursively applying this approach, the unwinder can decrypt all return addresses until it finds a matching exception handler. This approach requires that we trust that the unwinding library does not contain a memory disclosure bug.

E. Fine-Grained Code Randomization

LR^2 does not depend on any particular type of code randomization and can be combined with most of the diversifying transformations in the literature [29]. We choose to evaluate our approach using a combination of function permutation [27] and register-allocation randomization [14, 39] as both transformations add very little run-time overhead. As Backes and Nürnberger [3] point out, randomizing the layout at the level of code pages may help allow sharing of code pages on resource-constrained devices. Note that had we only permuted the function layout, adversaries may be able to harvest trampoline pointers and use them to construct an attack without knowing the code layout. Because these pointers only target function entries and return sites (instructions following a call) this constrains the available gadgets much like a coarse-grained CFI policy would. Therefore, we must assume that gadget-stitching attacks [19, 24] are possible. However, stitching gadgets together is only possible with precise knowledge of how each gadget uses registers; register randomization therefore helps to mitigate such hypothetical attacks.

F. Decoupling of Code and Data Sections

References between segments in the same ELF object usually use constant offsets as these segments are loaded contiguously. To prevent an attacker from inferring the code segment base address in LR^2, we replace static relocations that are resolved during link time with dynamic relocations. This allows us to load the segments independently from each other, because the offsets are adjusted at load time. By entirely decoupling the code from the data section we prevent the attacker from inferring any code addresses from data addresses. As a convenient side-effect of this approach, code randomization is possible without the need for position-independent code (PIC). PIC is necessary to make applications compatible with ASLR by computing addresses relative to the current program counter (PC). Since we replace all PC-relative offsets with absolute addresses to decouple the code and data addresses, we observed slightly increased performance relative to conventional, ASLR-compatible position-independent executables at the cost of slower program loading.

G. Implementation in LLVM

We implemented our proof-of-concept transformations for LR^2 in the LLVM compiler framework. Our approach is not specific to LLVM, however, and is portable to any compiler or static rewriting framework. However, access to compile-time analysis and the compiler intermediate representation (IR) made our implementation easier. In particular, the mask hoisting optimization described previously is easier at compile time, but not impossible given correct disassembly and rewriting.

Since blindly masking every load instruction is expected to incur a high performance overhead due to the high frequency of load instructions, we take a number of steps to reduce the number of necessary mask instructions. LLVM annotates memory instructions such as loads and stores with information about the type of value that is loaded. We can use this information to ensure that load masking is not applied to loads from a constant address. Such loads are used to access jump table entries, global offset table (GOT) offsets, and other constants such as those in the constant pool. These loads account for less than 2% of all load operations in SPEC CPU2006, so this optimization has a small impact.

LLVM-based SFI implementations (e.g., Sehr et al. [47]) operate purely on the machine instructions late in the backend, roughly corresponding to rewriting the assembly output of the compiler. This makes the insertion of fault isolation instrumentation easier, but misses opportunities for additional optimization that is specific to our load-masking techniques. In order to hoist the masking of potentially unsafe addresses to their definition and avoid redundant re-masking, we leverage static analysis information about the program available earlier in the compiler pipeline. Specifically, we begin by marking unsafe address values while the program values are still in

static single assignment (SSA) form [16]. This allows us to easily find the definition of address values used by load instructions, and mask these values. Since stack spilling takes place after this point in the compilation, we must be careful to remask any source addresses restored from the stack, since the attacker may have modified these values while on the stack. In particular, we add markers to values that we mask while the program representation is still in SSA form. During register allocation, we check if marked values are spilled to memory. In the case of spills, we insert a masking instruction when restoring this value from the untrusted stack.

As in Native Client (NaCl) [47], it is necessary to prevent the compiler from generating load instructions using both a base and offset register (known as register-register addressing), to be sure that masking will properly restrict the resulting addresses. We modify the LLVM instruction lowering pass, where generic LLVM IR is converted to machine-specific IR, to prevent register-register addressed instructions. Instead, we insert a separate add instruction to compute the effective address. We make an exception if the load is known to be safe (e.g., a jump table load).

Finally, we insert return address protection instrumentation, stack pointer checks, and trampolines for forward code pointers during compilation as described in the previous sections.

H. Full LR^2 Toolchain

1) Code-Data Separation: By masking all load addresses we effectively partition the memory into a readable and unreadable section. Our fully-fledged prototype system uses a slightly modified Linux kernel and dynamic loader to separate the process memory space into readable and unreadable sections (see Figure 2 for an overview of this separation). The kernel and dynamic loader normally load entire ELF objects contiguously. Data segments are usually loaded consecutively above the corresponding module's code. In LR^2, however, readable segments are placed exclusively in the lower 2GiB region of the process address space, while unreadable (code) segments must be placed in the higher 2GiB region. Consequently, this requires ELF objects to be split. We applied small patches to the Linux kernel (121 LoC) and `musl` dynamic loader (196 LoC) to load each ELF segment into the proper area.

Furthermore, we modified the usual kernel memory mapping mechanism to comply with our memory layout restrictions. By passing an internal flag to *mmap*, an application can specify which memory region the requested memory must be allocated in. This allows the loader to ensure that a program's data segment is mapped low enough in memory that the corresponding executable segment lies between 0x80000000 and 0xC0000000 which is where reserved kernel memory begins. Finally, our patch ensures that memory areas allocated by the kernel (e.g., stacks and heaps) are in the readable region.

We also needed to slightly modify the linker to prepare an executable for use with LR^2 memory layout. Specifically, we patched the gold linker to *not* mark executable sections as readable[4] and to assign these sections to high addresses. This type of patch is needed for all XoM solutions, since current linkers mark executable segments with read-execute, rather than execute-only permissions. Additionally, we added linker support for 32-bit offsets in Procedure Linkage Table (PLT) entries, which comes at the cost of one additional instruction per PLT entry. This is necessary because the PLT (unreadable memory) refers to the Global Offset Table (GOT) (readable memory), and therefore might be too far away for the 28-bit address offset previously used.

2) Libraries: For memory disclosure resilience, all code in an application needs to be compiled with LR^2, including all libraries. Since the popular C standard library `glibc` does not compile with LLVM/Clang, we tested our implementation with the lightweight replacement `musl` instead. It includes a dynamic loader, which we patched to support our code layout with the same approach as applied to the kernel. We use LLVM's own `libc++` as the C++ standard library, since the usual GNU `libstdc++` depends on `glibc` and GCC.

V. Performance Evaluation

We evaluate the performance of LR^2 using the CPU-intensive SPEC CPU2006 benchmark suite, which represents a worst-case, CPU-bound performance test. We measure the overall performance as well as the impact of each technique in our mitigation independently to help distinguish the various sources of overhead. In addition we measured the code size increase of our transformations, since code size is an important factor in mobile deployment. Overall, we found that with all protections enabled, LR^2 incurs a geometric mean performance overhead of 6.6% and an average code size increase of 5.6%. We summarize the performance results in Figure 4. Note that these measurements include results for the hmmer and soplex benchmarks, which are known to be very sensitive to alignment issues ($\pm$12% and $\pm$6%, respectively) [34].

We want to measure the impact of LR^2 applied to whole programs (including libraries), so we compile and protect a C and C++ runtime library with our modifications for use with the SPEC suite. Since the de-facto standard libraries on Linux, `glibc` and `libstdc++`, don't compile with LLVM/Clang, we use `musl` and LLVM's own `libc++` instead. We extended the `musl` loader to support our separated code and data layout.

The perlbench and namd benchmarks required small workarounds since they contain `glibc`/`libstdc++` specific code. h264ref on ARM fails for unknown reasons when comparing the computation result, both for the unmodified and the LR^2 run; since it completes the computation we include the running time nonetheless. Finally, the stack unwinding library used by LLVM's `libc++` fails with omnetpp, so we exclude it from all benchmark measurements. We report all measurements as the geometric mean over all other SPEC CPU2006 benchmarks. All measurements are from a Chromebook model CB5-311-T6R7 with an Nvidia Tegra Logan K1 System-on-Chip (SoC), running Ubuntu 14.04 with Chromium OS's Linux 3.10.18 kernel.

A. Forward-Pointer Hiding

We measured impact of forward-pointer hiding, which introduces an additional direct jump instruction for each indirect call. We found that this transformation resulted in an overhead of less than 0.3% on average over all benchmarks, with a maximum overhead of 3%.

[4]Note that the memory permission execute normally implies readable due to the lack of hardware support

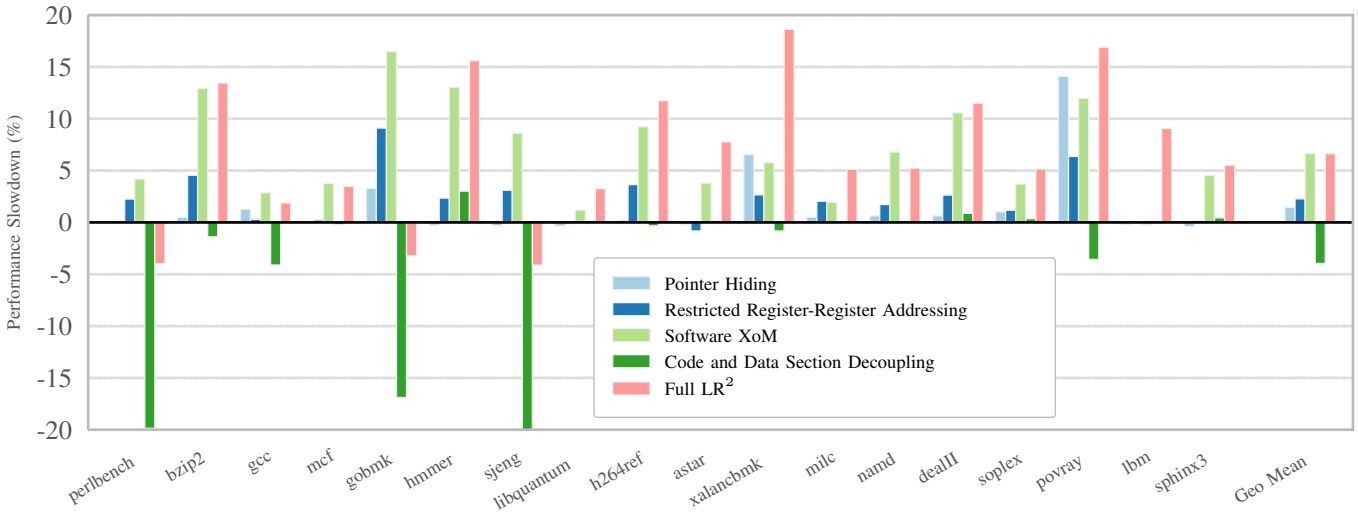

Figure 4: LR^2 overhead on SPEC CPU2006. We use the performance of unprotected position independent binaries as the baseline.

B. Return-Address Hiding

Return-address hiding requires one extra load and XOR at the entry of each function that spills the link register. At each function return it replaces the return instruction with one load, one XOR and one branch. We found that this instrumentation added an overhead of less than 1% on average, with a maximum overhead of 3% over the baseline time. Combining forward-pointer hiding and return-address hiding, we measured an average overhead of 1.4%. We show the combined results in Figure 4, labeled *Pointer Hiding*. This overhead compares favorably to Readactor's [14] 4.1% overhead for full code pointer hiding, since our return-address hiding scheme does not require expensive return trampolines for each call site.

For both forward-pointer and return-address hiding, we noticed that a few benchmarks ran slightly faster with the instrumentation than without. We attribute this variance to measurement error and slight code layout differences resulting in different instruction cache behavior.

C. Register-Register Addressing Scheme Restrictions

An important feature of the ARM instruction set is register offset addressing for array or buffer loads. As described in Section IV, we have to disable this feature in LR^2, since it interferes with XoM address masking. We measured the overhead that this restriction incurs by itself and found that restricting register addressing schemes incurs 2.3% overhead on average and a 9% worst-case overhead on the gobmk benchmark. Benchmarks like hmmer, bzip2 and sjeng are affected because a large portion of the execution time is spent in one hot loop with accesses to many different arrays with varying indices.

D. Software XoM

The last component to analyze individually is our XoM instrumentation—masking unsafe loads. We found that, after applying the optimizations outlined in Section IV-B, software-enforced XoM results in an overhead of 6.6% on average (labeled *Software XoM* in Figure 4), with a maximum overhead of 16.4% for one benchmark, gobmk. We attribute this primarily to to data dependencies introduced between the masking and load instructions, as well as hot loop situations such as mentioned above.

E. Code and Data Decoupling

Normally the code and data segments of a program have a fixed offset in memory, allowing PC-relative addressing of data. However, this also allows an attacker to locate the beginning of the code segment from any global data address. As we describe in Section IV-F, we decouple the location of the data segment from the code segment, allowing the loader to independently randomize the position of each. To do this, we replace the conventional PC-relative address computation with dynamic relocations assigned by the program loader. This change led to a geometric mean speedup of 4% (labeled *Code and Data Section Decoupling* in Figure 4).

F. Full LR^2

The aggregate cost of enabling all techniques in LR^2 is 6.6% on average (see *Full LR^2* in Figure 4). This includes the cost of pointer hiding, software-enforced XoM, register-register addressing restrictions, fine-grained diversity, and the impact of decoupling code and data. This means that our pure software approach to leakage resilient diversity for ARM has about the same overhead as hardware-accelerated leakage resilient diversity for x86 systems (6.6% vs. 6.4% [14]). Because the removal of PC-relative address computations yield a speedup, the cost of individual transformations sometimes exceed the aggregate cost of LR^2. An earlier version of our prototype that did not remove PC-relative address computations to decouple code and data sections had an average overhead of 8.4%.

G. Memory Overheads

Finally, in addition to running time, we also measured code section size of the protected SPEC CPU2006 binaries. Forward-pointer hiding had very little overall impact on code size, leading to an increase of 0.9%. Return-address hiding

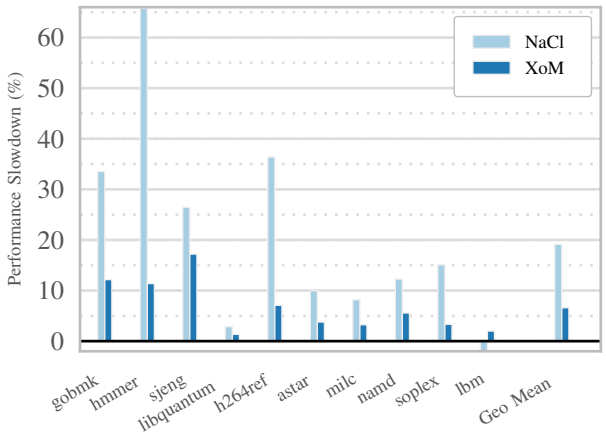

Figure 5: Comparing software XoM to SFI (NaCl) to quantify effect of load-mask optimization.

adds at least four instructions to most functions, which resulted in a 5.2% code size increase. The additional load address masking for software-enforced XoM increases the code size by another 10.2%. However, removing the PC-relative address computations decreases the code size by about 14% on average. Comparing the size of full LR^2 binaries to legacy position independent code shows an average increase of just 5.6%.

H. Impact of XoM-Specific Optimizations

Recall that the differences in threat models between software XoM and SFI (Section IV-B0b) allow us to protect multiple uses of a load address using a single masking instruction. To measure the impact of this optimization, we compare the running time of the SPEC CPU2006 benchmarks that run correctly when protected with NaCl to the cost of enforcing XoM. For this experiment, we used the latest version[5] of the NaCl branch (pnacl-llvm) that is maintained as part of the Chromium project. The results are shown in Figure 5. When enforcing XoM using load masking, the average overhead is 6.6% (for the set of benchmarks compatible with NaCl) whereas software-fault isolation, which also masks writes and indirect branches, costs 19.1% overhead. We stress that we are comparing two different techniques with different purposes and threat models. However, these numbers confirm our hypothesis that our XoM-specific load-masking instrumentation reduces overheads. A bigger impact can be seen when comparing code sizes: XoM led to a 5.8% increase, while NaCl caused an increase of 100%. This is a valuable improvement in mobile environments where memory is a scarce resource.

VI. Security Analysis

Our primary goal in LR^2 is to prevent disclosure of the code layout, which enables sophisticated attacks [50] against code randomization schemes [29]. By securing the code from disclosure we can then rely on the security properties of undisclosed, randomized code.

In order to launch a code-reuse attack the attacker must know the code layout. By applying fine-grained randomization, e.g., function or page reordering, we prevent all static code-reuse attacks, since these attacks are not adjusted to each target's randomized code layout. For our proof-of-concept we chose to build on function permutation as it is effective, efficient, and easy to implement. However, as all code randomization techniques, function permutation by itself is vulnerable to an attacker who discloses the code layout at run time [11, 20, 50]. Hence, we focus our security analysis of LR^2 on resilience against direct and indirect information-disclosure attacks targeting randomized program code.

A. Direct Memory Disclosure

Direct memory disclosure is when the attacker reads the memory storing randomized code. JIT-ROP [50] is a prominent example of this type of attack. JIT-ROP recursively discloses and disassembles code pages at run time until enough gadgets are disclosed to assemble and launch a ROP attack.

We prevent all direct disclosure attacks by masking memory loads in the protected application, i.e., we prevent loads from reading the program code directly. Masking the load address restricts any attempt to read the code section to the lower half of the memory space which contains only data. Naively masking every load operation is inefficient; we therefore apply the optimizations described in Section IV-B to reduce the number of masking instructions. Allowing some unmasked load operations may appear to increase the risk of an unsafe load instruction. However, we are careful to ensure that all unsafe loads are restricted, as we show in the following.

1) PC-Relative Loads: All PC-relative loads with a constant offset are guaranteed to be safe, since an attacker cannot influence the address used during the load operation and only legitimate data values are loaded in this manner. Therefore, we need not mask these load instructions.

2) Constant Offsets: We allow loads from known safe base addresses (i.e., already masked values) plus or minus a *small* constant offset (less than 4KiB). Thus, if we ensure that the base address must point into the data section, adding a guard page between the data and code sections prevents the computed address from reaching into the code section. We place an unmapped 8KiB (2 pages) *guard region* between the data and code sections to safeguard all possible constant offsets. In addition, the addresses above 0xC0000000 are reserved for kernel usage and will trigger a fault when accessed, so programs are already safe from address underruns attempting to read from the highest pages in memory by subtracting a constant from a small base address.

We also allow limited modification of masked addresses without re-masking the result. If an address has already been masked so that it is guaranteed to point into the data section, adding or subtracting a small constant will result in either an address that is still safe, or one that falls into the *guard region*. In either case, the modified address still cannot fall into the code section, and thus we do not need to re-mask it. We perform this optimization for all constant stack pointer adjustments.

3) Spilled Registers: When a program needs to store more values than will fit into the available registers, it stores (spills) a value temporarily onto the stack to free a machine register.

[5] As of August 10, 2015

As recently demonstrated, stack spills of sensitive register contents can allow adversaries to completely bypass code-reuse mitigations [11]. In our case, an attacker could attempt to bypass LR^2 by manipulating a previously masked register while it is spilled to writable memory. Therefore, we do not trust any address that is restored from writable memory and always re-mask it before the value is used to address memory.

B. Indirect Memory Disclosure

Mitigating direct memory disclosure alone does not fully prevent an attacker from leaking the code layout. An attacker can indirectly gain information about the code layout by leaking readable code pointers from the data section [11, 20]. The necessary number of leaked code pointers for a successful code-reuse attack depends on the granularity of the applied randomization. For instance, in the presence of page-based randomization, one code pointer allows the attacker to infer 4 KiB of code due to page alignment, whereas the attacker has to leak more code pointers in the presence function-level randomization to infer the same amount of code. To counter indirect memory disclosure we create trampolines for forward code pointers and encrypt return addresses.

1) Forward-Pointer Protection: An attacker cannot use function pointers to infer the code layout because they point to trampolines which reside in code segment. Hence, the destination address of a trampoline cannot be disclosed. The order of the trampolines is randomized to prevent any correlation between the trampolines and their target functions. This constraints the attacker to whole-function reuse attacks. To mitigate such attacks, we suggest using the XoM-based technique presented by Crane et al. [15] to randomize tables of function pointers. This extension should be completely compatible with the software-only XoM provided by LR^2 without modification and would protect against the most prevalent types of whole-function reuse: return-into-PLT and vtable-reuse attacks.

2) Return-Address Protection: Return addresses are a particularly valuable target for attackers because they are plentiful, easy to access, and useful for code-reuse attacks, even with some mitigations in place. For example, when attacking an application protected by function permutation, an attacker can leak return addresses to infer the address of the functions and in turn the addresses of gadgets within those functions [11]. We prevent this by encrypting each return address with a per-function 32-bit random number generated by a secure random number generator. However, our adversary model allows the attacker to leak all encrypted return addresses spilled to the stack. Whiles she cannot infer code addresses from the encrypted return addresses we conservatively assume that she can relate each return address to its corresponding call site.

We must also address reuse of unmodified, disclosed return addresses. In a previous indirect disclosure protection scheme, Readactor [14], return addresses were vulnerable to reuse as-is. Although Readactor prevented attackers from gaining information about the real location of code surrounding a call site, an attacker could potentially reuse call-preceded gadgets. An attacker could disclose the trampoline return address corresponding to a given call site and jump into that trampoline, which in turn jumps directly after the real call site. This allows attackers to reuse any disclosed return addresses. To mitigate this threat, the Readactor authors proposed additional randomizations (register and callee stack slot permutation) to attempt to disrupt data flow between call-proceeded gadgets and mitigate this threat.

In LR^2 arbitrary reuse of return addresses is impossible. By encrypting every return address with a per-callee encryption key, our system prevents the attacker from invoking a call-site gadget from anywhere but the corresponding callee's return instruction. In other words, encrypted return addresses can only be used to return from the function that originally encrypted the address. Thus, the attacker is confined to the correct, static control-flow graph of the program. This restriction is similar to static CFI policies. However, we further strengthen LR^2 by applying register-allocation randomization. During our analysis of state-of-the-art ROP attacks we determined that the success of these attack is highly dependent on the data flows between specific registers. Register randomization will disrupt the attacker's intended data flow between registers and hence, have unforeseen consequences on the control flow which will eventually result in a crash of the application.

While our XOR encryption scheme uses a per-function key, this key is shared across all invocations of a function. That is, each time a return address is spilled from a function F it is encrypted with the same key K_F. In most cases this is not a problem, since function permutation prevents an attacker from correlating return addresses encrypted with the same key. However, if a function F_1 contains two different calls to another function F_2, the return addresses, R_1 and R_2 respectively, are encrypted with the same key K_{F_2}. The attacker has a priori knowledge about these addresses, since with function permutation they are still placed a known (constant) offset apart. We believe this knowledge could be exploited to leak some bits of the key K_{F_2}. To prevent this known-plaintext attack we propose two options: (1) we can either apply more fine-grained code randomization, e.g., basic-block permutation to remove the correlation between return addresses or (2) fall back to using the trampoline approach to protect return addresses as presented by [14] when a function contains more than one call to the same (other) function. These techniques remove the a priori knowledge about the encrypted return addresses. In fact, return-address encryption even strengthens the second approach because it prevents trampoline-reuse attacks for return addresses.

C. Proof-of-Concept Example Exploit

We evaluate the effectiveness of LR^2 against real-world attacks by re-introducing a known security vulnerability (`CVE-2014-1705`) into the latest version of Chromium (v46.0.2485.0) and conducted our experiments on same setup we used in our performance evaluation. The vulnerability allows to overwrite the length field of a buffer object. Once this is done we can exploit this manipulated buffer object via JavaScript to read and write arbitrary memory.

We constructed a JIT-ROP style attack that first leaks the vtable pointer of an object O_{target} to disclose its vtable function pointers. Using one of these function pointers we can infer the base address of the code section of Chromium. Next, we use our information disclosure vulnerability to search the executable code at run time for predefined gadgets that allow

us to launch a ROP attack to mark data memory that contains our shellcode as executable. Finally, we overwrite the vtable pointer of O_{target} with a pointer to an injected vtable and call a virtual function of O_{target} which redirects control flow to the beginning of our shellcode to achieve arbitrary code execution.

There are currently some efforts by the Chromium community to achieve compatibility with the `musl` C library. By the time of writing this paper Chromium remains incompatible which prevents us from applying the *full* LR^2 toolchain. However, we applied our load-masking component while compiling Chromium and analyze the effect this load-masking would have on the memory disclosure we exploit.

Our analysis indicates that Chromium would immediately crash when the attempted code read was restricted into an unmapped memory area within the data section. Figure 6 shows how the function that this exploit uses to leak memory is instrumented. After instrumentation, all load instructions in the function cannot read arbitrary memory and must only read from addresses that point into the data segment. Thus, our proof-of-concept exploit would fail to disclose the code segment at all and would instead crash the browser with a segmentation violation.

VII. Discussion and Extensions

A. Auto-Vectorization

When loops have no data dependencies across iterations, consecutive loop iterations may be vectorized automatically by the compiler, using an optimization technique called auto-vectorization. This technique computes multiple loop iterations in parallel using vector instructions that perform the same operation on a contiguous set of values.

While investigating the source of the higher overhead for the hmmer benchmark, we found that one function—`P7Viterbi`—accounts for over 90% of the benchmark's execution time. The main loop of this function is amenable to vectorization as it exhibits a high degree of data parallelism [42]. Modern ARM processors support the NEON instruction set extension which operate on four scalar values at a time. Unfortunately, support for automatic vectorization in LLVM was only added in October 2012 and is still maturing. Using the older and more capable vectorization passes in GCC, ICC from Intel, and XLC from IBM may allow more loops to be vectorized [32].

In the context of LR^2, vectorization would not only reduce the running time by exploiting the data parallelism inherent to many computations; it would also reduce the number of required load masking operations by a factor of more than four. First of all, vectorized loads read four consecutive scalars into vector registers using a single (masked) address. Second, the NEON instructions operate on dedicated, 128-bit wide registers which means that fewer addresses would be spilled to the stack and re-masked when reloaded.

B. Assembly code

LLVM does not process inline assembly on an instruction level and therefore transformation passes can only work with inline assembly blocks as a whole. Therefore our current prototype does not handle inline assembly; this is not a fundamental limitation of our approach however. To make sure that every load is properly masked in the presence of assembly code, we could extend the LLVM code emitter or an assembly-rewriting framework such as MAO [26] with load-masking and code pointer hiding passes. Since the code is not in SSA form at this stage we can not apply our optimizations.

C. Dynamically Generated Code

JIT-ROP attacks are ideally mounted against browsers containing scripting engines. To ensure complete leakage-resilience, we must ensure that XoM and code-pointer hiding is also applied to just-in-time compiled code. Crane et al. [14] patched the V8 JavaScript engine used in the Chrome browser to make it compatible with XoM. To use this patch for LR^2, we would have to add functionality to ensure that every load emitted by the JIT compiler is properly masked. This would simply involve engineering effort to patch the JIT compiler.

A special property of JIT-compiled code is that it is treated as both code and data by the JIT compiler; when the compiler needs to rewrite or garbage collect the code, it is treated as read-write data, and while running it must be executable. When XoM is enforced natively by the hardware, the page permissions of JIT compiled code can be changed by updating the page tables used by the memory management unit. With software-enforced XoM, we can make JIT compiled code readable by copying it (in part or whole) into the memory range that is accessible to masked loads. However, that would require a special `memcpy` function containing unmasked loads. Therefore, we believe that a better solution would be to adopt the split-process technique presented by Song et al. [51]. The key idea of this work is to move the activities of the JIT compiler into a separate, trusted process. Specifically, the code generation, optimization, and garbage collection steps are moved to a separate process in which the JIT code cache always has read-write permissions. In the main process, the JIT code cache is always mapped with execute-only (or read-execute if XoM is unavailable) permissions. The two processes access the JIT code cache through shared memory. The main process invokes JIT compilation functionality in the trusted process through remote procedure calls.

D. Whole-Function reuse attacks

Since LR^2 raises the bar significantly for ROP attacks against mobile architectures, attackers may turn to whole-function reuse techniques such as the classic return-into-libc (RILC) technique [37] or the recent counterfeit object-oriented programming (COOP) attack [46]. Our core techniques—execute-only memory and code-pointer hiding—can be extended to mitigate RILC and COOP attacks, as proposed by Crane et al. [15]. To thwart COOP, we would split C++ vtables into a data part (rvtable) and a code part (xvtable) stored on execute-only pages. The xvtable contains trampolines, each of which replaces a pointer to a virtual method. Randomly permuting the layout of the xvtable breaks COOP attacks because they require knowledge of the vtable layout. We can break RILC attacks by similarly randomizing the procedure linkage table (PLT) or analogous data structures in Windows.

```
1  ldr     r0, [r1, #0]
2
3  mov     r12, #28
4  ldr     r3, [r0, #7]
5  ldr     r1, [r0, #11]
6  bfi     r0, r12, #0, #20
7
8  add     r1, r3
9  ldr     r0, [r0, #0]
10
11
12 ldr     r1, [r1, r2, lsl #2]
13 [...]
```

Before Instrumentation

```
1  ldr     r0, [r1, #0]
2  bic     r0, r0, #0x80000000
3  mov     r12, #28
4  ldr     r3, [r0, #7]
5  ldr     r1, [r0, #11]
6  bfi     r0, r12, #0, #20
7  bic     r0, r0, #0x80000000
8  add     r1, r3
9  ldr     r0, [r0, #0]
10 add     r1, r1, r2, lsl #2
11 bic     r1, r1, #0x80000000
12 ldr     r1, [r1]
13 [...]
```

After Instrumentation

Figure 6: Simplified disassembly of the function `v8::internal::ElementsAccessorBase::Get` that is used to read arbitrary memory. The load instruction in line 12 reads the memory from the base address provided in register `r1` plus the offset in register `r2`. After the instrumentation this load is restricted by masking the MSB (line 11) which prevents reads into the code segment.

E. Compatibility

Due to the nature of its load masking and return-address hiding scheme, LR^2 is fully compatible with unprotected third party libraries. However, if an unprotected library contains an exploitable memory-disclosure vulnerability it compromises the security of the entire process.

In some cases application developers use the `mmap()` function to map memory to a specific address. In LR^2 we do not allow mapping to arbitrary addresses because the application will fail when trying to read memory mapped into the XoM region. Hence, we only allow mapping memory into the data region. This is still consistent with the correct semantics of `mmap()` because the kernel considers requested addresses merely as a hint rather than a requirement

F. AArch64

Our implementation currently targets 32-bit ARMv7 processors. ARM recently released ARMv8, which implements the new AArch64 instruction set for 64-bit processing. LR^2 can be ported directly to AArch64. Though AArch64 does not provide a bit clear instruction with immediate operands, we can achieve the same effect with a bitwise AND instruction.

VIII. Related Work

Table I: Characterization of leakage-resilient defenses. The third column indicates whether the defense prevents read accesses to code pages. The fourth column indicates whether pointers can be used to leak the code layout.

	Applicable to mobile systems?	XoM	Ptr. Hiding
Oxymoron [3]	No. Bypassed [20]		(partial)
XnR [4]	No. Requires virtual memory.	✓	
Isomeron [20]	No. High memory requirements	N/A	(partial)
Opaque CFI [35]	No. High memory requirements	N/A	
HideM [22]	No. Requires virtual memory.	✓	
Readactor [14]	No. Requires HW X-only memory.	✓	✓
Readactor++ [15]	No. Requires HW X-only memory.	✓	✓
ASLRGuard	No. Requires 64-bit address space.		✓
TASR	No. Requires strict C compliance.	N/A	
LR^2	Yes. Requires W⊕X support.	✓	✓

Numerous papers have been published on software diversity in the last two decades. We refer to Larsen et al. [29] for an overview and limit the present discussion to recent work on leakage-resistant diversity. Table I shows recent approaches and the reasons why they are not ideal for mobile and embedded devices.

a) Leakage-Resilient Diversity: Backes and Nürnberger [3] were first to demonstrate a defense against JIT-ROP attacks. Their Oxymoron approach uses the vestiges of x86 segmentation features to hide code references between code pages which in turn prevents the recursive disassembly step in the original JIT-ROP attacks. Davi et al. [20] later showed that JIT-ROP attacks are still possible without recursive disassembly because the adversary can scan data pages to discover plenty of code pointers that leak the location of code pages. The eXecute-no-Read (XnR) approach by Backes et al. [4] provides increased resilience against memory disclosure vulnerabilities by emulating execute-only memory (XoM) on x86 processors. While the concept of XoM goes back to MULTICS [12], it is hard to support on x86 and other platforms that implicitly assigns read permissions to executable pages. The XnR approach is to mark code pages "not present" so any access invokes a page-fault handler in the operating system. If an access originates from the instruction fetcher, the page is temporarily marked present (and thus executable and readable), otherwise execution terminates. This prevents all read accesses outside a sliding window of recently executed pages. Gionta et al. [22] demonstrated that XoM can also implemented using a technique known as "TLB Desynchronization" on certain x86 processors. Whereas virtual addresses usually translate to the same physical address regardless of the type of access access, the HideM approach translates reads and instruction fetches to distinct physical pages. This means that HideM, in contrast to XnR, can support code that embeds data in arbitrary locations. However, Conti et al. [11] demonstrated that neither XnR nor HideM prevents indirect memory disclosure attacks where the adversary "harvests" code pointers stored in the stack, heap, or global data areas of the process.

Readactor by Crane et al. [14] provides comprehensive XoM on x86 processors by separating code and data during compilation. At run time, a lightweight hypervisor activates

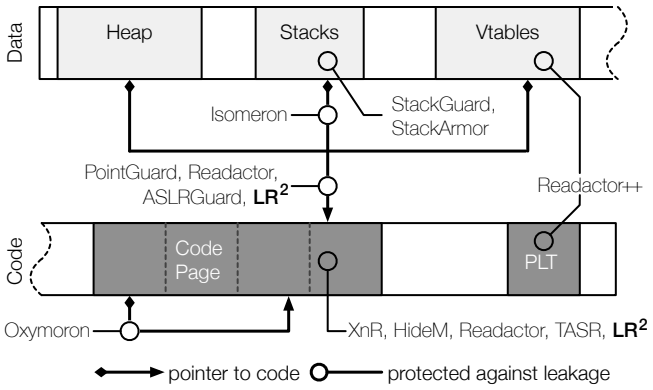

Figure 7: Relationship between the in-memory program representation and techniques that defend against memory leakage.

the extended page tables in modern x86 processors that allows read/write/execute permissions to be controlled independently. Readactor also seeks to prevent indirect memory disclosure attacks though code-pointer hiding, an indirection mechanism known as call and jump trampolines. A recent extension (Readactor++ [15]) adds resilience against whole-function reuse attacks such as counterfeit-object oriented programming (COOP) [46] and return-into-libc (RILC) [37].

The memory leakage resilience of LR^2 is similar to that of Readactor. Unlike the approaches by Crane et al. [14, 15], we do not require a CPU with hardware support for virtualization and a hypervisor; in fact, our approach does not rely on virtual memory at all and therefore applies to MMU-less chips commonly used in build embedded and real-time systems.

Another way to defend against information disclosure is "live re-randomization", where the program is periodically re-randomized to invalidate any current code pointers, thereby preventing the attacker from exploiting any knowledge they gain of the program. Giuffrida et al. [23] describe the first implementation of this idea. However, even with very short randomization periods, the attacker may still have enough time for an attack [4, 20]. Bigelow et al. [6] propose an improved approach, TASR, which only re-randomize programs when they perform input or output. Both approaches require that all code pointers are updated post-randomization, and rely on a modified C compiler to provide their locations. However, finding all code pointers in a C program is not always possible in the general case. Bigelow et al. describe a set of heuristics and assumptions they rely on to find the pointers, but real-world C code does not strictly comply with C's standard rules and often violates common sense assumptions about pointer use and safety [10].

Figure 7 maps key regions of the in-memory program representation to related work in leakage-resilient diversity. While many defenses focus on enforcing XoM or preventing indirect leakage through code pointers, only Readactor and LR^2 provide both to stop all variants of JIT-ROP. As described in Section VII-D, the vtable and procedure linkage table (PLT) randomization techniques are fully compatible with software-XoM and would increase resilience against COOP and RILC attacks.

b) Protecting Code Pointers: Cowan et al. [13] were first to protect code pointers. Their PointGuard technique XORs all pointers with a per-program key; PointGuard uses a weak threat model and therefore does not provide leakage resilience. G-Free [38] encrypts return addresses using a per-function key. Unlike our solution, the XOR key is not hidden from adversaries which makes their solution vulnerable in our threat model. Isomeron by Davi et al. [20] clones the code and switches between clones at each call site by randomly flipping a coin. If the coin came up heads, an offset is added to the return address before it is used. Because the result of the coin-flip is stored in a hidden memory area, adversaries cannot predict how the return addresses in a ROP payload will be modified by Isomeron. Since all code is cloned, Isomeron does not scale down to resource-limited systems.

ASLRGuard [31] also prevents code pointers from disclosing the code layout. ASLRGuard uses a nonce and index into a code pointer table, which is hidden using the vestiges of x86 segmentation; this is yet another example of an x86-specific defense. More importantly, ASLRGuard implicitly needs a 64-bit virtual memory space for ASLR itself to be secure from brute-force attacks. Shacham et al. [49] demonstrated that ASLR in a 32-bit address space is insecure due to low entropy. Most RISC devices have a 32-bit address space at most, and some even lack an MMU to provide virtual memory at all. Thus, ASLR and ASLRGuard are not ideal for these resource-limited devices.

Cryptographic CFI (CCFI) [33] encrypts code pointers. Specifically, CCFI uses the AES instructions of recent x86 processors to protect pointers and uses the storage location as a nonce during encryption to reduce the ability of the adversary to reuse encrypted pointers in replay attacks. However, CCFI is yet another defense tied to x86 hardware and has a much higher overhead than LR^2 (45% vs 6.6%) on the SPEC CPU2006 benchmarks.

XOR encryption has also been used to prevent non-control data attacks [5, 9]. These protections are orthogonal to ours.

c) Mobile-Oriented Defenses: The literature contains several other mobile-oriented code-reuse defenses. Bojinov et al. [8] describe the initial implementation of ASLR for Android devices. Recent releases of the Android platform adds support for high-entropy ASLR but remains vulnerable to information disclosure. XIFER [17] is a load-time software diversifier for Android and Linux that randomizes the code right before it starts executing. While effective against traditional ROP attacks, XIFER is not leakage resilient. Lee et al. [30] identified weaknesses with the Zygote process creation model in Android that weakens the effectiveness of ASLR and evaluated improvements.

MoCFI [18] and CFR [41] are mobile-oriented CFI solutions based on binary rewriting and compilation of iOS apps respectively. Both implementations use static analysis augmented by either heuristics (in the case of MoCFI) or programmer intervention (for CFR) to generate a control-flow graph (CFG) used to restrict program control flow. This adds a high degree of uncertainty to the CFG's accuracy. A CFG that is too coarse-grained, i.e., places too few restrictions on the control flow, is easily exploitable by attackers, so the security of these defenses depends on the quality (granularity) of the

generated CFGs.

d) Software-Fault Isolation: SFI isolates untrusted code so it cannot access memory outside the sandbox or escape confinement. SFI policies are typically enforced by inserting inline reference monitors [45, 53].

Since reads are far more frequent than writes, some SFI implementations only sandbox writes and indirect branches. Google's NaCl implementation for ARM [47] eschewed load-isolation initially but support was later added [36] to prevent untrusted plug-ins from stealing sensitive information such as credit card and bank account numbers. Like LR^2, NaCl for ARM uses a customized compiler and masks out the high bits of addresses. Unlike LR^2, NaCl also constrains writes and indirect branches. ARMor [54] is another SFI approach for ARM. It uses link time binary rewriting to instrument untrusted code. This makes ARMor less efficient than compile-time solutions and the authors report overheads ranging from 5-240%.

Several hardware-based fault isolation approaches appeared recently. Zhou et al. [55] present ARMlock which uses the memory domain support in ARM processors to create sandboxes that constrain the reads and writes, and branches of code running inside them with no loss of efficiency. While ARMlock prevents code from reading the contents of other sandboxes, it cannot support our use-case of preventing read accesses to code inside the sandbox. Santos et al. [44] use the ARM TrustZone feature to build a trusted language runtime (TLR); while this greatly reduces the TCB of an open source .NET implementations the performance cost is high. Unlike LR^2, these approaches rely on features that limit their applicability to certain hardware platforms.

One may consider the load-masks we insert as a type of SFI inline reference monitor. However, as explained in Section IV-B, we place masking instructions differently from SFI techniques due to the different threat model; using the same instrumentation for SFI would not be secure. Further, LR^2 does not need to constrain writes or indirect branches as the adversary must disclose the code layout before mounting a code-reuse attack.

Software that is vulnerable to memory corruption remains exposed to sophisticated code-reuse exploits. The problem of code reuse is not specific to x86 systems but threatens RISC-based mobile and embedded systems too. Code randomization can greatly improve resilience to code reuse as long as the code layout is not disclosed ex post facto. The combination of execute-only memory and code-pointer hiding provides comprehensive resilience against leakage of code layout information. Unfortunately, the implementation of these techniques has so far relied on x86-specific features or has increased resource requirements beyond reasonable limits for mobile and embedded devices.

Unlike previous solutions, our leakage-resilient layout randomization approach—LR^2—only requires that the host system enforces a W$\oplus$X policy. Our software enforcement of execute-only memory is inspired by prior work on software-fault isolation. However, since our threat model is fundamentally different from SFI (we protect trusted code whereas SFI isolates untrusted code), we are able to insert fewer load-masking operations than comparable SFI implementations. This significantly reduces overheads.

We reuse existing techniques to protect forward pointers but present a new optimized XOR pointer encryption scheme relying on XoM and function permutation to protect return addresses. Since LR^2 does not require any special hardware support, it can protect applications running on a broad range of non-x86 devices, including MMU-less micro-controllers. Even though LR^2 prevents memory disclosure purely in software, its performance is similar to defenses offering comparable security.

Acknowledgments

The authors thank Nikhil Gupta for helping with logistics and benchmarking.

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-15-C-0124, FA8750-15-C-0085, and FA8750-10-C-0237, by the National Science Foundation under award numbers CNS-1513837 and IIP-1520552 as well as gifts from Mozilla, Oracle, and Qualcomm.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government.

This work has been co-funded by the German Science Foundation as part of project S2 within the CRC 1119 CROSSING and the European Union's Seventh Framework Programme under grant agreement No. 609611, PRACTICE project.

References

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

[2] ARM Ltd. *ARM Compiler Software Development Guide v5.04*, 2013. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0471k/chr1368698593511.html.

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

[4] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2014.

[5] S. Bhatkar and R. Sekar. Data space randomization. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA, 2008.

[6] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi. Timely rerandomization for mitigating memory disclosures. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[7] A. Bittau, A. Belay, A. J. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *35th IEEE Symposium on Security and Privacy*, S&P, 2014.

[8] H. Bojinov, D. Boneh, R. Cannings, and I. Malchev. Address space randomization for mobile devices. In *ACM Conference on Wireless Network Security*, WiSec, 2011.

[9] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro. Data randomization. Technical Report MSR-TR-2008-120, Microsoft Research, September 2008. URL http://research.microsoft.com/apps/pubs/default.aspx?id=70626.

[10] D. Chisnall, C. Rothwell, R. N. M. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann. Beyond the PDP-11: Architectural support for a memory-safe C abstract machine. In *20th*

International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, 2015.

[11] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, M. Negro, M. Qunaibit, and A.-R. Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[12] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the MULTICS system. In *Joint Computer Conference*, AFIPS, 1965.

[13] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: protecting pointers from buffer overflow vulnerabilities. In *12th USENIX Security Symposium*, USENIX Sec, 2003.

[14] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *36th IEEE Symposium on Security and Privacy*, S&P, 2015.

[15] S. Crane, S. Volkaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. D. Sutter, and M. Franz. It's a TRAP: Table randomization and protection against function reuse attacks. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[16] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, 1989.

[17] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *19th Annual Network and Distributed System Security Symposium*, NDSS, 2012.

[18] L. Davi, A. Dmitrienko, S. Nürnberger, and A. Sadeghi. Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *8th ACM Symposium on Information, Computer and Communications Security*, ASIACCS, 2013.

[19] L. Davi, A. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium*, USENIX Sec, 2014.

[20] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (Just-In-Time) return-oriented programming. In *22nd Annual Network and Distributed System Security Symposium*, NDSS, 2015.

[21] J. Drake. Stagefright: scary code in the heart of Android. https://www.blackhat.com/us-15/briefings.html#stagefright-scary-code-in-the-heart-of-android, 2015.

[22] J. Gionta, W. Enck, and P. Ning. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *5th ACM Conference on Data and Application Security and Privacy*, CODASPY, 2015.

[23] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *21st USENIX Security Symposium*, USENIX Sec, 2012.

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

[25] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Librando: transparent code randomization for just-in-time compilers. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2013.

[26] R. Hundt, E. Raman, M. Thuresson, and N. Vachharajani. MAO – an extensible micro-architectural optimizer. In *9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO, 2011.

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

[28] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2014.

[29] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *35th IEEE Symposium on Security and Privacy*, S&P, 2014.

[30] B. Lee, L. Lu, T. Wang, T. Kim, and W. Lee. From zygote to morula: Fortifying weakened aslr on android. In *IEEE Symposium on Security and Privacy*, S&P, 2014.

[31] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. ASLR-Guard: Stopping address space leakage for code reuse attacks. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[32] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua. An evaluation of vectorizing compilers. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT, 2011.

[33] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. CCFI: cryptographically enforced control flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[34] M. Meissner. Tricks of a Spec master. https : / / gcc . gnu . org / wiki / summit2010 ? action = AttachFile&do = get&target=meissner2.pdf.

[35] V. Mohan, P. Larsen, S. Brunthaler, K. Hamlen, and M. Franz. Opaque control-flow integrity. In *22nd Annual Network and Distributed System Security Symposium*, NDSS, 2015.

[36] NaCL. Implementation and safety of nacl sfi for x86-64, 2015. https://groups.google.com/forum/#!topic/native-client-discuss/C-wXFdR2lf8.

[37] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine*, 11, 2001.

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

[39] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *33rd IEEE Symposium on Security and Privacy*, S&P, 2012.

[40] O. Peles and R. Hay. One class to rule them all: 0-day deserialization vulnerabilities in android. In *Workshop on Offensive Technologies*, WOOT, 2015.

[41] J. Pewny and T. Holz. Control-flow restrictor: Compiler-based CFI for iOS. In *29th Annual Computer Security Applications Conference*, ACSAC, 2013.

[42] S. Quirem, F. Ahmed, and B. K. Lee. Cuda acceleration of p7viterbi algorithm in hmmer 3.0. In *Performance Computing and Communications Conference*, IPCCC, 2011.

[43] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information System Security*, 15, 2012.

[44] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using ARM TrustZone to build a trusted language runtime for mobile applications. In *Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2014.

[45] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3, 2000.

[46] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *36th IEEE Symposium on Security and Privacy*, S&P, 2015.

[47] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary cpu architectures. In *18th USENIX Security Symposium*, USENIX Sec, 2010.

[48] J. Seibert, H. Okhravi, and E. Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2014.

[49] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2004.

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

[51] C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski. Exploiting and protecting dynamic code generation. In *22nd Annual Network and Distributed System Security Symposium*, NDSS, 2015.

[52] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *2nd European Workshop on System Security*, EUROSEC, 2009.

[53] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *14th ACM Symposium on Operating System Principles*, SOSP, 1993.

[54] L. Zhao, G. Li, B. De Sutter, and J. Regehr. Armor: Fully verified software fault isolation. In *9th ACM International Conference on Embedded Software*, EMSOFT, 2011.

[55] Y. Zhou, X. Wang, Y. Chen, and Z. Wang. Armlock: Hardware-based fault isolation for arm. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2014.