Oxymoron
Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing

Michael Backes
Saarland University, Germany
Max-Planck-Institute for Software Systems, Germany
backes@mpi-sws.org

Stefan Nürnberger
Saarland University, Germany
nuernberger@cs.uni-saarland.de

Abstract

The latest effective defense against code reuse attacks is fine-grained, per-process memory randomization. However, such process randomization prevents code sharing since there is no longer any identical code to share between processes. Without shared libraries, however, tremendous memory savings are forfeit. This drawback may hinder the adoption of fine-grained memory randomization.

We present Oxymoron, a secure fine-grained memory randomization technique on a per-process level that does not interfere with code sharing. Executables and libraries built with Oxymoron feature *'memory-layout-agnostic code'*, which runs on a commodity Linux. Our theoretical and practical evaluations show that Oxymoron is the first solution to be secure against just-in-time code reuse attacks and demonstrate that fine-grained memory randomization is feasible without forfeiting the enormous memory savings of shared libraries.

1 Introduction

Code reuse attacks manage to re-direct control flow through a program with the intent of imposing malicious behavior on an otherwise benign program. Despite being introduced more than 20 years ago, code reuse is still one of the three most prevalent attack vectors [1, 28], e.g., through vulnerable PDF viewers, browsers, or operating system services. Several code reuse mitigations have been proposed. They either detect the redirection of control flow [7, 12], or randomize a process's address space. Randomizations jumble the whole address space, with the intent of preventing code reuse attacks by making it impossible to predict where specific code resides.

Especially *Address Space Layout Randomization* (ASLR [23, 22]) has become widespread, but meanwhile has been shown to be ineffective [24, 25]. A promising avenue is the use of even finer randomization techniques that randomize at the granularity of functions, basic blocks or even instructions [18, 10, 16, 19, 17].

To be effective, fine-grained memory randomization must prevent an attacker from using information about the memory layout of one process to infer the layout of another process. This is a particular threat in the light of shared code originating from shared libraries. Hence, most recent fine-grained memory randomization solutions also randomize shared libraries for every single process [13, 21, 29]. As a result, there is no identical code in any two processes, which makes sharing impossible. A dysfunctional code sharing, however, increases the memory footprint of the entire system, likely on the order of Gigabytes, as we elaborate in Section 2.

To summarize: fine-grained randomization solutions presented so far come at the expense of tremendous memory overhead, which renders them impractical.

Oxymoron /ˌɒk.sɪˈmɔː.rɒn/ (noun)
Greek. A figure of speech that combines contradictory terms.

We present Oxymoron, which combines two seemingly contradictory methods: a secure fine-grained memory randomization with the ability to share the entire code among other processes. At the heart of Oxymoron is a new x86 calling convention we propose: *Position-and-Layout-Agnostic CodE (PALACE)*. This code uses no instructions that reference other code or data directly, but instead the instructions use a layer of indirection referred to by an index. This index uniquely identifies a target and hence remains identical when targets are randomized in memory. Consequently, the memory in which those instructions are stored does not change, thereby making it available to be shared with other processes.

Oxymoron cuts program code into the smallest sharable piece: a memory page. We randomize those pages and share them individually among processes. Each shared page appears at a different, random address in each process. We use the x86 processor's segmentation feature to disable access to the unique indices, which we organized in a translation table. This unique property of Oxymoron makes our solution more secure than fine-grained memory randomization solutions published so far.

To demonstrate the effectiveness and efficiency of Oxymoron, we implemented and evaluated a static binary rewriter for the Intel x86 architecture that emits PALACE executables and libraries with a very low run-time overhead of only 2.7%. By re-enabling code sharing, Oxymoron is the first memory randomization technique that reduces the total system memory overhead back to levels it was before fine-grained memory randomization, while simultaneously being the first solution that is secure against the just-in-time code reuse attacks recently proposed by Snow et al [26].

2 Problem Description

Before we describe our idea, we want to explain in more detail why any traditional fine-grained memory randomization necessarily makes sharing libraries impossible. The goal of fine-grained randomization is for every process to feature a memory layout that is as varied as possible from any other process. If we treat program code, which usually is en bloc, as a puzzle and shuffle the puzzle pieces throughout the entire address space, their combinatorial possibilities provide a high entropy. It is only possible to share those puzzle pieces individually as a memory page with other processes if the content of each piece is identical in each process. With traditional code, the content of those piece necessarily changes when their order in memory is rearranged, as we explain in the following:

Code references other code or other data using either absolute memory addresses, e.g., `call 0x804bd32`, or relative addresses, e.g., `call +42`. For absolute addresses it is obvious that different randomizations necessarily lead to different code and data addresses. As a result, the encoding of instructions that hold such addresses changes as well, thereby forfeiting the sharing with other processes. Relative addresses, in turn, make code independent of its load address in memory. However, in case of using code pieces that are randomized, the relative distances change as well. Here, for the same reason, those pieces cannot be shared across processes as they feature different relative addresses. Consequently, fine-grained memory randomization impedes common code sharing, which is a fundamental concept of *all* modern OSes.

Severity. Modern operating systems use code sharing automatically, and it is in effect because the running programs use the same libraries (C library, threading library etc.), i.e. their address spaces have identical code loaded.

To verify this claim, we conducted a simple experiment that shows the impact of code sharing and lack thereof. We used an unmodified Ubuntu 13.10 x86 operating system on a machine with 4 GB of RAM and evaluated how much RAM is saved due to code sharing. After booting to an idle desktop, the 234 running processes consumed a total 679 MB. Our analysis of memory page mappings in each process obtained from `/proc/<PID>/maps` revealed that most of the processes used the same set of shared libraries. As expected, most frequently the standard C-library `libc.2.17.so` was shared between all of the 234 processes. All mapped portions of `libc` sum up to 207,028 KB while only 885 KB of real memory are consumed. This is a savings of 206 MB for `libc` alone.

Figure 1 illustrates the top ten savings by library. In total, sharing instead of duplicating saved 1,388 MB of RAM on the idle Ubuntu desktop. When additionally starting the Firefox browser, the memory consumption was increased from 679 MB to 817 MB. The total amount of savings by sharing summed up to 1,435 MB of RAM, which is an additional savings of 47 MB.

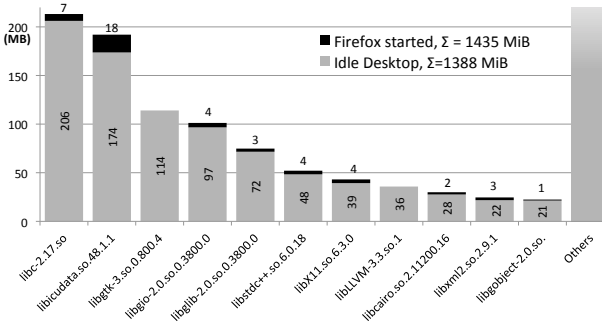

Figure 1: *Savings due to sharing of libraries. Idle desktop saves 1388 MB, with Firefox 47 MB more is saved.*

2.1 Threat Model

We assume a Linux operating system that runs a user mode process, which contains a memory corruption vulnerability. The attacker's goal is to exploit this vulnerability in order to divert the control flow and execute arbitrary code on her behalf. To this end, the attacker knows the process' binary executable and can precompute potential gadget chains in advance. The attacker can control the input of all communication channels to the process, especially including file content, network traffic, and user input. However, we assume that the attacker

has not gained prior access to the operating system's kernel and that the program's binary is not modified. Apart from that, the computational power of the attacker is not limited.

Moreover, for JIT-ROP attacks [26] to work, we assume that the process has at least one memory disclosure vulnerability, which makes the process read from an arbitrary memory location chosen by the attacker and report the value at that location. This vulnerability can be exploited any number of times during the runtime of the process. Note that the process itself performs the read attempt: both address space and permissions are implied to belong to the process.

3 High-Level Design of Oxymoron

To benefit from the best of both worlds – fine-grained memory randomization and code sharing – the challenge is to create a form of code that does not incorporate absolute or relative addresses, as we have already shown that both addressing schemes by definition suffer from being dependent on their randomization. An additional layer of indirection that translates unique labels to current randomized addresses allows the byte representation of code to remain the same, which enables code page sharing. However, this approach is difficult to realize as it is accompanied by four key challenges:

1. keeping the size of the translation table small in order not to increase the memory that we saved,
2. developing an efficient layer of indirection so that it is practical,
3. making the translation inaccessible by adversaries,
4. making the solution run on a commodity, unmodified Linux OS.

Overall Procedure. Oxymoron prevents code reuse attacks by shuffling every instruction of a program to a completely different position in memory so that no instruction stays at a known address, thereby making it infeasible for an adversary to guess or brute-force addresses. We use a three-step procedure (cf. Figure 2):

A) **Code Transformation**: The executable E is transformed to Position-and-Layout-Agnostic CodE (PALACE). The result is a PALACE-code executable P_E. The same applies to shared libraries, which can be treated like executables.

B) **Splitting**: The P_E code is then split into the smallest possible piece that can be shared among processes: a memory page. The code of P_E now consists of code pieces $P_E = p_1|p_2|\ldots|p_n$.

C) **Randomization**: At program load time, the pieces $p_1|p_2|\ldots|p_n$ are shuffled by the ASLR part of the operating system loader. In memory, their order is completely random and the pieces may have empty gaps of arbitrary size between them.

The first two steps only have to be done once, while the third step is performed at load-time of the executable P_E.

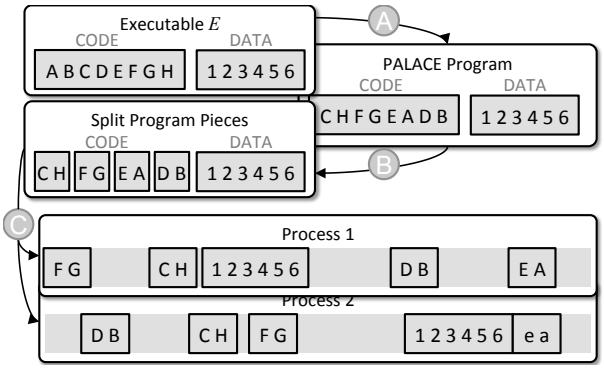

Figure 2: *The program is transformed and split once (A and B), then randomized at every process start-up (C).*

3.1 Code Transformation

To enable layout-agnostic code, all references to code and data are replaced with a unique label. Such a unique label is an assigned index into a translation table. This *Randomization-agnostic Translation Table* (RaTTle) in turn refers to the actual target (see Figure 3). This is the key to code sharing among processes, since the byte representation of the PALACE code does not change in the next step, when it is split and individual pieces are shuffled in memory.

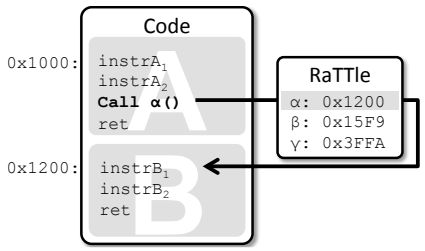

Figure 3: *Control-flow is redirected through the RaTTle rather than jumping to addresses directly.*

3.2 Splitting

Splitting ensures that the resulting pieces can be mapped into different processes at different addresses. As PALACE code references every target through a unique label in the RaTTle, it can be split without the need for traditional relocation, which rewrites addresses that hence have changed.

The PALACE code is split into page-sized pieces. If those pieces are later shuffled, it must be assured that the original semantics of the program are kept intact. This is essential when control flows from the end of one piece to the piece that was adjacent to it in the original program code. Thus, we need to insert explicit control flows between consecutive code pieces that might be moved away in a later stage of randomization. These explicit links only need to be inserted as the last instruction of a piece to ensure that control indeed flows to the intended successor (see Figure 4). After the links have been inserted, the code pieces can be randomized in memory without violating the original program semantics.

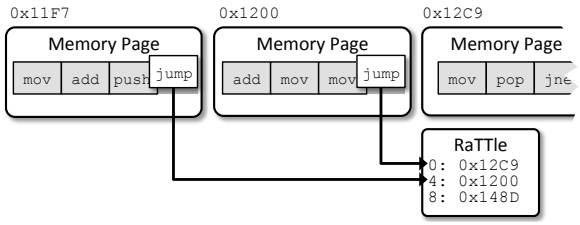

Figure 4: *Filling a page with instructions and linking them with explicit control flow transfers.*

3.3 Randomization

Modern OS loaders for shared libraries already support Address Space Layout Randomization (ASLR), i.e. they load the code, data, and stack segments at random base addresses. We leverage this fact by putting every memory page in its own loadable segment of the executable file or of the shared library. As the page-sized code pieces are already transformed to PALACE code, no relocation of addresses is needed. An ASLR-enabled commodity loader can blindly load all pieces at random addresses. Consequently, each process can have its own permutation of the randomization. Only the RaTTle needs to be kept up to date with a per-process randomization (see *"Populating the RaTTle"*).

3.4 Addressing the RaTTle

At first glance, it might seem we have only shifted the problem of addressing functions in code to securely addressing the RaTTle. However, our approach enables secure access to the RaTTle without access for adversaries. We first explain why we chose the more involved realization of the RaTTle and not existing approaches, such as a fixed address, a fixed register or the Global Offset Table (GOT). As already alluded to by Shacham et al. [25], the following techniques have drawbacks:

Fixed. Storing the RaTTle at a fixed address in memory allows for its address to be hard-coded in the instructions themselves. Unfortunately, a hard-coded address restricts the table to a fixed position. This fact can be exploited by an attacker.

GOT. Accessing the GOT is realized by using relative addresses, which forfeits sharing as discussed earlier. Moreover, several attacks are known that dereference the GOT [5].

Register. A dynamic address that is randomly chosen for every process could be stored in a dedicated machine register. However, this register would need to be sacrificed and every original use of that register must then be simulated with other registers or the stack. Moreover, a leakage vulnerability could reveal the address of the RaTTle.

Our Approach. Our RaTTle does not suffer from the aforementioned drawbacks. We use the x86 feature of memory *segmentation* to address and at the same time hide the RaTTle from adversaries. X86's segmentation is disused today because it has been superseded by memory paging. Memory paging, also called virtual memory, allows a fine-grained mapping of memory on a per-process basis and is much more versatile than segmentation. However, segmentation is still available in modern processors and in combination with paging allows for the security we need for the RaTTle. Additionally, as segmentation is a hardware feature and we can use it to implement the translation table, it is very efficient. Segmentation allows the memory to be divided in user-defined segments that may overlap. Segmentation is realized in the processor by adding a user-defined offset to all addresses the code handles (see Figure 5).

Segmentation allows different so-called *segment descriptors* to be created, each with their own *base address* and *limit*, i.e. the start and length of that segment. The list of these segment descriptors is kept in the *Global Descriptor Table* (GDT, see Figure 5). *Segment selectors* must then point to exactly one segment entry in that GDT. Segment selection is done using dedicated segment selector registers such as `CS` (Code Segment), `DS` (Data Segment),

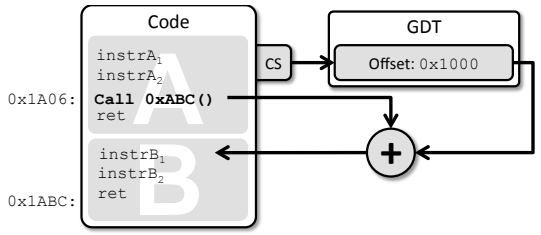

Figure 5: *Code using segments as offsets for addresses.*

SS (Stack Segment) and three general purpose segment selectors ES, FS and GS.

Position-and-Layout-Agnostic CodE (PALACE). The trick we use is the fact that segments can be selectively overridden on a per-instruction basis. In this way, a single instruction may use an addressing that is relative to the RaTTle, thereby indexing the RaTTle to change control flow or to access data. For example, `call *%fs:0x4` dereferences the double-word stored at `%fs:4` and calls the function stored at that double-word. If we let the segment selector FS point to the randomly chosen address of the RaTTle, we effectively index the RaTTle by an offset of 4 (see Figure 6).

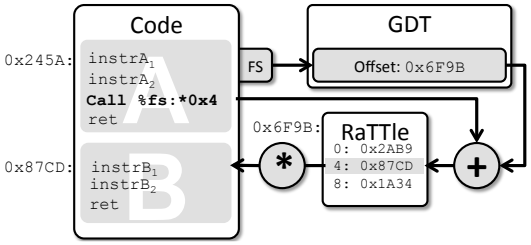

Figure 6: *The RaTTle in Action: Indexed through the GDT and dereferenced using an indirect call; all in* one instruction.

In PALACE code, we substitute each branch and jump instruction with an `%fs` segment override and a unique index. When not using the FS segment override, code does not have access to the RaTTle because it uses a different segment. The address of a segment, and hence of the RaTTle, cannot be read from user space because the local and global descriptor tables point to kernel space memory which is inaccessible from user space. This makes the address of the RaTTle inaccessible.

As a segment selector for the RaTTle, we chose the general purpose segment selector register FS, as already used in the example above. To the best of our knowledge, this register is unused. The only use we found is in the Windows emulator Wine that uses segmentation for its 16-bit Windows emulation.

Efficient Data Access. Data can be accessed in a similar way, but through the Global Offset Table (GOT). The GOT is used in position-independent code such as libraries anyway. We just need to substitute the way the address of the GOT is calculated with an indirection through the RaTTle. Further access is done through the GOT as in traditional position-independent code. This is explained in more detail in Section 4.5.

Populating the RaTTle. The RaTTle is the only part of the code that needs rewriting at load time. The RaTTle is empty in the ELF executable file on disk and its memory gets initialized by the loader with the help of relocation information. This relocation information points to the actual symbols that each RaTTle index refers to. The Linux loader automatically takes the relocation information to rewrite the RaTTle at program load [6].

4 Design Details

With the ingredients described earlier, we can put together our mitigation against code reuse attacks that is efficient, lightweight and shares code and data between processes.

4.1 Design Decisions

There are several ways to implement PALACE. A PALACE executable can be produced by a compiler, or it can be transformed from a traditional executable using static or load-time translation.

Compiler Support. The same way contemporary compilers support PIC, they can be augmented to emit PALACE code. Based on the principles of PALACE code introduced in the previous *Idea* section, the compiler needs to generate PALACE code and put it in subsequent memory-page-sized chunks. It is then ready to be loaded by a traditional loader that permutes the chunks prior to execution of the code.

Static Translation. If the source is not available, an existing executable can be transformed to PALACE by means of static translation [14, 15]. Static translation reads an executable or shared library file from disk, disassembles it, transforms the instructions, and writes a modified executable file back to disk. In our scenario, static translation keeps most of the instructions untouched while only replacing code and data references with the appropriate indirection through the RaTTle.

Load-time Translation. Load-time Translation can be regarded as a static translation that happens automatically at very load-time, after the executable or library has been read from disk into memory but before it starts execution. This method is often referred to as *binary rewrit-*

ing. Its advantage is that a process can be randomized at every startup. In our scenario, however, we do not need load-time translation as we can achieve a randomization at load-time with the specially crafted PALACE chunks in the executable file.

Our Choice. We want to stress that Oxymoron can be implemented by a compiler that simply emits PALACE code in the first place instead of traditional code. We could have implemented Oxymoron as a compiler solution. However, this would have required us to modify existing compilers. Instead, we built a legacy-compatible solution that uses static translation and can be built on an existing fine-grained memory randomization framework, which already uses static translation. We built Oxymoron on the existing framework Xifer provided by Davi et al. [13].

In theory, a static translation approach may seem fragile because it needs a perfect disassembly. However, static translation can be tuned to reliably disassemble code generated by a particular compiler with known and carefully chosen parameters. Besides, in this paper we use the translation from traditional x86 code to PALACE code as a comprehensible running example that demonstrates how PALACE code looks in contrast to traditional x86 code.

In both cases, compiler and static translation, the generated PALACE code of the executables and libraries can be read by a commodity Linux. The Linux OS loader will detect the executable as being ASLR-enabled and will randomize its base address. Unfortunately the commodity loader does not randomize the program segments individually but keeps their relative distances. For traditional position-independent code that was necessary so that code in the `.text` section can still reference objects in the `.data` section by their relative distance to the current instruction pointer. However, for PALACE this limitation is not required. We want to achieve a more fine-grained randomization by allowing an individual randomization of each program segment, which could be as small as a memory page. This can be achieved by requesting a special linker in the program header, which randomizes the segments individually.

4.2 Setting up the RaTTle

The RaTTle needs to be populated with all references in the executable and the table needs to be loaded at a random address. Moreover, one table does not suffice for the interaction of several shared libraries. Before we can use PALACE code, we need to set up the RaTTle as follows:

1. Assign every reference in code a unique number that will act as an index into the RaTTle,
2. Fill the RaTTle with the actual, current, random addresses of the original targets, and
3. Set up segmentation so that a free segment selector points to the RaTTle and we can index the RaTTle.

In step 1, the absolute addresses of the original program are saved in a hash set. Then, every address is assigned an ascending index. This ensures that the table does not grow unnecessarily large. Because the final, random addresses are unknown before the process is started, the RaTTle cannot be filled until start-up of the process. As we want to avoid modification of the operating system loader, we chose a method that is able to fill the RaTTle using only traditional features of the loader. Such a feature is relocation. Relocation information tells the loader which objects in the executable file or in the library must be overwritten with current addresses at load time. Therefore, we add relocation information for each RaTTle index to the final executable/library file. This ensures that the loader rewrites each index so that it points to the corresponding position of code or data that this index represents. As a result, the randomized addresses of the code pieces are automatically written into the RaTTle by the operating system loader.

4.3 Setting up Segmentation

In order to find the RaTTle in memory, we need to set up segmentation so that a pre-defined segment points to the beginning of the table. Unfortunately, we cannot use relocation information for this purpose, because neither setting up segmentation nor setting segment selectors is supported by relocation information. Setting up segmentation via the Global Descriptor Table (GDT) would require kernel modifications. Since the goal is to avoid operating system modifications in order to stay legacy compatible, this is not an option. Luckily, the x86 architecture additionally supports a so-called *Local Descriptor Table* (LDT). The LDT can be switched for every address space, so that Linux emulates a per-process LDT. This is a perfect feature for enabling Oxymoron on a per-process basis.

The set-up of the LDT and the segment selector that points into the LDT is done in initialization code. To this end, we leverage the ELF executable format's initialization code that resides in the `.init` section. Code in this section is ensures to be executed before any other code. This init code figures out the address at which the RaTTle has been randomly loaded by the loader and sets up the LDT accordingly. After the initialization code has run, the segment selector `FS` points to the random address

of the RaTTle. The PALACE code can now work as intended.

4.4 Control Flow and Data

Code. Control flow branches or function calls that target another memory page need to be replaced with an indirection through the RaTTle. The simplest case is a direct `call` or an unconditional `jmp` to a different place in code:

Address	Before	After
8050512:	`call 0x8050c08`	`call %fs:4`
RaTTle:		[0] ..... [4] `0x8050c08`

Only branches that reference code outside of the current memory page must go through the RaTTle. Code and data access within one memory page may be encoded position-relative (e.g., `call +90`).

If the to-be-replaced instruction is an indirect jump, the translation is slightly larger due to the fact that x86 does not support two levels of indirection. It is either possible to use the RaTTle to get the address of the second indirection and then dereference that using an indirect jump or to use a trampoline. We use a trampoline because it is slightly faster:

Address	Before	After
8050512:	`jmp *0x80a00012`	`jmp %fs:4`
80a00012:	`8050c08`	`8050c08`
RaTTle:		[0] ..... [4] `jmp *80a00012`

A slightly more involved case is a conditional jump because there is no equivalent conditional indirect jump. Our solution is a bit more involved:

Address	Before	After
8050512:	`cmp %eax, %ebx`	`cmp %eax, %ebx`
8050514:	`jne 0x8050590`	`jne 0x8050518`
8050516:		`jmp 0x805051a`
8050518:		`jmp *%fs:4`
RaTTle:		[0] ..... [4] `0x8050590`

An indirect jump, such as `jmp *%eax` does not need to be replaced at all. However, the used register (in this example `%eax`) must point to the correct randomized position in memory. This is either ensured by the compiler that generated PALACE code or by the translation from traditional code. In either case, a register is loaded with a code address. Optionally, this address is modified to mimic jump tables or C++ vTables, and then the indirect jump transfers control flow to the address stored in the register. To load a code address to the register before it is modified, a fixed address is copied to the register. This is similar to `mov $0x8402dbc, %eax`. In the case of PALACE, this step needs an indirection to conceal the actual address and to make the address exchangeable by the RaTTle. In PALACE code this register loading looks like this: `mov %fs:$0x4, %eax`. This copies an address stored as an entry in the RaTTle to the register `%eax`. Then, some mathematical operations can be performed, such as adding the offset into C++ vTables and finally the indirect jump is performed as in traditional x86: `jmp *%eax`.

Data Access. Accessing data through the RaTTle is done in exactly the same way. An indirect memory operation is used to read or write data from or to an address stored in the RaTTle. `mov %fs:$0x4, %ebx` is used to read the first entry (4 bytes) of the RaTTle into register `%ebx` and vice versa the operation `mov %ebx, %fs:$0x8` copies the register `%ebx` to the second entry (8 bytes) of the RaTTle.

4.5 Inter-Library Calls and Data

Control flow and access to data is not restricted to one library or executable. Naturally, these code elements frequently use each other's functions and data. Some operating systems, like Windows, use relocation information to directly patch the control flow so that it points into a library after it has been loaded. Linux, on the other hand, uses the procedure linkage table (PLT) to link calls to libraries with the advantage of lazy loading.[1] In contrast, we use an indirection through the RaTTle for every library call or access to global library data because this approach conceals the actual address of the loaded library and has only minor performance impact.

Inter-Library Data. Libraries can export data to be used by the executable main process or other shared libraries. Since it is known a priori which data is accessed in another library, each reference gets a place-holder in the GOT which can be accessed as described above. When the appropriate library is loaded by the loader, it automatically updates the GOT thanks to the relocation info pointing to this entry in the GOT.

The following is an example of typical position-independent code that uses a GOT to access data: The code is first calling the next instruction, thereby pushing its own address as a return address to the stack. Following, this very address is popped off the stack to get the

[1] First, the PLT entries do not point to the actual procedure inside a library because it has not been loaded yet. Instead, they point to code that loads the library and then rewrites the PLT to link the call to the actual target procedure.

absolute address of the currently running code. The address of the GOT is calculated by adding a known offset.

Address	Before	
8050512:	`call 0x8050517`	Call next instruction
8050517:	`pop %ebx`	ebx ← 8050517
8050518:	`add $1234, %ebx`	ebx ← GOT
805051e:	`mov 4(%ebx), $1`	GOT[4] ← 1

When transforming this piece of code to PALACE, only the calculation of the GOT needs to be substituted. In this case, the three former instructions get compressed to a single instruction with segment override. Interestingly, this is a faster method of accessing the GOT than the currently used PC-relative addressing.

Address	After	
8050512:	`mov %fs:4, %ebx`	ebx ← GOT
805051e:	`mov 4(%ebx), $1`	GOT[4] ← 1
RaTTle:	`0x805174B`	Points to GOT

Inter-Library Calls. Inter-library calls are calls from one loaded library to another or from the main executable to a library. In theory, these calls are no different from a call within the same library or executable because the RaTTle can simply point to code in another library. However, in practice, this would require the RaTTle to reflect all possible combinations of loaded libraries. Therefore, we resort to a solution in which every loaded library brings its own RaTTle and an inter-library call acts as a trampoline that changes the segment selector `FS` to point to the corresponding RaTTle of another library prior to jumping into that library (see in Figure 7).

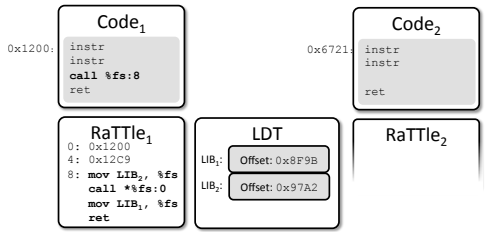

Figure 7: *Inter-Library Calls: Because the indices overlap, a new RaTTle needs to be set up before those calls.*

Please note the missing "*" in the `call %fs:8` of Figure 7, which means the RaTTle is not de-referenced rather than used as a trampoline. This trampoline then lets `FS` point to the index of the other library's RaTTle without the need to know the exact address. Suppose the function that we want to call is stored at index 0 in RaTTle_2, but RaTTle_1 is currently active. The code in Figure 7 first sets `FS` to point to RaTTle_2. RaTTle_2 is the second selector in the LDT. Hence, the trampoline code in RaTTle_1 assigns $10111_{bin} = 23$ to `FS`, which corresponds to a segment selector of "2" (see Appendix A). The trampoline code then jumps to index 0, which now corresponds to currently active RaTTle_2. Because the trampoline uses a `call` instruction to finally call into the other library, control flow returns to the trampoline where `FS` is restored to its former value.

4.6 Debugging

Debugging information augments the executable or library file with annotations describing which memory addresses correspond to which variables or lines of code. These stored addresses must be compatible with Oxymoron randomized addresses. Since Oxymoron is implemented as a static translation tool, the original debugging information needs to be translated as well. Currently Oxymoron supports the common DWARF [3] file format which can be read by the `gdb` or other debuggers. This way, it is possible to teach `gdb` the randomized addresses so that `gdb` can still step through the code, inspect variables etc. like for the non-randomized executable.

5 Evaluation

In this section, we evaluate the effectiveness of Oxymoron empirically as well as theoretically. In order to demonstrate the efficiency, we used the de-facto standard performance benchmark SPEC CPU2006 as well as micro benchmarks to measure cache hit/miss effects.

First, we inspect the security of the RaTTle itself to verify that it did not open the flood gates for other attack vectors. Then, we compare the slightly different randomization of memory pages that this solution entails to the more classical memory randomization solutions in order to get an understanding of the implied security.

5.1 Practical Security Evaluation

We tested our randomization solution against real-life vulnerabilities and exploits. The documented vulnerabilities `CVE-2013-0249` and `CVE-2008-2950` both allow arbitrary code execution by means of return-oriented programming [2]. `CVE-2013-0249` targets the `libcurl` library which handles web requests and is used in dozens of popular programs, including ClamAntiVirus, LibreOffice, and the Git versioning system. The exploit for this vulnerability is crafted in such a way that it triggers a buffer overflow in `libcurl` with the ability to overwrite a return address and ultimately execute a chain of ROP gadgets. The severity of this bug lies in the fact that it can be triggered remotely when `libcurl` accesses

a prepared resource that is under the control of the adversary. In order to test the exploit, we used the 'curl' downloader executable in version 7.28.1, which internally uses `libcurl`. We could successfully run arbitrary code by assembling ROP gadgets at our discretion. After curl had been rewritten to use Oxymoron, the exploit was no longer possible as the addresses that are needed to successfully mount the attack are unknown due to the randomization at every program start.

Similarly, the vulnerability `CVE-2008-2950` allows for arbitrary code reuse in the PDF library `poppler`, which is used by many popular programs such as LibreOffice, Evince and Inkscape. A specially prepared PDF file can trigger an arbitrary memory reference in the `poppler` library, ultimately leading to a code reuse attack. After our attacks against `pdftotext` using `libpoppler` 0.8.4 were successful, we applied Oxymoron. Since the memory address of the PALACE-protected process were no longer known, the exploit was rendered unsuccessful after applying Oxymoron to the pdftotext executable.

5.2 Security of the RaTTle

Because processes are protected by $W \oplus X$ (stack execution prevention), no code can be injected by an attacker. Hence, the only possibility is to reuse existing code. This existing (PALACE) code is littered with `%fs`-prefixed instructions that implicitly point to the RaTTle due to the sheer fact they incorporate a reference to `%fs`. However, the situation is identical to finding ROP gadgets in a classical program, as an attacker needs to know their randomized position in memory in order to chain them together. The fact that this address is not known to an attacker prevents the reuse of code. In fact, the probability of guessing a correct address is negligible (see subsection *"Theoretical Security Evaluation"*).

The RaTTle holds lots of random addresses and, at first glance, seems like a valuable target for an attacker. The security of the RaTTle originates from the fact that its address is unknown and that its content cannot be accessed. All `%fs`-instructions are mere replacements for control flow branches and as such only use the RaTTle as a layer of indirection without ever knowing the actual address of the landing position. If an `%fs`-instruction is a replacement for data access, the same holds true: The RaTTle is only used for indirect access of the actual data. In general, the x86 architecture does not support revealing addresses that segments point to. The only way to read the address is to parse the GDT or LDT which both reside in kernel space. To access the LDT, a user mode program needs to issue a special syscall. Even if a program would consist of ROP gadgets to issue this syscall, he would still need to know the addresses of the required ROP gadgets. So this can be reduced to finding special instructions that can be used as ROP gadgets. This has a negligible probability as explained in *"Theoretical Security Evaluation"*.

5.3 Enhanced Security of the RaTTle

It is possible to further enhance the security of the RaTTle by making it completely inaccessible. The segmentation principle of the x86 architecture allows to distinguish code access from data access. This way, it is possible to set up two different RaTTles, one for code going through `%fs` and one for data going through `%gs`. First of all, in a program without self-modifying code, there should be no instructions that read data using the `%fs` code segment selector. Even if there were, the processor would prohibit such access. Further, it is possible to move the RaTTle completely outside of the normal, otherwise flat[2] data segment (`%ds`). This results in the inability for code to ever access the RaTTle without using proper segment selectors, because it no longer resides in the accessible segment. This is an effective protection against leakage and disclosure attacks (see subsection *"Disclosure Attacks"*). Also, the call stack could be protected using this method. If return addresses are not saved on the regular stack, but rather on a side stack in a reserved area inside the RaTTle, there is no way for memory disclosure vulnerabilities to ever read return addresses and thus they cannot gain information about function addresses.

5.4 Theoretical Security Evaluation

In this subsection we elaborate on why the entropy of memory page granularity randomization is still sufficient for fine-grained randomization and why it is much higher than traditional ASLR.

First, we show that the entropy induced by a page-granular randomization is high enough in the sense that the adversary has only negligible probability of successfully guessing an address. We model the adversary's goal as mounting a code reuse attack against a running program consisting of the executable and its loaded libraries. Hence, his goal is to know the address of either a particular function f of interest (return-into-libc attack) or of several particular instructions $i_1 \dots i_k$ to build gadgets from (ROP attack). Since the contents of a memory page can be extracted from the executable file, the attacker can determine in which memory page the instruction in question resides. Therefore, the success of the adversary re-

[2] A flat segment is a segment that covers the entire address space, i.e. `0x00000000` to `0xFFFFFFFF` on a 32-bit system. This is the default for Windows, Linux and MacOS.

lies on the probability of knowing the address of a particular memory page.

Every memory page is assigned a random address at load-time. Thus, the first page can choose 1 out of n possible page-aligned address slots. The second 1 out of $n-1$ and so forth. For p total process pages to lay out in memory, this yields a total of $\frac{n!}{(n-p)!}$ combinations. The adversary's probability of correctly guessing one address is hence the reciprocal $\frac{(n-p)!}{n!}$. In a 32 bit address space, we have $n = 2^{19} = 524,288$ possible page addresses. The probability of guessing one page correctly therefore is 2^{-19}. That scenario is intuitively identical to ASLR which only randomizes the base address of the code. However, when finding ROP gadget chains, the page granularity drastically lowers the chance of success compared to ASLR because several pages have to be guessed correctly. For a 128 kB ($p = 32$ pages) executable to lay out in memory, the adversary's probability of guessing the correct memory layout therefore is:

$$Pr\left[Adv_{layout}\right] = \frac{(n-p)!}{n!} = \frac{(2^{19}-2^5)!}{2^{19}!} = 2^{-608}$$

Leakage Attacks in ASLR. A leakage vulnerability inadvertently reveals a valid, current address inside the running program. If the adversary additionally knows which object or function has been leaked, he knows the address of that object/function. In the case of ASLR, he can then infer the current addresses of all other objects or functions because ASLR has shifted the entire code segment in memory by changing its base address. Consequently, the relative distances between functions stay exactly the same.

To model the leakage attack, we assume the adversary exploits an existing leakage vulnerability thereby learning a valid address. We assume that this address depicts the beginning of a particular function that the adversary knows. That such a leaked address actually constitutes a function pointer is not very likely but here it models the best-case scenario for the adversary. Hence, the following calculations give a upper bound of success for an adversary.

More formally, the adversary has access to an oracle that can tell which function f has leaked and the adversary can use the leakage vulnerability to learn the current address of A_f of the function f. The adversary can then calculate their difference in memory by calculating their difference in the executable file. As their relative positions did not change in ASLR, the adversary can infer the current address of f' by calculating the difference to the leaked function f. In the case of traditional ASLR, the address of any function f' can be calculated with probability 1. Ultimately, the success probability of the adversary entirely depends on the likelihood of finding such a leakage vulnerability.

Leakage Attacks in Oxymoron. In our case of memory page granularity shuffling, the relative distance between functions varies in general since the code segment is not just shifted en bloc. For any leaked pointer f, there is a chance that it resides in the same memory pages as the desired function f'. For an equal distribution of f' in p pages, the likelihood of f' being in the same page as f is $\frac{1}{p}$. For a program of a total size of only one memory page (4kB), both functions f and f' must reside in the same memory page. Under the assumption that both functions are uniformly distributed, the probability for both to appear in the same memory page is $\frac{1}{p}$ for a program size of p pages. Hence

$$Pr[Adv^{PALACE}_{ret2libc}] \leq \frac{1}{p} \quad \text{and} \quad Pr[Adv^{PALACE}_{ROP}] \leq \frac{1}{p^k}$$

Disclosure Attack. We distinguish between a leakage and a disclosure vulnerability. A disclosure vulnerability allows an attacker to read arbitrary memory content given its address. Snow et al. proposed just-in-time code reuse, which showed that a disclosure vulnerability can significantly reduce the security of fine-grained memory randomization [26]. Just-in-time code reuse repeatedly exploits a memory disclosure vulnerability to map portions of a process' address space with the objective of reusing the so-discovered code in a malicious way. In a fine-grained randomization, the memory pages are scattered across the address space and scanning with arbitrary memory addresses is very likely to end up in unmapped memory. In order not to trap into unmapped memory, they rely on a leakage attack to learn a valid address and then start from this address by disassembling the code in order to follow control flow instructions. Even fine-grained randomization can be reversed using their technique by transitively following the control flow.

However, in our setting of PALACE code, no control flow branch can be followed by reading memory as such a branch only constitutes an offsets into the RaTTle. In order to resolve branches such as `call *%fs:4`, the attacker would need to know the address of the RaTTle or `%fs`, which is not possible, as alluded to earlier. The only chance an attacker has is to rely on a leakage vulnerability to get a valid address. If that address points to data it is useless to the attacker. If it points to code, the attacker can only use a disclosure vulnerability to get the contents of up to a whole memory page (4KB). Otherwise, he is likely to overrun the page and end up in unmapped memory which triggers a page fault that kills the program.

5.5 Effectiveness of Memory Page Sharing

To have a set basic programs one would typically find on a Linux machine, we used the *busybox* project, which incorporates 298 standard Linux commands. Those command line programs were started and their memory footprint was measured using `/proc/<PID>/maps`. On average, they mapped 14.9% more code pages than their unmodified original. Their data pages were unmodified. Only the RaTTle consumes memory (see Subsection 6.1). Compared to fine-grained memory randomization solutions that impede code page sharing, Oxymoron on average saves about 85% of program memory.

6 Performance Evaluation

To evaluate the efficiency of Oxymoron, we did not only use standard command line tools from busybox but conducted CPU benchmarks with PALACE-enabled programs using the de facto standard SPEC CPU2006 integer benchmark suite. All benchmarks were performed on an Intel Core i7-2600 CPU running at 3.4 GHz with 8 GB of RAM.

Static Translation Overhead. Before the executable and libraries can be shuffled in memory, they either need to be compiled with an PALACE-enabled compiler or they must be converted using static translation (cf. section 3). Even though the translation only needs to be performed once, it must be efficient. We measured the rewriting time for all benchmark programs of the Spec CPU suite. The rewriting process is not exactly linear, but on average achieves between 35,000 and 700,000 instructions per second. An overview of the timings of several programs is given in Table 1.

Benchmark	Total # of Instructions	Rewriting Time (s)
483.xalancbmk	1,111,779	4.321
403.gcc	942,244	3.667
471.omnetpp	238,978	0.316
400.perlbench	322,084	1.084
445.gobmk	226,661	6.744
464.h264ref	170,942	0.396
456.hmmer	54,582	0.116
458.sjeng	40,438	0.101
473.astar	32,502	0.032
401.bzip2	28,087	0.056
462.libquantum	15,788	0.024
429.mcf	12,268	0.024

Table 1: *Timings for static rewriting that needs to be done at least once. The total # of instructions include the executable and all its shared libraries.*

The number of instructions per benchmark reflect the total number of instructions from the executable file itself plus its dependent libraries. Note, that this measurement rewrites the entire C-library and other dependent libraries again for each benchmark and is hence slower than just translating the main executable.

Run-Time Overhead. The run-time overhead introduced by the translation through the RaTTle as well as the introduction of `jmp` instructions to connect pages (cf. section 3) is measured in Figure 8. The average run-time overhead of all benchmarks is only 2.7% for the PALACE code and 0.1% for the additionally needed chunking in memory page-sized pieces (4096 bytes).

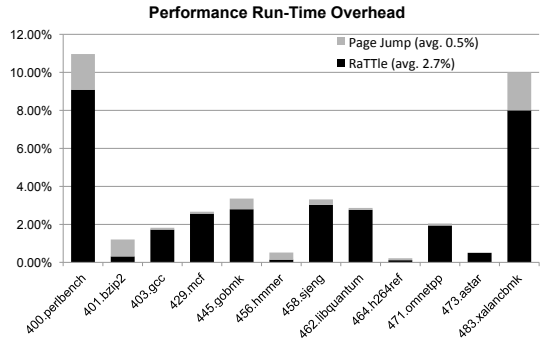

Figure 8: *SPEC CPU2006 integer benchmark results.*

Cache Miss Penalty. We also evaluated the cache effects of our randomization. This is important, since modern processors assume locality of code, which might be thwarted by wild jumping in the code due to the randomization. Keeping cache effects in mind, our implementation optimizes jumping behavior in order to optimize performance under real-life conditions. Our cache experiments showed that PALACE and the randomization have no measurable cache effect.

For this impact to be measured, we handcrafted code consisting of interdependent `add` instructions with a total length of one L1 cache line. These instructions are aligned in memory in such a way that they start at the beginning of a cache line and re-occur such that every cache set and every cache line is filled after execution. We inserted equidistant `jmp` instructions and measured the overhead of 100,000 runs on an Intel Core i7-2600 (32 KB L1 cache, 64 bytes per line). Our results show that the performance impact is not measurable up to every seventh instruction being a `jmp`. If every sixth instruction is a `jmp`, a negligible overhead of 0.4% is introduced. Our analysis of the `busybox` code showed that after translating it to PALACE, on average indeed every 6^{th} instruction was a branch or jump.

6.1 Memory and Instruction Overhead

Compared to a traditional program, the introduction of PALACE code replaced control flow branches with other, `%fs`-relative, instructions. For all SPEC2006 benchmark executables, on average 9% +-1.7% of all instructions are calls that needed to be replaced by indirections through the RaTTle. GOT indirect calls through the RaTTle are only 0.03% of all instructions.

Additionally, a PALACE binary executable file is slightly larger than a traditional executable file because each code page (4 KB) is a separate ASLR-enabled section in the executable file.

During run-time, the memory footprint also slightly increases because the RaTTle has to be kept in memory. Of course, this run-time memory usage is accompanied with the achieved goal of memory savings due to the sharing of code pages with other processes.

Encapsulating each memory page in a separate segment in the ELF file requires the allocation of one section header and one program header per page. A section header is 40 bytes and the ELF program header is 32 bytes which leads to an overhead of 72 bytes per 4096 byte memory page, or $\approx 1.76\%$. Figure 9 depicts both the increase of instructions due the static translation as well as the increase of the ELF section and program headers.

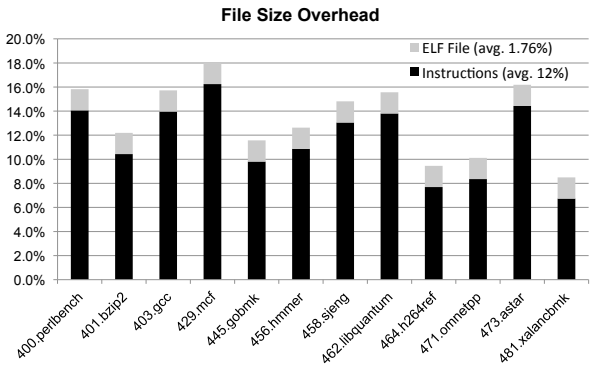

Figure 9: *Memory overhead after static translation.*

Run-Time. The size of the RaTTle depends on how many references the code has. If a target is referenced more then once, e.g., the GOT, only one index is saved in the RaTTle. For all files that belong to the SPECint CPU2006 benchmark, on average 19% of the code segment had to be added in the form of a RaTTle.

7 Related Work

Over the course of the last several years, code reuse attacks and their mitigation has been an ongoing cat and mouse game. Some of the code reuse mitigation techniques address the problem at its roots by preventing buffer overruns or by confining the control flow to the destined control-flow graph. Other mitigation techniques make it hard for the adversary to guess or brute-force addresses that are necessary for successful execution of malicious code.

In this section, we focus on approaches that use fine-grained memory randomization as a means to mitigate code reuse attacks and work that nullifies memory randomization or even fine-grained memory randomization.

One way to categorize fine-grained memory randomization solutions is by their implementation: There exist compiler-based solutions, static or load-time translations, and dynamic translations. Another category dimension is whether they randomize only once, every time the program starts, or even continuously during program execution.

Compiler-Based Solutions. If a program is not randomized, an adversary can learn the layout, i.e. addresses, of all functions and gadgets and hence use them in a ret2libc or ROP attack. The idea of compiler-based approaches is to randomize the layout of a program and to install differently randomized copies on different computers so that the program layout is not predictable for an adversary.

Cohen et al. [10] suggested compiling different versions of the same program. In a modern setting this technique can be applied within an AppStore to distribute individually randomized software. Similarly, Franz et al. [16, 19] have suggested automating this compiler process and generate a different version of a program for every customer. The authors suggest that app store providers integrate a multicompiler in the code production process. However, those approaches have several shortcomings: First, app store providers have no access to the app source code. This requires the multicompiler to be deployed on the developer side, who has to deliver possibly millions[3] of app copies to the app store. Second, the proposed scheme requires software update processes to correctly patch app instances that in turn differ from each other. Finally, the most severe drawback of compiler-based solutions is the fact that the diversified program remains unchanged until an update is provided, which increases the chance of an adversary compromising this particular instance over time.

[3] According to Gartner [4], the number of app downloads is about 102 billion in 2013.

Similar to Oxymoron is the idea of using a compiler-based solution to divide a shared library into even more fragments. Code Islands [30] follows this path and compiles groups of functions to several shared libraries instead of one shared library containing all the functions. These (potentially thousands of shared library files) are then put in a container whose format is understood by a modified loader which maps the libraries in the particular process. However, their solution needs a modified loader to support the proprietary format. Executables then need to load literally thousands of shared libraries, while each library constitutes a single function.

In contrast, Bhatkar et al. [8] presented a source code transformer and its implementation for x86/Linux. The main idea is to augment any source code with the capability of self-diversification for each run. In particular, features are added to the source code that allow the program to re-order its functions in memory in order to mitigate code reuse attacks. Their tool can also be applied to shared libraries if their source code is available. However, their solution induces a run-time overhead of 11% and apparently needs access to the source code.

Static Translation. Static translation reads an executable or shared library file from disk, disassembles it and transforms the instructions according to a predefined pattern within the executable file itself. Kil et al. [20] use static translation for their Address Space Layout Permutation (ASLP). ASLP performs function permutation without requiring access to source code. The proposed scheme statically rewrites ELF executables to permute all functions and data objects of an application. The presented scheme is efficient and also supports re-diversification for each run. However, only the functions themselves are permuted, not their content.

Pappas et al. proposed randomizing instructions and registers within a basic block to mitigate return-oriented programming attacks [21]. However, the proposed solution cannot prevent return-into-libc attacks (which have been shown to be Turing-complete [27]), since all functions remain at their original position.

Load-Time Translation. Load-time translation solutions are similar to static translation but apply the translation at load time in order for the processes to benefit from a re-randomization at each run. This can be achieved by several means, such as rewriting the binary file after it has been loaded but before execution [29, 13]. Such solutions usually suffer from the fact that each execution either needs a translation/rewriting phase each time a process is started or they need a prior static analysis phase [29].

Dynamic Translation. Dynamic translation leaves the original file untouched and does not apply binary rewriting but the program undergoes a *dynamic translation*, i.e. the instructions are transformed as they are executed. Dynamic translation is very similar to Just-in-Time (JIT) compilation but usually translates from and to the same instruction set architecture. For example, Bruening proposed the DynamoRIO framework in his PhD thesis [9]. DynamoRIO is able to perform run-time code manipulation. ILR (instruction location randomization) [18] randomizes the location of each single instruction in the virtual address space. For this, a program needs to be analyzed and re-assembled during a static analysis phase. This is why ILR induces a significant performance overhead (on average 13%), and suffers from a high space overhead, i.e., the rewriting rules reserve on average 104 MB for only one benchmark of the SPEC CPU benchmark suite. For direct calls, ILR can only randomize the return address in 58% of the calls, meaning that for a large number of return instructions, ILR needs to do a live translation for un-randomized return addresses to runtime addresses.

Constant Re-Randomization. To the best of our knowledge, there are only two papers that actually implemented and benchmarked re-randomization. Curtsinger et al. [11] have implemented an LLVM compiler modification that injects code, which adds the functionality to re-randomize the address of functions every 500 ms. According to [11], their overhead of code, heap and stack (re-)randomization is 7%.

Giuffrida et al. [17] changed the Minix microkernel to re-randomize itself every x seconds. This is achieved by maintaining the intermediate language of the LLVM compiler for the compiled kernel modules. However, this procedure has a significant run-time overhead of 10% for a randomization every $x = 5$ seconds or even 50% overhead when applied every second.

Common Shortcomings and Nullification. All the related work on fine-grained memory randomization has in common that they either do not randomize shared libraries, or if they do, the difference introduced in the shared libraries prohibits code sharing.

Furthermore, it is unclear whether fine-grained memory randomization alone is enough to protect against code reuse attacks. Recently, Snow et al. [26] showed that given a memory disclosure vulnerability it is possible to assemble ROP gadgets on-demand without knowing the layout or randomization of a process. They explore the address space of the vulnerable process step by step by following the control flow from an arbitrary start position. After they have discovered enough ROP gadgets

they compile the payload so that it incorporates the actual current addresses that were discovered on-site.

Snow et al. also proposed potential mitigations of their own attack. However, the proposed solutions are either very specific to their heap spraying exploitation or are as general and slow as frequent re-randomization of a whole process. The latter is not even secure if the attack takes place between two randomization phases.

To the best of our knowledge, in this paper we present the first solution that addresses both problems: (1) It is secure against the new just-in-time ROP by Snow et al. (2) It profits from code sharing despite secure randomization.

8 Discussion

In this section we would like to discuss the general applicability of Oxymoron but also its limitations.

The PALACE code presented in this paper only relies on segmentation as an additional hardware feature. Hence, Oxymoron also works in virtualized environments. We successfully tested Oxymoron in software and hardware virtual machines as well as on a para-virtualized Linux using the Xen hypervisor.

The solution presented herein was implemented for the 32 bit x86 architecture. While its 64 bit successor has limited supported for segmentation, the necessary offset functionality of `%fs` segment registers is still available. However, in 64 bit mode, segments do no longer support to set a limit, which makes the RaTTle accessible as data if its address is known.

Another interesting avenue that we did not investigate is just-in-time (JIT) compiled code, such as the Java runtime environment. Those JIT-compilers would need to be adapted in order to emit PALACE-enabled code, otherwise the traditional code they emit is not protected.

9 Conclusion

We presented a novel technique for fine-grained memory randomization that still allows sharing of code among processes. This makes fine-grained memory randomization practical as the memory overhead is significantly reduced in contrast to other randomization solutions. Oxymoron is effective, i.e., code reuse attacks can be mitigated, memory leakage vulnerabilities can no longer be used to revert the randomization, and we presented the first solution to be secure against just-in-time code reuse attacks. The randomized addresses are protected by hardware means, which is an unprecedented security level with a run-time overhead of only 2.7%.

An interesting side effect of our PALACE code is that accessing the Global Offset Table (GOT) uses fewer instructions than the state-of-the-art technique of using PC-relative addressing. Maybe our method could be a slightly faster alternative for accessing the GOT.

References

[1] Common Weakness Enumeration – Top Software Vulnerabilities. `http://cwe.mitre.org/top25/index.html`.

[2] Database of Common Security Vulnerabilities and Exposures. `http://cve.mitre.org`.

[3] Dwarf 2.0 debugging format standard. `http://www.dwarfstd.org/doc/dwarf-2.0.0.pdf`.

[4] Gartner Says Mobile App Stores Will See Annual Downloads Reach 102 Billion in 2013. `http://www.gartner.com/newsroom/id/2592315`.

[5] How to hijack the Global Offset Table with pointers for root shells. `http://www.open-security.org/texts/6`.

[6] Executable and Linking Format (ELF). Tool Interface Standards Committee, May 1995.

[7] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)* (2005), ACM, pp. 340–353.

[8] BHATKAR, S., SEKAR, R., AND DUVARNEY, D. C. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium* (2005), USENIX Association.

[9] BRUENNING, D. *Efficient, Transparent and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[10] COHEN, F. B. Operating system protection through program evolution. *Computer & Security 12*, 6 (Oct. 1993), 565–584.

[11] CURTSINGER, C., AND BERGER, E. D. Stabilizer: statistically sound performance evaluation. In *ACM SIGARCH Computer Architecture News* (2013), vol. 41, ACM, pp. 219–228.

[12] DAVI, L., DMITRIENKO, A., EGELE, M., FISCHER, T., HOLZ, T., HUND, R., NÜRNBERGER, S., AND SADEGHI, A.-R. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *Symposium on Network and Distributed System Security (NDSS)* (2012).

[13] DAVI, L. V., DMITRIENKO, A., NÜRNBERGER, S., AND SADEGHI, A.-R. Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and arm. In *8th ACM SIGSAC symposium on Information, computer and communications security (ACM ASIACCS 2013)* (2013), ACM, pp. 299–310.

[14] DE SUTTER, B., DE BUS, B., AND DE BOSSCHERE, K. Link-time binary rewriting techniques for program compaction. *ACM Transactions on Programming Languages and Systems (TOPLAS) 27*, 5 (2005), 882–945.

[15] EUSTACE, A., AND SRIVASTAVA, A. Atom: A flexible interface for building high performance program analysis tools. In *Proceedings of the USENIX 1995 Technical Conference Proceedings* (1995), USENIX Association, pp. 25–25.

[16] FRANZ, M. E unibus pluram: massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 workshop on New security paradigms* (2010), ACM, pp. 7–16.

[17] GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proceedings of the 21st USENIX conference on Security symposium* (2012), USENIX Association, pp. 40–40.

[18] HISER, J. D., NGUYEN-TUONG, A., CO, M., HALL, M., AND DAVIDSON, J. W. ILR: Where'd My Gadgets Go? In *IEEE Symposium on Security and Privacy* (2012).

[19] JACKSON, T., SALAMAT, B., HOMESCU, A., MANIVANNAN, K., WAGNER, G., GAL, A., BRUNTHALER, S., WIMMER, C., AND FRANZ, M. Compiler-generated software diversity. In *Moving Target Defense*, vol. 54 of *Advances in Information Security*. Springer New York, 2011, pp. 77–98.

[20] KIL, C., JUN, J., BOOKHOLT, C., XU, J., AND NING, P. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *ACSAC* (2006).

[21] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE Symposium on Security and Privacy* (2012).

[22] PAX TEAM. `http://pax.grsecurity.net/`.

[23] PAX TEAM. PaX Address Space Layout Randomization (ASLR). `http://pax.grsecurity.net/docs/aslr.txt`.

[24] SHACHAM, H. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)* (2007).

[25] SHACHAM, H., JIN GOH, E., MODADUGU, N., PFAFF, B., AND BONEH, D. On the Effectiveness of Address-space Randomization. In *ACM Conference on Computer and Communications Security (CCS)* (2004).

[26] SNOW, K. Z., MONROSE, F., DAVI, L., DMITRIENKO, A., LIEBCHEN, C., AND SADEGHI, A.-R. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy* (2013).

[27] TRAN, M., ETHERIDGE, M., BLETSCH, T., JIANG, X., FREEH, V., AND NING, P. On the expressiveness of return-into-libc attacks. In *Proceedings of the 14th international conference on Recent Advances in Intrusion Detection* (2011), Springer-Verlag.

[28] VAN DER VEEN, V., CAVALLARO, L., BOS, H., ET AL. Memory errors: the past, the present, and the future. In *Research in Attacks, Intrusions, and Defenses*. Springer, 2012, pp. 86–106.

[29] WARTELL, R., MOHAN, V., HAMLEN, K. W., AND LIN, Z. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *ACM Conference on Computer and Communications Security (CCS)* (2012).

[30] XU, H., AND CHAPIN, S. Address-space layout randomization using code islands. In *Journal of Computer Security* (2009), IOS Press, pp. 331–362.

A LDT Selector Bits

The actual value that a segment selector must hold is not merely an index to the GDT/LDT, but is defined by the architecture set as follows:

Bits 15 - 3	Bit 2	Bit 1 - 0
Number of the entry	0=GDT, 1=LDT	Privilege Level

As user mode is in Ring 3, bits 0 and 1 must be set to 11_{bin}. The use of the LDT forces us to set bit 2 to 1_{bin}. The index "0" of the LDT yields a valid value for the segment selector of 111_{bin} or 7 in decimal.