Oxymoron

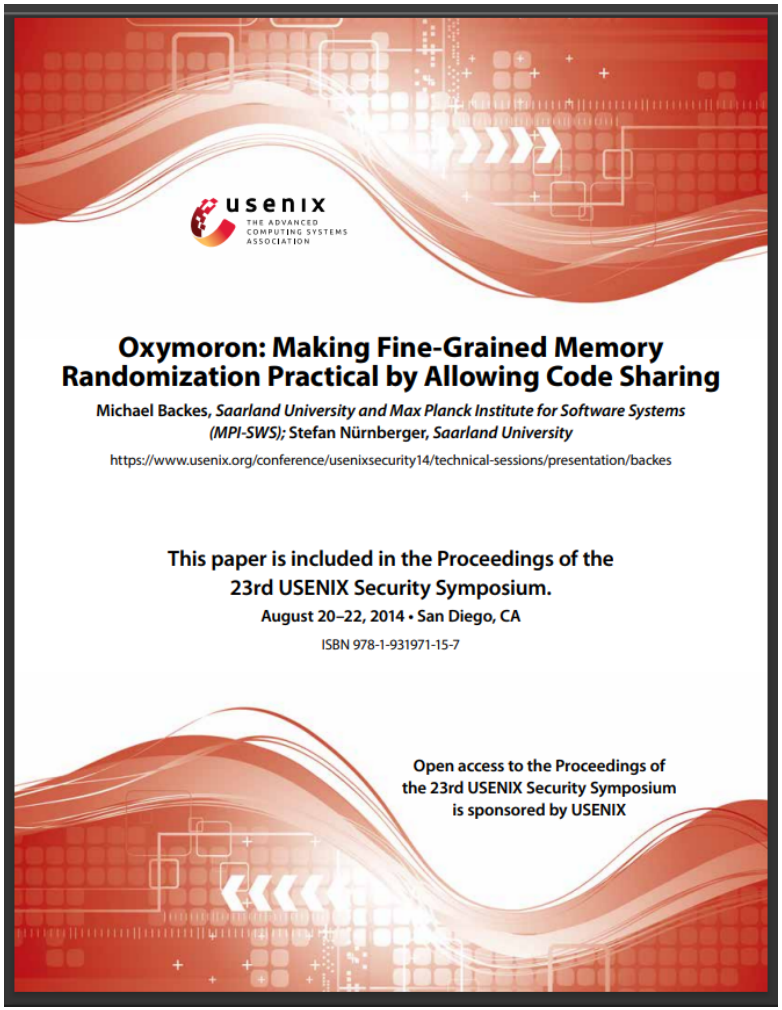
Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing

Raghuveer Shivakumar (Raghu)

University of Colorado, Boulder

2022-09-28

Oxymoron



Authors: Michael Backes, Saarland University and Max Planck Institute for Software Systems (MPI-SWS); Stefan Nürnberger, Saarland University

Oxymoron /ˌɑksɪˈmɔːrɑn/ (noun) *Greek. A figure of speech that combines contradictory terms.*
Def: A secure fine-grained memory randomization with the ability to share the entire code among other processes.

Threat Model

- Linux OS that runs a user process, which contains a memory corruption vulnerability
- Attacker's goal is to exploit this vulnerability in order to divert the control flow and execute arbitrary code on her behalf.
- The attacker can control the input of all communication channels to the process.
- However, we assume that assume that the attacker has not gained access to the operating system kernel, and that the program binary is not modified.

Background

- **Absolute Addressing**: `call 0x804bd32`
- **Relative Addressing**: `call +42`

Background (cont.)

ASLR

- Randomizes the base address of the process's virtual address space/stack/heap/shared libraries

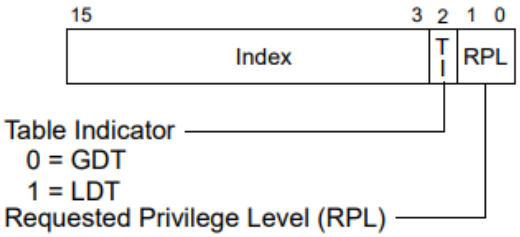
Fine Grained Memory Randomization

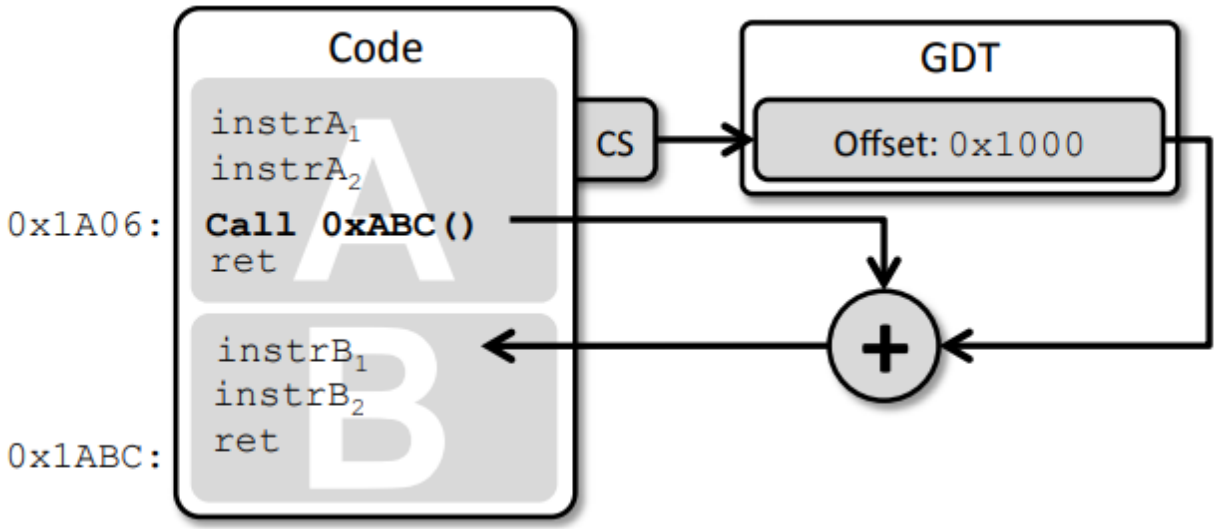
- Randomizes the base address of each memory page

Background (cont.)

Segmentation

- Segmentation: A process is divided into segments, each of which has a base address and a limit (size)
  - Segment registers(fs/gs/cs) → Segment selectors(Idx,TI,RPL) → Segment descriptors(GDT/LDT) → Segment base address



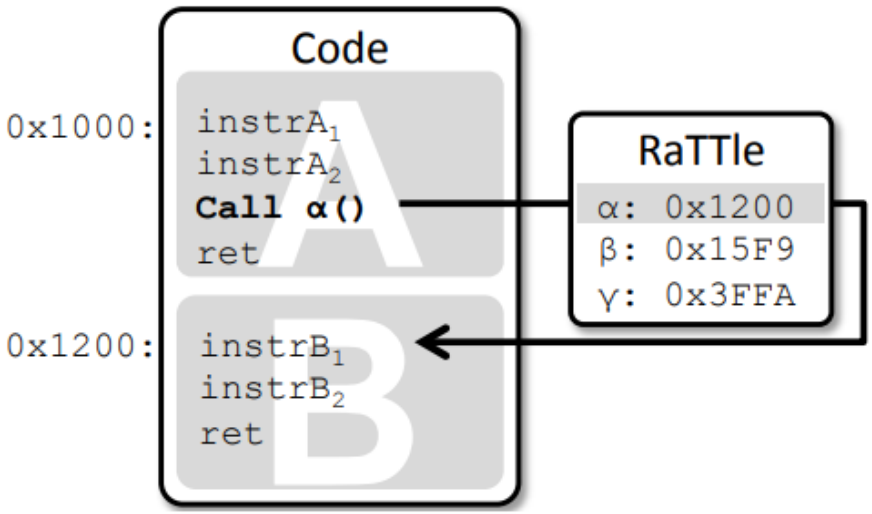


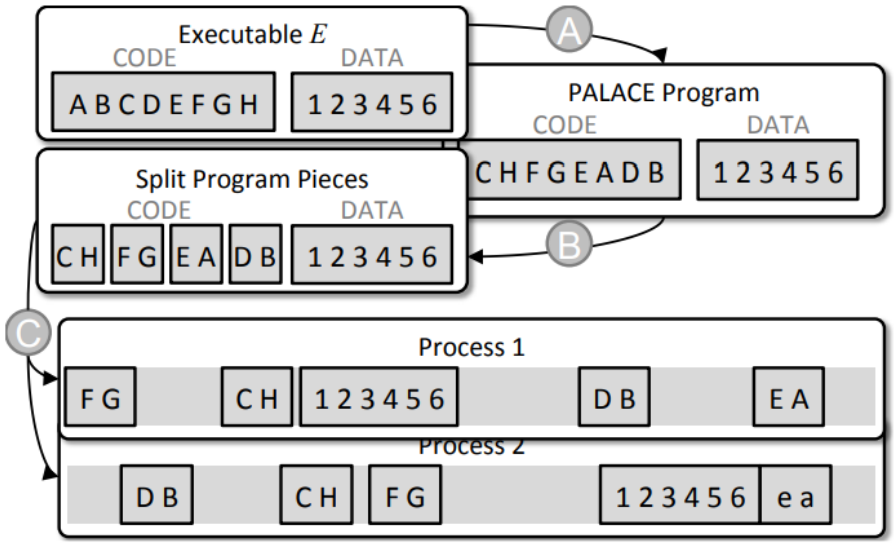
`Segment Selector:` fs, gs, cs, ss, ds, etc

`Segment Descriptor Table:` GDT, LDT,IDT, etc

Oxymoron Core Ideas

- At the heart of Oxymoron is a new x86 calling convention called: *Position-and-Layout-Agnostic CodE* ***(PALACE)***, which is made possible using an address table called ***RaTTle*** (*Randomization-agnostic Translation Table*)





Implementation

1. Splitting
2. Randomization
3. Addressing the RaTTle

Splitting

- Split the PALACE code into page-sized pieces.
- Use jumps to stitch the pieces together and maintain the control flow.

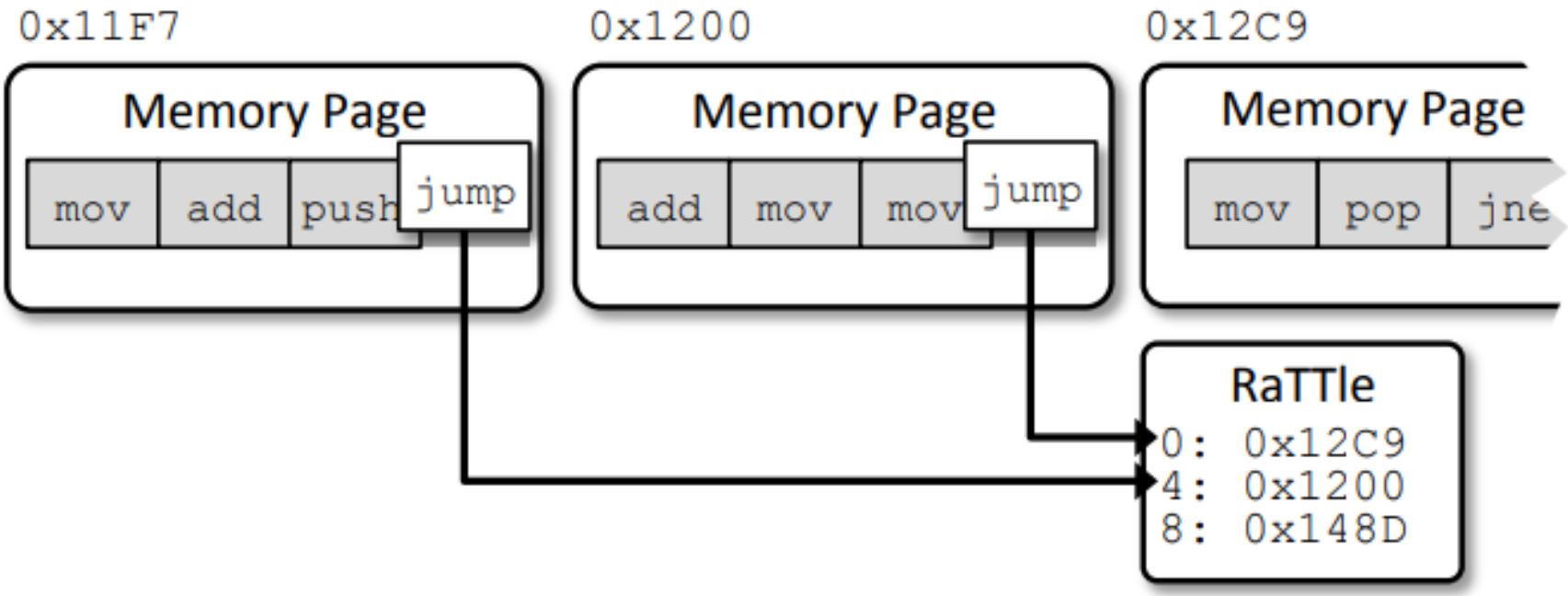


Figure 4: *Filling a page with instructions and linking them with explicit control flow transfers.*

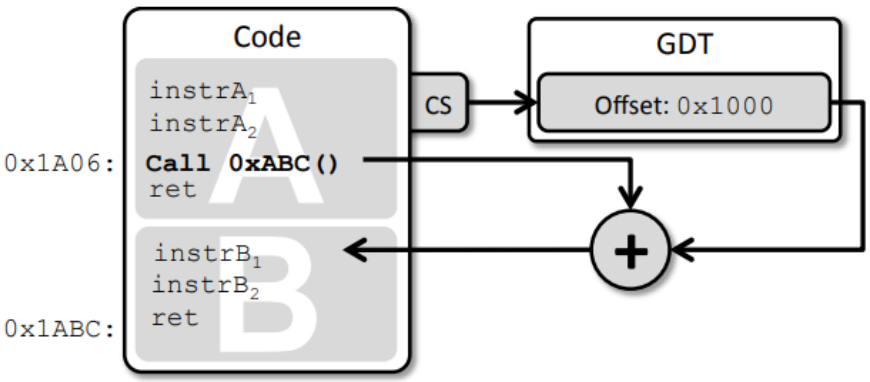
Randomization

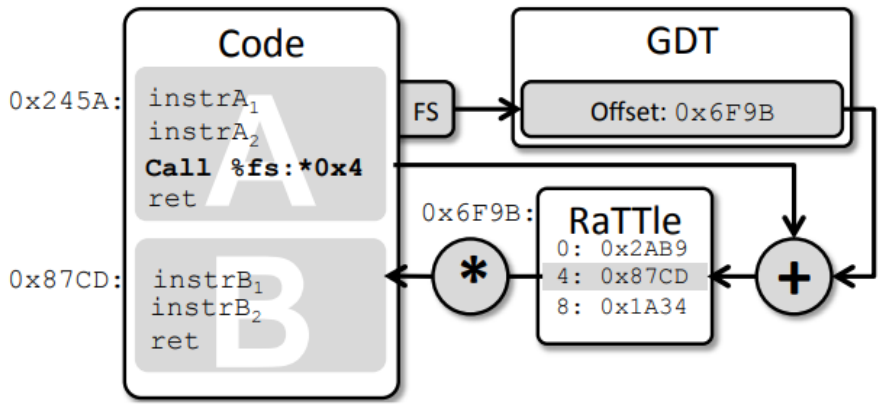
- Randomize the base address of each page.
  - By requesting a special linker in the program header that randomizes the segments individually.
  - remember: ASLR only randomizes the base address of the entire process's virtual address space.
- Keep RatTle entries consistent with the randomized addresses across processes.

Addressing the RaTTle

Hint: Segmentation

- Modern day OSes have segmentation support but mostly use paging for memory management.
- Wine (Windows Emulator) is the only tool that uses explicit segmentation for memory management.
- We use segmentation to leverage the GDT/LDT to disable direct access to the page addresses.





Design Details

PALACE Translation

- Compiler Pass - Emit PALACE code and put in page sized chunks
- Static translation - Disassemble $\rightarrow$ Transform to PALACE $\wedge$ Add RaTTle entries $\rightarrow$ Assemble
- Binary Rewriting/Load-time Translation - Same as static translation but at load time. Adv: randomizes at every load.

Choice: Static translation (for now)

Setting up Rattle

- Cannot introduce high overheads - use a hash table entry for every unique reference to a page.

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

Examples

Example 1: Direct addressing

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

Example 2: Indirect addressing

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

Evaluation

- CVE-2013-0249 - Libcurl attack
- 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
- After re-writing the binary with Oxymoron, the attack was no longer successful due to randomized indirection at every start.

Evaluation (cont.)

Theoretical Evaluation:

Randomization Possibilities:

- Choosing the first page of n pages: $1/n$, Choosing a second page: $1/n - 1$, and so on.
  - For p total process pages to lay out in memory: $n!/(n-p)!$

Brute Force Possibilities:

- For p total process pages, it would be the reciprocal of the randomization possibilities: $(n-p)!/n!$

Evaluation (cont.)

In 32 bit address space, we have $2^19 = 524288$ possible page addresses. In ASLR, the possibily of guessing would be $1/524288$. In Oxymoron, the possibily of guessing would be based on the block size. For eg, if we have have a block size of 128 kb (p = 32 pages) to lay out in memory, the possibily of guessing would be:

$$P(layout) = \frac{(2^{19} - 2^5)!}{2^{19}!} = 2^{-608}$$

Evaluation (cont.)

- All benchmarks were performed on **Intel Core i7-2600 CPU running at 3.4 GHz with 8 GB of RAM.**

Translation Overhead:

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.*

Evaluation (cont.)

Runtime Overhead:

- Control flow maintenance overhead: 0.5% of the total runtime
- Indirection overhead: 2.7% of the total runtime

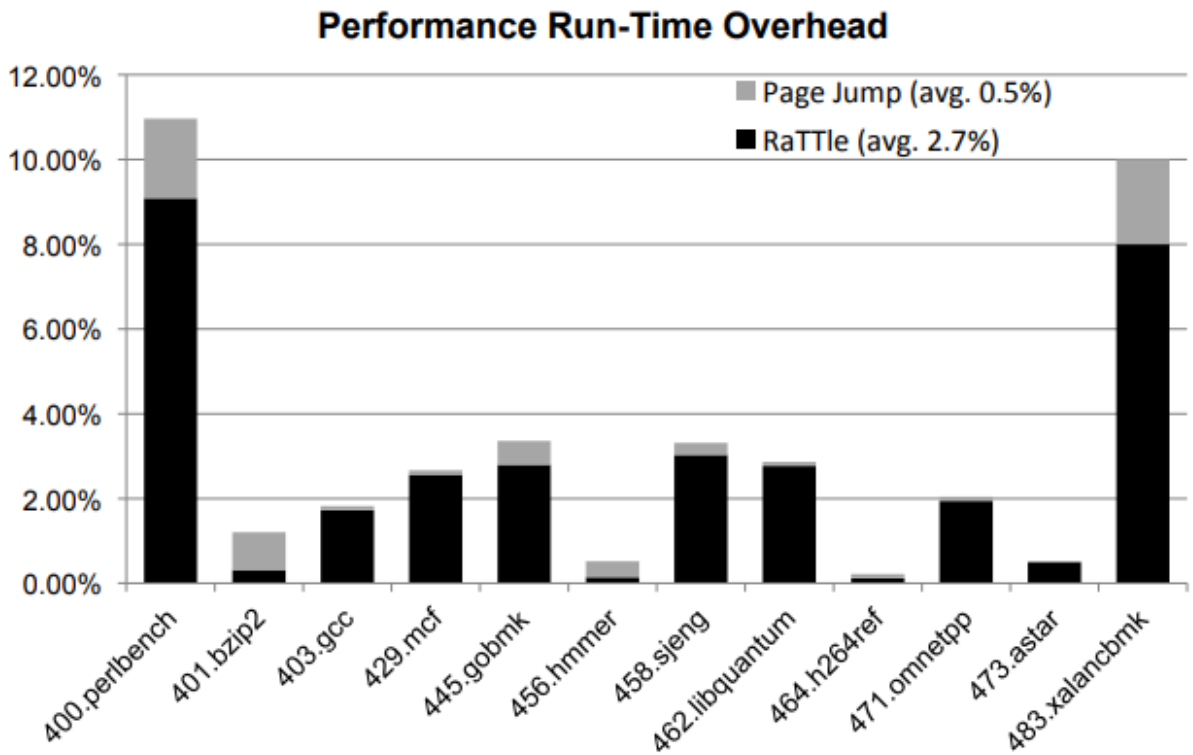


Figure 8: *SPEC CPU2006 integer benchmark results.*

Evaluation (cont.)

Memory Overhead:

- **Section header:** 40 bytes, **ELF program header:** 32 bytes => 72 bytes per page -> 1.76% of the total memory footprint

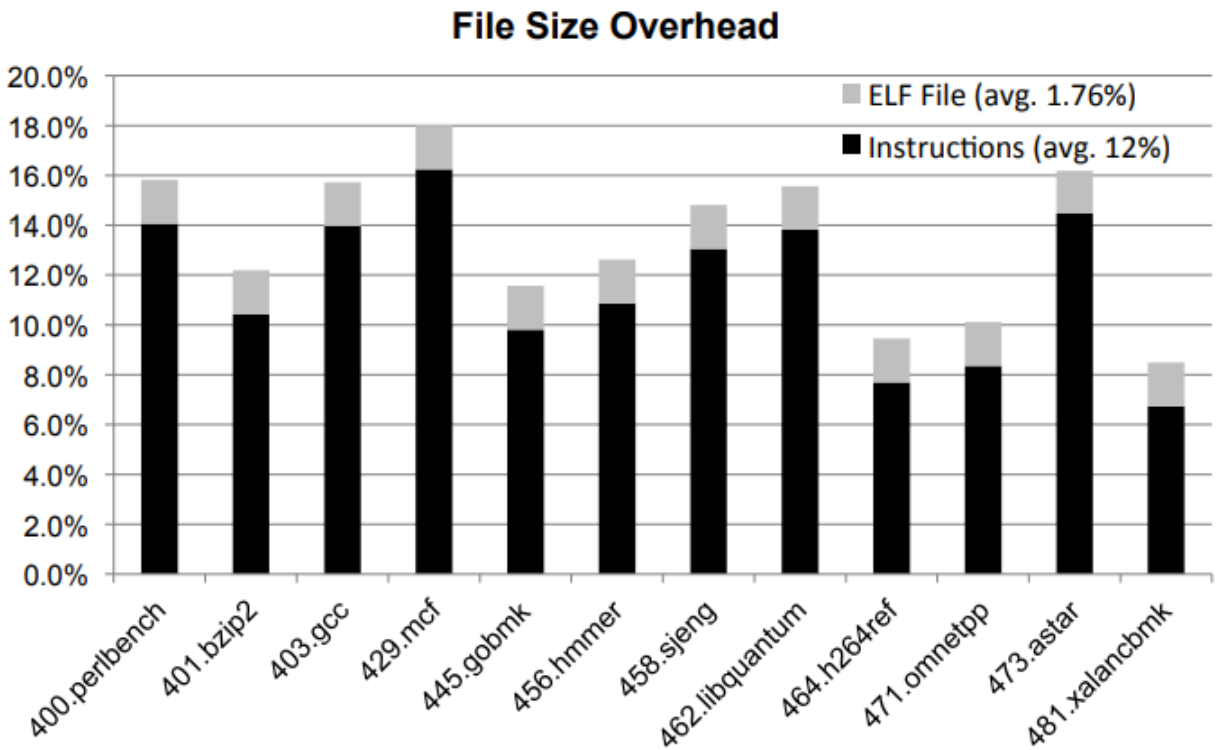


Figure 9: *Memory overhead after static translation.*

Questions?