Targeted Memory Runtime Analysis

David Piché

December 8th, 2023

Polytechnique Montreal

DORSAL Laboratory

Agenda

1. Introduction
2. General approach
3. Implementation
4. Results
5. Discussion
6. Conclusion

Introduction

- Memory issues in C/C++ are still prevalent
  - Use-after-free
  - Memory leaks
  - Out-of-bound writes
  - And much more…

- Runtime memory analysis
  - Collects data on runtime execution
  - Uses this data to detect memory errors and act accordingly

Introduction : State of the Art

- Current memory runtime analysis tools uses similar techniques:
  - Redzones
  - Shadow Memory
  - Pointer Tagging

- Memcheck of the Valgrind suite uses shadow memory to detect memory errors.
  - Slowdown factor of 22.2 .
- AddressSanitizer requires compile-time instrumentation but achieves a slowdown factor of 1.71 .
  - Raises the memory overhead significantly by the addition of redzones.

The General Approach

- In many cases, the developer will have some information on the class of objects predisposed to memory errors (such as object size).
- As opposed to using compile-time or runtime instrumentation to verify memory accesses (which protects all allocated memory) , we let protected memory accesses trigger a SIGSEGV signal, with an option to protect only a subset of allocations based on those aforementioned factors.
- The smaller subset of memory allocations selected by the developer will ensure lower overall time overhead.

The General Approach

- This means, for our library, these important steps:
  1. Get control before the access
  2. Verify a valid access
  3. Unprotect the object
  4. Perform the access
  5. Re-protect the object
  6. Continue the execution

- We focus on the Intel x86_64 architecture.

Implementation: Protecting memory objects

To protect dynamically allocated objects, we have implemented two methods:

- **Pointer tainting** using bits 47 to 63
  - System call arguments may be tainted!
- Protect **memory pages** with *mprotect()* and the PROT_NONE flag
  - Currently we allocate an entire page per object

Implementation: Bounds checking

In order to verify the access, use bounds checking

- We need information regarding the memory access:
  - Which register contains the tainted address
  - Information on base, index, scale, offset to compute address for bounds checking

- Use **Capstone** to disassemble instruction and retrieve relevant information.

Figure 1: Capstone logo [1]

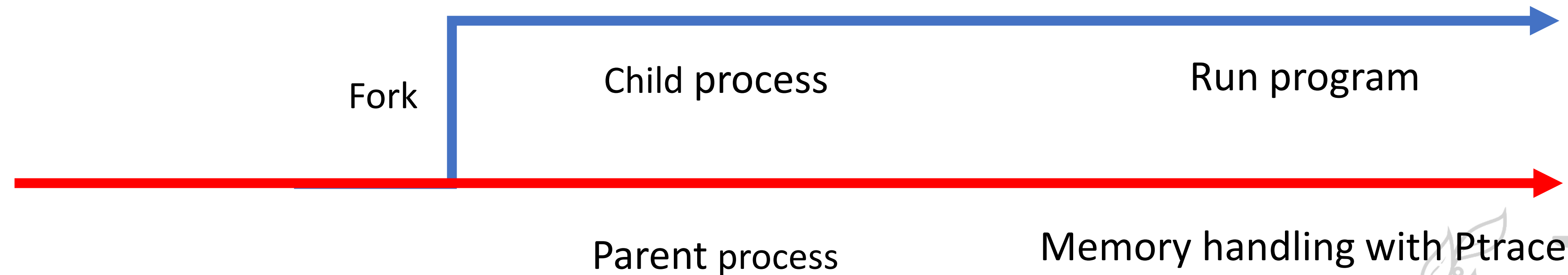
Implementation: Gaining control of the program

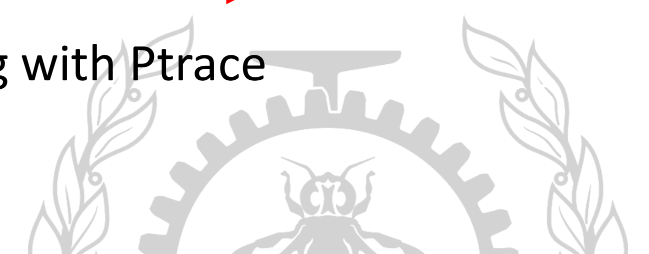
In order to unprotect the memory zone before the memory access instruction and re-protect it after, we consider two main approaches:

- **PTrace**
- **Out-of-line code execution**

Implementation: Ptrace

- Use Ptrace with 2 different processes:
  - The child process runs the program with the special allocators
  - The parent process takes care of memory handling
  - Ptrace used for communication between processes and single-step
  - Using the CLONE_VM flag with clone() to make communication between the two threads easier



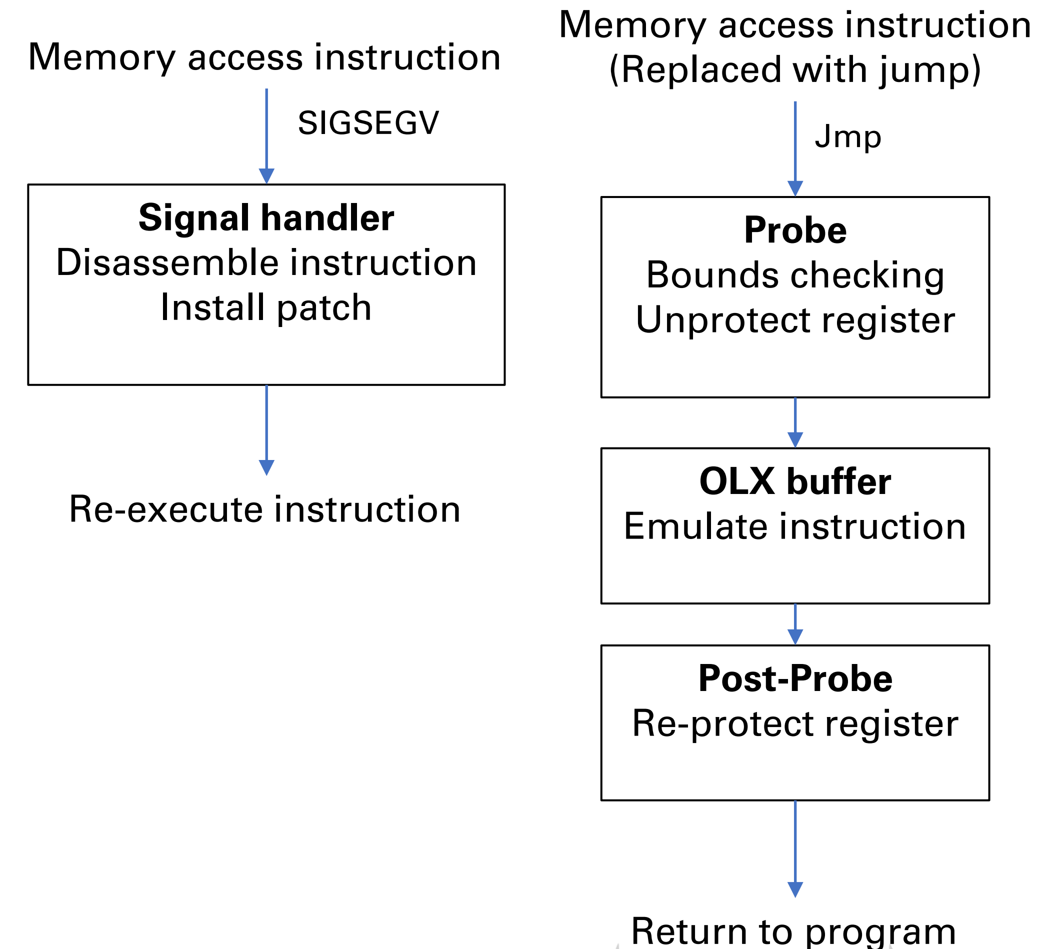


Implementation: LibOLX

- The **LibOLX** [2] library from Olivier Dion specializes in out of line code execution.
- Produces binary instructions that emulate the memory access instruction.
- Binary instructions may be specified by the developer to be run before and after the emulated instruction.
- Most of the time overhead using this library comes from the SIGSEGV signal handling of our approach.

Implementation: Libpatch

- The **Libpatch** [3] library from Olivier Dion specializes in inserting probes at runtime.
- Install patch at first encounter of instruction.
- OLX buffer emulates instruction.
- Post-probe allows us to re-protect address.
- SIGSEGV signals are not raised for subsequent executions of the same instruction, reducing overhead.



Results: Configuration

- For each approach explored, 4 combinations are tested:
  - **Ptrace-mprotect**: Using *Ptrace* while protecting memory with *mprotect*
  - **Ptrace-taint**: Using *Ptrace* with pointer tagging
  - **OLX-taint**: Using the LibOLX library with pointer tagging
  - **Patch-taint**: Using the LibPatch library with pointer tagging

- The benchmarks were done on a AMD Ryzen 7 5700g with 32 Gb of RAM.

- The operating system was Ubuntu 22.04.2 LTS with the 5.19.0-50-generic Linux kernel.

Results: Allocation Distribution

- Analyzing the distribution of the sizes of allocated objects is important if we wish to use it as a factor to select a subset of allocations to protect.

- For 15 benchmarks of SPEC CPU 2017 benchmark suite [4], a wrapper library tracks every object allocation/deallocation.

Results: Allocation Distribution

- The majority of the benchmarks have a large number of allocations of small objects (less than 127 bytes).
- The current implementation of the *mprotect* approach when protecting memory pages will incur a significant memory overhead.

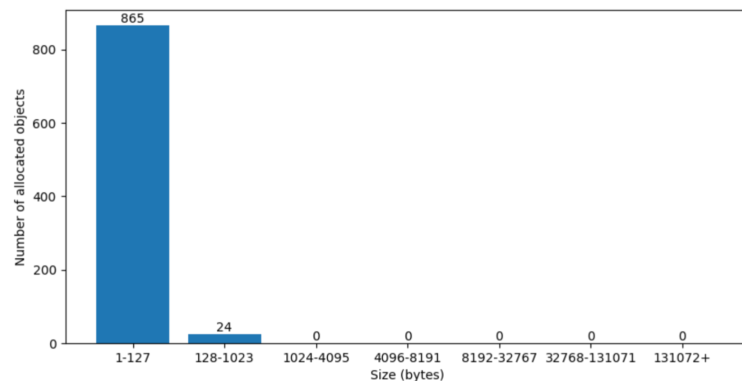


Figure 2: Allocation distribution of object sizes for 523.xalancmbk_r

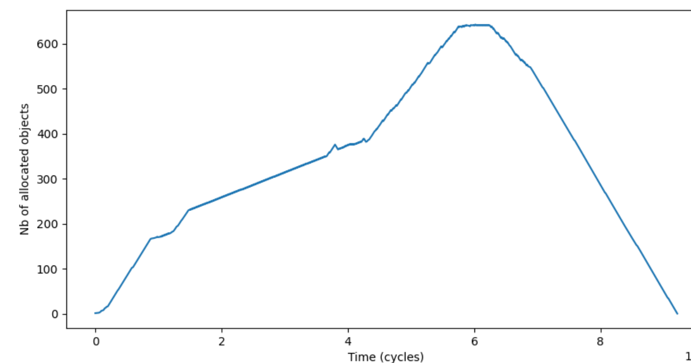


Figure 3: Number of allocated objects over time for 523.xalancmbk_r

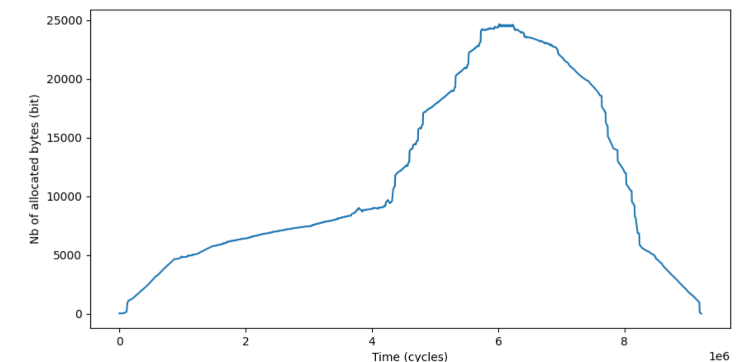
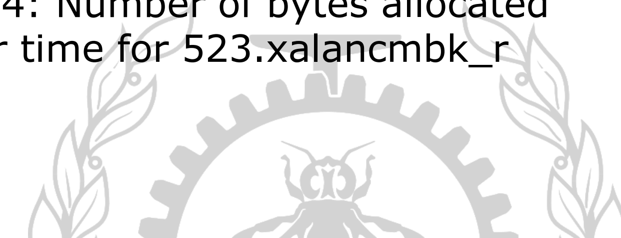


Figure 4: Number of bytes allocated over time for 523.xalancmbk_r



Results: Instruction Distribution

- During the first execution of an instruction:
  - We must disassemble the instruction.
  - The OLX buffer must be created with the **OLX-taint** approach.
  - The instruction must be instrumented with the **Patch-taint** approach.
- The distribution of the frequency of memory access instructions is measured.

Results: Instruction Distribution

- For all 7 test benchmarks analyzed, many instructions are run a high number of times.

Benchmark	Number of instructions run				
	1 times	2-10 times	11-127 times	128-1023 times	1024 times and more
505.mcf_r	129	30	11	43	317
523.xalancbmk_r	6209	3424	3183	1631	395
531.deepsjeng_r	128	58	0	0	2
541.leela_r	184	133	498	166	261
519.lbm_r	109	13	4	37	234
526.blender_r	743	836	451	249	290
544.nab_r	193	151	165	66	248

Figure 5: Instruction distribution of the frequency of memory access instructions

Results: Time overhead

- **Ptrace-mprotect**: tests done with 7 benchmarks from the SPEC CPU suite.
- **Ptrace-taint, OLX-taint, Patch-taint**: tests done with a custom micro-benchmark.

Approach Used	Time (s)	Overhead per Instruction
No protection	0.017	-
Ptrace-mprotect	173.120	8.6 µs
Ptrace-taint	293.619	14.7 µs
OLX-taint	22.649	1.13 µs
Patch-taint	1.320	65 ns

TABLE 4.4 Micro-benchmark results

Figure 6: Time overhead per memory instruction

```
1 crc = 0;
2 for (i = 0; i < loop_size; i++) {
3     for (j = 0; j < size; j++)
4     {
5       line = table[j];
6       crc += 1;
7       crc += line[0];
8     }
9 }
```

Figure 7: Micro-benchmark code causing a memory error

Discussion

- Many aspects of our approach may be applied to other architectures.
- When selecting a subset of allocations to protect based on object size, the **patch-taint** approach is most interesting in terms of overhead.
- Using another factor to select a subset of allocations could make the **OLX-taint** approach more interesting.

Conclusion

- This work explores different techniques in order to reduce the overall overhead of runtime memory analysis.

- The **Patch-taint** approach is the most promising in terms of overhead, which is lowered significantly by selecting a small subset of allocations to protect, using object size for example.

References

[1] “Capstone : The Ultimate Disassembler,” 2023. [Online]. Available : http://www.capstone-engine.org/

[2] O. Dion, “LibOlx,” 2023. [Online]. Available : https://git.sr.ht/~old/libolx

[3] O. Dion, “LibPatch,” 2023. [Online]. Available : https://git.sr.ht/~old/libpatch

[4] Standard Performance Evaluation Corporation, “SPEC CPU 2017,” 2017. [Online]. Available : https://www.spec.org/cpu2017