Memory Sanitization for Native Applications at the Binary Level

Adel Belkhiri

February 2, 2026

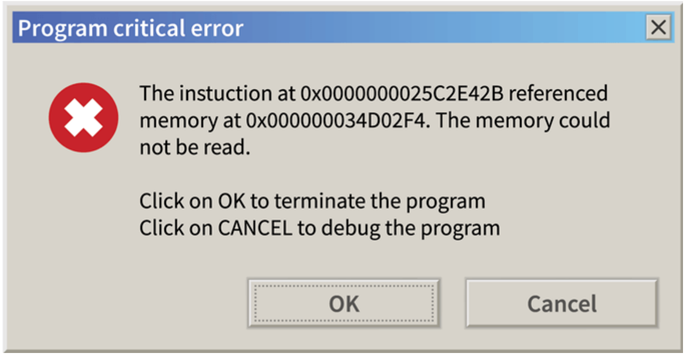
Polytechnique Montréal

Agenda

1. Introduction
2. MallocSan Design
    - Post-Handler Deferring Optimization
    - VSIB Addressing Support
    - Multithreading Support
3. Overhead Evaluation
4. Demo
5. Future Work

Memory Access Safety

- C and C++ expose raw memory to programmers, trading safety for performance and control
- The explicit control over memory provides flexibility, at the cost of :
  - Out-of-band
  - Use after free
  - Use before initialization
  - Memory Leak
  - Etc.



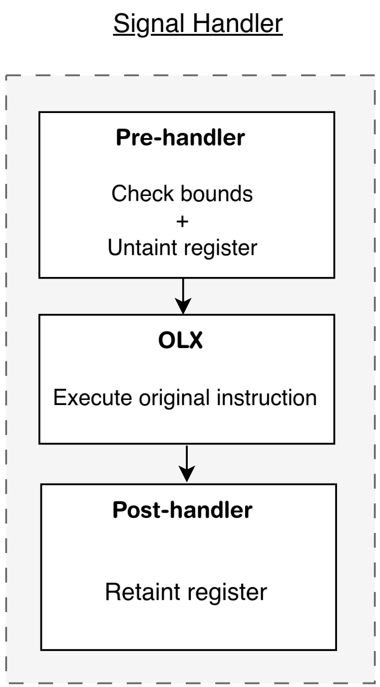
Motivation

- Stack safety has improved through mature compiler-based defenses (e.g., stack canaries, ASLR, CFI)
- Heap buffer overflows and use-after-free consistently rank among the primary root causes of actively exploited vulnerabilities*
- Heap safety solutions are impractical in scenarios where recompilation or full access to the source is impossible
    - Proprietary or closed-source software
    - Legacy binaries with unavailable build environments
    - Third-party libraries and SDKs

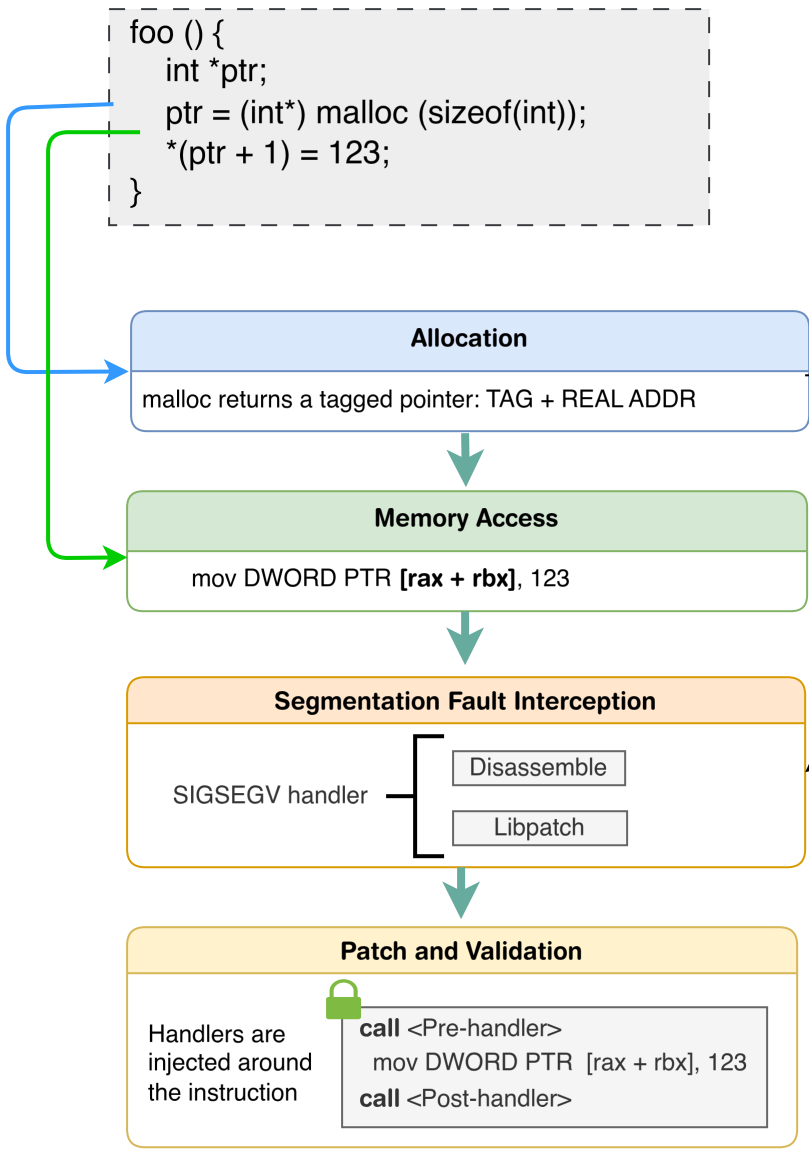
** According to 2023 Known Exploited Vulnerabilities Report:*
Link: https://cwe.mitre.org/top25/archive/2023/2023_kev_insights.html

MallocSan Design (1)

- Heap pointers are tainted at allocation using wrapped malloc APIs
- Dereferencing a tainted pointer triggers a controlled SIGSEGV fault
- Libpatch is invoked from MallocSan's SIGSEGV handler to patch the faulting instruction
- Handlers enforce access safety
  - **Pre-handler:** validate the access and temporarily untaint the pointer
  - **Post-handler:** re-taint registers to restore protection



MallocSan Design (2)



Object Table

index	base address	size	alloc site
0x1	0x00005b49425332a0	4	foo()

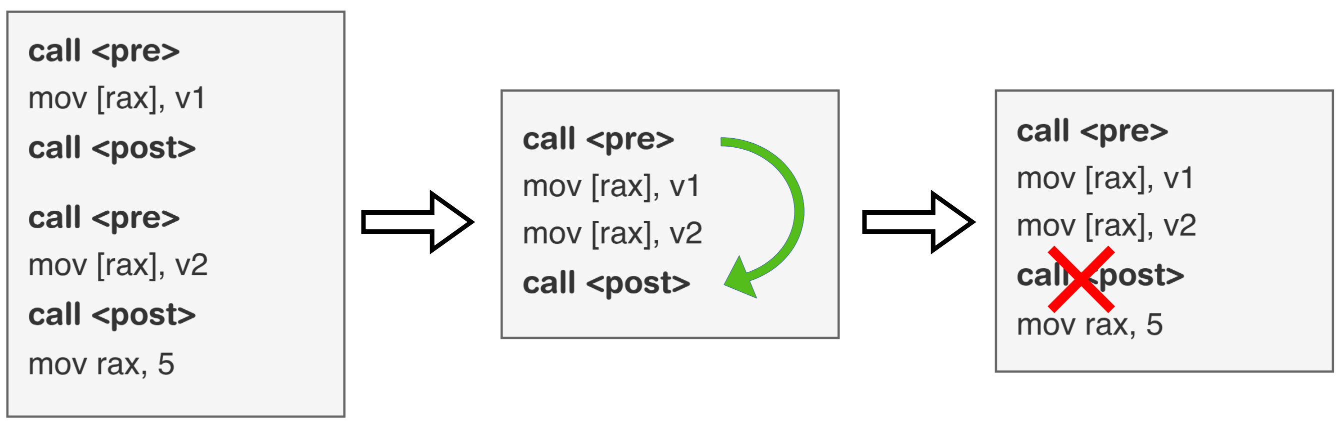
Instruction Table

fault @	arg_m[]	arg_r[]	disass
0x55555650	1	0	mov DWORD PTR [rax + rbx], 123

base	index	disp	scale
rax	rbx	0	0

Post-Handler Deferring Optimization (1)

- Executing a post-handler after every tainted memory access is **expensive** and often unnecessary, as registers tend to remain unchanged over many instructions
- If the tainted register is overwritten in the next instructions, we can skip installing a post-handler



Post-Handler Deferring Optimization (2)

Algorithm 1 Defer Post-Handler

```
INIT
    safeAddr ← fault location
    similarAccessCount ← 0
BEGIN
for each next instruction do
    if control-flow reached or (taint read / modified) then
        return safeAddr
    end if
    if taint-independent then
        if non-memory (and patchable) then
            safeAddr ← instruction address
        end if
        continue
    end if
    if fault-like memory access then
        similarAccessCount ← similarAccessCount + 1
        continue
    end if
    if taint overwritten then
        postHandler ← false
        return safeAddr
    end if
end for
END
```

VSIB (Vector Scaled Indexed Base)

- A single instruction may perform multiple memory accesses (e.g., vgather*, vscatter*)

$$\mathbf{addr}_i = \mathbf{base} + \mathbf{index}_i \times \mathbf{scale} + \mathbf{disp}$$

- Vector instructions can be compiler-generated (-O3, -march=native,...) or explicitly written using SIMD intrinsics
- SIMD intrinsics and VSIB semantics depend on the target ISA (e.g., AVX, AVX-512)
- MallocSan generalizes memory safety from scalar to vector memory access instructions

VSIB (Vector Scaled Indexed Base)

Example

```
vgatherdps ymm0, [rdi + ymm1*4], ymm2
```

Where:

- **rdi** → base address
- **ymm1** → vector of 32-bit indices
- **4** → scale factor
- **ymm2** → mask (per-lane enable)

Multithreading

- Instruction table refactored as a **lock-free hash table** containing only immutable metadata (index/base registers, displacement, scale, …)
- Mutable runtime state such as register values and metadata updates moved to thread-local storage
- Any thread may decode and patch a faulting instruction using a per-thread Capstone context
- When multiple threads fault on the same instruction, one thread initializes the instruction entry while the others wait for it to become ready
- Object table redesigned as a lock-free Treiber stack

Runtime Overhead

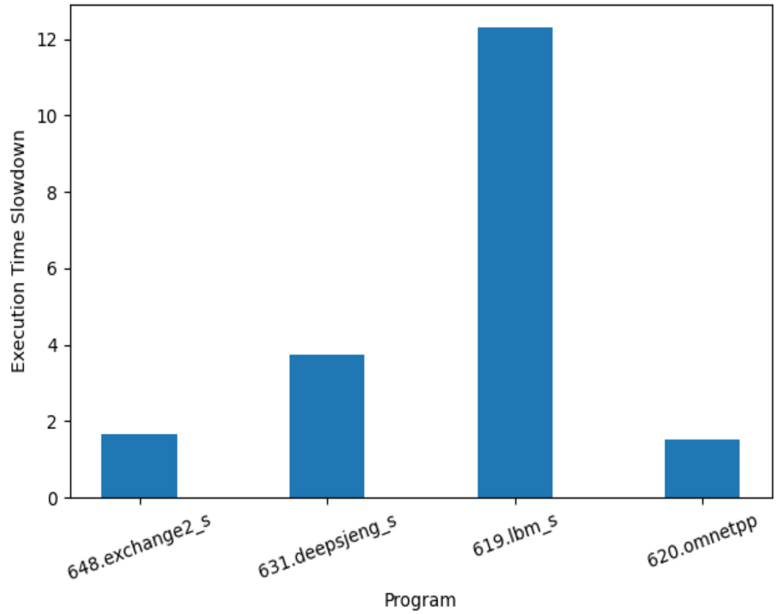


Fig. 1: Execution time slowdown induced by MallocSan across SPEC CPU 2017 applications

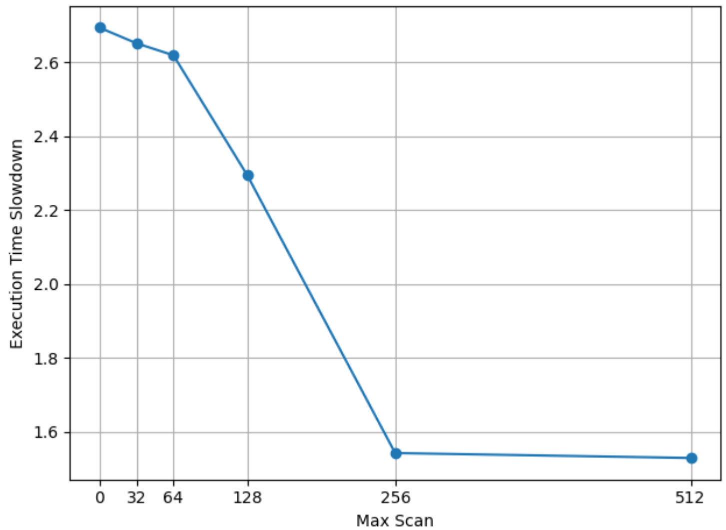


Fig. 2: Execution time slowdown of exchange2 as a function of the *Max Scan* param in the post-handler deferring algorithm

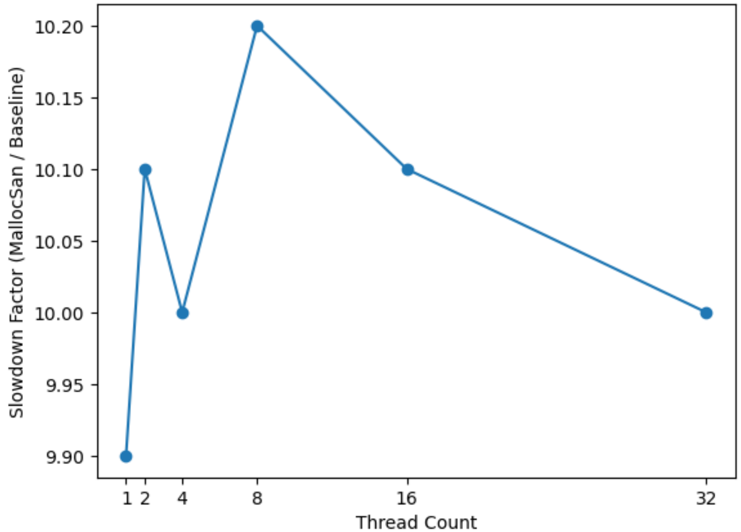


Fig. 3: Memcached execution time slowdown as the number of worker threads increases.

Evaluation Results

- Instruction patching with the TRAP strategy significantly impacts MallocSan's performance
- Libpatch's Alias and Punning patching algorithms may overwrite consecutive memory-access instructions
  - Supporting overlapping patches in libpatch will improve significantly MallocSan's performance
- Libpatch conservatively saves the full execution context when invoking handlers, including registers that are not always required
  - **Example:** the unconditional saving of XMM registers contributes to unnecessary overhead in MallocSan

Demo

Future Work

- MallocSan is a binary-level memory sanitizer that does not require source code access or recompilation
- While MallocSan's current performance is acceptable, several optimizations remain to be explored:
    - **Example:** Cache instruction disassembly results to avoid repeatedly disassembling the same instructions
- Re-evaluate MallocSan's overhead on representative multi-threaded applications

Questions?

https://github.com/adel-belkhiri/MallocSan