Memory Sanitization for Native Applications at the Binary Level

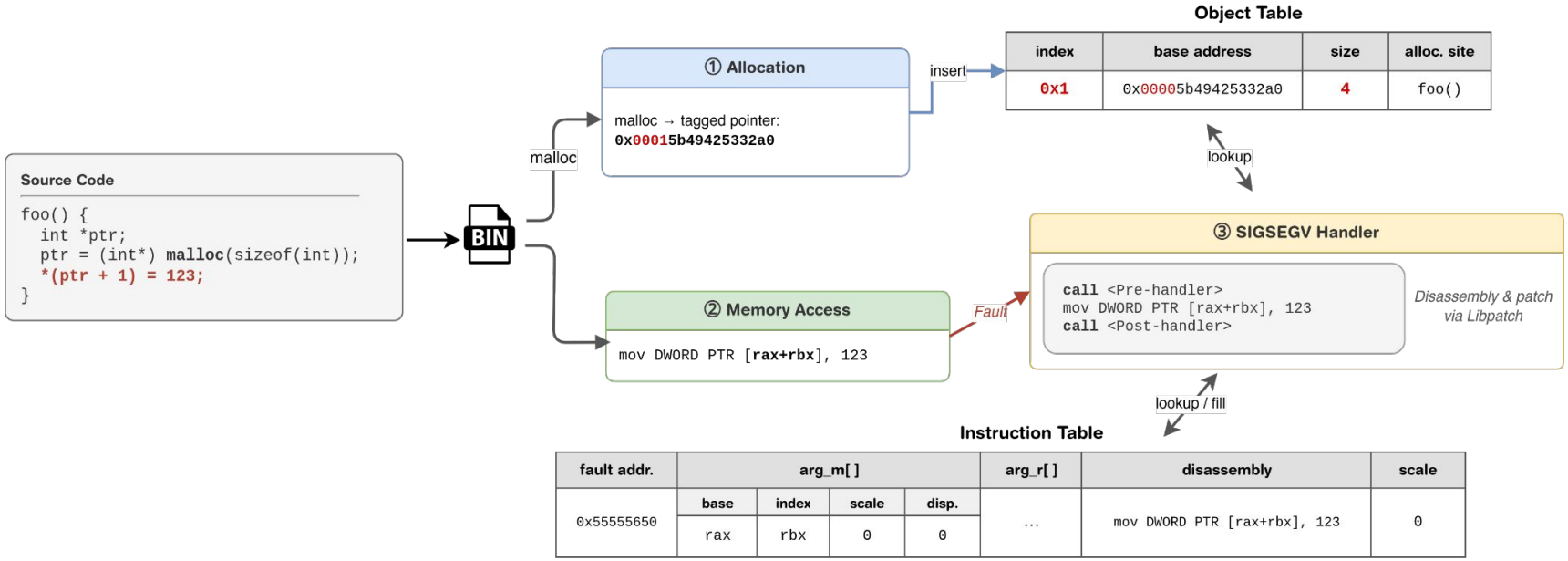
Adel Belkhiri

Progress Report Meeting

Montreal, July 10, 2026

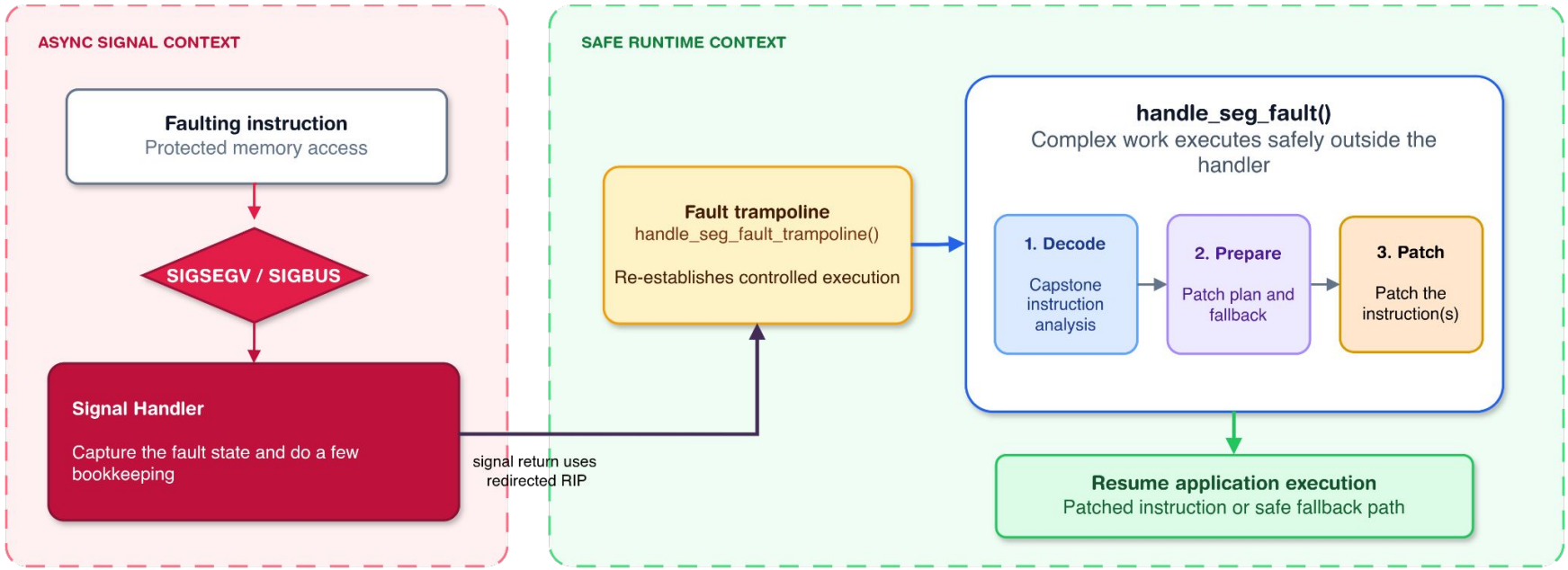
MallocSan: A Short Recap

- A memory sanitizer for native x86-64 applications. Its key concepts are:
  1) Intercepts calls to the **malloc-family** functions
  2) Returns heap pointers with **embedded tags** (OID)
  3) Handles SIGSEGV faults when invalid **tagged pointers are dereferenced**
  4) **Rewrites instructions** using libpatch to support tagged-pointer checking



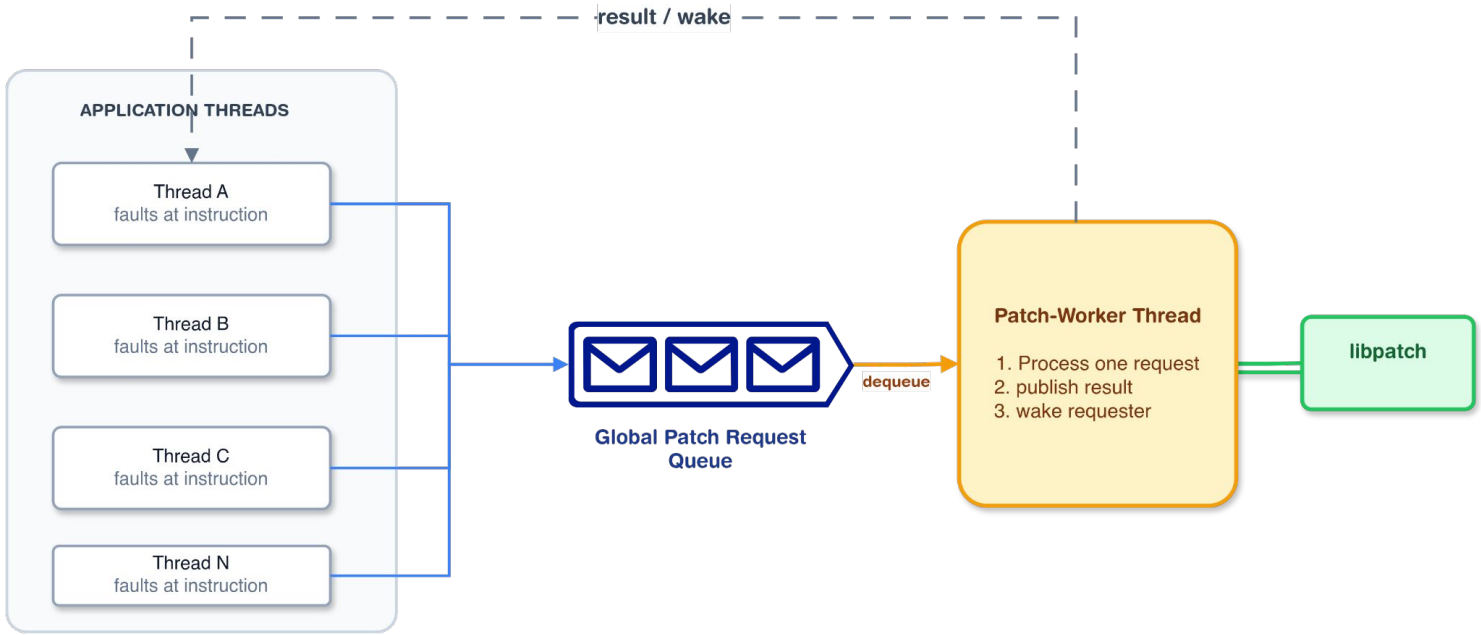
Capstone Is Not Signal Safe

- Expensive work such as instruction decoding and patch preparation is moved outside the asynchronous signal context
- The SIG fault handler now captures the fault state and redirects execution to a trampoline and then to a function that decodes, classify, and patch safely.



Libpatch Is Not Thread Safe

- Single-threaded program uses libpatch to patch instructions synchronously
- A dedicated worker serializes patching once the application becomes multithreaded
- Application threads enqueue patch requests and only the worker calls libpatch, preventing races in shared states



“MOV” Instruction Emulation

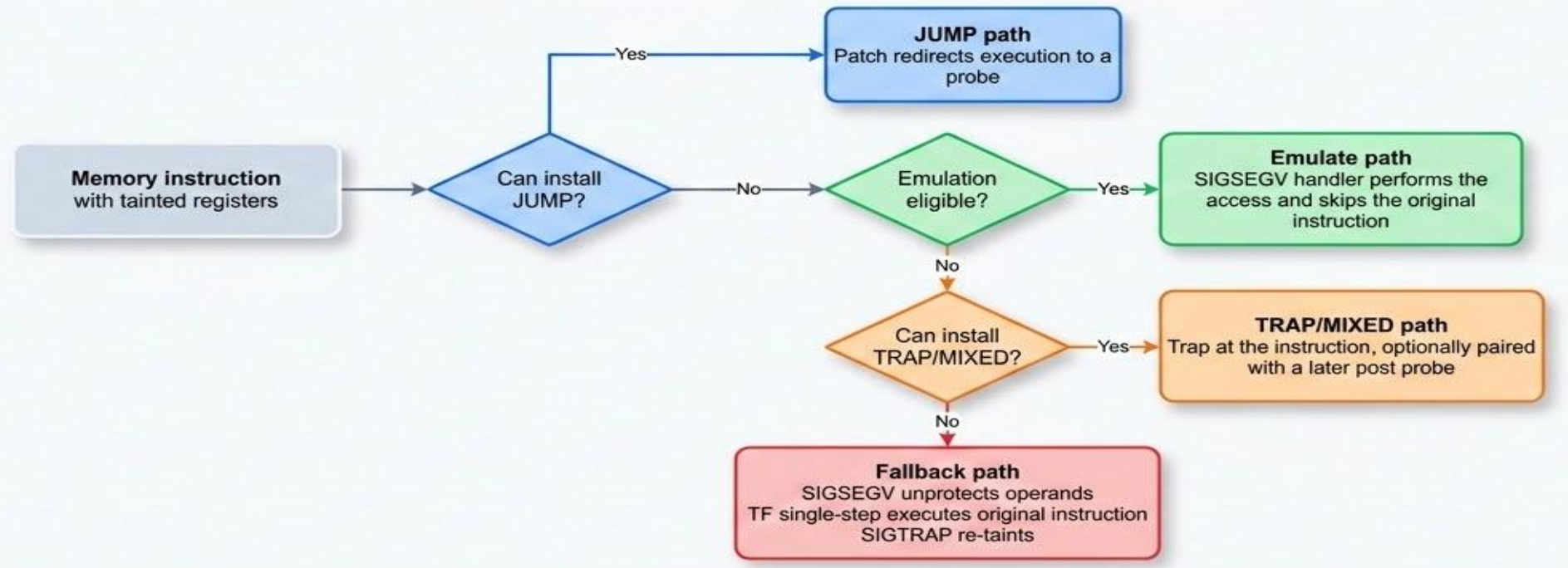
- Hot instructions patched with TRAP kills MallocSan performance; handling the memory access directly in the SIGSEGV path is cheaper
- Many memory accesses are simple to emulate, such as fixed-field mov instructions:

mov rax, [rdi + 0x10]

- In the SIGSEGV handler, MallocSan computes the untainted address, checks bounds, performs the load/store, and advances RIP without modifying the tainted registers.
- Emulation preserves the same protection model for eligible mov sites, avoids TRAP dispatch, re-taint restoration, and is **1.6x faster**
- Complex instructions still fall back to the existing TRAP/single-step mechanism, preserving correctness.

Single-Step/Trap Fallback Path

- For unpatchable instructions, MallocSan temporarily untaints the registers in the SIGSEGV handler, and sets the Trap Flag so the faulting instruction executes exactly once
- The following SIGTRAP restores pointer taint, clears the flag, and resumes execution



Post-handler Deferral Optimization

Algorithm 1 Defer Post-Handler

```
1:  INIT
2:      safeAddr ← fault location
3:      similarAccessCount ← 0
4:  BEGIN
5:  for each next instruction do
6:      if control-flow reached or (taint read / modified) then
7:          return safeAddr
8:      end if
9:      if taint-independent then
10:         if non-memory (and patchable) then
11:             safeAddr ← instruction address
12:         end if
13:         continue
14:     end if
15:     if fault-like memory access then
16:         similarAccessCount ← similarAccessCount + 1
17:         continue
18:     end if
19:     if taint overwritten then
20:         postHandler ← false
21:         return safeAddr
22:     end if
23: end for
24: END
```

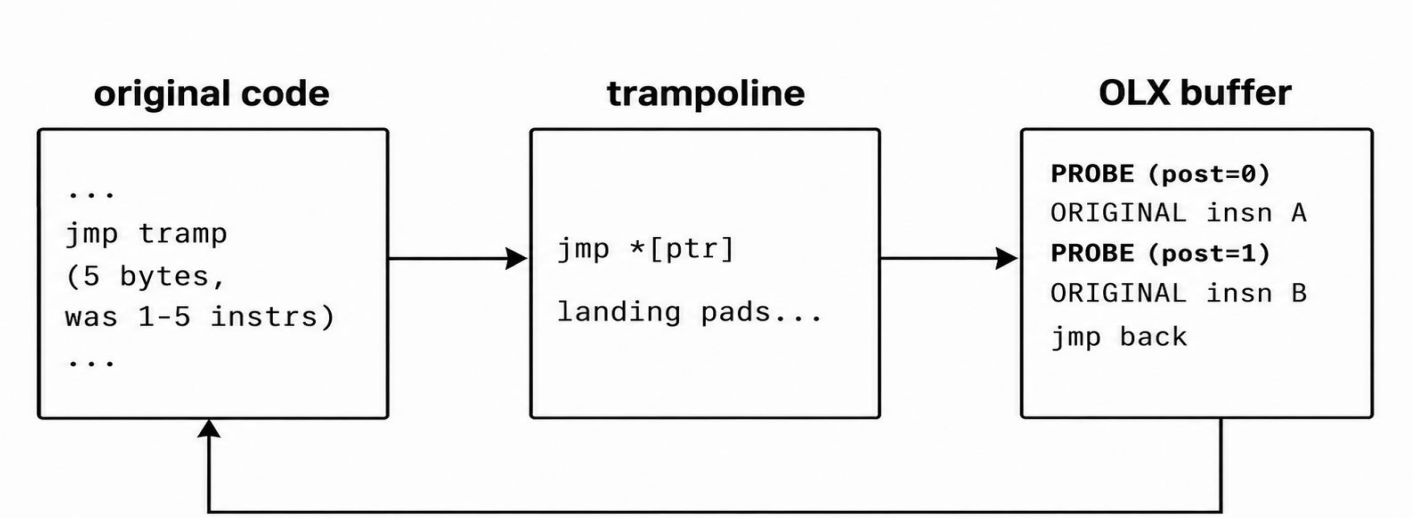
```
mov   eax, [rdi + 50]      ; FAULT
mov rbx, 1
add   rdi, 4

.loop:
   add   eax, [rdi]
   dec   rcx
   jne   .loop

mov   [rip + 123], rax
```

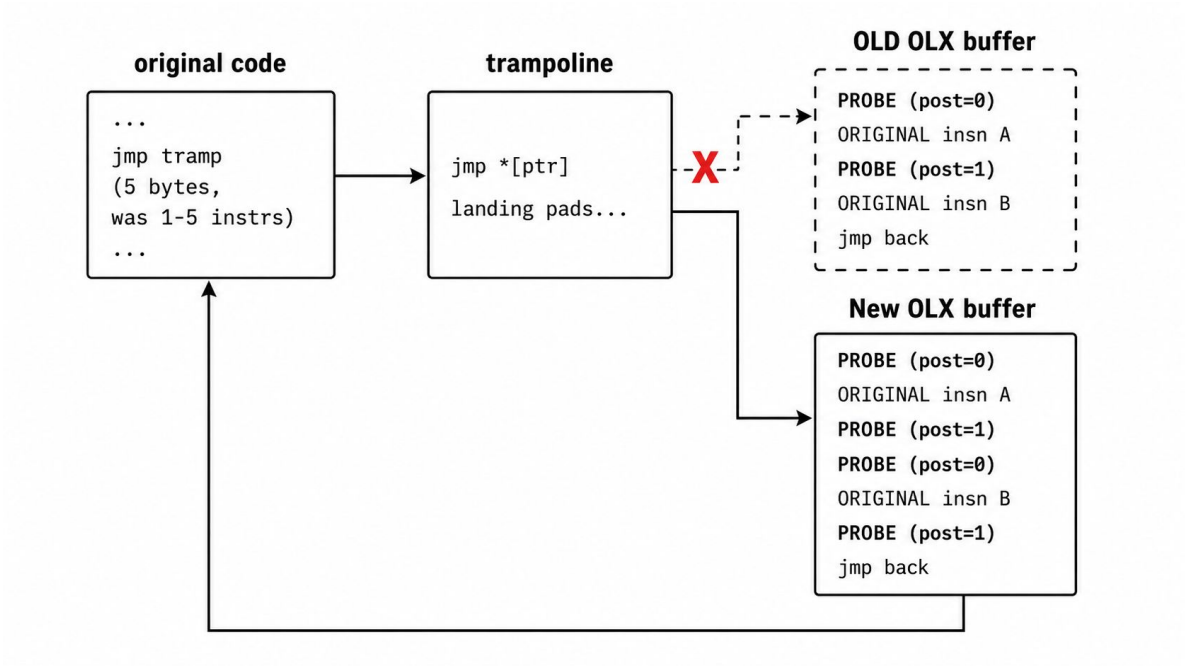
Overlapping Patches (1)

- When a relocated instruction faults, execution is forced onto the fallback path, because libpatch cannot patch relocated instructions directly
- The trampoline transfers control to the OLX, where the probe and relocated instructions will be executed before jumping back to the program's continuation point
- The trampoline reaches the OLX through an indirect pointer, which becomes the key mechanism for updating the patch later
- A probe placed on a relocated instruction is treated as a child of the parent patch. No new trampoline is created



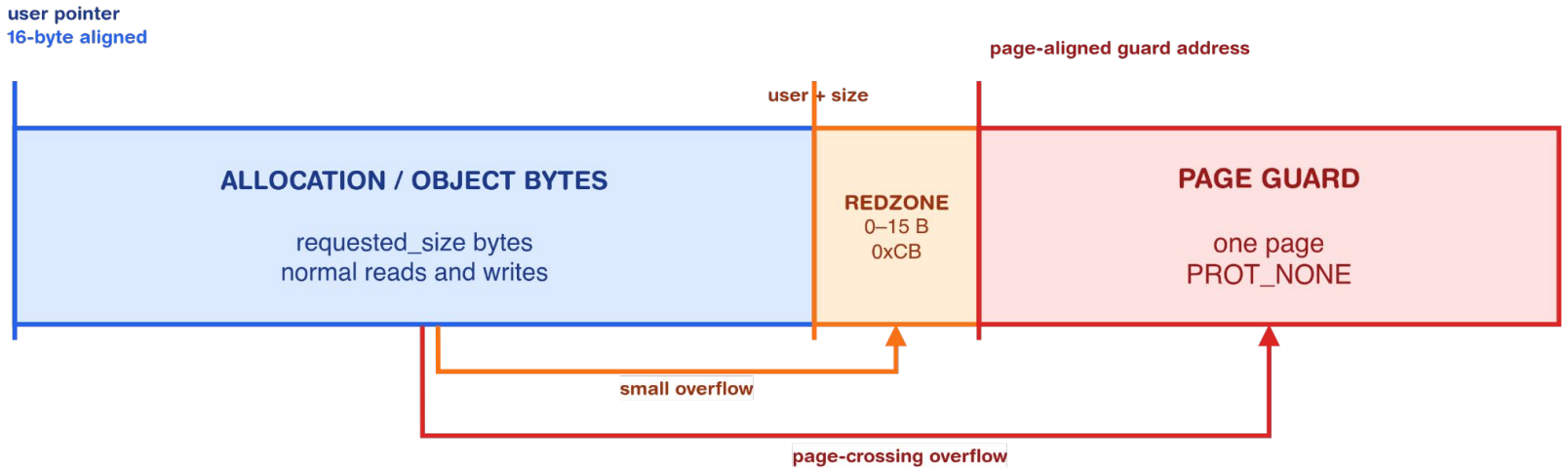
Overlapping Patches (2)

- Libpatch rebuilds the parent's OLX into a fresh replacement buffer, and emits the child pre- and post- handlers around the relocated target instruction
- The rebuilt OLX is atomically published through the parent trampoline, and the old OLX buffer is retired lazily
- During the handoff a thread observes either the old or the new buffer — both are mapped and valid. No stop-the-world



Guard Page Backend (1)

- Eligible large allocations are placed immediately before a PROT_NONE guard page
- Page-crossing overflows enter the guard page and raise SIGSEGV in hardware
- Small trailing overflows may stop in the alignment gap, which is filled with patterned canary bytes
- At free(), the canary bytes are checked for corruption indicate a trailing overflow



Guard Page Backend (2)

- **DW_GUARD_MIN_SIZE**: minimum guarded allocation size
- **DW_GUARD_MAX_OBJECTS**: maximum live guarded objects

- **Advantage/Limitation:**

  + Guarded objects use canonical, untainted pointers which removes the overhead associated with the per-access instrumentation

  - Current implementation protects only the end of the allocation, so object underflows are not detected

  - Small trailing reads in the alignment gap may be missed; small trailing writes are detected only during redzone checks at free()/realloc()

Runtime Overhead: MallocSan vs. Valgrind

Fig. 1: Execution-time overhead

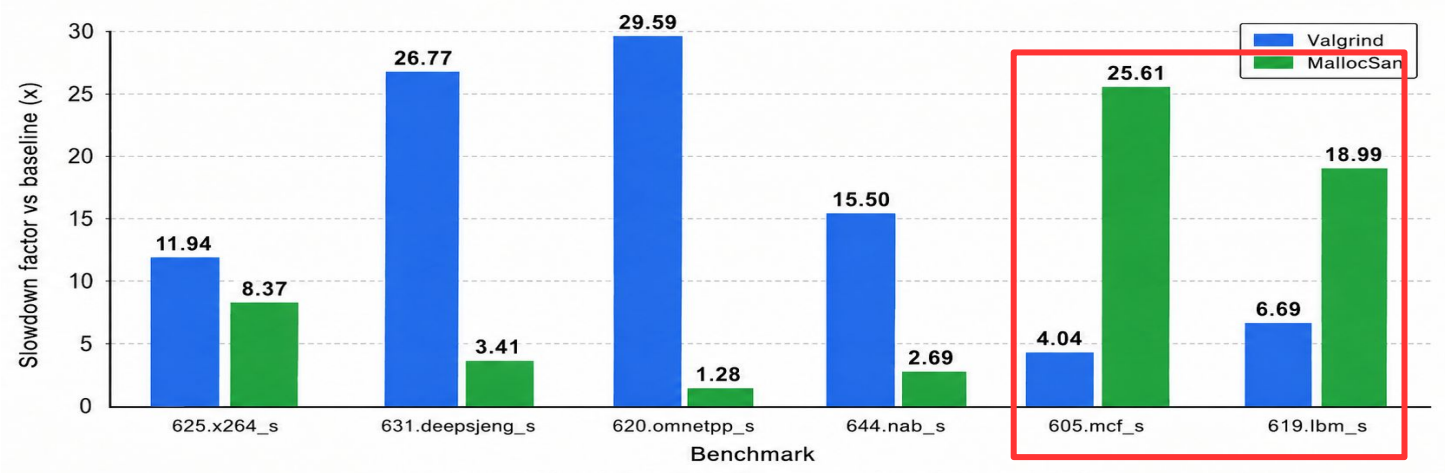
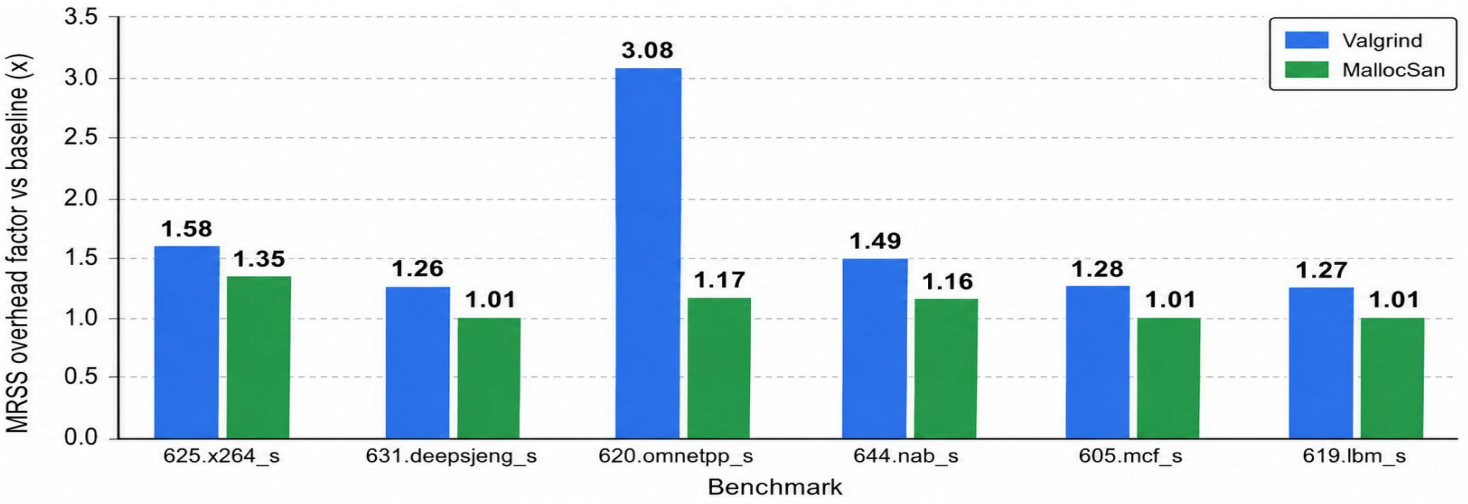


Fig. 2: Memory overhead



Runtime Overhead: MallocSan vs. Valgrind

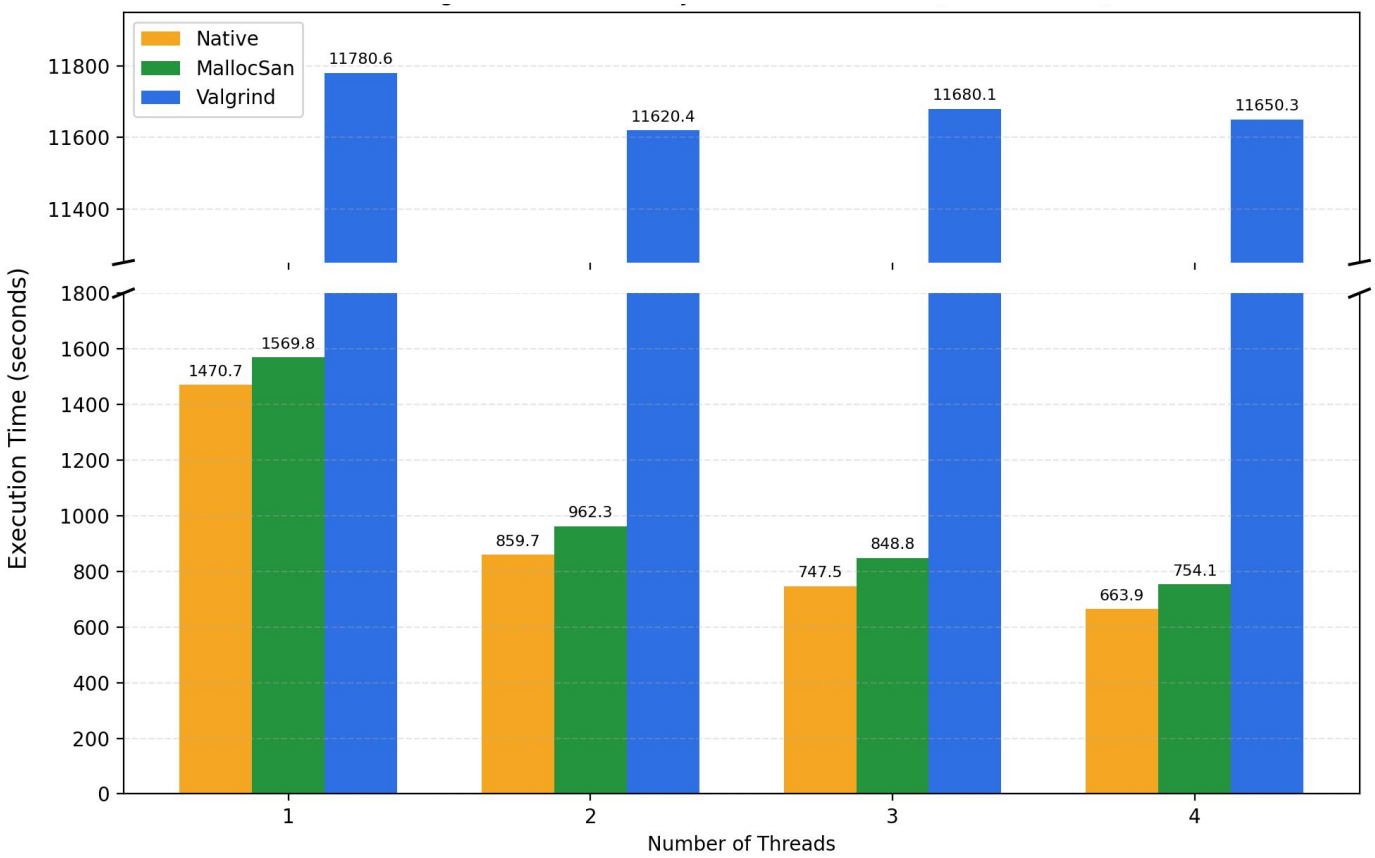


Figure: Execution time comparison of Native, MallocSan, and Valgrind by number of threads using a broken y-axis

Conclusion

- MallocSan outperforms Valgrind on most SPEC benchmarks and provides better support for extended instruction sets (e.g., AVX-512) and multithreaded applications
- However, MallocSan's performance can be significantly affected by:
  - TRAP-patching of hot instructions, which is unavoidable in some binaries. E9Patch can sometimes help mitigate this issue
  - A low success rate for post-handler deferral when hot functions are dominated by control-flow instructions, such as calls and jumps
- **Possible optimization directions:**
  - Coalescing probes to reduce the number of jumps inserted into the runtime binary
  - Reducing the cost of per-access checks, for example by emulating jump-based instructions or firing probes conditionally

Questions?

adel.belkhiri@polymtl.ca