Runtime Memory Access Validation Using Dynamic Instrumentation

Adel Belkhiri

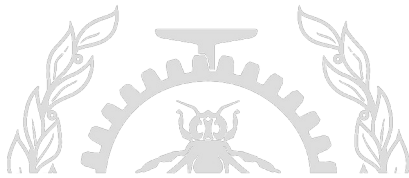
May 12, 2025

Polytechnique Montréal

DORSAL Laboratory

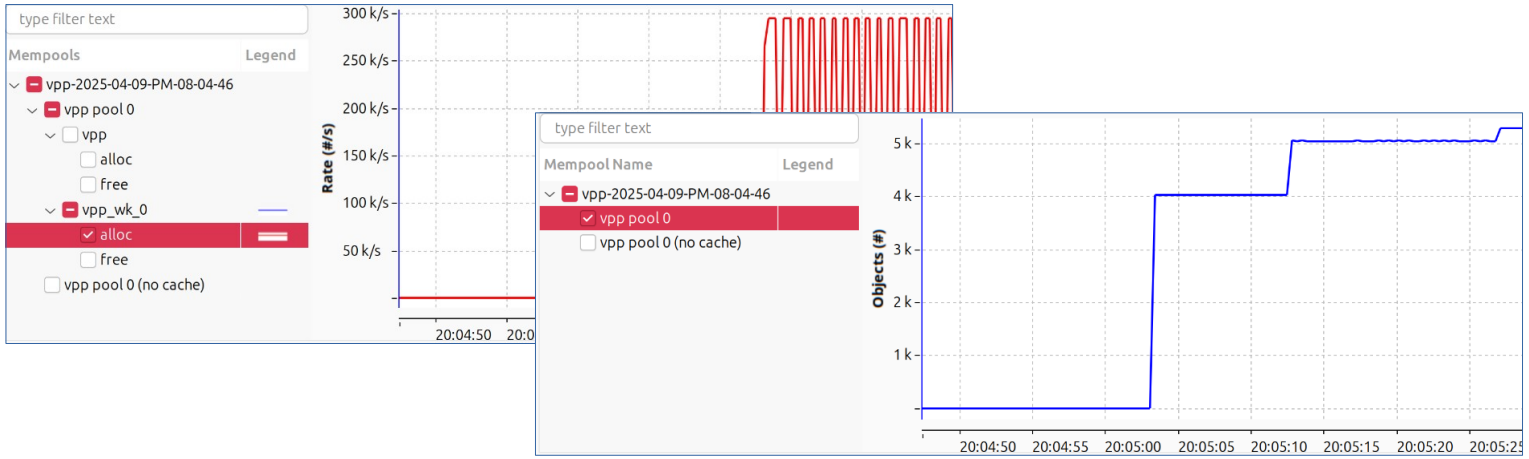
Agenda

1. Previous Work
2. Memory access validation techniques
   - Redzones, shadow memory, and pointer tainting
3. Beyond Datawatch
4. Future Work



Previous Work

- Integrated DPDK mempool analyses into Trace Compass Incubator

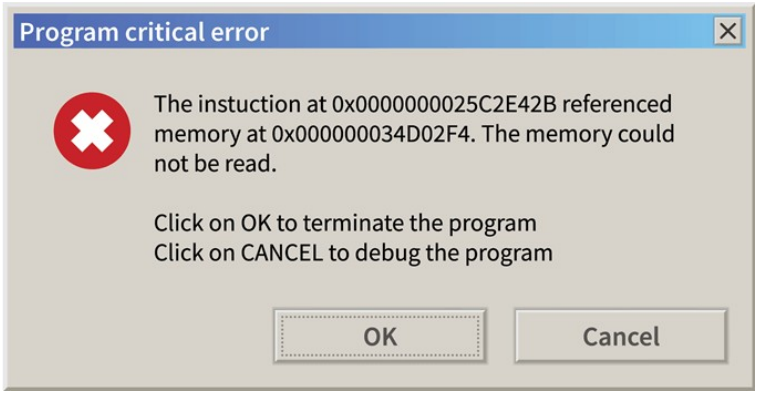


- Submitted an article to Runtime Verification conference.

A. Belkhiri, A. Fiorini, M. Khouzam and H. Lee, A Transparent and Efficient Performance Analysis Approach to Enhance DPDK Observability, Runtime Verification, Austria, 2025.

Memory Access Validation

- Low-level languages like C/C++ lack automatic memory management, leading to many issues related to memory accesses
  - Out-of-band
  - Use after free
  - Use before initialization
  - Memory Leak
  - Etc.



Red zones and shadow memory

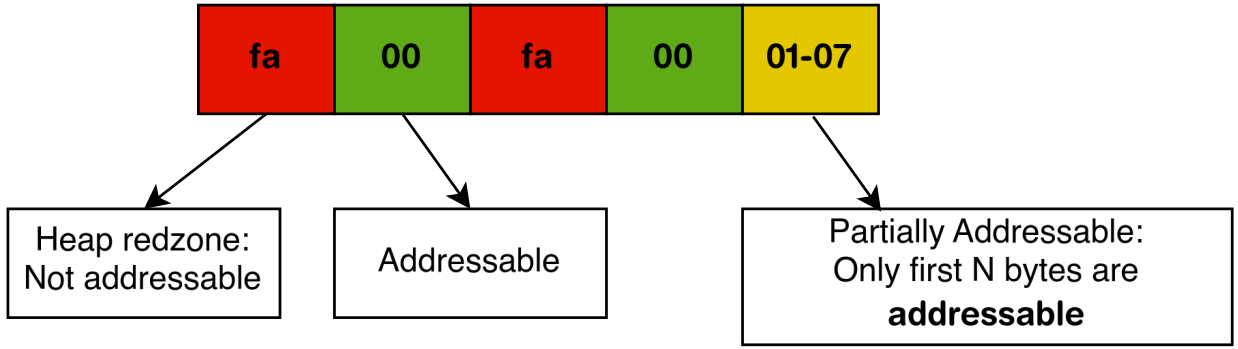
Red zones:

Special memory areas placed around allocated memory blocks to detect overflows

Each 8-byte region of application memory is mapped to a single shadow byte that encodes its accessibility status

Shadow Memory:

A parallel memory structure that stores metadata about application data



Pointer tainting (1)

- Embeds metadata (e.g., an object-ID) in the top 2 bytes of a 64-bit pointer (x86-64) to enable memory access validation at runtime

- **How it works?**

  1) Heap allocator returns a tainted pointer.

  2) When the pointer is dereferenced the CPU flags the non-canonical address, triggering SIGSEGV/SIGBUS.

  3) The signal handler:

     a) Removes the taint,

     b) Validates the access against shadow metadata,

     c) Executes the access instruction with the untainted pointer

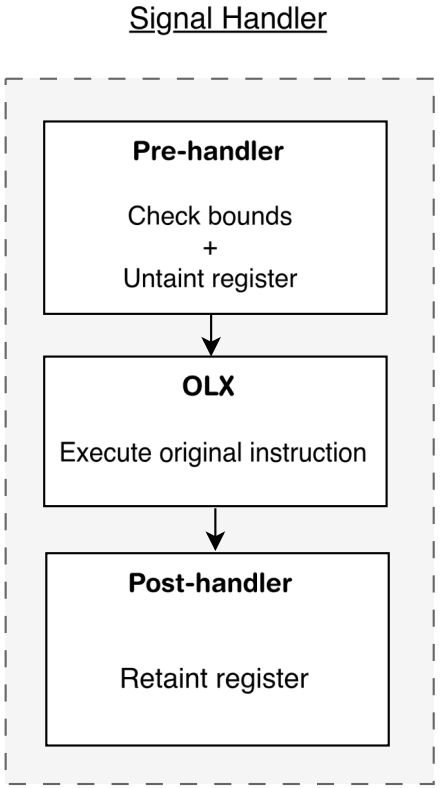
     d) Re-applies the taint, and resumes execution.

Pointer tainting (2) — Tools

- Valgrind Memcheck = 20x to 50x slowdown
- Addess Sanitizer (ASan) = 2x to 3x slowdown
- **Address Monitor (AMon)**
  - Inserts taint/untaint code at compile time via custom LLVM passes.
  - Advantages / Limitations:
    - \+ Low runtime overhead.
    - \+ Efficient for source-available code.
    - – Requires recompilation and full access to the source.

Pointer tainting (3) — Tools

- **Beyond DataWatch**
  - Jason Puncher developed Datawatch at Ciena to detect invalid memory accesses (for 32-bit applications - PowerPC)
  - Olivier Dion developed *Libpatch* to dynamically instrument applications with pre and post handlers around target instructions
  - David Piché prototyped tainted pointers for user-level x86-64 applications with MallocSan, using different methods, including ptrace and Libpatch



Pointer tainting (4) — Tools

- **MallocSan**
  - Advantages / Limitations:
    - \+ No recompilation required
    - \+ Supports on-the-fly, selective protection (e.g., per object or allocation site)
    - – Slightly higher cost due to first-hit patching and thread synchronization
    - – Wrapping all glibc syscalls is complicated, which may introduce blind spots

Possible Optimization — MallocSan

- Libpatch cannot presently instrument and place pre and post handlers on every possible instruction:
  - For short instructions (less than 5 bytes) the post handler is not called directly after the target instruction
  - It is not possible to instrument two consecutive short instructions
  - For branching instructions, the post handler cannot be called

→ Consequently, MallocSan must often fall back on using the much slower trap-based instrumentation

Examples of short instructions

```
mov al, [rax]     ;  Load (2 bytes)
mov [rax], al     ;  Store (2 bytes)
```

Possible Optimization — MallocSan

- Remove redundant untaint/check/retaint:
  - For two consecutive accesses to the same memory region (with same size), untaint/check/retaint only once
  - If the untainted register is not reused (quickly overwritten), avoid retainting

- **Examples**:

```
int *ptr = malloc(4)
*ptr = 42;    // write to the address pointed by the pointer
int val = *ptr;  // read from the same address and same size
```

```
int *ptr = malloc(4)
For (int i = 0; i < 19; i++) {
      *ptr = i;  // writing to the same address repeatedly
}
```

Future Work

- Dynamic instrumentation enables memory access validation without recompilation, while allowing selective and targeted checks

- **Next Steps:**
  - Extend Libpatch to support finer-grained jump-based patching
  - Enhance MallocSan: improve coverage & reduce runtime overhead

Questions?

adel.belkhiri@polymtl.ca