Low-overhead Spatial Memory Safety Verification

Farzam Dorostkar with Prof. Michel Dagenais

May 30th 2024

Polytechnique Montreal

DORSAL Laboratory

Since Last Meeting

ThreadMonitor (TMon)

Post-mortem data race detector for C/C++ programs that use Pthreads

- Compile-time instrumentation updated to LLVM 17
- Trace decoder updated to Perf 6.8
- Vince Bridgers and Ankush Tyagi joined us from Ericsson!
- github.com/farzamdorostkar/tmon

Since Last Meeting

AddressMonitor (AMon) - New Project

Detects heap spatial access violations in C programs on X86-64

- Dynamic analysis based on pointer tainting
- Two variants: on-the-fly and post-mortem
  - AMon-OTF: runtime analysis
  - AMon-PM: traces a program execution using Intel ptwrite
    - Uses Intel's ptwrite packets
    - User-generated 64-bit payload
    - Uses the trace data to emulate the same runtime verification performed by AMon-OTF
- Minimal data and instruction memory overhead, low runtime overhead

Problem Definition:
Lack of Spatial Safety in C

Problem Definition: Lack of Spatial Safety in C

- C provides developers direct control over various memory operations
- Ability to directly access and manipulate memory addresses
  - Advantageous in scenarios demanding high performance
  - Absence of built-in mechanisms to verify the safety of memory accesses
  - Source of bugs
- Spatial Safety Violation
  - Write to or read from memory locations outside the intended boundary of an object

Common Approaches

Common Approaches

Shadow-based Approaches:

- Allocate shadow memory to track the status of application memory
- Unchanged memory layout
- Incomplete in detecting all spatial violations
- High memory overhead

Pointer-based Approaches:

- Encode bounds information within each pointer
- Fat vs low-fat pointers
- Enforce complete spatial safety
- No data memory overhead

Methodology: Pointer Tainting

Methodology: Pointer Tainting

Approach

When Allocating Heap Objects

- Assign a unique taint to each allocated object
- Build and maintain an object table [taint, base address, size]
- Embed the taint into the 2 MS bytes of the returned address
- On the Intel 64 bits architecture the first 2 bytes are unused

When Accessing Memory

- Retrieve the taint
- Use the taint to look up the object's bounds in the object table
- Ensure the accessed memory falls within the object's bounds
- Raise an alert if a violation is detected

Methodology: Pointer Tainting

Challenge

Embedded taint changes the address layout

- Although unused, 2 MS address bytes are not ignored
- Changes the effective address
- Dereferencing a tainted pointer causes segmentation fault

Solution

To verify and dereference a tainted pointer

1. Use the taint to verify the spatial safety of the memory access
2. Untaint the pointer
3. Dereference the untainted pointer
4. Re-taint the pointer

AddressMonitor (AMon): Implementation

AddressMonitor (AMon): Implementation

Two variants

1. On-the-fly (AMon-OTF)
2. Post-mortem (AMon-PM)

Each variant consists of two main modules

1. Runtime library (libamon.so)
2. Compile-time transformation

AddressMonitor (AMon): Implementation

Runtime Library (libamon.so)

- Intercepts standard heap allocation functions
  - To add taint to returned pointers
- Intercepts other standard C functions as well
  - To untaint possibly tainted arguments
- Maintains the object table
- Defines the bounds checking logic
- Defines environment variables to control the behavior of AMon
  - On-the-fly vs post-mortem analysis modes, supported object sizes, etc.
- It is preloaded

AddressMonitor (AMon): Implementation

Compile-time Transformation

At LLVM IR level

- Function pass
- Traverses each function to identify the memory access instructions (Loads and Stores).
- For each load/store instruction:
  - Creates a new equivalent instruction where the dereferenced pointer is untainted
  - Replaces the old instruction with the new one
  - Re-taints the dereferenced pointer

This part is common between the two variants of AMon.

AddressMonitor (AMon): Implementation

Compile-time Transformation: Variant-specific

AMon-OTF

- Inserts the bounds checking logic immediately before each access

AMon-PM

- Instruments each access with a single ptwrite instruction
- Uses a ptwrite packet to record the required runtime information for each access
  - Base address, taint, and the access size
- The post-mortem analyzer uses the trace data to emulate the same runtime verification performed by AMon-OTF

Preliminary Evaluation Study & Discussion

Preliminary Evaluation Study & Discussion

- Two SPEC CPU 2017 benchmarks
- Under ASan, AMon-OTF, and AMon-PM
- Compared to native compilation

Benchmark	ASan		AMon-OTF		AMon-PM	
	Time (x)	MRSS (x)	Time (x)	MRSS (x)	Time (x)	MRSS (x)
470. mcf	1.6 x	2.5 x	1.2 x	1.1 x	2.9 x	2.1 x
444.namd	1.9 x	3.5 x	1.4 x	1.2 x	3.0 x	1.5 x

Preliminary Evaluation Study & Discussion

- ASan causes high memory overhead due to its use of shadow memory and red zones
- The reported memory overhead for AMon-PM is associated with the tracer (Linux Perf)
  - Trace data collection activities conducted by the Perf tool
  - More flexible and less restrictive than the direct memory overhead caused by ASan
- The low memory overhead of AMon-OTF is mostly associated with allocating an object table
- For ASan and AMon-OTF, the reported execution time overheads are associated with on-the-fly bounds checking operations
- For AMon-PM, the reported execution time overhead is mostly associated with the tracer, with a smaller impact from the untainting and re-tainting process

Conclusion & Future Work

- AddressMonitor (AMon): detects heap spatial access violations in C programs on X86-64
- Dynamic analysis tool with two variants: on-the-fly (AMon-OTF) and post-mortem (AMon-PM)
- Pointer tainting and compile-time transformation
- Minimal data and instruction memory overhead, low runtime overhead
- AMon is already capable of detecting temporal access violations to some extent (e.g. use-after-free).

Thanks!

Questions? Comments?

farzam.dorostkar@polymtl.ca
https://github.com/farzamdorostkar
https://farzamdorostkar.github.io/