AddressMonitor (AMon)

Low-overhead Spatial and Temporal Memory Safety for C

Farzam Dorostkar

DORSAL
Polytechnique Montréal

Progress Report Meeting, May 2025

Introduction
Detection Capabilities

- AddressMonitor (AMon) detects **heap spatial and temporal violations** at runtime
  - Out-of-bounds access
  - Use-after-free
  - Double-free
  - Memory leak
- AMon is also capable of detecting additional unsafe practices
  - Reading uninitialized memory
  - Passing non-base pointers to `free` or `realloc`

Introduction
Design

- AMon is based on pointer tainting and compile-time code transformation
  - Runtime library (libamon.so)
  - Compiler pass (compiler dependent)
- AMon is designed to have minimal memory overhead
  - Specifically compared to AddressSanitizer (ASan)

Pointer Tainting

Object-specific Analysis

- When allocating a heap object
  - Assign a unique taint (ID) to each allocated object
  - Maintain an object table [taint, base address, size, temporal status]
  - Most architectures have unused address bits
    - For instance, the first 2 bytes are unused on most 64-bit architectures
  - Embed the taint into the unused bits (pointer tainting)
- When dereferencing a tainted pointer
  - Retrieve the taint
  - Use the taint to verify the access against the object table
  - Dereferencing a tainted pointer causes segmentation fault (Requires pointer untainting)

AMon in Action

Detecting a Buffer Overflow

object table

index	base address	size	temporal status	stack trace
0x0001	0x00005b49425332a0	4	ALLOCATED	#0 foo+0x19

overflow.c

```
bar (int *ptr) {
  *(ptr + 1) = 2025; /* overflow! */
}

foo () {
  int *ptr;
  ptr = (int*)malloc(sizeof(int));
  bar (ptr);
}
```

libamon.so

```
void* malloc (size_t size) {
  void *res;
  res = __libc_malloc (size); /* res = 0x00005b49425332a0 */
  amon_protect (res);         /* res = 0x00015b49425332a0 */
  objtbl_add (res, size);     /* add to the object table */
  return res;                 /* return the tainted pointer */
}
```

Pointer Tainting and Object Table Population

AMon in Action

Detecting a Buffer Overflow

overflow.c

```
bar (int *ptr) {
  + amon_write ((void*)(ptr + 1), 4);
  + int *uptr = (ptr + 1) & UNTAINT_MASK;
  + *uptr = 2025;
  - *(ptr + 1) = 2025; /* overflow! */
}

foo () {
  int *ptr;
  ptr = (int*)malloc(sizeof(int));
  bar (ptr);
}
```

object table

index	base address	size	temporal status	stack trace
0x0001	0x00005b49425332a0	4	ALLOCATED	#0 foo+0x19

libamon.so

```
void amon_write (void *tptr, size_t size) {
  uintptr_t taint = tptr ≫ 48;              /* taint = 0x0001 */
  uintptr_t base = tptr & UNTAINT_MASK;    /* base = 0x00005b49425332a4 */

  objtbl[taint].temp == ALLOCATED ? :  report_temporal_violation (); ✓
  base < objtbl[taint].base &&
  base + size < objtbl[taint].base + objtbl[taint].size
      ?  :  report_spatial_violation(); ✗
}
```

Runtime Verification and Pointer Untainting

AMon in Action

Detecting a Buffer Overflow

object table

index	base address	size	temporal status	stack trace
0x0001	0x00005b49425332a0	4	ALLOCATED	#0 foo+0x19

overflow.c

```
bar (int *ptr) {
  + amon_write ((void*)(ptr + 1), 4);
  + int *uptr = (ptr + 1) & UNTAINT_MASK;
  + *uptr = 2025;
  - *(ptr + 1) = 2025; /* overflow! */
}

foo () {
  int *ptr;
  ptr = (int*)malloc(sizeof(int));
  bar (ptr);
}
```

stderr

```
ERROR: AddressMonitor:  heap-buffer-overflow on address 0x5b49425332a4

  » Violating access:  write of size 4 at 0x5b49425332a4
      [The call stack at the point of violation]
      #0 overflow_amon(bar+0x35)
      #1 overflow_amon(foo+0x27)
      #2 ...

  » Intended object bounds:  4-byte region [0x5b49425332a0,0x5b49425332a4)
      [The stack trace at the point of heap allocation]
      #0 overflow_amon(foo+0x19)
      #1 ...
```

Error report generated by AMon upon detecting an out-of-bounds access

AMon LLVM Compiler Pass

Objectives

- Code analysis
  - Detect heap pointers
- Code transformation
  - Instrument code with untainting logic at compile-time
  - Insert runtime checks

AMon LLVM Compiler Pass: Updates

Pointer Flow Analysis in LLVM IR

- Starts by constructing an initial set of *original* heap pointers
  - Pointers returned from standard heap allocation functions (`malloc` family)
  - Function arguments of pointer type
  - Pointers returned from function calls
  - Module-level global variables storing pointers
- Performs data-flow analysis to identify all *derived* heap pointers - tracks the propagation of original heap pointers
  - Pointer arithmetic
  - Type casting
  - Control-flow merges
  - Indirect propagation through memory access
    - When a tainted pointer is stored in memory and later read

AMon LLVM Compiler Pass: Updates

Untainting Heap Pointers

- Identifies all IR instructions that perform memory access
  - Memory access operations (such as `load`, `store`, and `cmpxchg`)
  - Memory-related intrinsics (such as `llvm.memcpy` and `llvm.memset`)

- Untaints dereferenced heap pointer(s)
  - Extracts the pointer operand(s)
  - Checks if the pointer belongs to the set of identified heap pointers
  - If so, generates an untainted version of the pointer using `llvm.ptrmask` intrinsic
  - Replaces the pointer in the current instruction with the untainted version using the `setOperand` method

AMon LLVM Compiler Pass: Updates

Inlined Temporal and Spatial Checks

- AMon now supports **inlining** runtime checks directly into the instrumented code
  - Eliminating the need for function calls to `libamon` (`amon_read` and `amon_write`)
  - Reduced runtime overhead
  - Slightly larger executables

Industrial Test

Evaluated by Ericsson

- AMon has been tested by a group of developers at Ericsson (Austin branch)
  - Dedicated 32-bit architecture (where unused address bits are available)
  - Dedicated memory layout
- Successfully integrated AMon into their computing environment with minimal modification
  - $\sim$ 40 LOC in the compiler pass
  - $\sim$ 200 LOC in the runtime library
- They have conducted initial tests on small cases
- AMon detected memory issues in their test cases
  - Detected out-of-bounds accesses and uses of uninitialized memory

Performance Analysis

SPEC CPU 2017 Benchmarks - Reference Workloads

Benchmark	Native		ASan		EffectiveSan		AMon	
	Time (sec)	MRSS (MB)	Time (sec)	MRSS (MB)	Time (sec)	MRSS (MB)	Time (sec)	MRSS
505.mcf	232.4	623.7	285.3 (22.8 %)	919.8 (47.5 %)	508.3 (118.8 %)	624.8 (0.2 %)	370.6 (59.4%)	660.0 (5.8%)
519.lbm	123.5	420.3	128.3 (3.9 %)	476.8 (13.4 %)	198.1 (60.4 %)	420.8 (0.1 %)	221.0 (78.9%)	420.8 (0.1%)
544.nab	247.6	149.7	290.4 (17.3 %)	540.8 (261.2 %)	332.3 (34.2 %)	163.4 (9.2 %)	257.5 (3.9%)	164.4 (9.8%)
538.imagick	256.9	293.0	423.8 (64.9 %)	823.8 (181.2 %)	561.9 (118.7 %)	296.9 (1.3 %)	535.2 (108.3%)	296.5 (1.2 %))
557.xz	245.6	794.1	327.3 (33.3 %)	1051.0 (32.4 %)	384.3 (56.5 %)	794.3 (≈ 0%)	368.8 (50.1%)	794.7 (≈ 0%)
Average Overhead			28.5 %	107.2 %	77.8 %	2.2 %	60.1 %	3.4 %

MY ONE-WEEK VISIT TO CIENA

GCC Proof of Concept for AMon

- Organized by Mohammad and François
- Goal: Implement a GCC-based proof of concept for AMon
- Gained first exposure to GCC internals, including GIMPLE passes and instrumentation workflow
- Implemented an initial prototype!

AMon

GIMPLE Intermediate Representation

Quick Introduction

- GIMPLE is an intermediate representation used for target- and language-independent optimizations and instrumentation (e.g. ASan)
- Static Single Assignment (SSA) form, which facilitates data-flow analysis
- GIMPLE passes plug into GCC Middle-End and are applied in a predefined order

AMon GCC Compiler Pass

Initial Prototype

- Register the Pass
  - Defined as a `gimple_opt_pass`
- Walk Through the Code
  - Iterates over functions, basic blocks, and GIMPLE statements
- Detect Simple Memory Accesses
  - Identifies loads (`gimple_assign_load_p`) and stores (`gimple_store_p`)
  - Extracts pointer operand and access size
- Untaint the Pointer
  - Casts to 64-bit int
  - Masks taint bits
  - Casts back to original pointer type
  - Replaces the pointer operand
- Insert Runtime Checks
  - `amon_read` or `amon_write` call to `libamon`

DEMO TIME

And yes, it is the GCC variant of AMon!

Thanks!

Questions? Comments?

farzam.dorostkar@{polymtl.ca, gmail.com}

github.com/farzamdorostkar

farzamdorostkar.github.io