AddressMonitor (AMon): Low-overhead Spatial and Temporal Memory Safety for C

Farzam Dorostkar

Supervisor: Prof. Michel Dagenais

Co-supervisor: Prof. Heng Li

Polytechnique Montreal

DORSAL Laboratory

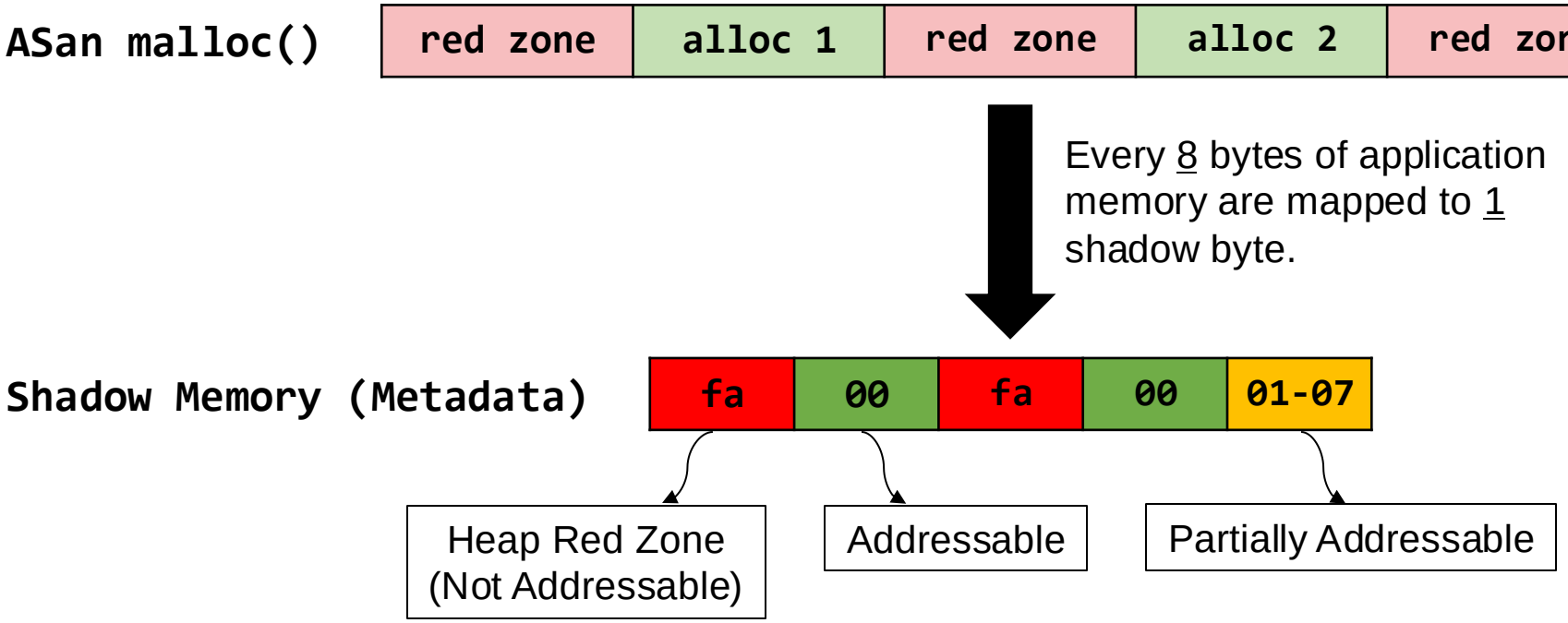
Dec. 5th 2024

AddressMonitor (AMon): Introduction

- AddressMonitor (AMon) is a dynamic tool designed to detect **heap spatial and temporal violations**
  - ❑ Out-of-bounds accesses (buffer overflows)
  - ❑ Use-after-frees
  - ❑ Double-frees
  - ❑ Memory leaks
- Currently targets C programs running on X86-64
- Pointer tainting
- Adds untainting code at **compile-time**
- Designed to have minimal overhead
  - ❑ Compared to AddressSanitizer (ASan): Much lower memory overhead + More precision in detecting buffer overflows

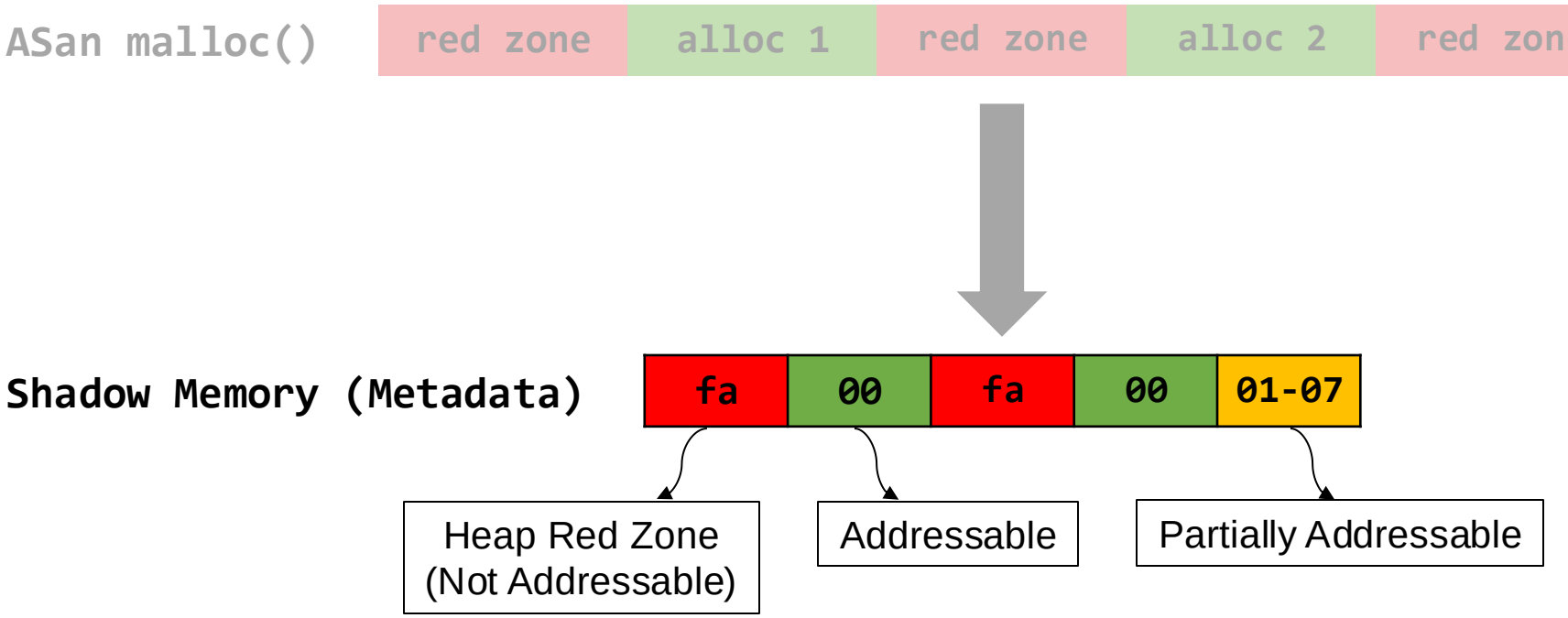
ASan: High Memory Overhead and Limited Overflow Detection

ASan: High Memory Overhead



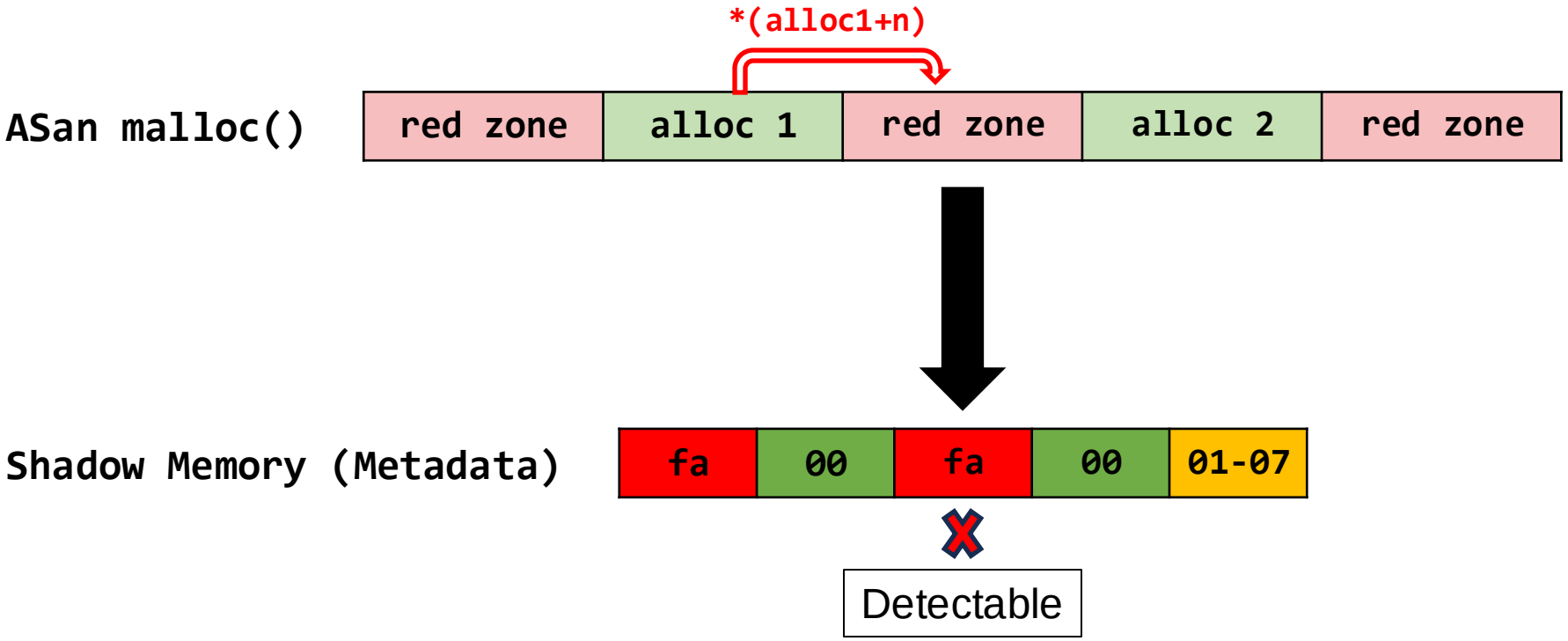
Red zones and **shadow memory** are key contributors to ASan's **high memory overhead** (2x-4x).

ASan: Limited Overflow Detection



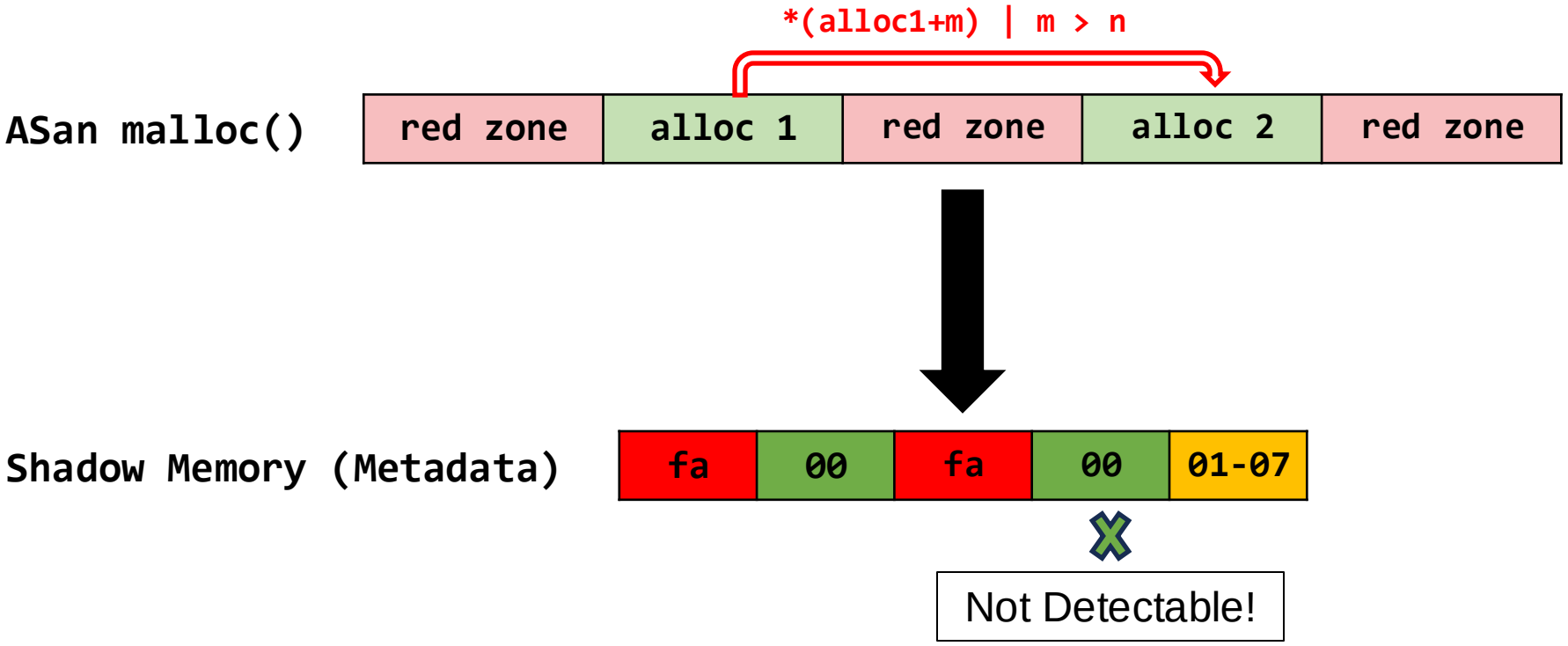
Shadow memory encodes a binary indicator (**addressable or not**) **without object-specific details**, leading to potential **false negatives**.

ASan: Limited Overflow Detection



Shadow memory encodes a binary indicator (**addressable or not**) **without object-specific details**, leading to potential **false negatives**.

ASan: Limited Overflow Detection



Shadow memory encodes a binary indicator (**addressable or not**) **without object-specific details**, leading to potential **false negatives**.

Pointer Tainting

Pointer Tainting

When Allocating Heap Objects

- Assign a unique taint (ID) to each allocated object
- Build and maintain an object table [taint, base address, size, staus]
- On most 64 bits architecture the first 2 bytes are unused
- Embed the taint into the 2 MS bytes of the returned address (**pointer tainting**)

When Accessing Memory

- Retrieve the taint
- Use the taint to verify the access against the object table
- Unlike shadow-based techniques, **pointer tainting provides object-specific analysis**.

But...

- Dereferencing a tainted pointer causes **segmentation fault**!

What Distinguishes AMon?

What Distinguishes AMon?

AMon instruments code with untainting logic at compile-time.

Similar tools, such as DataWatch, rely on costly dynamic techniques

- Let memory accesses trigger a SIGSEGV signal
- Perform dynamic patching
- Typically also require to re-taint
  - For instance, when modifying a register containing a tainted pointer, which might be shared across multiple memory-accessing instructions.
- Usually used for targeted protection, not full program verification

AMon instruments code at compile-time to call untainting

- **No re-tainting logic is required as it operates on IR registers rather than physical registers**
- Full program coverage

AMon: Implementation (Runtime Library & Compile-time Transformation)

AMon: Implementation

Runtime Library (libamon.so)

- Intercepts standard heap allocation and de-allocation functions
  - To add taint to returned pointers, etc.
- Intercepts other standard C functions as well
  - To untaint possibly tainted arguments
- Maintains the object table
- Defines the verification logic
- Defines environment variables to control the behavior of AMon
  - Supported object sizes
- It can be either preloaded using `LDPRELOAD` or added as a dependency using `patchelf`

AMon: Implementation

Compile-time Transformation

Implemented as an LLVM module pass

1. Identifies all IR instructions that perform pointer dereferencing
   - All Memory Access and Addressing Operations (such as `load`, `store`, and `cmpxchg`)
   - Memory-related Standard C Library Intrinsics (such as `llvm.memcpy` and `llvm.memset`)
2. Untaints the dereferenced pointer in each identified instruction
   - Extracts the pointer operand
   - Generates an untainted version of the pointer using the `llvm.ptrmask` intrinsic
   - Replaces the pointer in the current instruction with the untainted version using the `setOperand` method
3. Instruments each identified instruction for runtime verification
   - `amon_access(tainted pointer, access size, is_write)`

AMon: Implementation

AMon is easy to use.

- Compile your source code to LLVM IR
```
clang -S -emit-llvm test.c -o test.ll
```

- Apply the 'amon' LLVM pass to the generated IR
```
opt -passes='amon' -S test.ll -o test_amon.ll
```

- Compile the transformed IR to executable
```
clang test_amon.ll -o test_amon
```

- Run the executable while preloading libamon.so
```
LD_PRELOAD=./libamon.so ./test_amon
```

AMon: Implementation

```c
#include <stdlib.h>

int main(int argc, char *argv[])
{
int tmp;
int *ptr_taint = (int*)malloc(sizeof(int));
*ptr_taint = 2024;
tmp = *(ptr_taint + 1);
free(ptr_taint);
*ptr_taint = 2025;
free(ptr_taint);
return 0;
}
```

test.c

AMon: Implementation

objtbl

Taint	Base Address	Size	Call Stack	Status
		...		
0x03ec	0x6231ba38b1b0	4	main:6	Allocated

ptr_taint = 0x03ec6231ba38b1b0

```c
#include <stdlib.h>

int main(int argc, char *argv[])
{
int tmp;
int *ptr_taint = (int*)malloc(sizeof(int));
*ptr_taint = 2024;
tmp = *(ptr_taint + 1);
free(ptr_taint);
*ptr_taint = 2025;
free(ptr_taint);
return 0;
}
```

test.c

```c
static void*
  malloc(size_t size)
{
  ... // internal checks
  void *result = __libc_malloc(size);
  result = amon_protect(result); // taint added
  objtbl_add(result, size); // added to objtbl
  return result;
}
```

libamon.so

AMon: Implementation

objtbl

Taint	Base Address	Size	Call Stack	Status
		...		
0x03ec	0x6231ba38b1b0	4	main:6	Allocated

ptr_taint = 0x03ec6231ba38b1b0

```c
#include <stdlib.h>

int main(int argc, char *argv[])
{
int tmp;
int *ptr_taint = (int*)malloc(sizeof(int));
*ptr_taint = 2024;
tmp = *(ptr_taint + 1);
free(ptr_taint);
*ptr_taint = 2025;
free(ptr_taint);
return 0;
}
```

test.c

```
; Function Attrs: noinline nounwind uwtable
define dso_local i32 @main(i32 %argc, ptr %argv) {
entry:
  ...
  %ptr_taint = call noalias ptr @malloc(i64 noundef 4)
  store i32 2024, ptr %ptr_taint, align 4
  ...
}
```

Segmentation Fault!

test.ll

AMon: Implementation

objtbl

Taint	Base Address	Size	Call Stack	Status
...				
0x03ec	0x6231ba38b1b0	4	main:6	Allocated

ptr_taint = 0x03ec6231ba38b1b0

Adds untainting code at Compile-time

```
1 ; Function Attrs: noinline nounwind uwtable
2 define dso_local i32 @main(i32 %argc, ptr %argv) {
3 entry:
4   ...
5   %ptr_taint = call noalias ptr @malloc(i64 noundef 4)
6   store i32 2024, ptr %ptr_taint, align 4
7   ...
8 }
```

Segmentation Fault!

test.ll

```
1 ; Function Attrs: noinline nounwind uwtable
2 define dso_local i32 @main(i32 %argc, ptr %argv) {
3 entry:
4   ...
5   %ptr_taint = call noalias ptr @malloc(i64 noundef 4)
6   %1 = call ptr @llvm.ptrmask.p0.i64(ptr %0, i64 0x0000FFFFFFFFFFFF)
7   call void @amon_access(ptr %ptr_taint, i64 4, i1 true)
8   store i32 2024, ptr %1, align 4
9   ...
10 }
```

test_amon.ll

Runtime verification against objtbl

AMon: Implementation

objtbl

Taint	Base Address	Size	Call Stack	Status
		...		
0x03ec	0x6231ba38b1b0	4	main:6	Allocated

ptr_taint = 0x03ec6231ba38b1b0

```c
#include <stdlib.h>

int main(int argc, char *argv[])
{
int tmp;
int *ptr_taint = (int*)malloc(sizeof(int));
*ptr_taint = 2024;
tmp = *(ptr_taint + 1);
free(ptr_taint);
*ptr_taint = 2025;
free(ptr_taint);
return 0;
}
```

test.c

```
; Function Attrs: noinline nounwind uwtable
define dso_local i32 @main(i32 %argc, ptr %argv) {
entry:
  ...
  %add.ptr = getelementptr inbounds i32, ptr %ptr_taint, i64 1
  %2 = load i32, ptr %add.ptr, align 4
  ...
}
```

Segmentation Fault!

test.ll

AMon: Implementation

objtbl

Taint	Base Address	Size	Call Stack	Status
...				
0x03ec	0x6231ba38b1b0	4	main:6	Allocated

ptr_taint = 0x03ec6231ba38b1b0

Adds untainting code at Compile-time

```
1 ; Function Attrs: noinline nounwind uwtable
2 define dso_local i32 @main(i32 %argc, ptr %argv) {
3 entry:
4   ...
5   %add.ptr = getelementptr inbounds i32, ptr %ptr_taint, i64 1
6   %2 = load i32, ptr %add.ptr, align 4
7   ...
8 }
```

Segmentation Fault!

test.ll

```
1 ; Function Attrs: noinline nounwind uwtable
2 define dso_local i32 @main(i32 %argc, ptr %argv) {
3 entry:
4   ...
5   %add.ptr = getelementptr inbounds i32, ptr %ptr_taint, i64 1
6   %3 = call ptr @llvm.ptrmask.p0.i64(ptr %add.ptr, i64 0x0000FFFFFFFFFFFF)
7   call void @amon_access(ptr %add.ptr, i64 4, i1 false)
8   %4 = load i32, ptr %3, align 4
9   ...
10 }
```

test_amon.ll

Runtime verification against objtbl

amon_access(0x03ec6231ba38b1b4, 4, false)

AMon detects the out-of-bounds read violation

AMon: Implementation

objtbl

Taint	Base Address	Size	Call Stack	Status
		...		
0x03ec	0x6231ba38b1b0	4	main:9	Freed

ptr_taint = 0x03ec6231ba38b1b0

```c
#include <stdlib.h>

int main(int argc, char *argv[])
{
int tmp;
int *ptr_taint = (int*)malloc(sizeof(int));
*ptr_taint = 2024;
tmp = *(ptr_taint + 1);
free(ptr_taint);
*ptr_taint = 2025;
free(ptr_taint);
return 0;
}
```

test.c

```c
static void*
  free(void *ptr)
{
  ... // internal checks
  if(amon_is_protected(ptr)) amon_free_protect(ptr);
  else __libc_free(ptr);
}
```

libamon.so

AMon: Implementation

Compile-time Transformation: Optimizations

Not all identified instructions require monitoring

1. Filtering out accesses to stack
    - Filters out instructions where the pointer operand originates directly or indirectly (through pointer arithmetic) from an `alloca` instruction.
2. Not instrumenting memory accesses that can be analyzed statically (out-of-bounds)
    - AMon additionally incorporates a degree of static analysis
    - When both the allocation size and the access size are known at compile time
    - No instrument (runtime check) needed, but still needed to untaint the pointer
3. If all accesses to a pointer can be analyzed statically, no need to taint it
    - For instance, replaces a `malloc` with a direct `__libc_malloc`

Results & Conclusion

Results

- Three SPEC CPU 2017 benchmarks
- Under ASan and AMon
- Compared to native compilation

Benchmark	Native			ASan			AMon		
	Time (Sec)	MRSS (MB)	Exec. Size	Time (x)	MRSS (x)	Exec. Size	Time (x)	MRSS (x)	Exec. Size
505.mcf_r	7.8	292	83 KB	1.4 x	2.1 x	11 MB	1.3 x	≈ 1 (1.03) x	120 KB
519.lbm_r	1.3	420	64 KB	2.6 x	1.2 x	11 MB	1.6 x	≈ 1 (1.02) x	92 KB
544.nab_r	1.1	2.6	506 KB	1.4 x	5.5 x	12 MB	1.2	1.3 x	665 KB
557.xz_r	0.7	531	588 KB	1.7 x	1.2 x	12 MB	1.4	≈ 1 (1.02) x	763 KB

Conclusion

- **AddressMonitor (AMon):** A tool designed to detect heap-based spatial and temporal memory access violations in C programs targeting the x86-64 architecture
- **Core approach:** Leverages pointer tainting and introduces a novel compile-time transformation for efficient pointer untainting.
- **Compared to ASan:** Achieves significantly lower memory overhead, relatively reduced runtime overhead, and enhanced precision in identifying buffer overflow vulnerabilities
- **Implementation Flexibility:** Can be integrated with compilers beyond LLVM

Thanks!

Questions? Comments?

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