Configurable Data Confidentiality and Integrity with DataShield

Scott A. Carr, Mathias Payer

Purdue University

AsiaCCS 2017

Contents

- Motivation
- Introduction
- Design
- Implementation
- Evaluation
  - Performance and Security
- Conclusion

Motivation

Heartbleed

- Missing bounds check in OpenSSL
- Information leak
- Up to 66% of websites

ShellShock

- Bash executes trailing strings in env vars
- Code injection attack
- Effectively all Linux servers

StageFright

- Parser bugs for video/image messages
- Remote code execution
- 850M Android devices

Introduction

Data Confidentiality and Integrity

- Some data are more sensitive (worth paying overhead)
  - Ex: stack canaries, W^X, CFI, CPI[1]
- Let the programmer choose
- Compiler inserts dynamic checks to protect sensitive data

1. Code Pointer Integrity. Kuznetsov et al. OSDI 2014.

Our Assumptions

- Rewriting from scratch is impractical
- Programmer will not write a complete specification
- Only low overhead is acceptable
  - 5-10% may undetectable by the user [1]
- Have program source code

1. Everything You Want to Know About Pointer-Based Checking. Nagarakatte et al. SNAPL 2015.

Attacker Model

- Original program is buggy but benign
- Attackers exploit bugs to read/write unintended data
- At runtime, code is not writable
  – Data Execution Protection (DEP)
  – Standard on modern desktops and servers

What is Memory Safety?

- Reads/writes through pointers read/write the object to which the pointer was most recently assigned
- Spatial: pointer arithmetic (bounds checks)
- Temporal: heap/stack variable lifetimes
- Confidentiality – memory safety for reads
- Integrity – memory safety for writes

Why not Complete Memory Safety?

- Protecting all data is too costly
  - ~100% overhead [1,2]
- Overhead is a function of # of dynamic checks

1. SoftBound: Highly Compatible and Complete Memory Safety for C. Nagarakatte et al. PLDI 2009
2. CETS: Compiler-Enforced Temporal Safety for C. Nagarakatte et al. ISMM 2010.

DCI Contributions

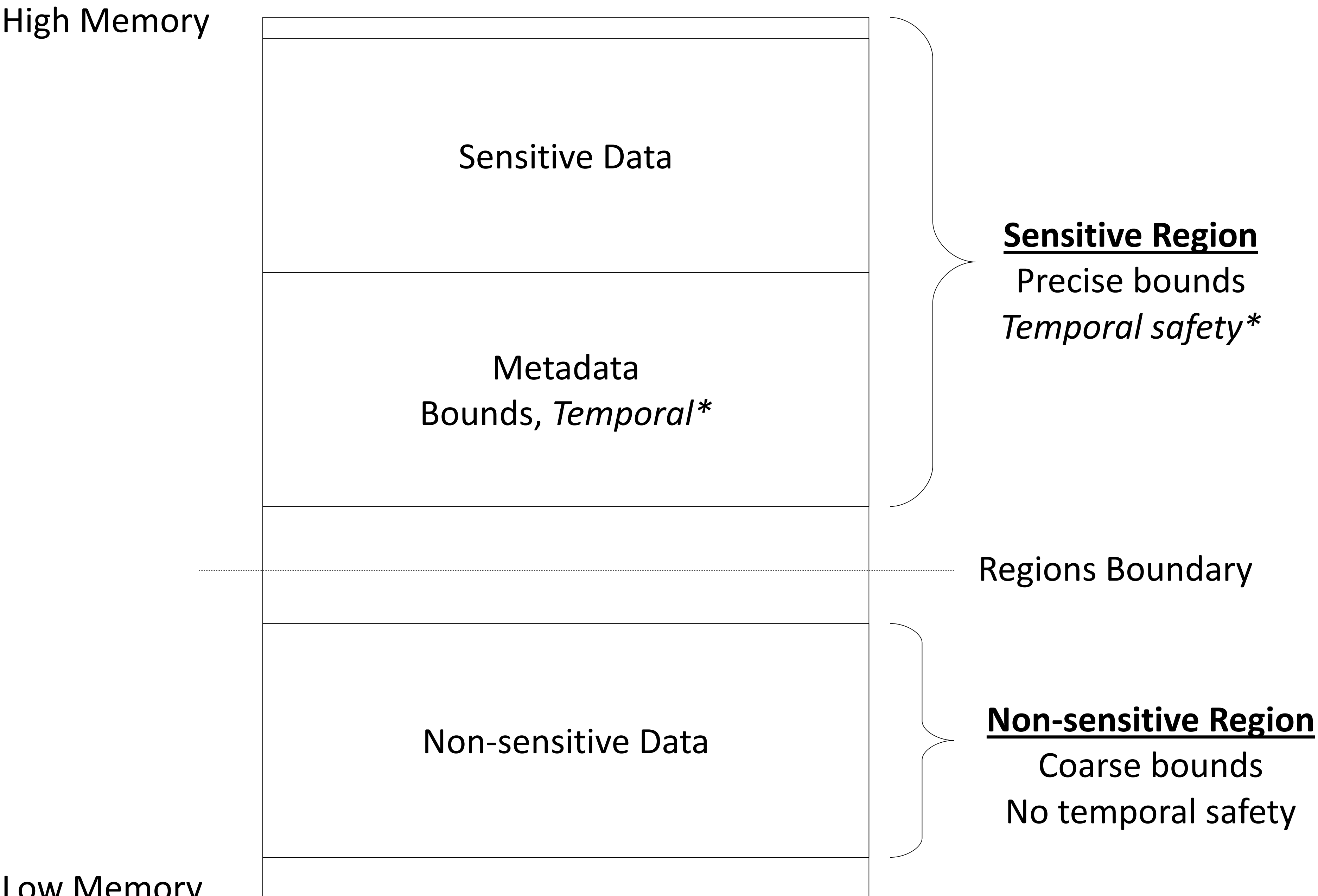
- New policy for protecting selected data
- Open-source LLVM-based implementation [1]
- Lower overhead than full memory safety
- Three coarse bounds check implementations
- Security evaluation
  - Detects attacks in mbedTLS

1. https://github.com/HexHive/datashield

Design

DCI Policy

- *Sensitive pointers* can only access the intended sensitive object
- *Non-sensitive pointers* can access any non-sensitive data
- Explicit data-flow between sensitive and non-sensitive objects is forbidden



*Not in prototype currently

Annotations

- Type based
- Marks type and members
- Cannot be cast away
- Mixed sensitivity `structs` not allowed

```
__attribute__((annotate("sensitive")))
```

Type Sensitivity

- All nested types have same sensitivity
- Pointers to sensitive types are sensitive
- Nesting a primitive type *P* does not make **<u>every</u>** *P* sensitive

```
struct S {
  int x;
  struct T *t;
};

struct T {
  float z;
  struct U *u;
}

struct U {
  …
};
```

Implementation

Annotation Example (1)

```
struct foo {
    char* name;
    int x,y,z;
};
```

```
__attribute__((annotate("sensitive"))) struct foo ignore;
```

Sensitive Allocation Example

```
struct foo* ptr = malloc(sizeof(struct foo));
```

Sensitive Allocation Example

```
struct foo* ptr = malloc(sizeof(struct foo));
struct foo* ptr = sensitive_malloc(sizeof(struct foo));
```

```
table[&ptr].base = ptr;
table[&ptr].end = ptr + sizeof(struct foo);
```

Sensitive Access Example

```
bounds = table[&ptr]
assert(bounds.base <= &(ptr->x));
assert(&(ptr->x) + sizeof(ptr.x) < bounds.end);


ptr->x = 5;
```

Non-sensitive Example

```
int* arr = malloc(sizeof(int)*100);

int* idx = arr+8;

*idx = 42;
```

Non-sensitive Example

```
int* arr = malloc(sizeof(int)*100);
int* arr = non_sensitive_malloc(sizeof(int)*100);

int* idx = arr+8;

*idx = 42;

int* idx_masked = idx & mask;
*idx_masked = 42;
```

Automatic Promotion

- Avoid annotated local and temporary variables
- Ex:

```
struct foo* myfunc(struct sens* a, struct sens*b) {
    int tmp = a->x + b->x;
    a->x = tmp;
    return a;
}
```

Automatic Promotion (2)

- Promotion is safe
- Just bounds check more variables
- Automatically clone function based on context

Implementation Overview

1. Compile time analysis
    - Module-level analysis
    - Inter-procedural type and context sensitive analysis
2. Runtime
    - Separate sensitive/non-sensitive variables
        - Heap
        - Stack
        - Global

Coarse Bounds Checks

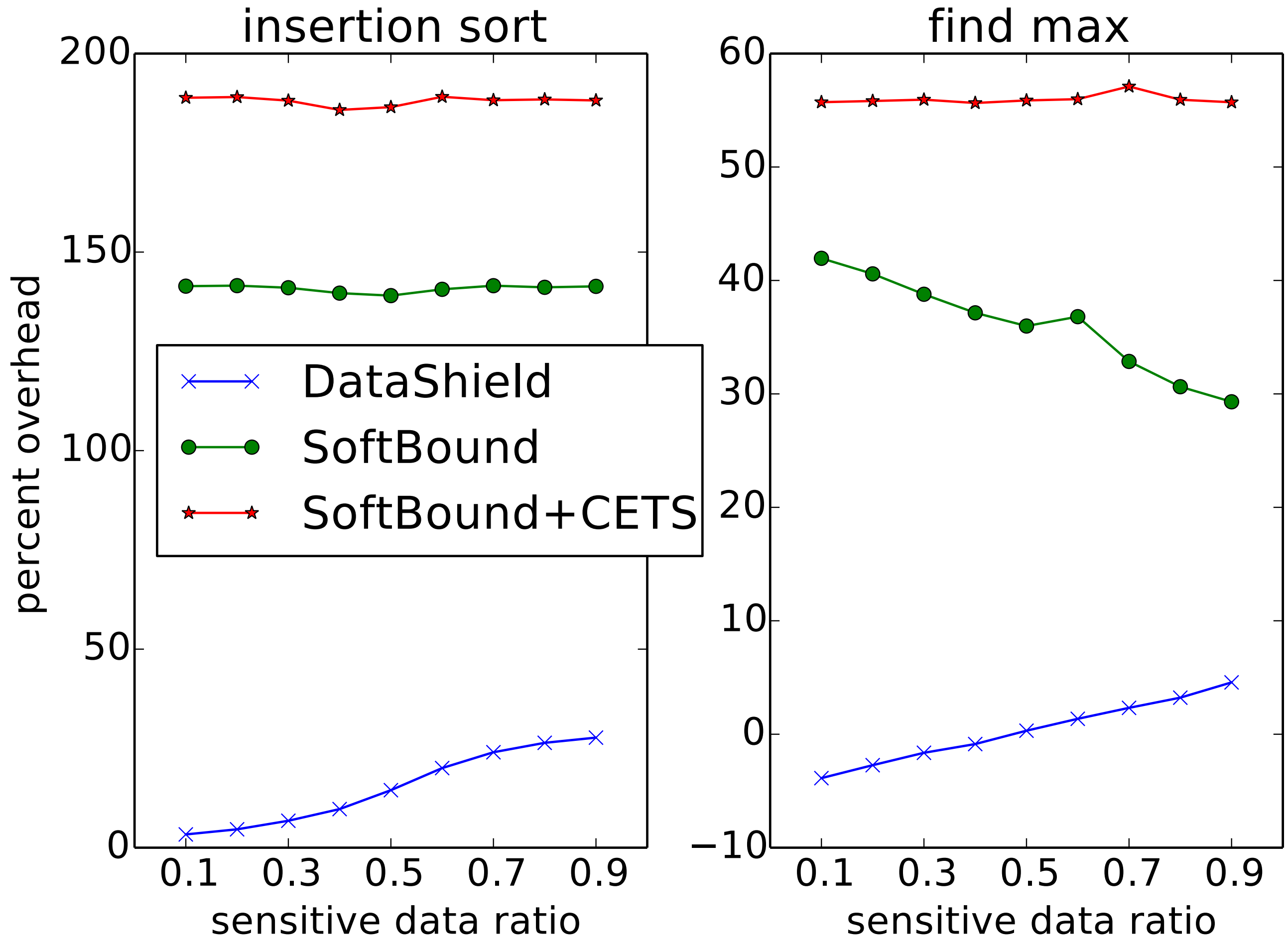
- Software Masking
- Intel MPX
- Address Override Prefix

Evaluation

Evaluation Sensitivity Ratio

- How does the portion of sensitive objects effect the overhead?
- Two microbenchmarks:
  - Insertion sort (quadratic complexity)
  - Find max (linear complexity)
- Vary percent sensitive 10% to 90%
- Compare against SoftBound

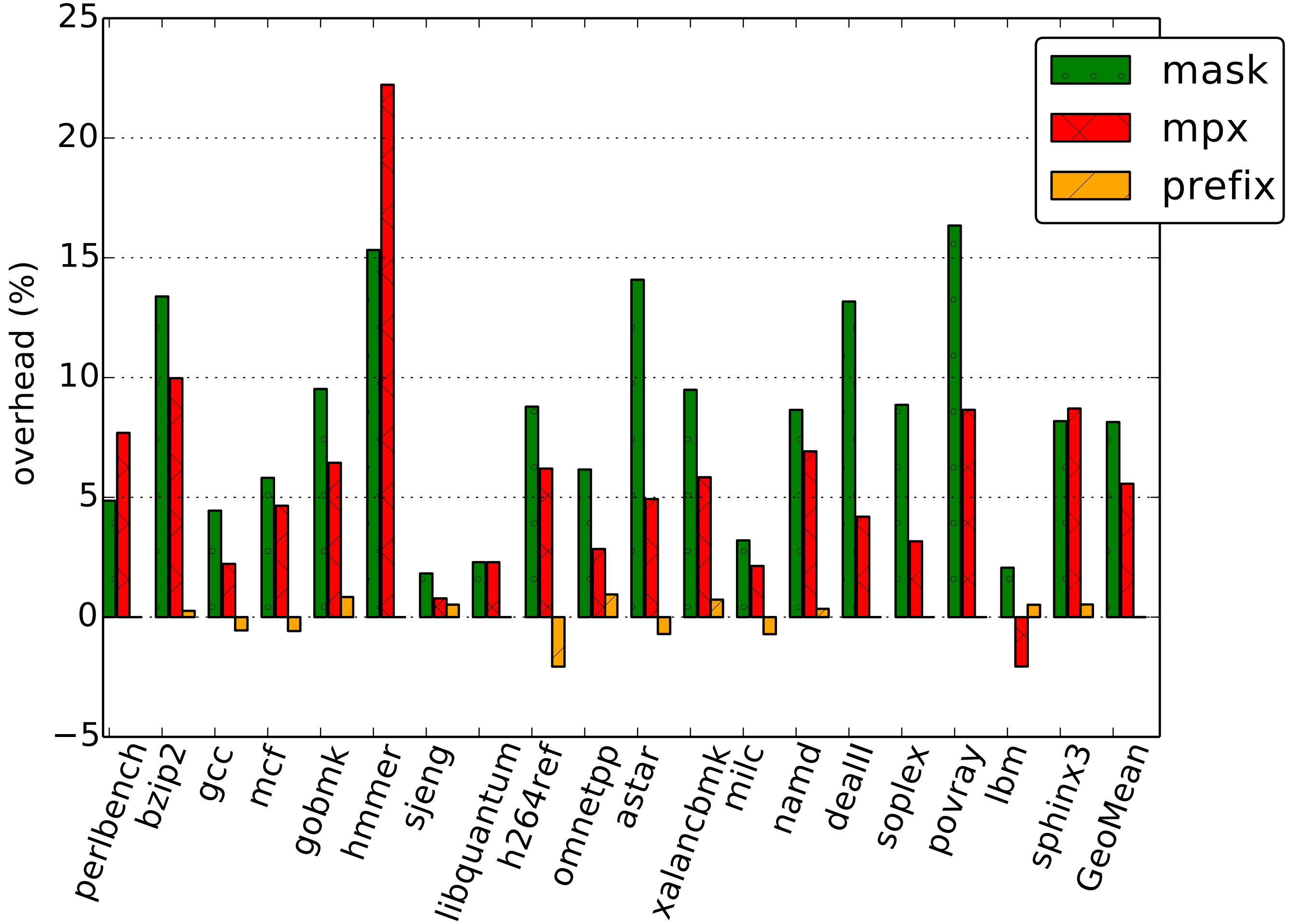
Evaluation - Sensitivity Ratio



Evaluation – SPEC CPU2006

- No annotations added
- Sensitive regions bounds still enforced
- Measured overhead of code that does not access sensitive data
- Three bounds implementations:
  1. Software Masking
  2. Intel MPX
  3. Address Override Prefix

Evaluation – Coarse Bounds Checks



Security Evaluation

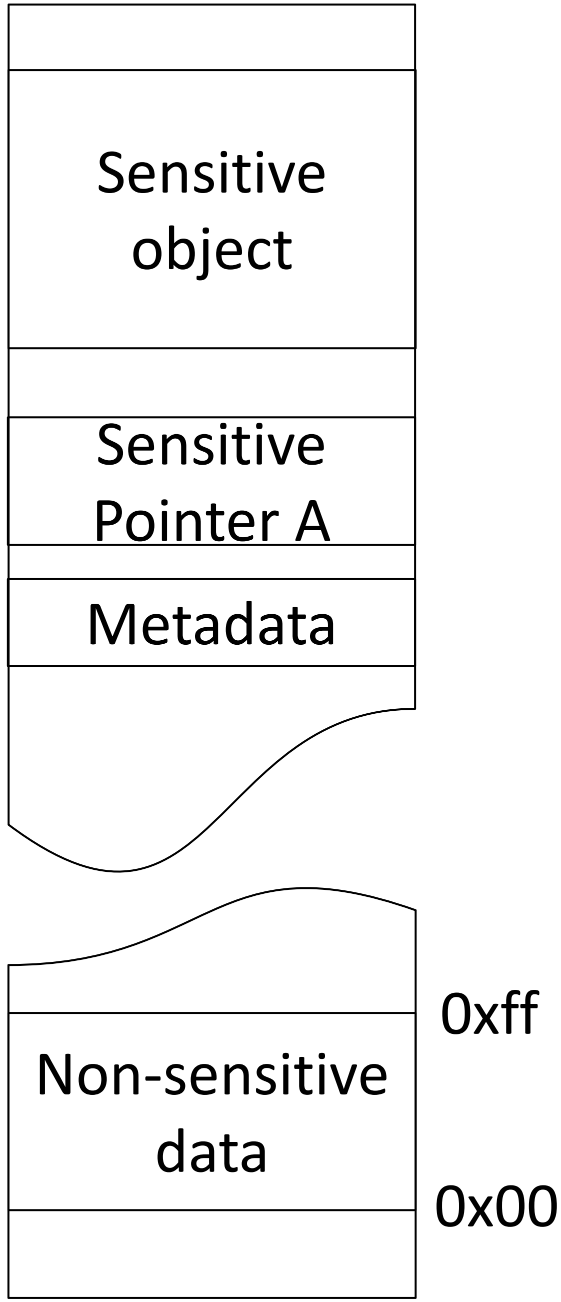
- CVE-2015-5291 from mbedTLS
- Malicious session ticket causes *buffer overflow*
- Proof of concept exploit publically available
- Compiled pre-patch version of mbedTLS
- Ran exploit
- Detected by DCI

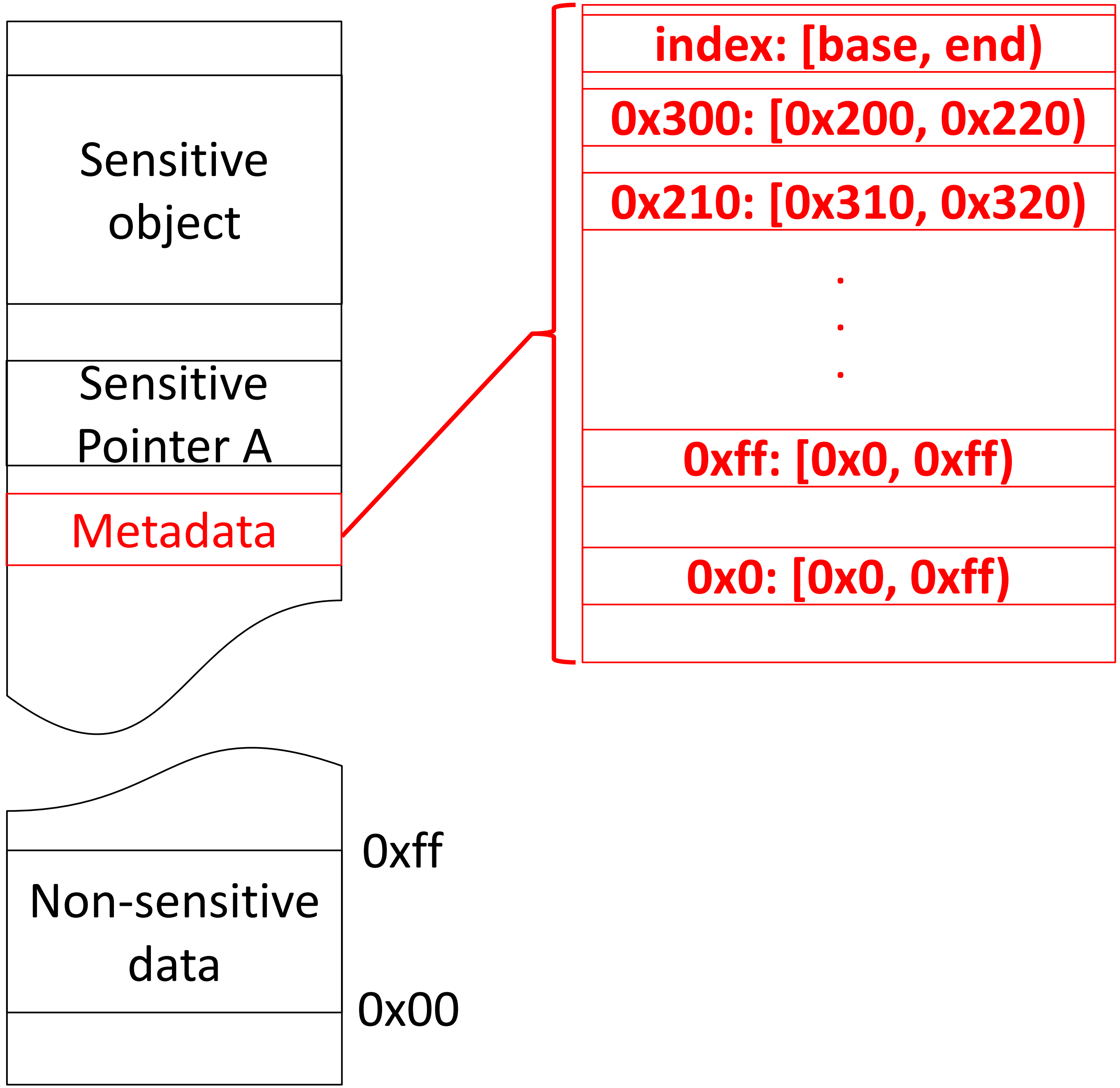
DCI Summary

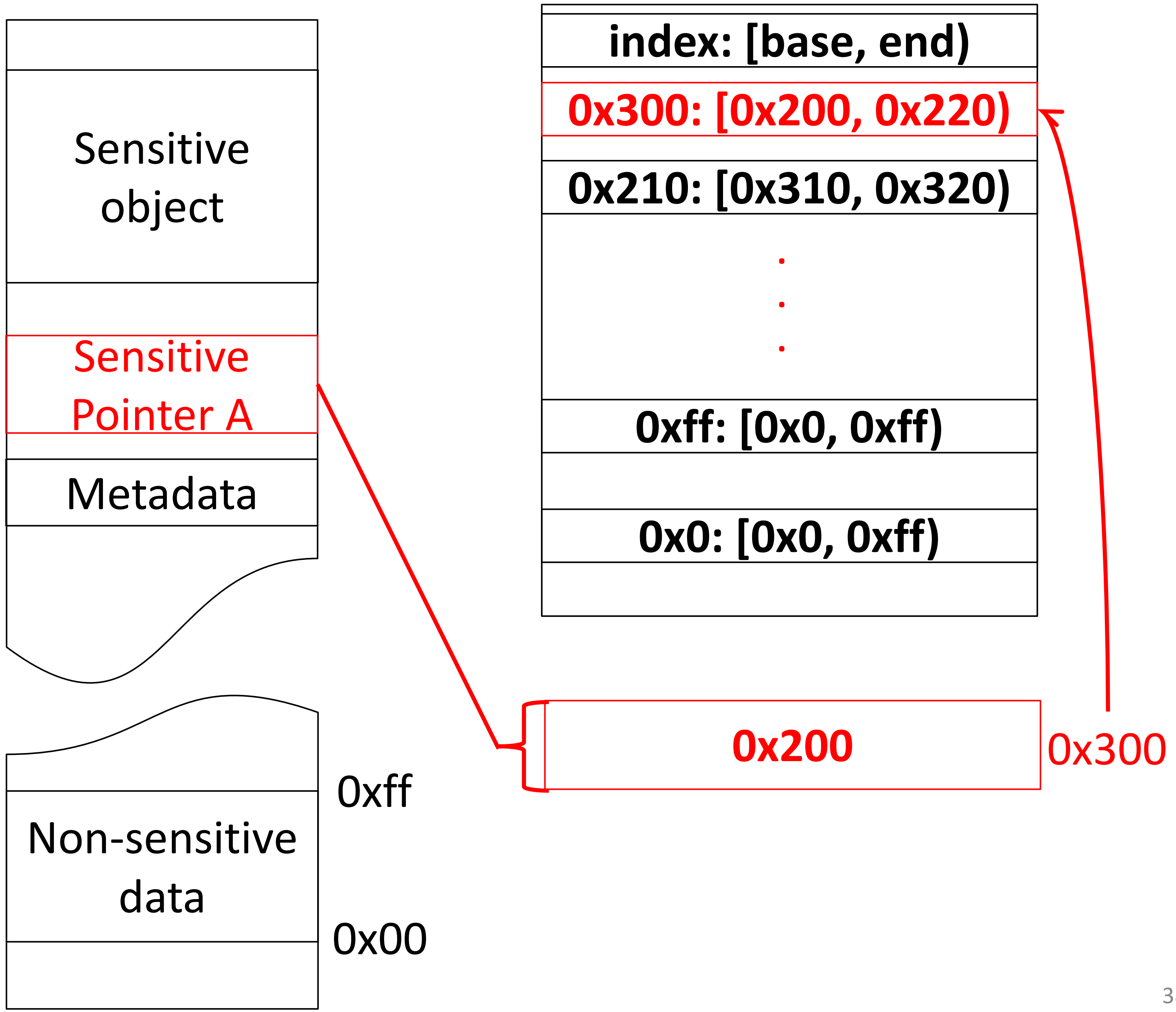
- *Strong* protection for *sensitive data*
- *Weaker* (but lower overhead) protection for *non-sensitive*
- Compiler analysis and runtime library
- C/C++ support
- Compatible standard libraries

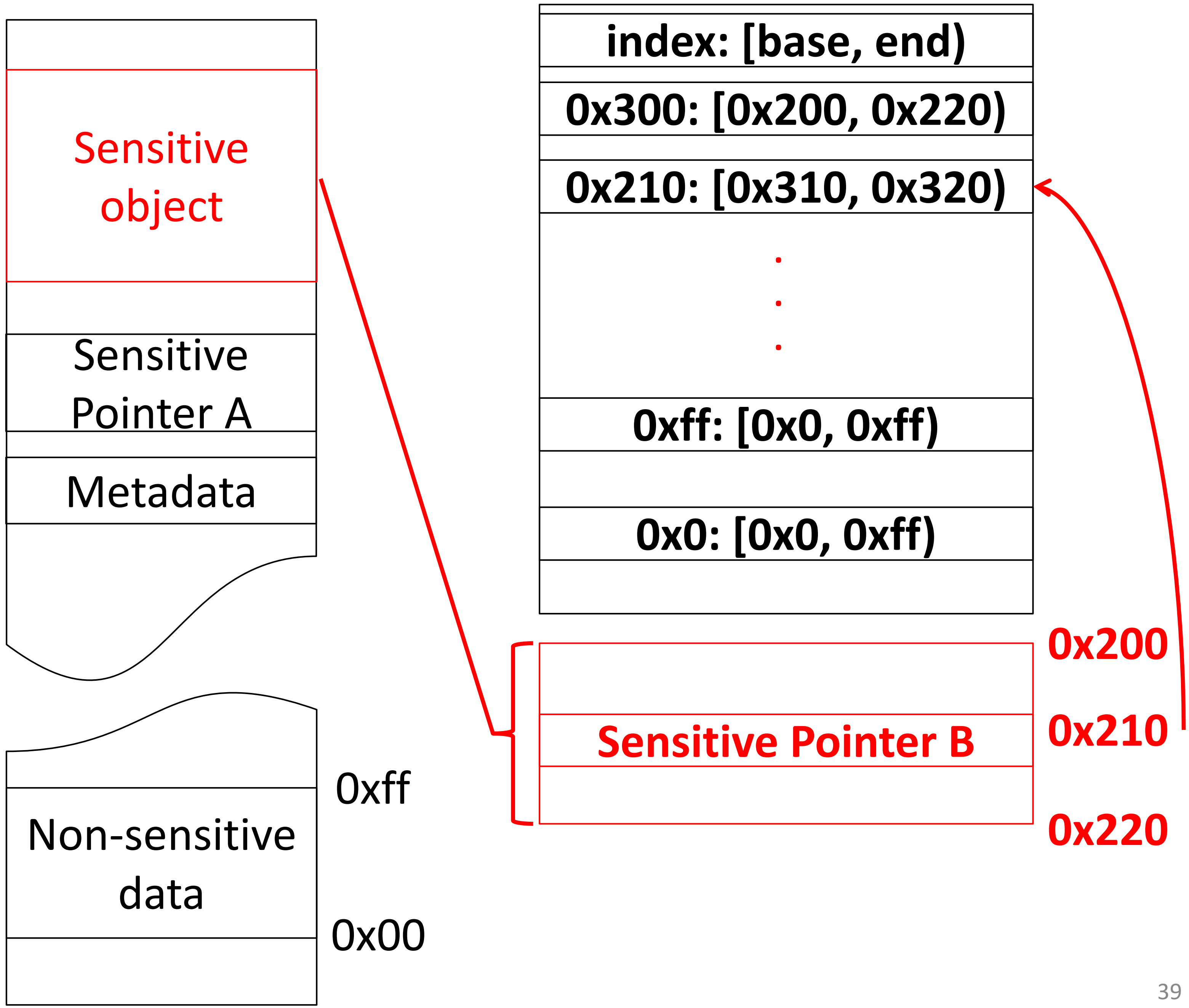
Questions?

Extra Slides


Sensitive object
Sensitive Pointer A
Metadata
0xff
Non-sensitive data
0x00


Sensitive object
Sensitive Pointer A
Metadata
Non-sensitive data
0xff
0x00
index: [base, end)
0x300: [0x200, 0x220)
0x210: [0x310, 0x320)
.
.
.
0xff: [0x0, 0xff)
0x0: [0x0, 0xff)


Sensitive object
Sensitive Pointer A
Metadata
0xff
Non-sensitive data
0x00
index: [base, end)
0x300: [0x200, 0x220)
0x210: [0x310, 0x320)
0xff: [0x0, 0xff)
0x0: [0x0, 0xff)
0x200
0x300



Sensitive object
Sensitive Pointer A
Metadata
Non-sensitive data
0xff
0x00
index: [base, end)
0x300: [0x200, 0x220)
0x210: [0x310, 0x320)
0xff: [0x0, 0xff)
0x0: [0x0, 0xff)
Sensitive Pointer B
0x200
0x210
0x220

Standard Library Support

- Option 1: per-application lib
  - Rewrite specialized versions of each library function on-demand
  - Same analysis/rewriting as application
  - Con: Requires unique lib per application
  - Pro: Internal checks
- Option 2: drop in replacement lib
  - Make all library allocated data non-sensitive
  - Use wrappers and copies for sensitive
  - Con: No internal checks
  - Pro: Allows single, compatible lib

Evaluation - astar

- C++: 4,285 LoC
- Relaxed policy: separation mode
  - Primitive arithmetic does not propagate sensitivity
  - Reduces overhead 96% -> 9.12%
  - Reduce sensitive bounds checks by 10^6 times

Evaluation - mbedTLS

- C: 30,000 LoC
- Instrument sample server and client
- Annotate `ssl_context`
- 35.7% overhead

- Challenge: sensitive data passed to callee through function pointer

Limitations

- Variadic arguments as sensitive
- Temporal metadata not implemented
- Region-based temporal protection if one sensitive type
  - Similar to Cling [1]

1. Cling: A Memory Allocation to Mitigate Dangling Pointers. P. Akritidis. USENIX Security 2010