SoftBound: Highly Compatible and Complete Spatial Memory Safety for C

Santosh Nagarakatte, Jianzhou Zhao,
Milo M. K. Martin, Steve Zdancewic

Presented by Jaehwang Jung

Memory Safety

No undefined behavior while accessing memory.

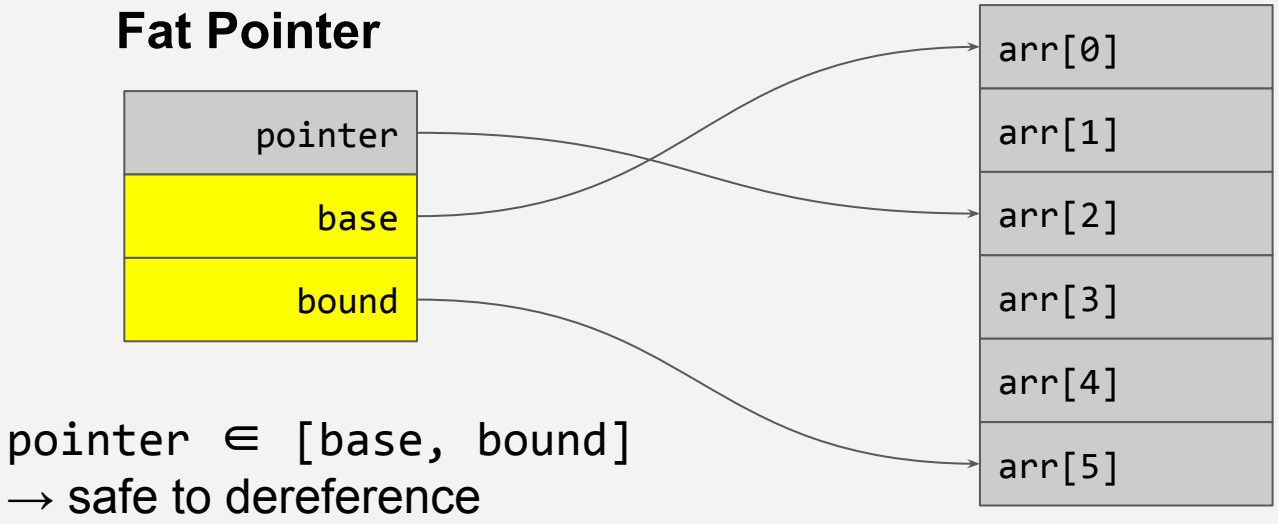
- **Spatial memory safety**:
  All pointer dereferences are within bounds of a valid object.
  - e.g. absence of out-of-bound array indexing
- Temporal memory safety:
  All pointer dereferences are valid at the time of the dereference.
  - e.g. absence of use-after-free

Retrofitting C with spatial memory safety is difficult

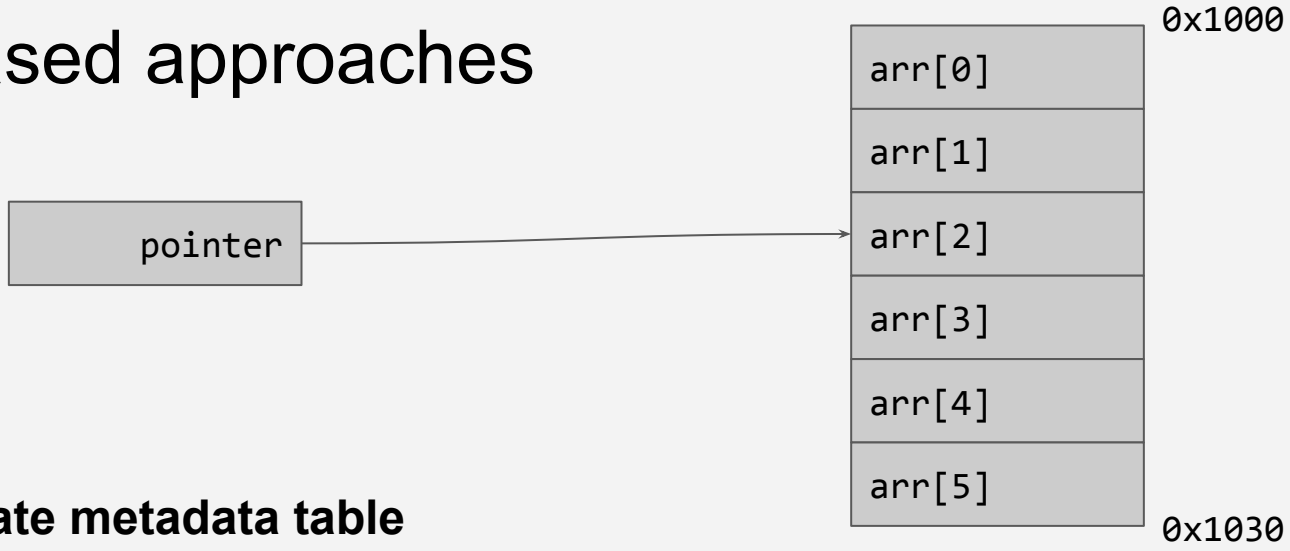
- conflation of arrays and pointers
- unchecked array indexing
- pointer arithmetic
- pointers to the middle of objects
- arbitrary casts user-visible memory layout
- structures with internal arrays
- ...

Previous Approaches for Enforcing Spatial Safety in C

Pointer-based approaches



Object-based approaches



Separate metadata table

bounds of arr: [0x1000, 0x1030]
bounds of asdf: [..., …]
….

`pointer` is within the bounds of `arr`
→ safe to dereference

Pointer-based vs. Object-based

- Low compatibility

(changes object layout)

+ High compatibility

Compatibility

Fat pointer changes the object layout →

- Not compatible with un-instrumented code
- May require source code modification if the code relies on the object layout!

On the other hand, object-based approach instrumentation isn't intrusive.

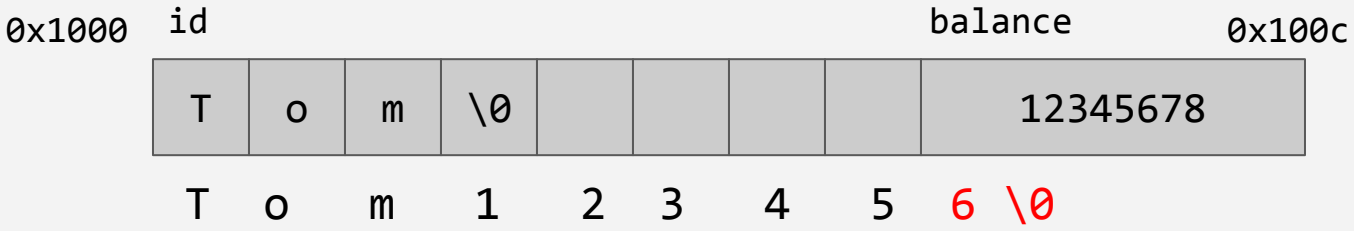
Pointer-based vs. Object-based

Pointer-based	Object-based
+ Complete spatial safety	+ High compatibility
- Low compatibility (changes object layout)	- Incomplete spatial safety (sub-object overflow)

Sub-object problem

```
struct Account {
    char id[8];
    int balance;
} tom;

char* ptr = tom.id;
strcpy(ptr, "Tom123456");
```



bounds of tom_accont: [0x1000, 0x100c]
….

`&tom = &tom.id = 0x1000`
`&tom.id` inherits the bound of `&tom`!
→ Can't detect the overflow inside the struct!

SoftBound
= Completeness of Pointer-based
+ Compatibility of Object-based

Key Idea

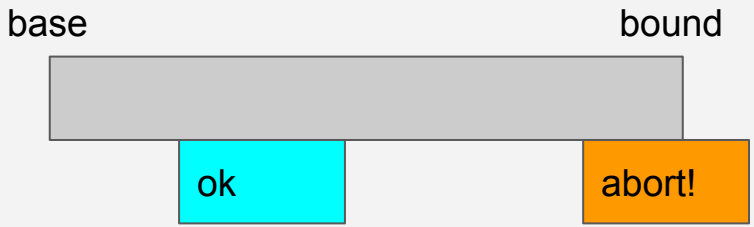
- Pointer-based, but implemented like object-based (for pointers in memory)
  - Metadata for every pointer (like pointer-based),
  - stored in a disjoint metadata space (like object-based)
- Instrumentation (on LLVM IR)
  - Insert bound check before pointer dereference
  - Propagate the metadata after pointer creation

Pointer dereference check

```
check(ptr, ptr_base, ptr_bound, sizeof(*ptr))

value = *ptr


void check(ptr, base, bound, size) {
    if (ptr ∉ [base, bound - size])
        abort();
}
```



A variable name maps to a register in LLVM IR.

Pointer creation

- create metadata

```
int* ptr = malloc(size)
int* ptr_base = ptr
int* ptr_bound = ptr + size;

int arr[100];
int* ptr = &arr[0];
int* ptr_base = &arr[0];
int* ptr_bound = ptr_base + sizeof(arr);
```

Pointer arithmetic and array indexing

- inherit the metadata

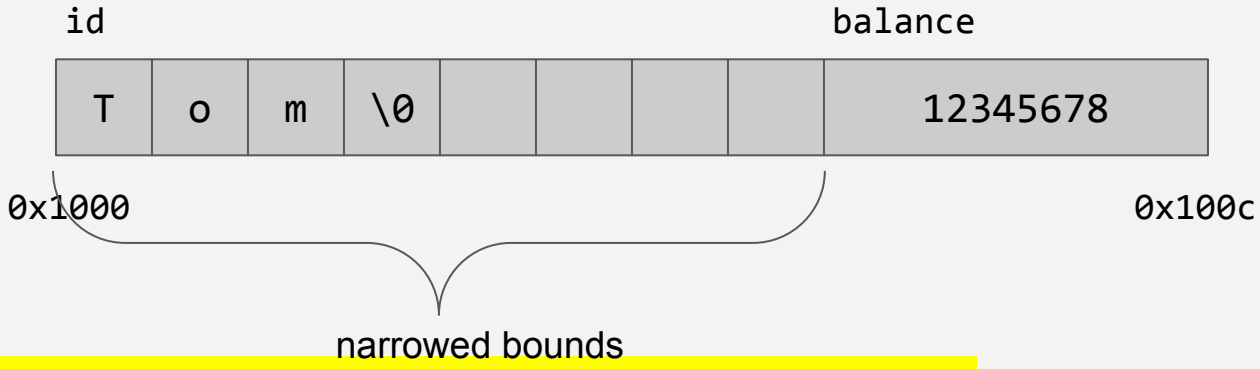
```
int* newptr = ptr + index;
```

```
int* newptr_base = ptr_base;
int* newptr_bound = ptr_bound;
```

Struct field accesses

- Combination of pointer arithmetic and derefences
- Bounds can be narrowed in order to detect sub-object overflow.

```
struct Account {
    char id[8];
    int balance;
} *tom;
...
char* ptr = tom->id;
```




```
ptr_base = tom->id;
ptr_bound = ptr_base + sizeof(tom->id);
```

Loading a pointer from memory

- Load metadata from the table

```
int** ptr;
int* new_ptr;
...
check(ptr, ptr_base, ptr_bound, sizeof(*ptr));
newptr = *ptr;
newptr_base = table_lookup(ptr)->base;
newptr_bound = table_lookup(ptr)->bound;
```

Storing a pointer in memory

- Store metadata in the table

```
int** ptr;
int* new_ptr;
...
check(ptr, ptr_base, ptr_bound, sizeof(*ptr));
(*ptr) = new_ptr;

table_lookup(ptr)->base = newptr_base;
table_lookup(ptr)->bound = newptr_bound;
```

Functions

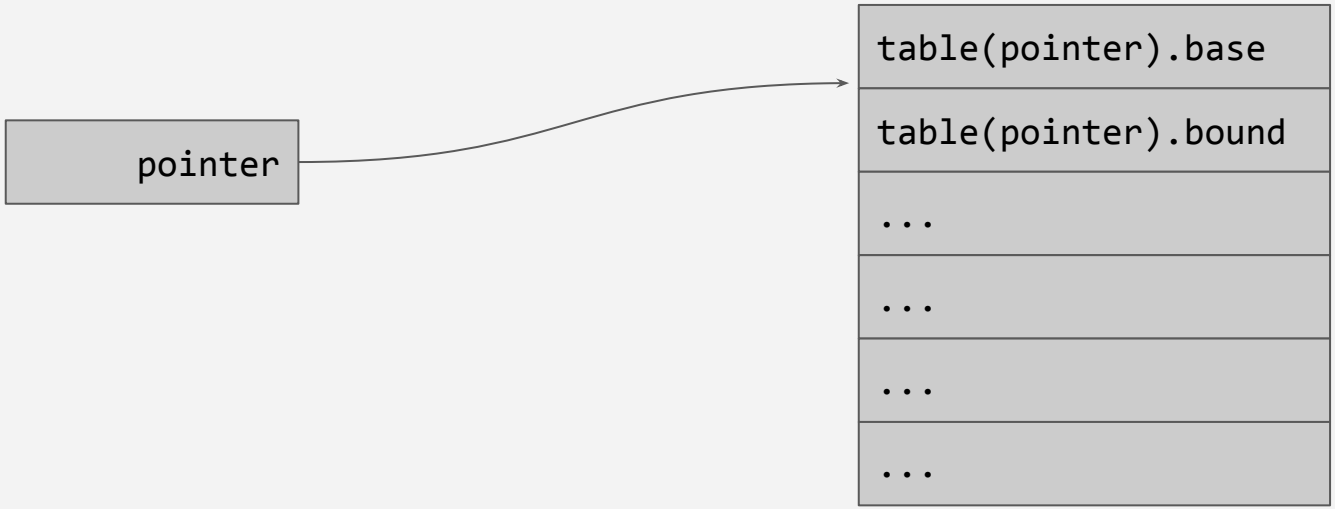
- Procedure cloning → naturally supports separate compilation!

```
char* func(char* s);
char* new_ptr = func(ptr);
```

```
typedef struct {
    pointer: char *s;
    base: char *s;
    bound: char *s;
} char_ptr;
char_ptr sb_func(char* s, void* s_base, void* s_bound);
char_ptr new_ptr = sb_func(ptr, ptr_base, ptr_bound);
```

Metadata table implementation

- Hashmap with simple hash (shift & mask) and open-addressing
- Shadow space



Proof that SoftBound detects spatial violations

Similar to usual proof of soundness of type systems.

- Operational semantics for simplified C
- Add metadata check and propagation to the operational semantics
- Define Well-formedness on program state using bounds info
- Preservation and progress
  - preservation: Each command preserves well-formedness.
  - progress: Result of each command is either OK or ABORT.

Evaluation

- Implemented as a LLVM pass.
  - compile to LLVM IR → optimization → SoftBound → optimization → ...
- Benchmark programs with varying proportion of memory operations and 2 network server applications (total 272kloc)
- Various security-related benchmarks.

Evaluation

- Spatial safety: Detected all the spatial violations and prevents all the security vulnerabilities security violation benchmark and real-world spatial bugs without any false positives.
  - Other tools miss >50% of the test cases.
  - Store-only mode (only instrument stores) is enough to prevent security vulnerability.
- Compatibility: Didn't require source code modification for benchmark programs and 2 network server applications (total 272 kloc)

Performance evaluation

	Hash table	Shadow space
Full	● runtime +93% ● memory +87%	● runtime +67% ● memory +64%
Store-only	● runtime +54%	● **runtime +22%**

Better than or similar to other tools!

The End?

No.

Implementation considerations and limitations

- Linking against un-instrumented library requires wrappers for the library functions.
  - Much better than pointer-based approaches, though.
- Handling `memcpy()` on objects containing pointers requires identification of all of the internal pointers.
- Handling integer-pointer cast requires manual annotation.
- Narrowing results in false violations!
  - Intrusive data structures: acquires pointer the the container struct from the pointer to the field.
  - Consecutive struct fields as an array
- Pointers stored in memory are susceptible to sub-object overflow.
  - Metadata table is not aware of narrowing.

Wrap-up

Summary

SoftBound = Spatial memory safety with

- Completeness of pointer-based approach
- Compatibility of object-based approach
- Good performance

Related

- Hardbound: Architectural Support for Spatial Safety of the C Programming Language (ASPLOS'08)
  - bounded pointers using hardware-managed shadow space
  - inspired SoftBound
- CETS: Compiler Enforced Temporal Safety for C (ISMM'10)
  - another work from the authors of SoftBound
- CCured: Type-Safe Retrofitting of Legacy Software (TOPLAS'05)
  - Pointer-based approach
  - Uses a novel type system to identify pointers that don't require bounds checks
    - Low runtime overhead, but requires non-trivial code modification..
  - to be presented by Jaemin

The paper contains ...

- Proof: definition of well-formedness
- More implementation issues
- In-depth comparison vs. CCured
- In-depth performance analysis