Complete Memory-Safety with SoftBoundCETS

Slides courtesy of
Prof. Santosh Nagarakatte

Project goal:
Make C/C++ safe and secure

Why?

Lack of ***memory safety*** is the root cause of serious **bugs** and **security vulnerabilities**

Security Vulnerabilities due to Lack of Memory Safety

Adobe Acrobat – **buffer overflow**

CVE-2013-1376- *Severity: 10.0 (High)*

January 30, 2014

Oracle MySQL – **buffer overflow**

CVE-2014-0001 - *Severity: 7.5 (High)*

January 31, 2014

Firefox – **use-after-free vulnerability**

CVE-2014-1486 - *Severity: 10.0 (High)*

February 6, 2014

Google Chrome– **use-after-free vulnerability**

CVE-2013-6649 - *Severity: 7.5 (High)*

January 28, 2014

DHS/NIST National Vulnerability Database:

- Last three months: **92 buffer overflow and 23 use-after-free disclosures**
- Last three years: **1135 buffer overflows and 425 use-after-free disclosures**

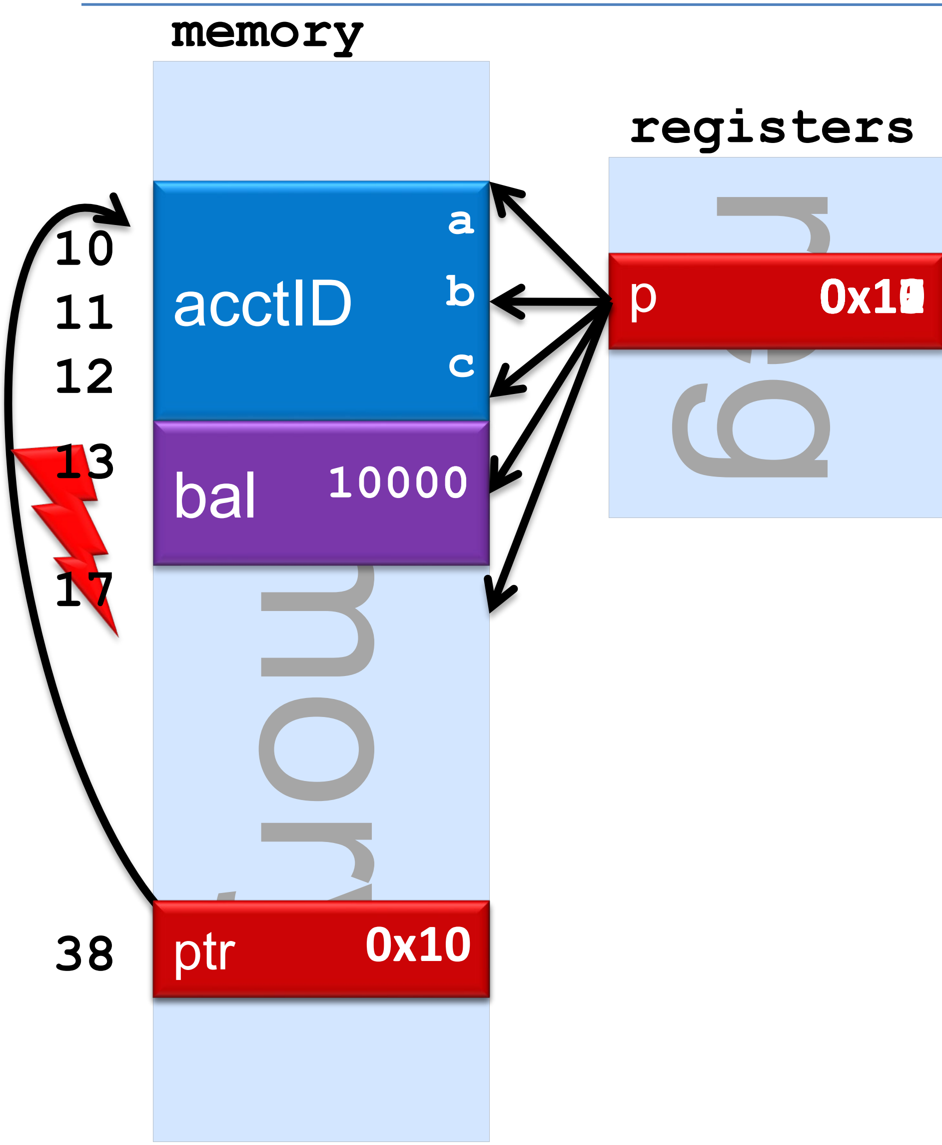
Photo Credit: Roger Halbheer

Nobody Writes New C Code, Right?

- More than a million new C-based applications!
  - Over last few years, publically available. Evidence?

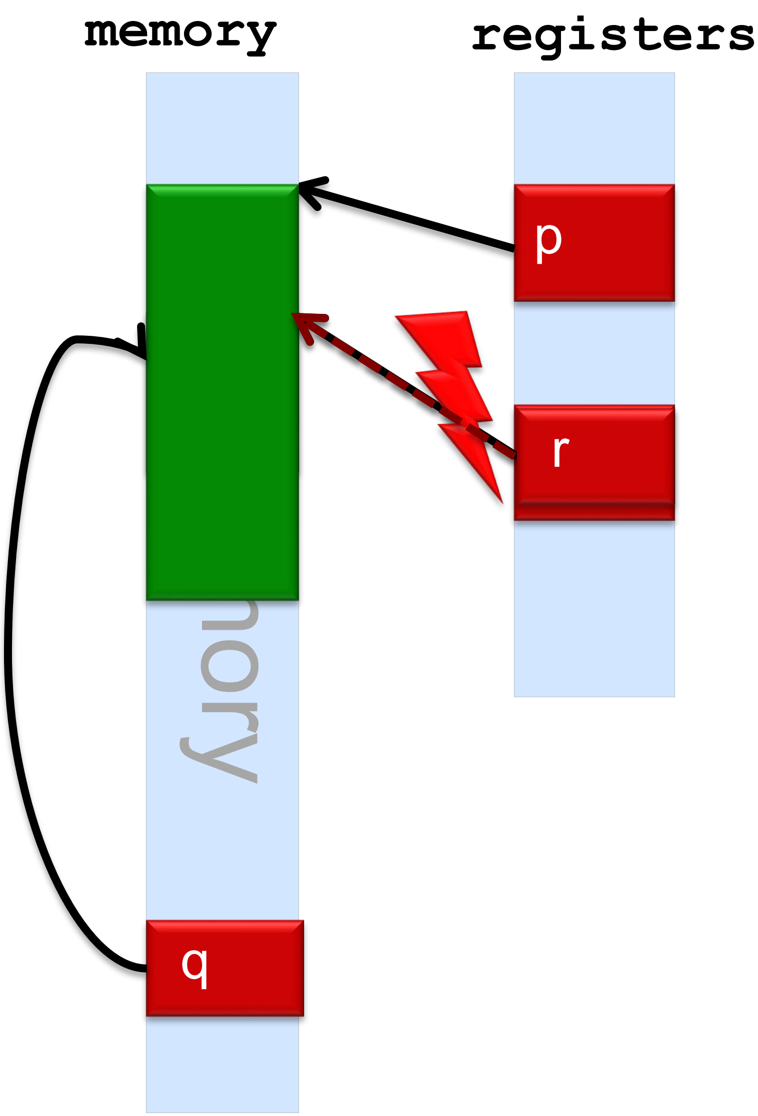
Background on Enforcing Memory Safety

Bounds Violation Example




```
struct BankAccount {
    char acctID[3];  int balance;
} b;
b.balance = 0;
char* ptr = &(b.acctID);
…
…
char* p = ptr;
…
…
do {
    char ch = readchar();
    *p = ch;
    p++;
} while(ch);
```

Dangling Pointer Example



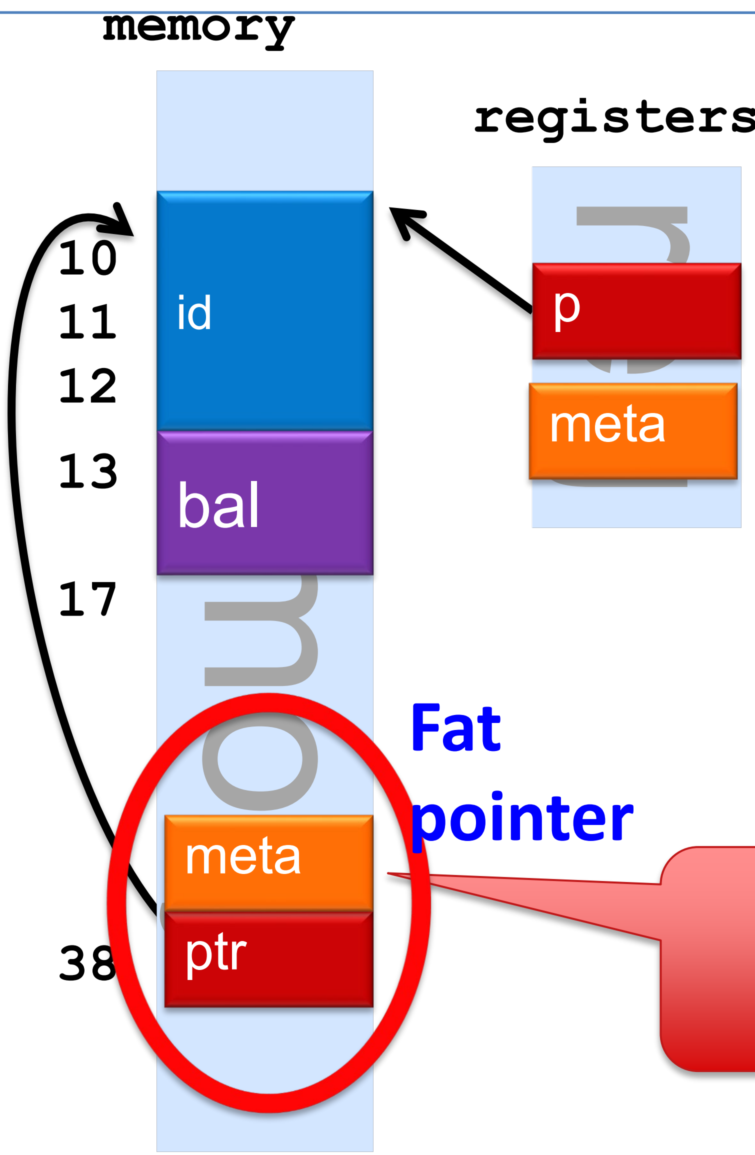

```
struct BankAcct  *p, *q, *r;
…
q = malloc(sizeof(BankAcct));
…
r = q;
…
free(q);
…
p = malloc(10*sizeof(BankAcct));
….
*r = …..
```

What is Void * C?

```
int foo (void * c);
```

Abstractions Not Enforced!

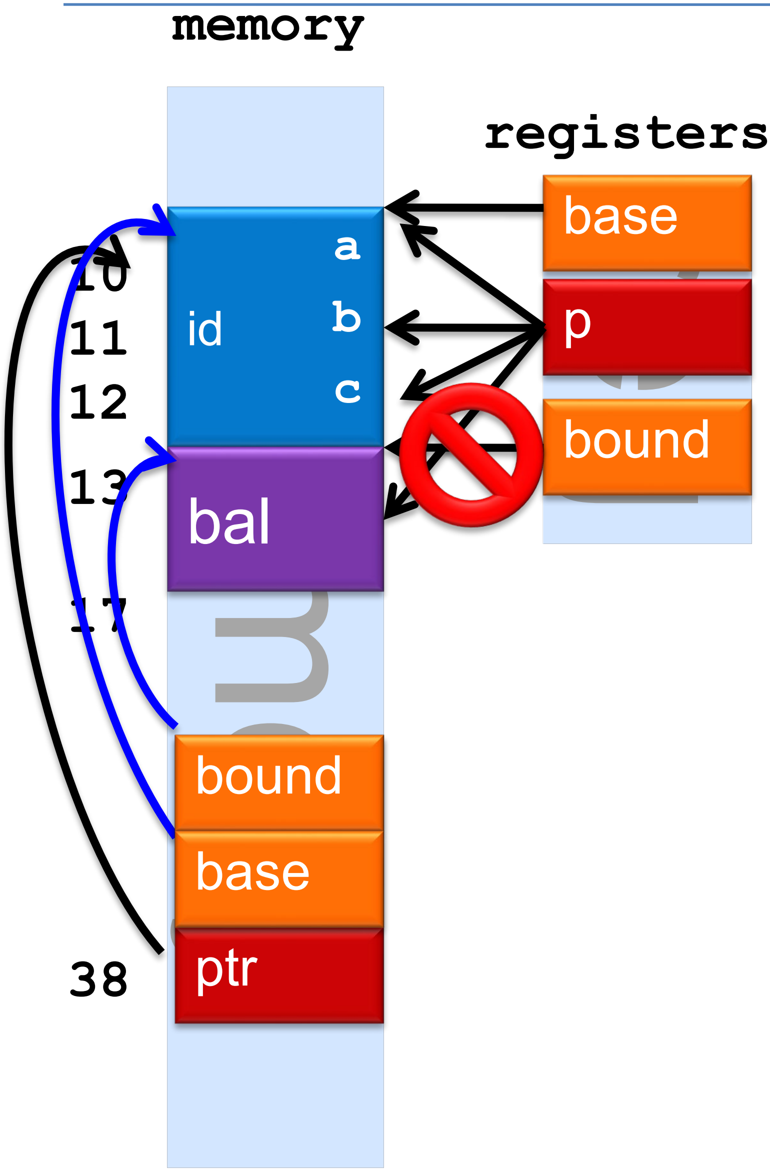
Pointer Based Checking



- Ccured, MSCC, P&F, SafeC
- Maintain metadata with pointers
- Each pointer has a “view of memory it can access”

Pointer based metadata for both registers and memory

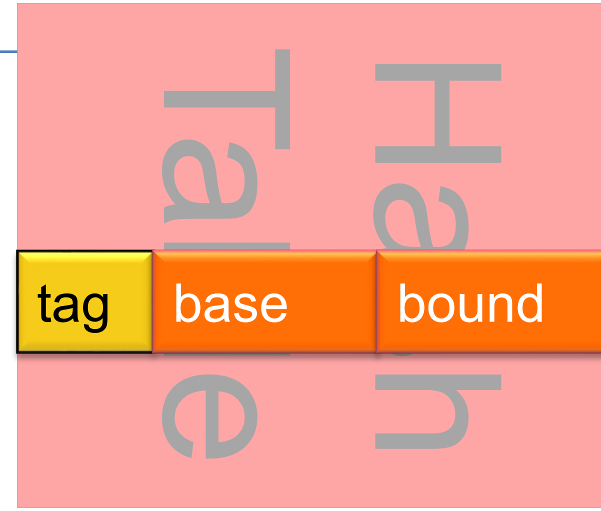
Pointer Based Checking: Spatial Safety

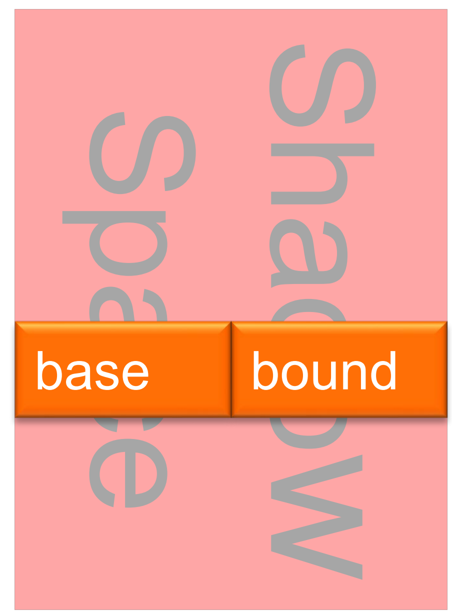



```
struct BankAccount {
    char acctID[3];  int balance;
} b;
b.balance = 0;
char* ptr = &(b.acctID);
…
…
char* p = ptr;
…
do {
    char ch = readchar();
    *p = ch;
    p++;
} while(ch);
```

SoftBound Base/Bound Storage

- **Registers**
- For memory: **hash table**
  - Tagged, open hashing
  - Fast hash function (bitmask)
  - Nine x86 instructions
    - Shift, mask, multiply, add, three loads, cmp, branch
- Alternative: **shadow space**
  - No collisions ➔ eliminates tag
  - Reduce memory footprint
  - Five x86 instructions
    - Shift, mask, add, two loads





Pointer Dereference Checks

- All pointer dereferences are checked

```
if (p < p_base) abort();
if (p + size > p_bound) abort();
value = *p;
```

- Five x86 instructions (cmp, br, add, cmp, br)
- Bounds check elimination not focus
  - Intra-procedural dominator based
  - Previous techniques would help a lot

Pointer Creation

Heap Objects

```
p = malloc(size);
p_base = p;
p_bound = p + size;
```

Stack and Global Objects

```
int array[100];
p = &array;
p_base = p;
p_bound = p + sizeof(array);
```

Base/Bound Metadata Propagation

- Pointer assignments and casts
  - Just propagate pointer base and bound
- Loading/storing a pointer from memory
  - Loads/stores base and bound from metadata space
- Pointer arguments to a function
  - Bounds passed as extra arguments (in registers)

```
int f(char* p) {…}
```

```
int  _f(char* p, void* p_base, void* p_bound)  {…}
```

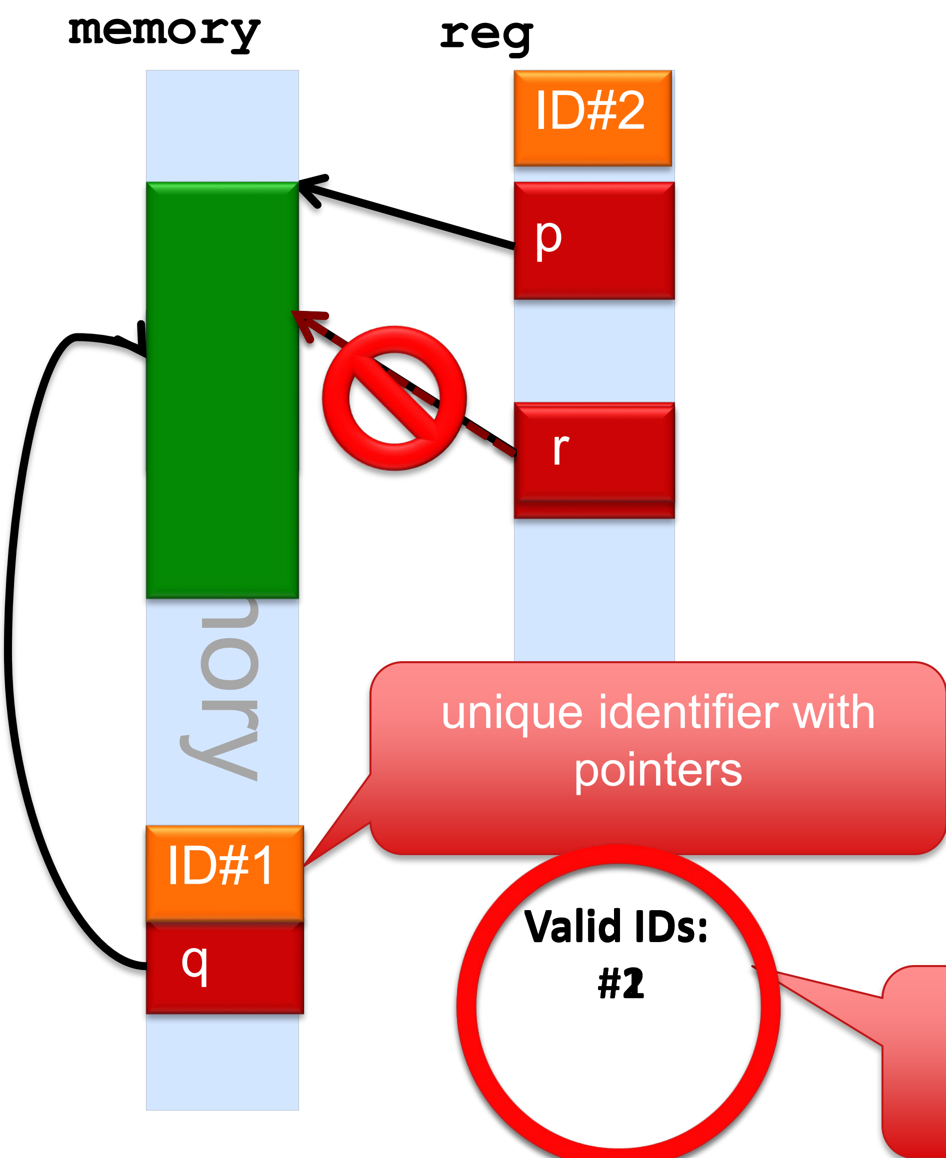
Pointers to Structure Fields

```
struct {
    char acctID[3];  int balance;
 } *ptr;
char* id = &(ptr->acctID);
```

option #1	option #2
<u>Entire Structure</u>	<u>Shrink to Field Only</u>
id_base = ptr_base; id_bound = ptr_bound;	id_base = &(ptr->acctID); id_bound = &(ptr->acctID) + 3;

Programmer intent ambiguous;
optional shrinking of bounds

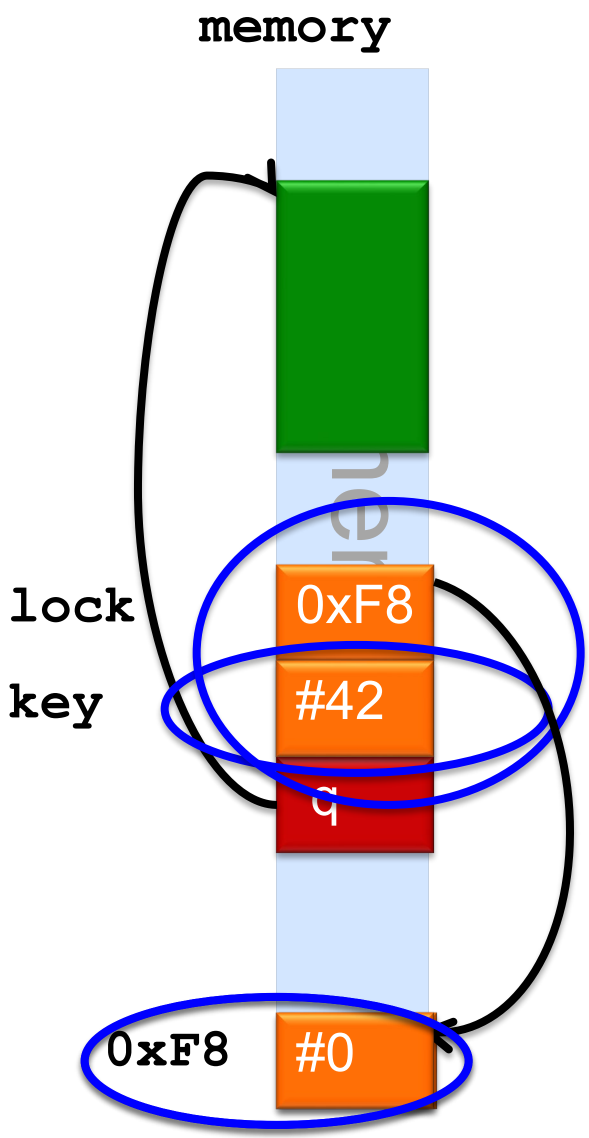
Pointer Based Checking: Temporal Safety




```
struct foo *q, *r;
struct bar  *p;
...
q = malloc(sizeof(struct foo));
...
r = q;
...
free(q);
...
p = malloc(sizeof(struct bar));
....
*r = .....
```

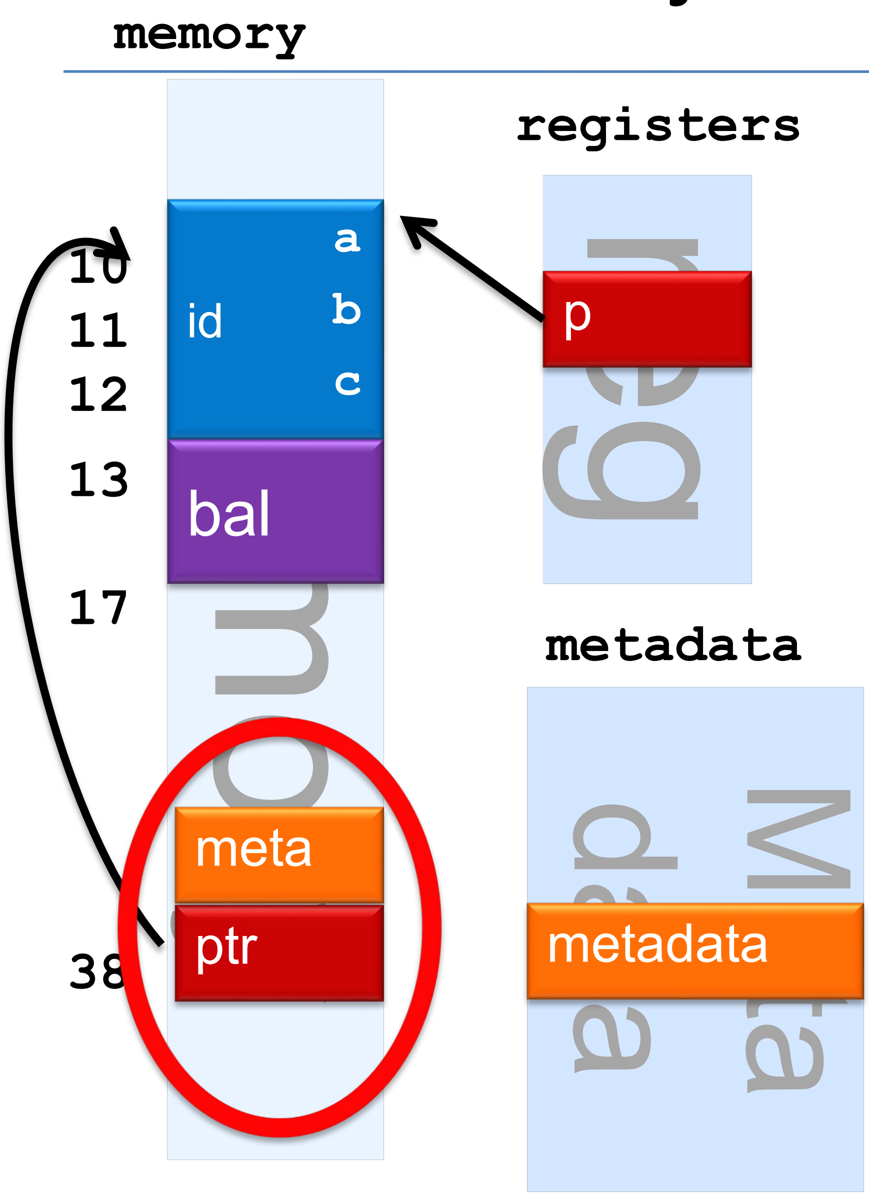
Maintain the set of valid identifiers

Pointer Based Checking: Lock & Key



- Split identifier
  - Lock & Key
- Invariant: valid if
  memory[lock] == ptr.key
- Allocation
  memory[lock] = key
- Check: exception if
  memory[lock] != key
- Deallocation
  memory[lock] = 0

Disjoint Metadata



- **Memory layout changed ➔ library compatibility lost**
- **Arbitrary type casts ➔ comprehensiveness lost**

Real World ‘C’ with Disjoint Metadata

- Key issue: type casts

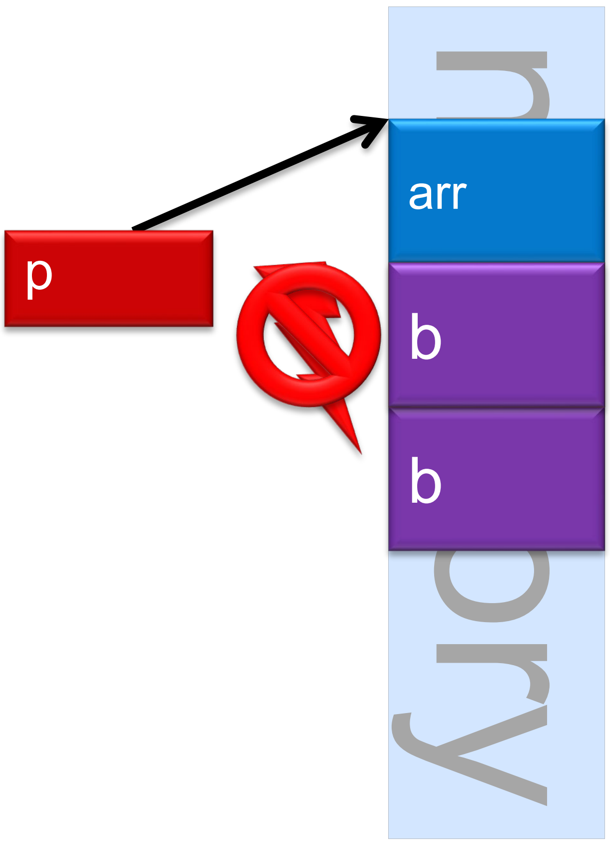
Disallow casts??

Insight: casts can only manufacture pointers but not metadata

```
struct  foo{
   int* arr;
   size_t b;
} ;
  struct foo *p;
  struct bar *q;
```

```
struct bar{
   size_t x;
   size_t y;
};
```

```
...
q = (struct bar *) p;
...
*q = …
```



Accesses to Disjoint Metadata Space

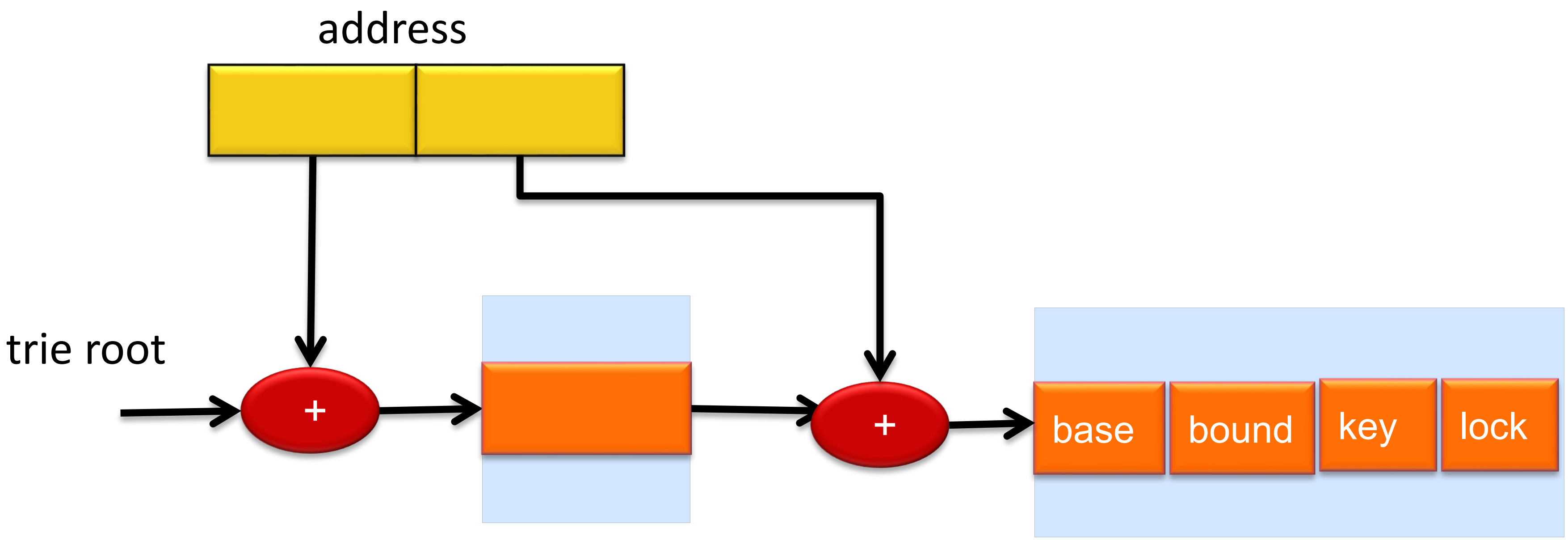
```
int *p;
int **q;
...
p_meta = load_meta(q);
p = *q;
```

Metadata accesses using address of the pointer than what pointer points to

How Do We Organize the Metadata Space?

- Shadow entire virtual address space
  - Allocate entries on demand
  - 32 bytes metadata for every word
  - 12 x86 instructions
    - (6 loads/stores, 2 adds, 2 shift, mov and mask)

Translation using a trie, a page table like structure



Performance Design Choice

Disjoint metadata accesses are expensive

Metadata with non-pointers→ Performance overhead

- Design choice: Metadata only with pointers
  - Programs primarily manipulate data
  - Metadata propagation on only pointer operations

- Type casts between pointers is allowed

- Casting an integer to a pointer is disallowed
  - Pointer obtains NULL/Invalid metadata
  - Dereferencing such a pointer would raise exception

Pointer Metadata Allocation/Propagation

Memory allocation

```
p = malloc(size);
  p_base = p;
  p_bound = p + size;
  p_key = allocate_key():
  p_lock = allocate_lock();
```

Memory deallocation

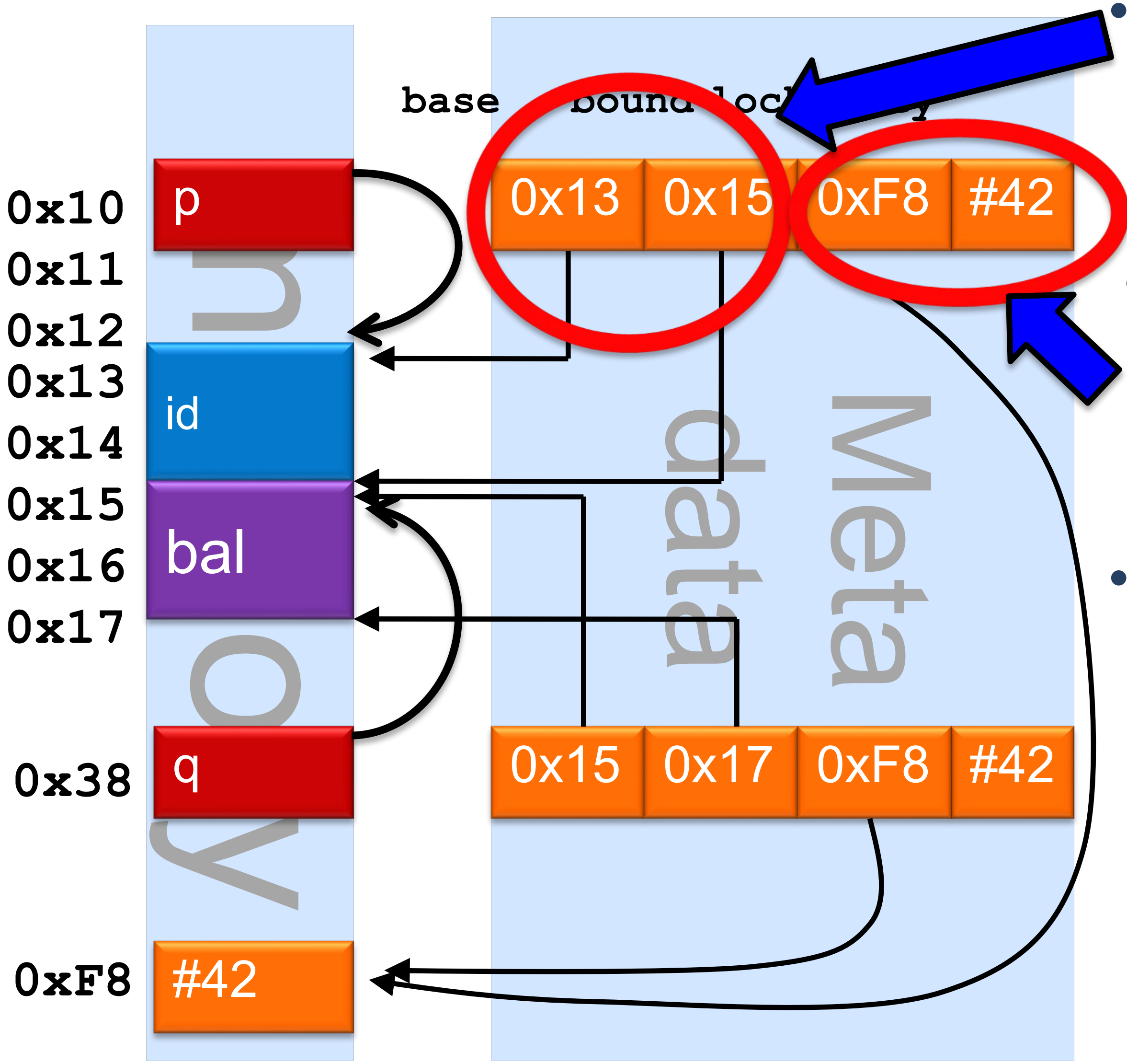
```
check_double_frees();
 free(p);
*(p_lock) = INVALID_KEY;
deallocate_lock(p_lock);
```

Pointer arithmetic/copies

```
 p = q + 10;
p_base = q_base;
p_bound = q_bound;
p_key  = q_key;
p_lock = q_lock;
```

Summary: Pointer Based Disjoint Metadata

memory disjoint metadata



- **Bounds Check**
  - **Easy once you have "base" & "bound"**
- **Temporal Check**

  Check if

  key = mem[lock]
- **Disjoint shadow space**
  - **Memory layout intact**
  - **Protects metadata**
  - **Allocated on-demand**
  - **But, hurts locality**

Where to Perform Pointer-Based Checking?

- Source-to-source translation
  - Pointers are readily available
  - Added code confuses the optimizer
- Compiler instrumentation
  - Pointers need to be optimized
  - Can operate on optimized code
- Binary instrumentation
  - Pointer identification is hard
  - Extra code translates into overhead
- Hardware injection
  - Pointers identifications is hard
  - Streamlined injection necessary

Compiler instrumentation provides best of both

Hardware injection can streamline the extra code added

SoftBoundCETS Compiler Instrumentation

- **Goal: reduce performance overheads**
  - How to identify pointers?
  - How to propagate metadata across function calls?
  - How to perform instrumentation?

- **Approach: perform instrumentation over LLVM IR**

Background on LLVM IR – C Code

```
struct node_t {
  size_t value;
  struct node_t* next;
};
typedef struct node_t node;

int main(){
 node* fptr = malloc(sizeof(node));
 node* ptr = fptr;
 fptr -> value = 0;
 fptr -> next = NULL;

 for (i= 0; i < 128 ; i++){
  node* new_ptr = malloc(sizeof(node));
  new_ptr->value = I;
  new_ptr->next = ptr;
  ptr  = new_ptr;
 }
 fptr->next = ptr;
}
```

Pointer store

Background on LLVM IR

```
%node_t = type {i64, node_t*};

define i32 @main(i32 %argc, i8** argv){
entry:
  %call = call i8* malloc(i64 16)
  %0 = bitcast i8* %cal to %node_
  %value = gep %node_t* %0, i32 0, i32 0
  store i64 0, i64* %value
  %next = gep %node_t* %0, i32 0, i32 1
  store %node_t* null, %node_t** next
  br label %for.cond

for.cond:
  %ptr.0 = phi %node_t* [%0, %entry], [%1, %for.inc]
  %i.0 = phi i64 [0, %entry], [%inc, %for.inc]
  %cmp = icmp ult i64 %i.0, 128
  br i1 %cmp, label %for.body, label %for.end
```

Explicitly typed

Pointer arithmetic using gep

IR is in SSA
phi nodes merge values from predecessors

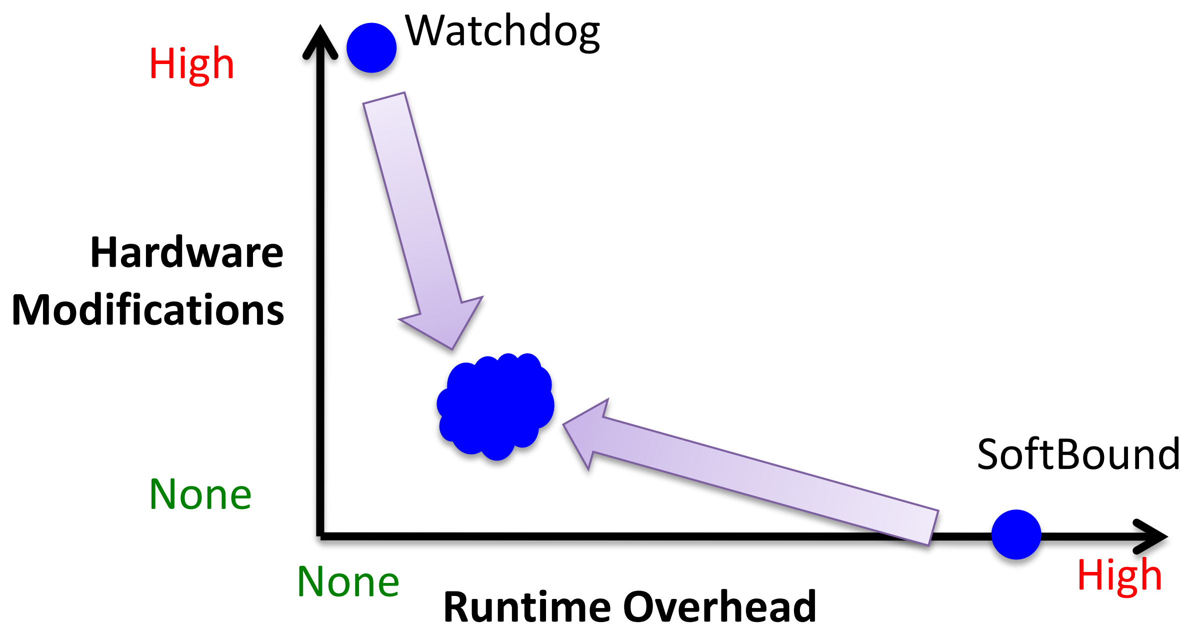
How Do We Instrument IR Code?

- Introduce calls to C functions
  - Checks, metadata accesses all written in C code

- SoftBoundCETS Instrumentation Algorithm
  - Operates in three passes
  - First pass introduces temporaries for metadata
  - Second pass populates the phi nodes
  - Third pass introduces calls to check handlers

Simple linear passes over the code, enabled us extract an implementation from the proofs

Exploring the Hardware/Software Continuum



Compiler does pointer identification and metadata propagation and hardware accelerates checks

Hardware vs Software Implementation

Task	Watchdog [ISCA 2012]	SoftBoundCETS [PLDI 2009, ISMM 2010]
Pointer detection	Conservative	Accurate with compiler

Hardware vs Software Implementation

Task	Watchdog [ISCA 2012]	SoftBoundCETS [PLDI 2009, ISMM 2010]
Pointer detection	Conservative	Accurate with compiler
Op Insertion	Micro-op injection	Compiler inserted instructions

Hardware vs Software Implementation

Task	Watchdog [ISCA 2012]	SoftBoundCETS [PLDI 2009, ISMM 2010]
Pointer detection	Conservative	Accurate with compiler
Op Insertion	Micro-op injection	Compiler inserted instructions
Metadata Propagation	Copy elimination using register renaming	Standard dataflow analysis

Hardware vs Software Implementation

Compiler can do these tasks efficiently

Hardware can accelerate checks & metadata accesses

Task	Watchdog [ISCA 2012]	So… [PLDI 2009, ISMM 2010]
Pointer detection	Conservative	Accurate with compiler
Op Ins…	…jection	Compiler inserted instructions
Metad… Propagation	…ation using register renaming	Standard dataflow analysis
Checks	+ fast checks (implicit) - no check optimization	- Instruction overhead + Check optimization
Metadata Loads/Stores	+ Fast lookups	- Instruction overhead

Hardware Support

Hardware acceleration with new instructions for compiler based pointer checking

Instructions added to the ISA

- Bounds check & use-after-free check instructions
- Metadata load/store instructions

Pack four words of metadata into a single wide register

- Single wide load/store → eliminates port pressure
- Avoid implicit registers for the new instructions
- Reduces spills/restores due to register pressure

Spatial (Bound) Check Instruction

int p;

…

~~if(q < q_base ||~~

~~q + sizeof(int) >= q_bound){~~

~~abort();~~

}

p = *q;

5 instructions for the spatial check

Schk.size imm(r1), ymm0

- Supports all addressing modes
- Size of the access encoded
- Operates only on registers
- Executes as one micro-op
- Latency is not critical

Temporal (Use-After-Free) Check Instruction

```
int p;
 …
~~if( q_key!= *q_lock){
   abort();
}~~
```

```
p = *q;
```

3 instructions for the temporal check

Tchk ymm0

Performs a memory access

Executes as two micro-ops

Latency is not critical

Metadata Load/Store Instructions

```
int *p, **q;
...
```

~~p_metadata = table_lookup(q);~~

Metaload %ymm0, imm(%rax)

```
p = *q;
..
```

~~table_lookup(q) = p_metadata~~

Metastore imm(%rax), %ymm0

```
*q = p
```

14 instructions for the metadata load

16 instructions for the metadata store

Performs a wide load/store

Executes as two micro-ops

– address computation

-- wide load/store uop

Shadow space for the metadata

See Papers For ….

- Compiler transformation to use wide metadata
- Metadata organization
- Check elimination effectiveness
- Effectiveness in detecting errors
- Narrow mode instructions
- Comparison of related work

Evaluation

- Three questions
  - Effective in detecting errors?
  - Compatible with existing C code?
  - Reasonable overheads?

Memory Safety Violation Detection

- Effective in detecting errors?
  - NIST Juliet Suite – 50K memory safety errors
  - Synthetic attacks [Wilander et al]
  - Bugbench [Lu05]: overflows from real applications

Benchmark	SoftBoundCETS	Mudflap	Valgrind
Go	Yes	No	No
Compress	Yes	Yes	Yes
Polymorph	Yes	Yes	No
Gzip	Yes	Yes	Yes

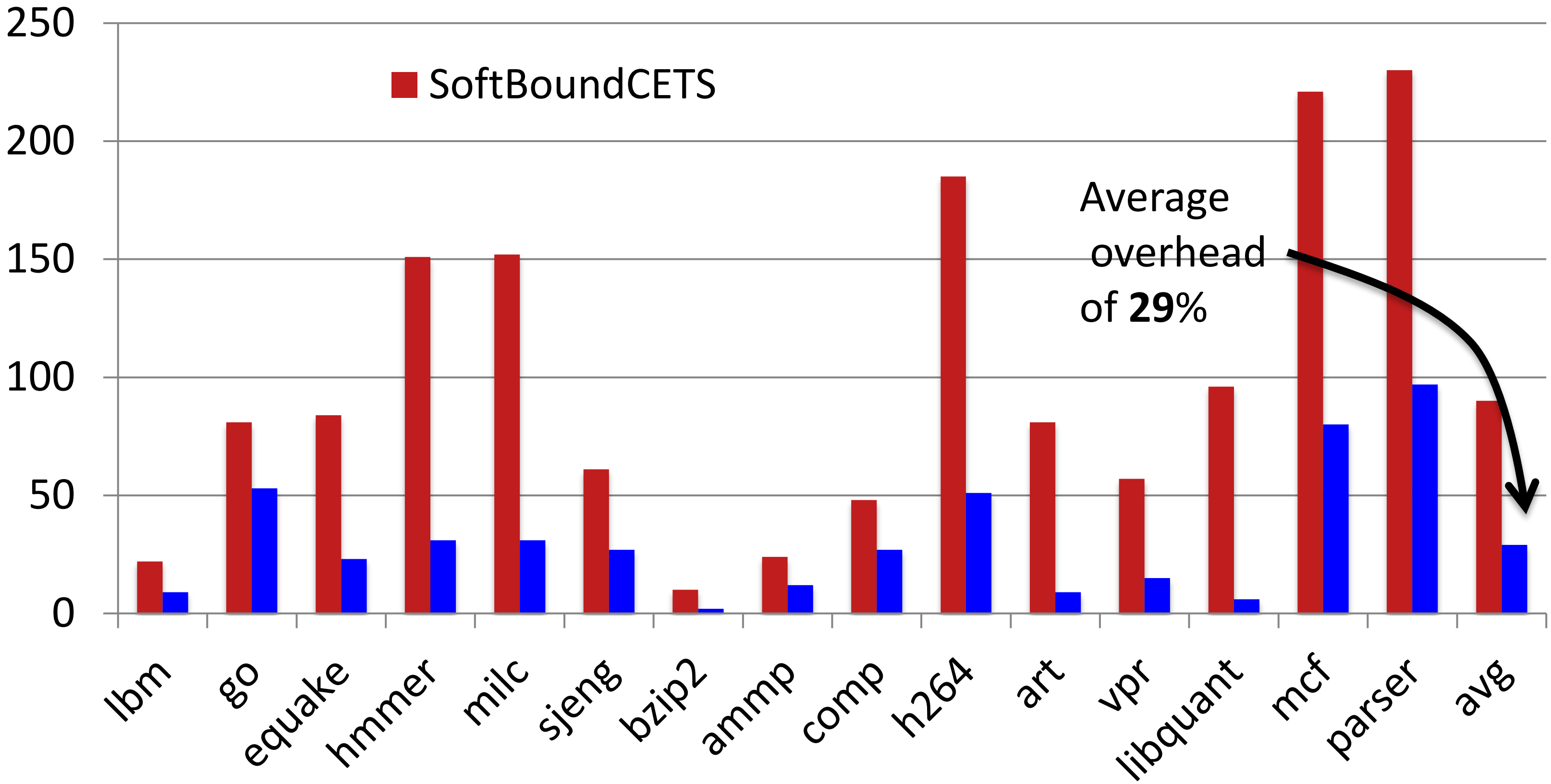
- Found unknown new bugs
  - H.264, Parser, Twolf , Em3d, Go, Nullhttpd, Wu-ftpd, ..

Source Compatibility Experiments

- Compatible with existing C code?

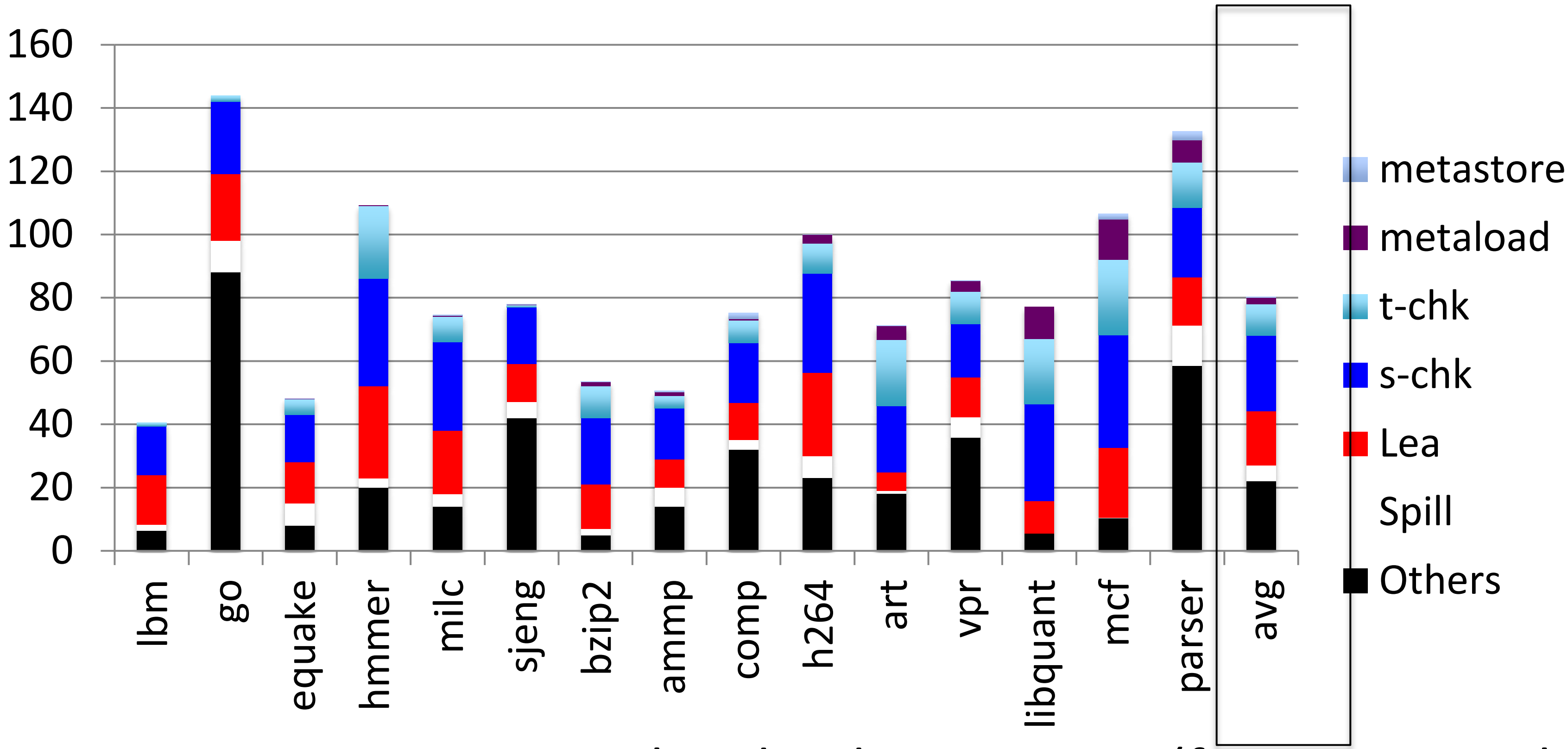
- Approximately one million lines of code total
  - 35 benchmarks from Spec, Olden
  - BugBench, GNU core utils, Tar, Flex, …
  - Multithreaded HTTP Server with CGI support
  - FTP server
- Separate compilation supported
  - Creation of safe libraries possible

Evaluation – Performance Overheads



- Timing simulations of wide-issue out-of-order x86 core
- Average performance overhead: **29%**
  - Reduces average from 90% with SoftBoundCETS

Remaining Instruction Overhead



- Average instruction overhead reduces to 81% (from 180% with SoftBoundCETS)
- Spatial checks → better check optimizations can help
- Lea instructions → change code generator

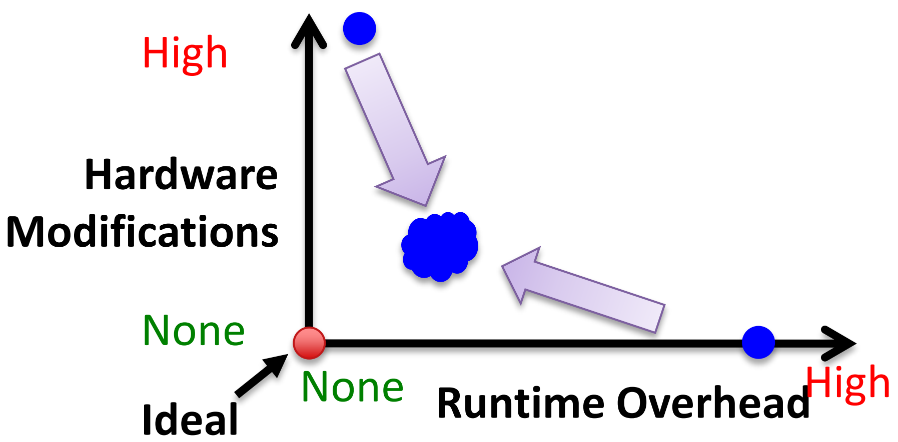
Intel MPX

- In July 2013, Intel MPX announced ISA specification
  - Similar hardware/software approach
    - Pointer-based checking: base and bounds metadata
    - Disjoint metadata in shadow space
    - Adds new instructions for bounds checking
  - Differences
    - Adds new bounds registers vs reusing existing AVX registers
    - Changes calling conventions to avoid shadow stack
    - Backward compatibility features
      - Interoperability with un-instrumented and instrumented code
      - Validates metadata by redundantly encoding pointer in metadata
      - Calling un-instrumented code clears bounds registers
    - Does not perform use-after-free checking

Conclusion

- Safety against buffer overflows & use-after-free errors
  - Pointer based checking
  - Bounds and identifier metadata
  - Disjoint metadata
- SoftBoundCETS with hardware instructions
  - Four new instructions for compiler-based pointer checking
  - Four new instructions
  - Packs the metadata in wide registers

Leveraging the compiler enables our proposal to use simpler hardware for comprehensive memory safety



Thank You

Try SoftBoundCETS for LLVM-3.4

http://github.com/santoshn/softboundcets-34/