SoftBound: Highly Compatible and Complete Spatial Safety for C

Santosh Nagarakatte, Jianzhou Zhao,
Milo Martin, Steve Zdancewic

University of Pennsylvania

{santoshn, jianzhou, milom, stevez}@cis.upenn.edu

University of Pennsylvania
Architecture + Compilers Group

Who Cares About Spatial Safety, Anyway?

June 2, 2009: iTunes-8.2
Open URL, stack overflow

May 12, 2009: libxml, Safari-3.2.3,
Visit website, heap overflow

Feb 20, 2009: Acrobat Reader
Open PDF, overflow

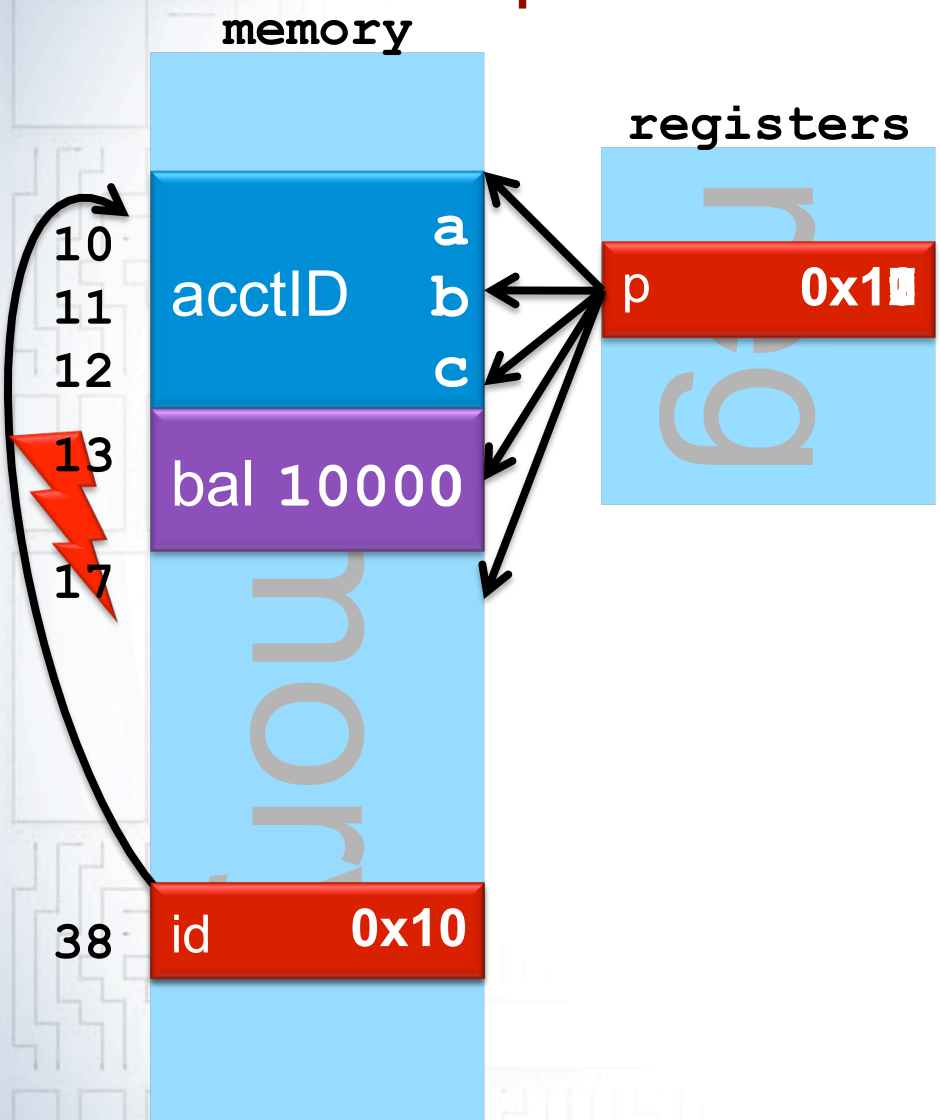
Jan 22, 2009: Windows,
RPC packet, overflow (Conficker worm)

These buffer overflows are security vulnerabilities

SoftBound: Spatial Safety for C

- Compiler transformation to enforce spatial safety
  - Inspired by fat pointer schemes
- **Compatible** – no source code modifications
  - Key: disjoint fat pointers ➔ memory layout unchanged
- **Simple analysis** – intra-procedural
  - Separate compilation, creation of safe libraries
- **Effective** – observed no false positives/negatives
- **Low overhead**
  - All loads and stores – 67% overhead
  - Only stores – 21% overhead

Spatial Violation Example




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

Preventing Spatial Violations

treat the symptoms

address space randomization

non-executable stack/heap

protect return addresses

use static analysis

use safe language

write in Java / C#

add bounds checking to C

what about false positives?

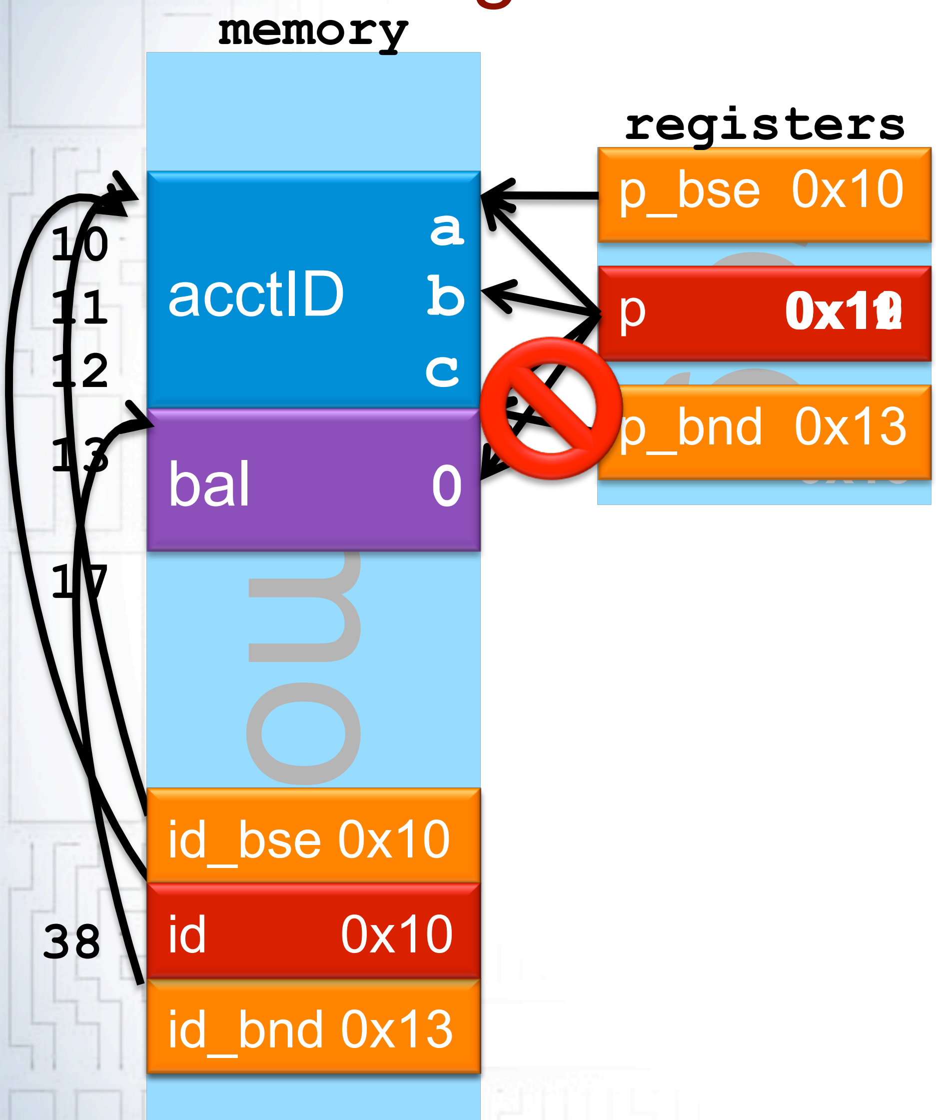
what about complete

all in (existing) C code?

Background: Bounds Checking for C

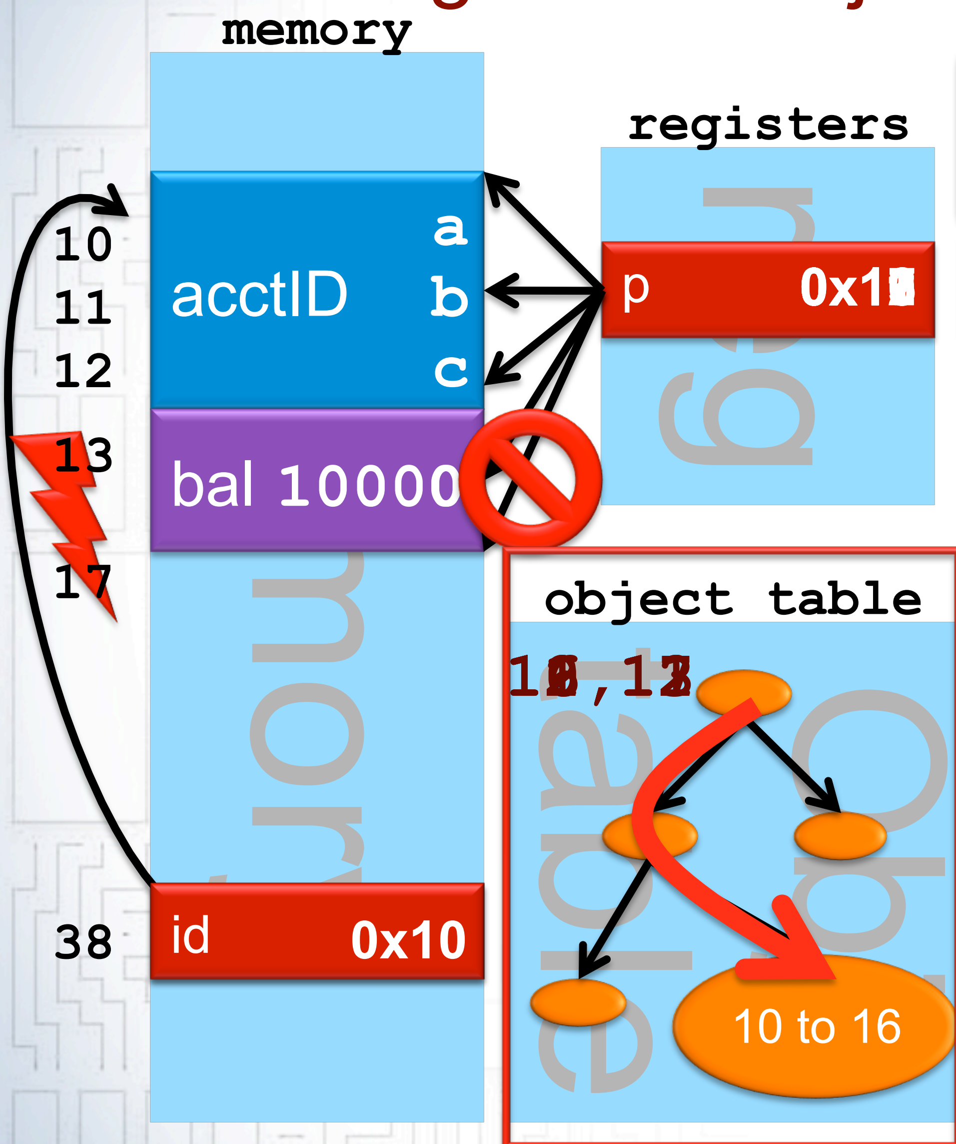
- **Tripwires** e.g., Purify, Valgrind …
  - Few bits of state for each byte in memory
  - A "red-zone" block between objects
- **Pointer based** e.g., SafeC, Cyclone, CCured, MSCC, …
  - Pointer becomes a fat pointer (ptr, base, bound)
  - Pointer dereferences are checked
- **Object based** e.g., Jones & Kelly, CRED, SafeCode, SVA, …
  - Checks pointer manipulations
  - Must point within same object
- **All have one or more challenges:**
  - **High runtime overheads**
  - **Incompleteness, handling arbitrary casts**
  - **Incompatible pointer representations, code incompatibilities**

Background: Fat Pointer Approach




```
struct BankAccount {
   char acctID[3];  int balance;
} b;
b.balance = 0;
char* id = &(b.acctID);
char* id_bse = &(b.acctID);
char* id_bnd = &(b.acctID)  + 3;
char* p = id;
char* p_bse = id_bse;
char* p_bnd = id_bnd;
do {
   char ch = readchar();
   check(p, p_bse, p_bnd);*p = ch;
   p++;
} while(ch);
```

Background: Object Based Approach

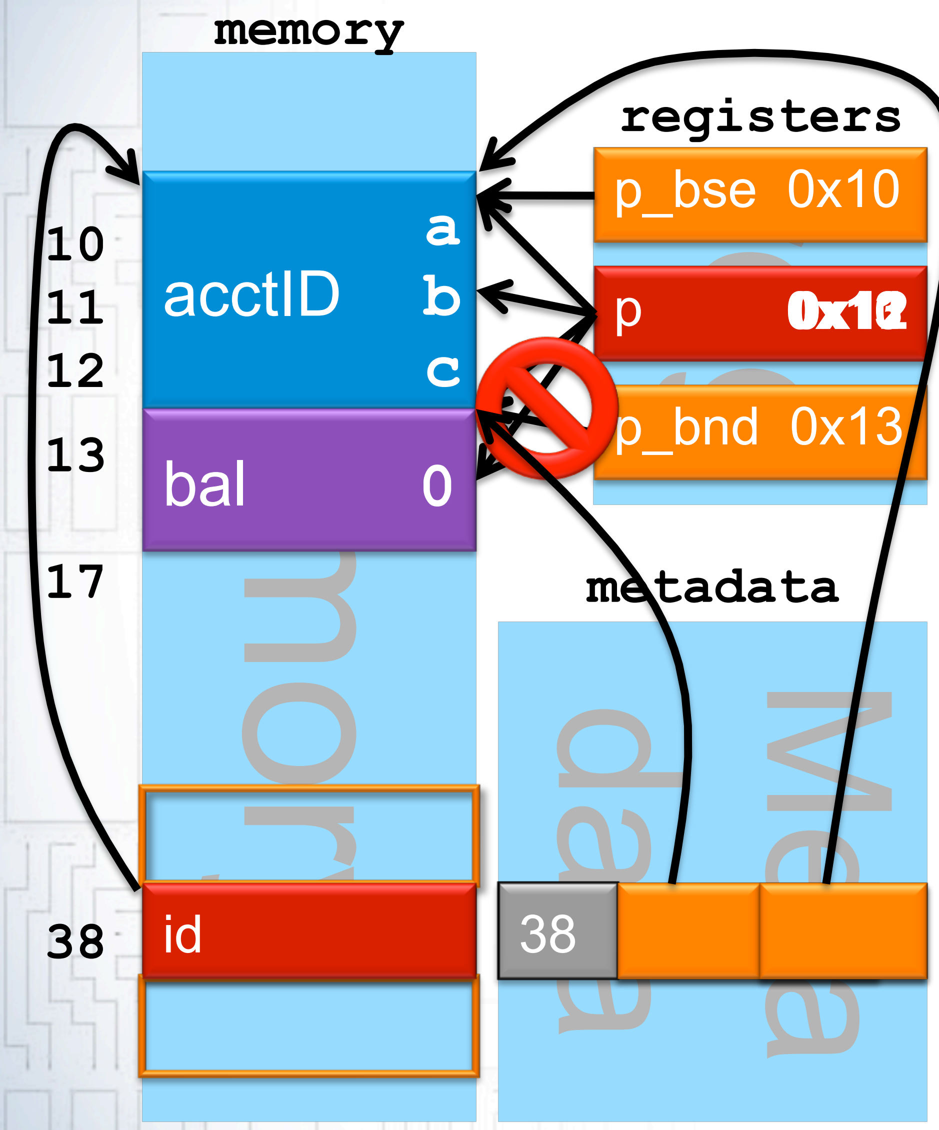



```
struct BankAccount {
    char acctID[3];  int balance;
} b;
insert(b, &b, &b+sizeof(b));
b.balance = 0;
char* id = &(b.acctID);
…
char* p = id;
…
…
do {
    char ch = readchar();
    *p = ch;
    p++; p = lookup(p, p + 1);
} while(ch);
```

Comparison of Approaches

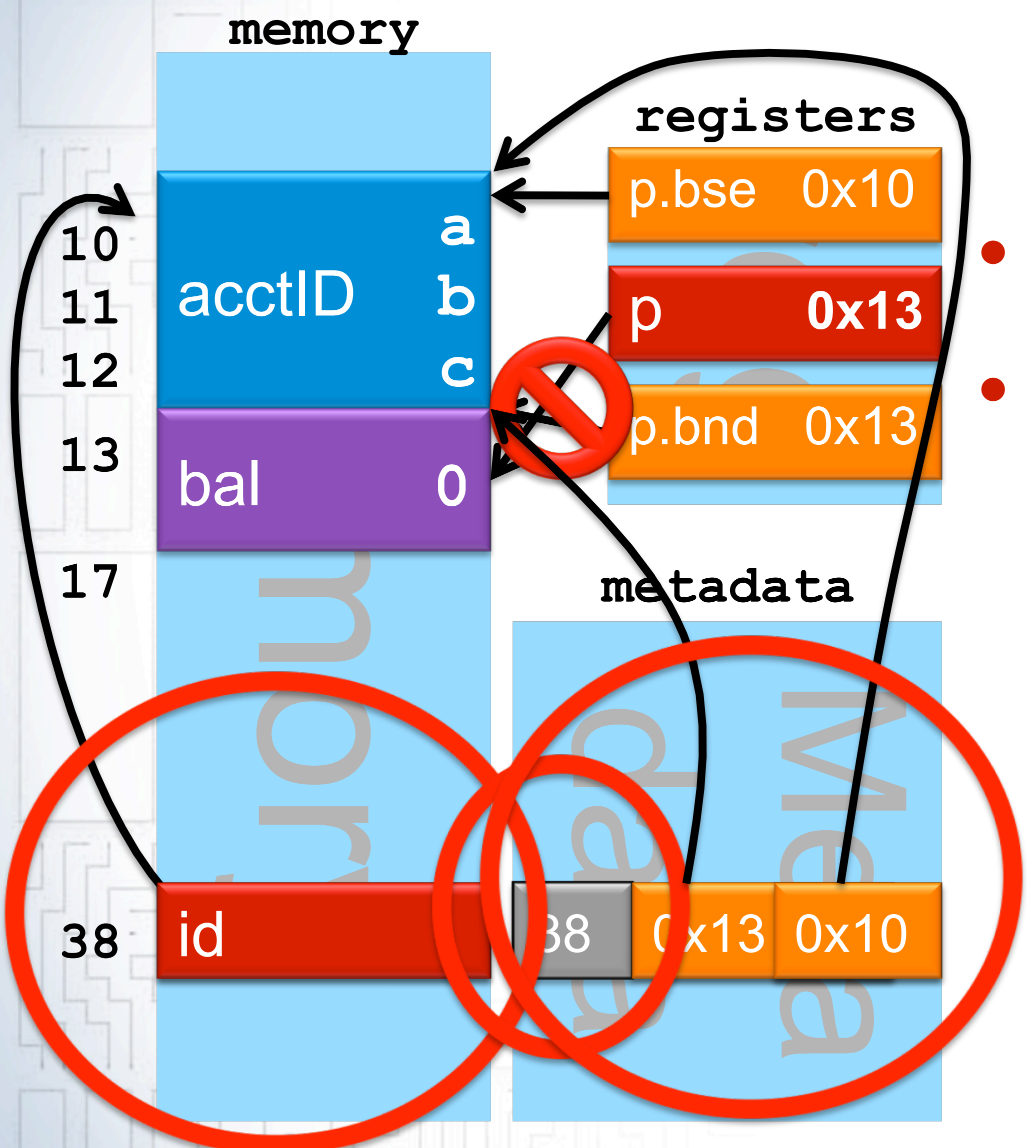
- **Object based**
  - + **Disjoint metadata** ➔ memory layout unchanged
    ➔ high source compatibility
  - – Cannot detect sub-object overflows
  - – Range lookup overhead
- **Fat pointers**
  - + Can detect sub-objects overflows
  - – **Inline metadata** ➔ memory layout changes
    ➔ low source compatibility
- **Both**
  - – Fail to protect against arbitrary casts
    (unless augmented, such as CCured's WILD pointers)

SoftBound Approach




```
struct BankAccount {
   char acctID[3];  int balance;
} b;
b.balance = 0;
char* id = &(b.acctID);
lookup(&id)->bse = &(b.acctID);
lookup(&id)->bnd = &(b.acctID) + 3;
char* p = id;
char* p_bse = lookup(&id)->bse;
char* p_bnd = lookup(&id)->bnd;
do {
   char ch = readchar();
   check(p, p_bse, p_bnd);*p=ch;
   p++;
} while(ch);
```

SoftBound Approach



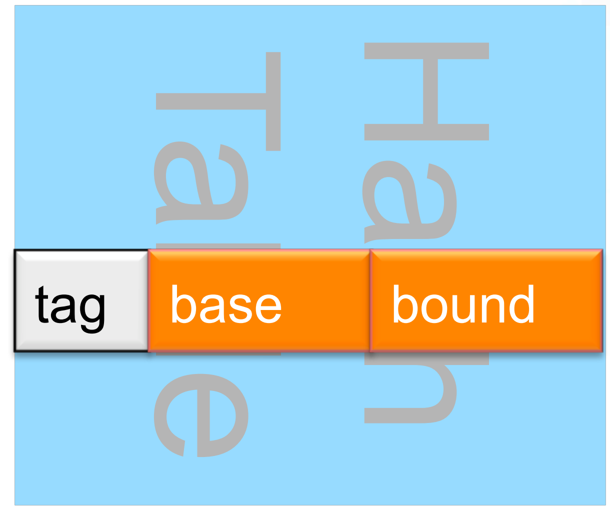
- Pointer based
- Disjoint metadata
  - Unchanged memory layout
  - Safe with arbitrary casts

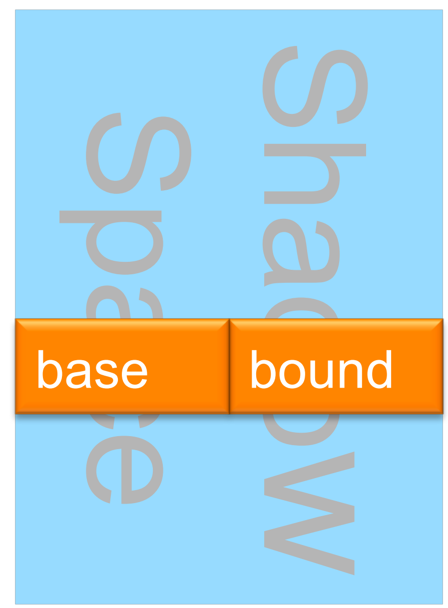
Rest of Talk

- SoftBound handling of base/bound metadata…
  - … Storage
  - … Checking on pointer dereference
  - … Creation
  - … Propagation
- SoftBound prototype
- Experiments

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

Entire Structure

```
id_base = ptr_base;
id_bound = ptr_bound;
```

option #2

Shrink to Field Only

```
id_base = &(ptr->acctID);
id_bound = &(ptr->acctID) + 3;
```

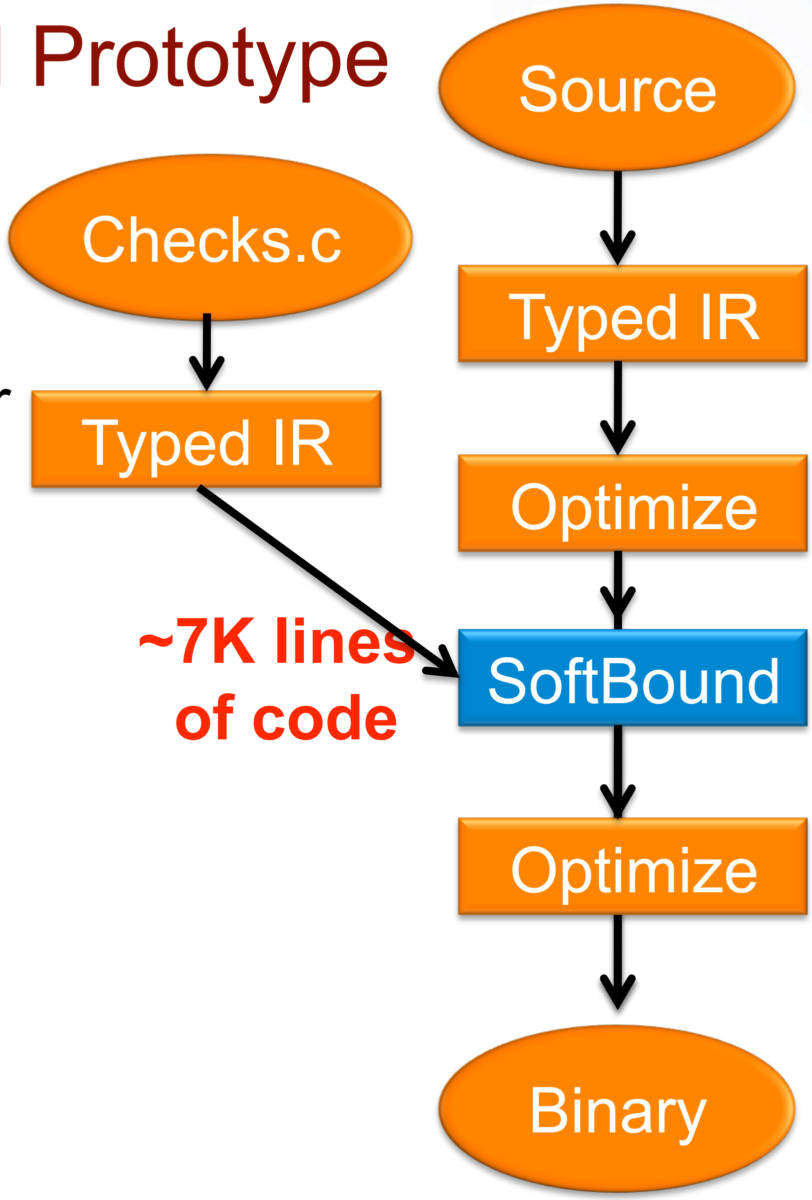
Programmer intent ambiguous;
optional shrinking of bounds

See Paper For…

- Proof of spatial safety guarantees
  - Region delineated by pointer metadata is always valid
  - Formalized a rich subset of C
    - Includes arbitrary casts, recursive structures, etc…
  - Mechanized proof in Coq
    - Online at: http://www.cis.upen.edu/acg/softbound/
- Handling various aspects of C
  - Separate compilation and library code
  - memcpy()
  - Function pointers
  - Variable argument functions
  - Etc…

SoftBound Prototype

- LLVM as its foundation
  - Typed IR helps in pointer identification



Experiments

- Three questions
  - Can SoftBound detect overflows?
  - Does SoftBound work with existing C code?
  - Does SoftBound have low overhead?

Spatial Violation Detection

- Can SoftBound detect overflows?
  - Synthetic attacks [Wilander et al]
    - Prevented all these attacks
  - Bugbench [Lu05]: overflows from real applications

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

No false negatives encountered

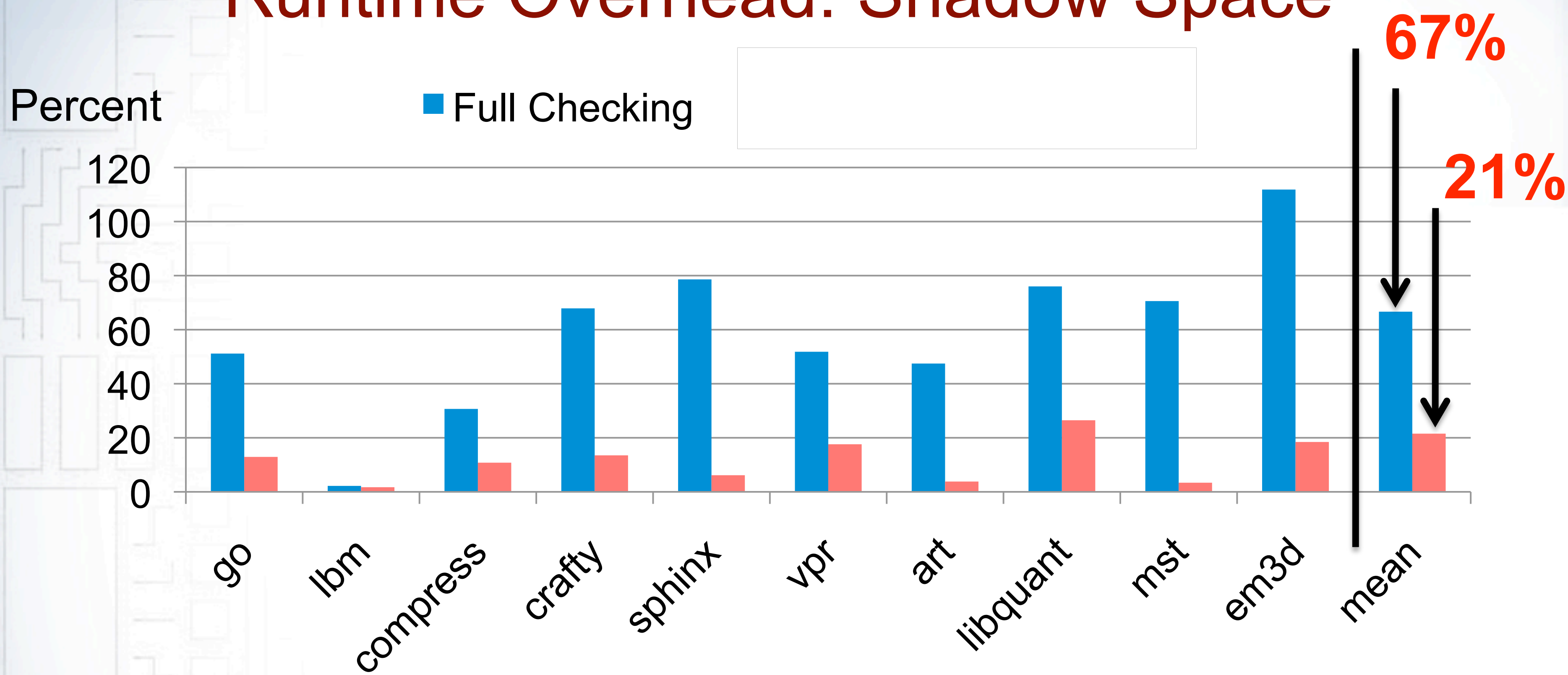
Source Compatibility Experiments

- Does SoftBound work with existing C code?

- 272K lines of code total
  - 23 benchmarks from Spec, Olden
  - BugBench
  - Multithreaded HTTP Server with CGI support
  - FTP server

No false positives encountered

Runtime Overhead: Shadow Space



- Check only stores [Yong03, Castro06]
  - Attacks predominantly use stores

Full Checking: default for development & testing

Store-only: for security critical apps, production code

Experiments Recap

- Can SoftBound detect overflows? **Yes**
- Does SoftBound work with existing C code? **Yes**
- Does SoftBound have low overhead? **Yes**
  - Full checking overhead - 67%
  - Store only checking overhead - 21%

Future Work

- Static optimizations
  - Removing redundant checks
- OS support
  - Shadow space management
- Hardware support
  - Heavyweight hardware support [Devietti, ASPLOS 08]
  - Lightweight hardware support
- Temporal safety
  - Dangling pointers
- C++

Our Experience with LLVM

- 4 months from first use to a PLDI submission
  - SoftBound pass – 7k lines of code
- Typed IR was crucial
  - Pointers already identified
  - Instrument post-optimized code
    - Versus source-to-source translation
  - Portable – ISA independent
- Leveraged existing optimizations

Couldn't have done it without LLVM

Conclusions

- SoftBound provides spatial safety for C
  - Fat pointer approach, but with disjoint metadata
  - Provides spatial safety guarantees

- SoftBound is:
  - **Compatible** (no false positives, no source changes)
  - **Effective** (no false negatives)
  - **Fast enough for…**
    - Debugging & testing: full checking
    - Security-critical software: store only checking

Want to try it out?

http://www.cis.upenn.edu/acg/softbound/

ACG

UNIVERSITY *of* PENNSYLVANIA

ARCHITECTURE + COMPILERS GROUP

Penn

UNIVERSITY *of* PENNSYLVANIA

Few Issues

- Instruction Combine

```
%struct.node_t = type { i64, i64, %struct.node_t* }
…..
ptr  = (struct temp*) malloc(sizeof(struct temp));
ptr->t1 = 0; ptr->t2  = 0;
```

⬇

```
%0 = malloc [3 x i64]                        ; <[3 x i64]*> [#uses=3]
%.sub9 = getelementptr inbounds [3 x i64]* %0, i64 0, i64 0 ; <i64*> ..
store i64 0, i64* %.sub9, align 8
%1 = getelementptr inbounds [3 x i64]* %0, i64 0, i64 2 ; <i64*> ..
store i64 0, i64* %1
```

Loss of Type Information: Multiple Ret Values

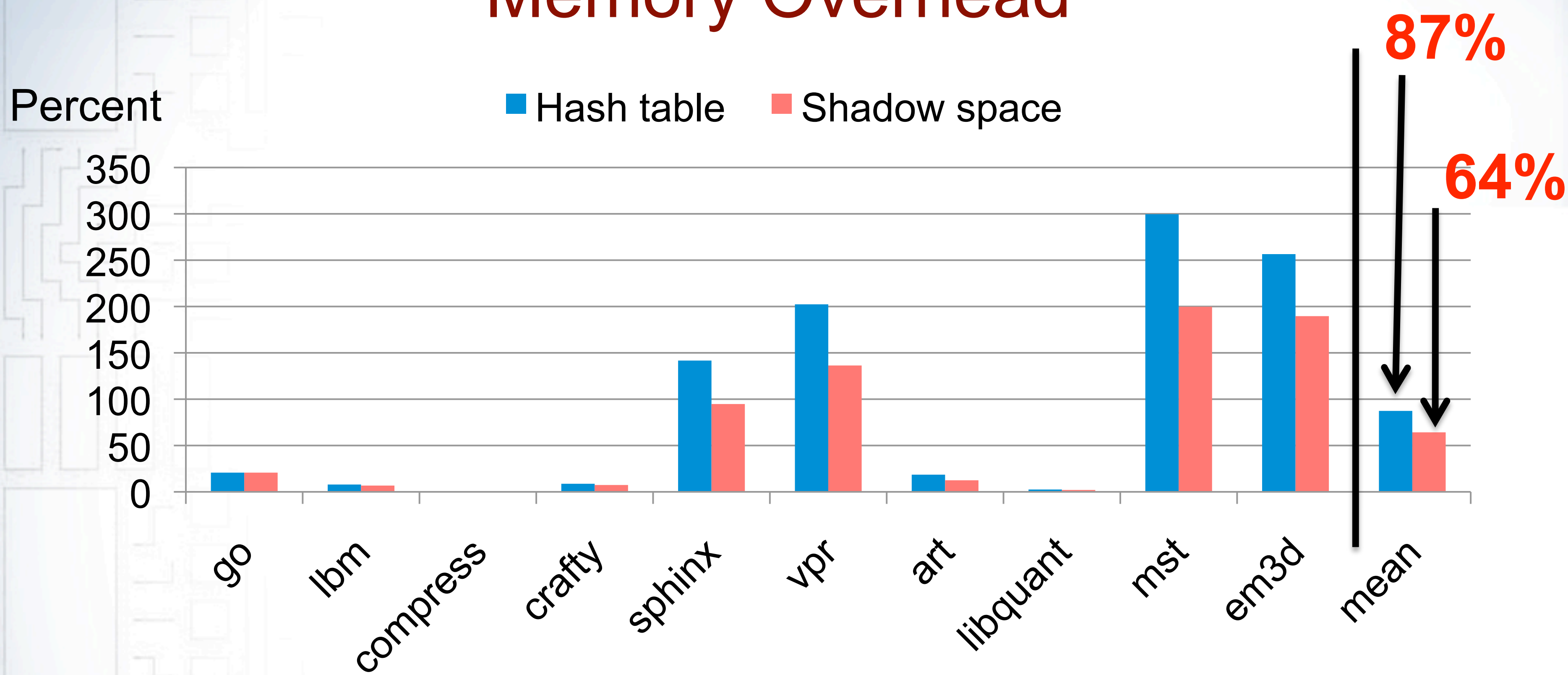
- From em3d benchmark:

```
typedef  struct t { node* n1, node* n2} graph_t;
…
graph_t initialize_graph() { …. }
```

↓

```
%0 = type{i64, i64}
define %0 @initialize_graph() nounwind{
….
}
```

Memory Overhead



Average memory overhead – full checking: 84%

Average memory overhead – store only: 64%