Vulnerability Prevention

Holistic Software Security

Aravind Machiry

Can we prevent vulnerabilities?

- Prevent => Making sure that a program does not have vulnerabilities.

- Why does a program has vulnerabilities?

How do we write programs?

I want to write code to do X:

1. Think about "How to do X" -> Algorithm.

2. Code <-> Test.

Development mindset => Will the code **do "X"**?

Security mindset => I want the code **NOT to do Y**.

Possible Y's:

- Buffer overflow.
- Out-of-bounds access.
- etc.

Bridging the gap!

- Train developers to have security mindset:
  - Secure coding training.

- Enable developers to write code that “cannot” have vulnerabilities:
  - Provide Memory safe/Type safe languages:
    - Java, Python, C#, etc.

Memory Safety

- Spatial memory safety: Ensuring all memory dereferences are within the objects allocated space.
  - Out of bounds access, buffer overflow, underflow, etc.
  - `arr[i]`

- Temporal memory safety: Ensuring that memory dereferences are valid at the time of access.
  - Use-after-free, double free, etc.
  - `free(p); *p = 0;`

Type Safety

- Objects are well-typed and conversion between types is well-defined:
  - Ex: In Java, type conversion is allowed only with in subtypes.

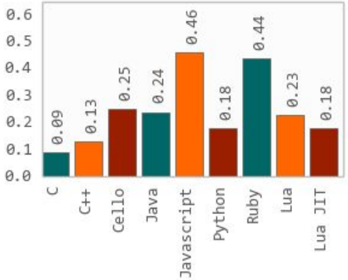
- Is Python type safe?
- Is Java type safe?
- Is C++ type safe?

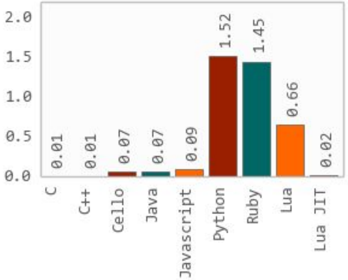
How is safety implemented?

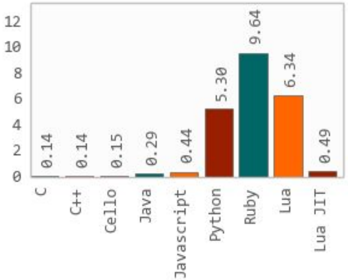
- Runtime checking:
  - Language runtime: Java JRE.
  - Memory accesses are checked for violations.
  - Castings are also checked at time.

Safety is not free!

Performance: Time and Space.







* http://libcello.org/learn/benchmarks

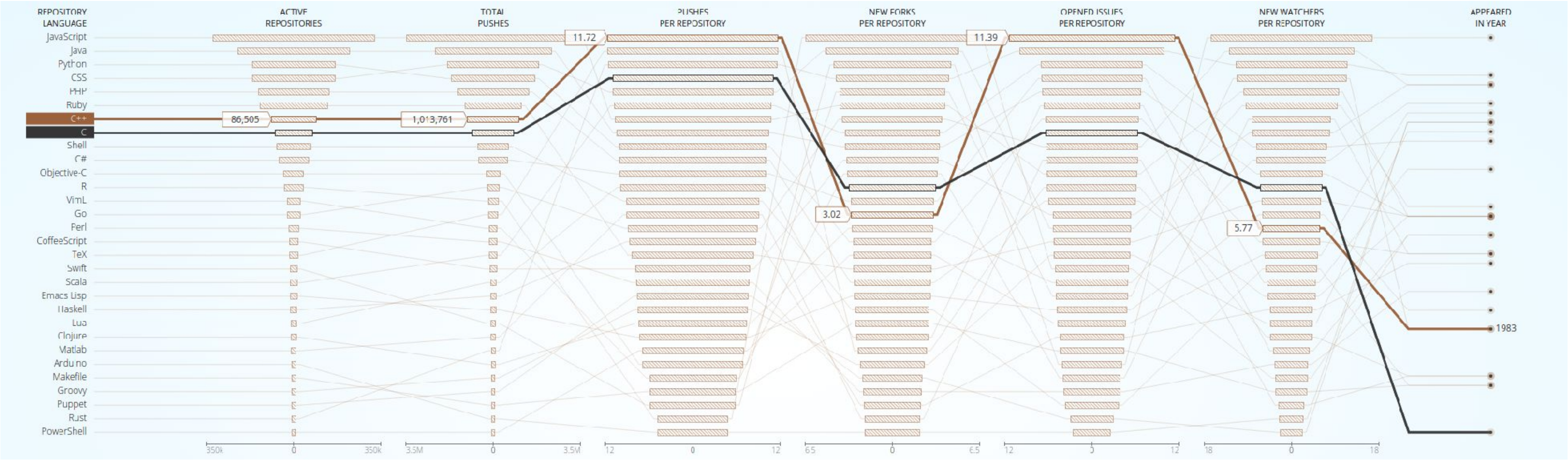
High-performant safe languages

- Rust/Go:
  - Similar to C/C++, faster than Java, Python, etc.

- Lets always use Rust/Go!

What is the catch?

What about legacy code?



Can we ask all developers to convert their code to safe languages?

Retrofitting Techniques

- Retrofit safety to unsafe languages:
  - Modify language semantics so that certain safety properties can be achieved.
  - Performance overhead?
    - Space and Time.
  - Automated or manual?
    - Does developer has to make changes to the existing code?

Retrofitting Techniques: Principles

- Spatial memory safety (SMS):
  - An efficient way to track bounds (start and end) of the object being referenced.

- Temporal memory safety (TMS):
  - An efficient way to track lifetime of objects.

SoftBound: SMS

For each pointer variable (p) : Add two variables to track bounds (start : p_base) and end: p_bound).

Check each pointer dereference to be with in bounds.

```
value = *ptr;
```

→ SoftBound

```
check(ptr, ptr_base, ptr_bound, sizeof(*ptr));
value = *ptr;
```

```
void check(ptr, base, bound, size) {
  if ((ptr < base) || (ptr+size > bound)) {
    abort();
  }
}
```

SoftBound: Tracking Pointers

```
ptr = malloc(size);
```

→

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

```
newptr = ptr + index;
```

→

```
newptr = ptr + index;
newptr_base = ptr_base;
newptr_bound = ptr_bound;
```

```
p = &(n->num);
```

→

```
p = &(n->num);
p_base = max(&(n->num), n_base);
p_bound = min(p_base + sizeof(n->num), n_bound);
```

SoftBound: Tracking Pointers

```
int** ptr;
int* new_ptr;
(*ptr) = new_ptr;
```

SoftBound: Tracking Pointers

```
int** ptr;
int* new_ptr;
(*ptr) = new_ptr;
```

⇒

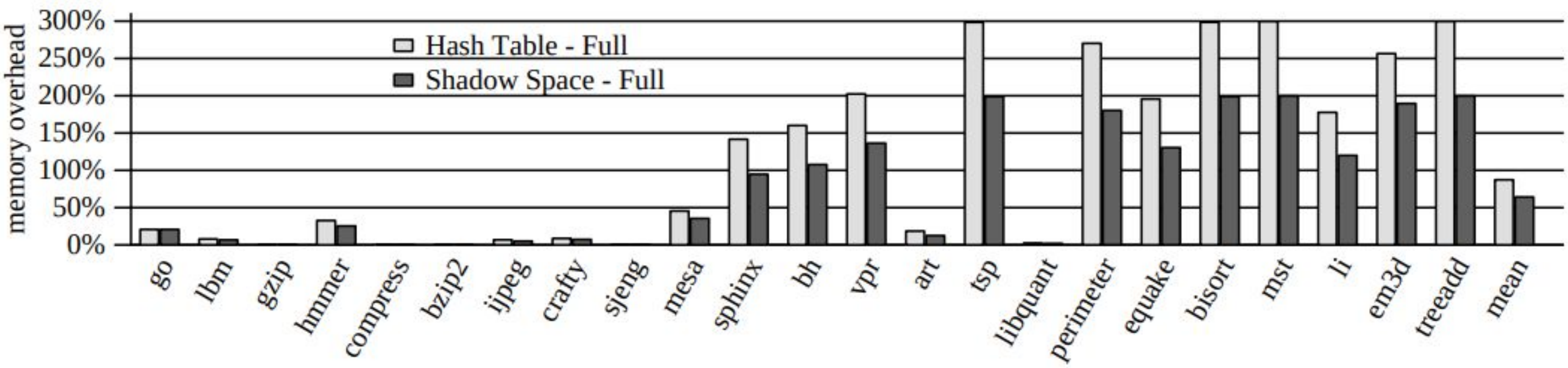
```
int** ptr;
int* new_ptr;
(*ptr) = new_ptr;
table_lookup(ptr)->base = newptr_base;
table_lookup(ptr)->bound = newptr_bound;
```

```
newptr = *ptr;
```

⇒

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

SoftBound: Performance



SafeCode: SMS

- Use splay trees to store the bounds information of pointers:
  - **Temporal locality**: Recently accessed object will be accessed again.
  - Splay trees favors temporal locality:
    - Stack behaving tree: Recently inserted object will be fast to access.

SafeCode: Novelty

- Use pool allocation: Objects size fall into one of the predefined sizes. E.g., 16, 32, 64, etc.

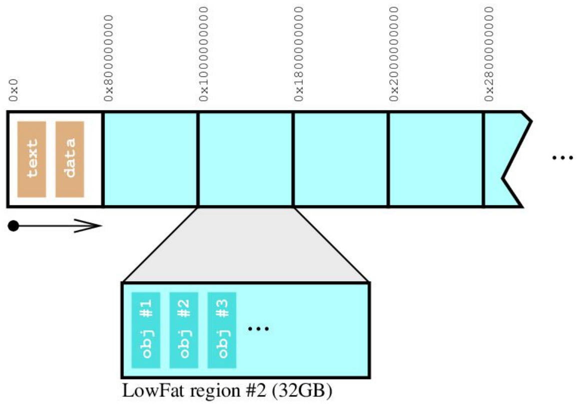
- Split the global splay tree into multiple small splay trees:
    - One for each size.

- Given a pointer => Find its pool and check for the bounds in the splay tree of the corresponding pool.

Low Fat Pointers: SMS

We can smartly allocate and know the base and bounds from the pointer itself.

Each region will only store objects of specific size. E.g., 0x800000000-0xffffffff for objects of size < 16 bytes



Low Fat Pointers

p = malloc(10); // p: 0x8997f2820

q = p + 5; // q = 0x8997f2825

```
char get(char *q, int i) {
  return q[i];
}
```

→

```
char get(char *q, int i) {
  char *q_base = base(q);
  size_t q_size = size(q);
  char *r = q + i;
  if (r < q_base || r >= q_base + q_size)
    report_oob_error();
  return *r;
}
```

What is base(q) and size(q)?

base(q) =

size(q) =

What is base(q) and size(q)?

base(q) = **0x8997f2820**

size(q) = **16**

Since q is within the range (0x800000000..0xfffffffff), we know that the allocation size of the object pointed to by q is 16 bytes.

Base address should be: q - (q mod 16) = 0x8997f2820.

Handling pointer arithmetic

```
int list_length(Node *list)
{
    int len = 0;
    void *list_base = base(list);
    size_t list_size = size(list);
    while (list != NULL)
    {
        len++;
        Node **next = &list->next;
        void *next_base = list_base;
        size_t next_size = list_size;
        if (isOOB(next, next_base, next_size))
            error();
        list = *next;
        list_base = base(list);
        list_size = size(list);
    }
    return len;
}
```

Overhead

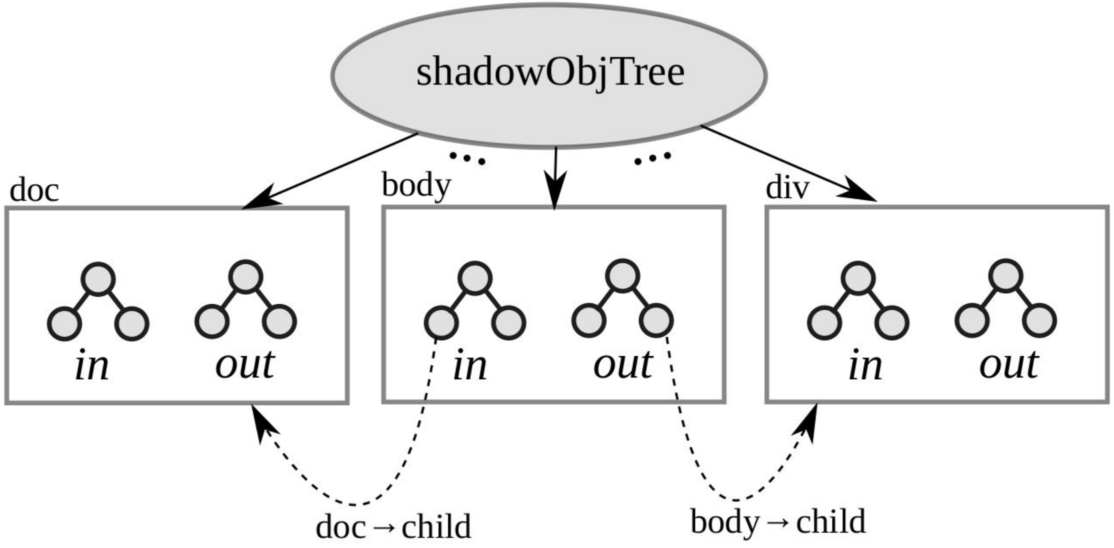
- 56% for reads+writes
- 13% for writes-only

DANGNULL: TMS

- Handles temporal memory safety:
  - Should keep track of object life times.

- Keep tracks of heap objects in a red-black tree (shadowObjTree).
  - Each object has in-bound and out-bound pointers.
  - In-bound: Pointers that are pointing to the current object.
  - Out-bound: Objects to which the current object points to.

DANGNULL


shadowObjTree
doc
body
div
in
out
in
out
in
out
doc → child
body → child

DANGNULL: Instrumentation

```
// (a) memory allocations
Document *doc = new Document();
Body *body = new Body();
Div *div = new Div();

// (b) using memory: propagating pointers
doc->child = body;
body->child = div;

// (c) memory free: doc->child is now dangled
delete body;

// (d) use-after-free: dereference the dangled pointer
if (doc->child)
    doc->child->getAlign();
```

DANGNULL

```
// (a) memory allocations
+ Document *doc = allocObj(Document);
+ Body *body = allocObj(Body);
+ Div *div = allocObj(Div);

// (b) using memory: propagating pointers
doc->child = body;
+ trace(&doc->child, body);

body->child = div;
+ trace(&body->child, div);

// (c) memory free: unsafe dangling pointer, doc->child,
//                  is automatically nullified
+ freeObj(body);

// (d) use-aftre-free is prevented, avoid dereferencing it
if (doc->child)
    doc->child->getAlign();
```

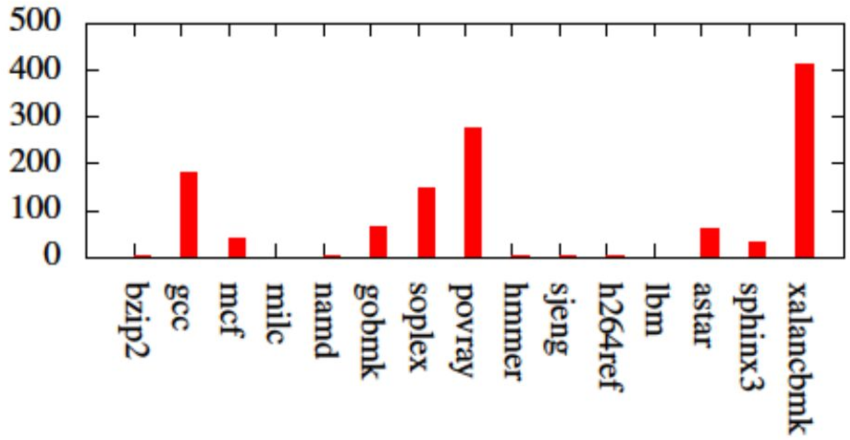
DANGNULL: Helper functions

```
def allocObj(size):
    ptr = real_alloc(size)
    shadowObj = createShadowObj(ptr, size)
    shadowObjTree.insert(shadowObj)
    return ptr
```

```
# NOTE. lhs <- rhs
def trace(lhs, rhs):
    lhsShadowObj = shadowObjTree.find(lhs)
    rhsShadowObj = shadowObjTree.find(rhs)

    # Check if lhs and rhs are eligible targets.
    if lhsShadowObj and rhsShadowObj:
        removeOldShadowPtr(lhs, rhs)
        ptr = createShadowPtr(lhs, rhs)
        lhsShadowObj.insertOutboundPtr(ptr)
        rhsShadowObj.insertInboundPtr(ptr)
    return
```

```
def freeObj(ptr):
    shadowObj = shadowObjTree.find(ptr)

    for ptr in shadowObj.getInboundPtrs():
        srcShadowObj = shadowObjTree.find(ptr)
        srcShadowObj.removeOutboundPtr(ptr)
        if shadowObj.base <= ptr < shadowObj.end:
            *ptr = NULLIFY_VALUE

    for ptr in shadowObj.getOutboundPtrs():
        dstShadowObj = shadowObjTree.find(ptr)
        dstShadowObj.removeInboundPtr(ptr)

    shadowObjTree.remove(shadowObj)

    return real_free(ptr)
```

DANGNULL: Performance



Cost of automation

- High performance penalty.

- Not backward compatible:
  - E.g., regular pointers cannot co-exist with low fat pointers.

- Maintenance overhead: Should have these features in the latest compilers.