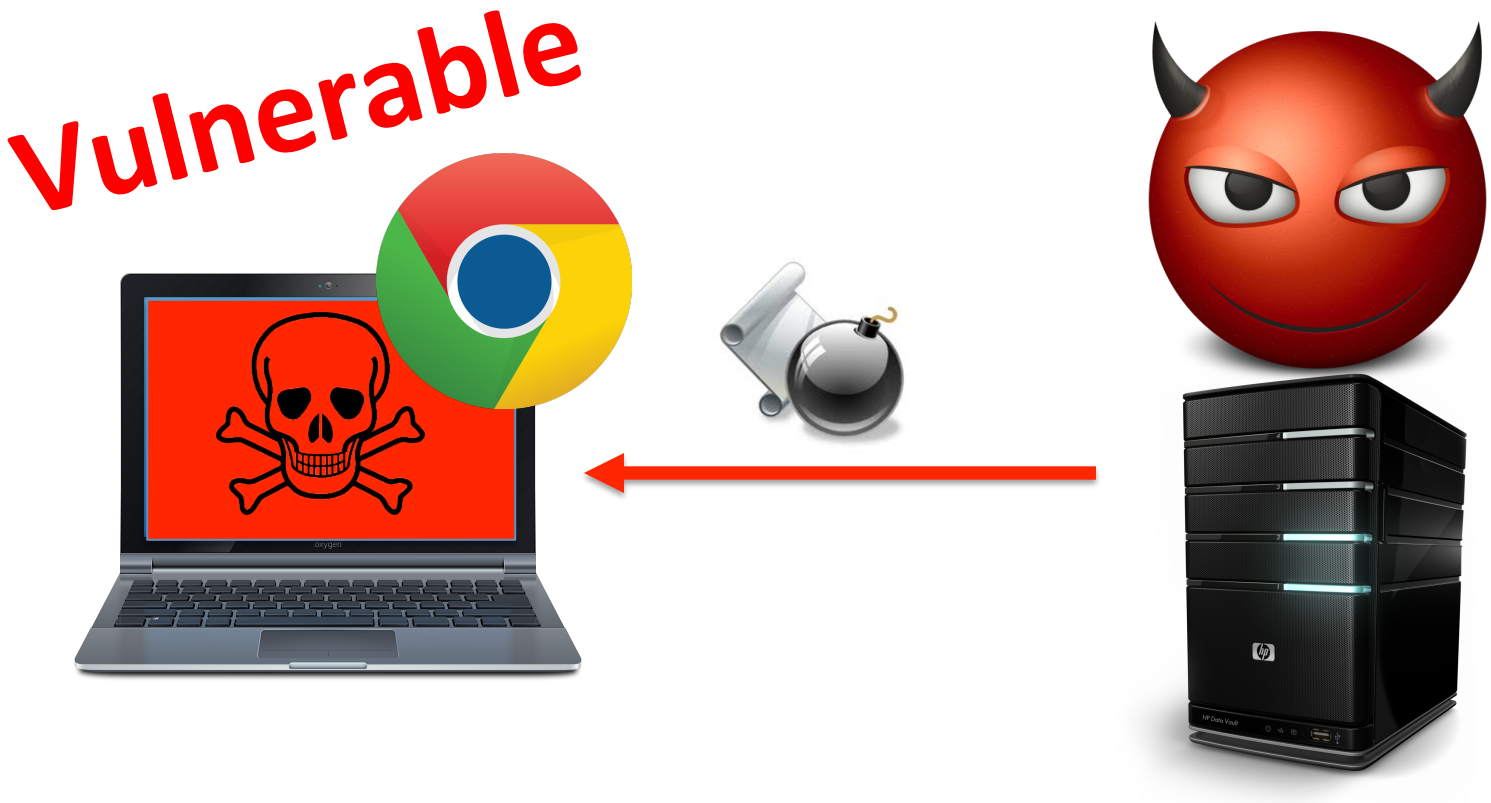
SafeDispatch

Securing C++ Virtual Function Calls from Memory Corruption Attacks

Dongseok Jang
UC San Diego

Zachary Tatlock
University of Washington

Sorin Lerner
UC San Diego


Vulnerable

Control Flow Hijacking

Lead Program to Jump to Unexpected Code

That does what attacker wants

Example: Stack Buffer Overflow Attacks

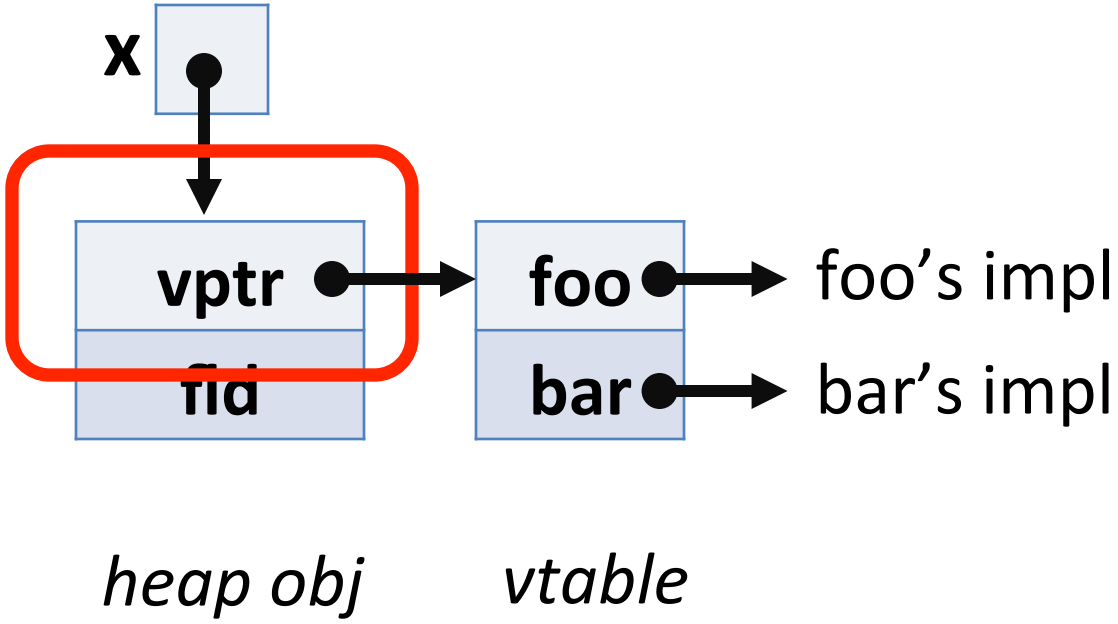
Well studied and hard to be critical by itself

New Frontier : Vtable Hijacking

Vtable Pointers

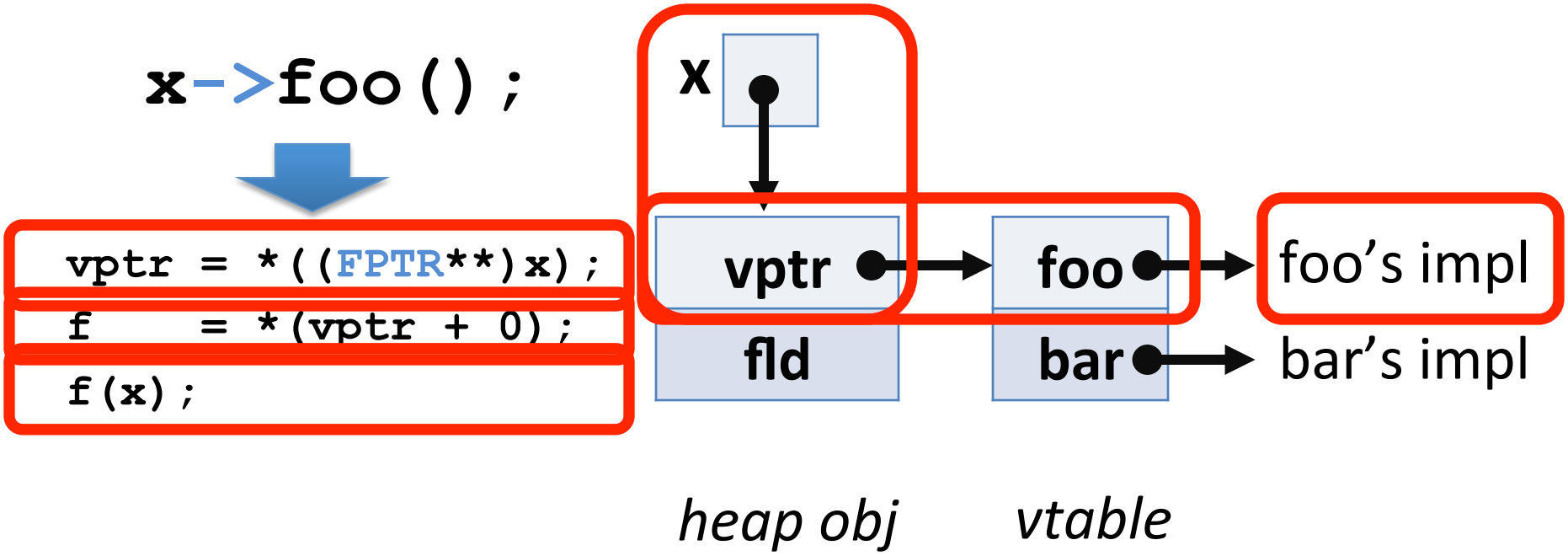
Mechanism for Virtual Functions

```
class C {
 virtual int foo();
 virtual int bar();
 int fld;
};
...
C *x = new C();
```

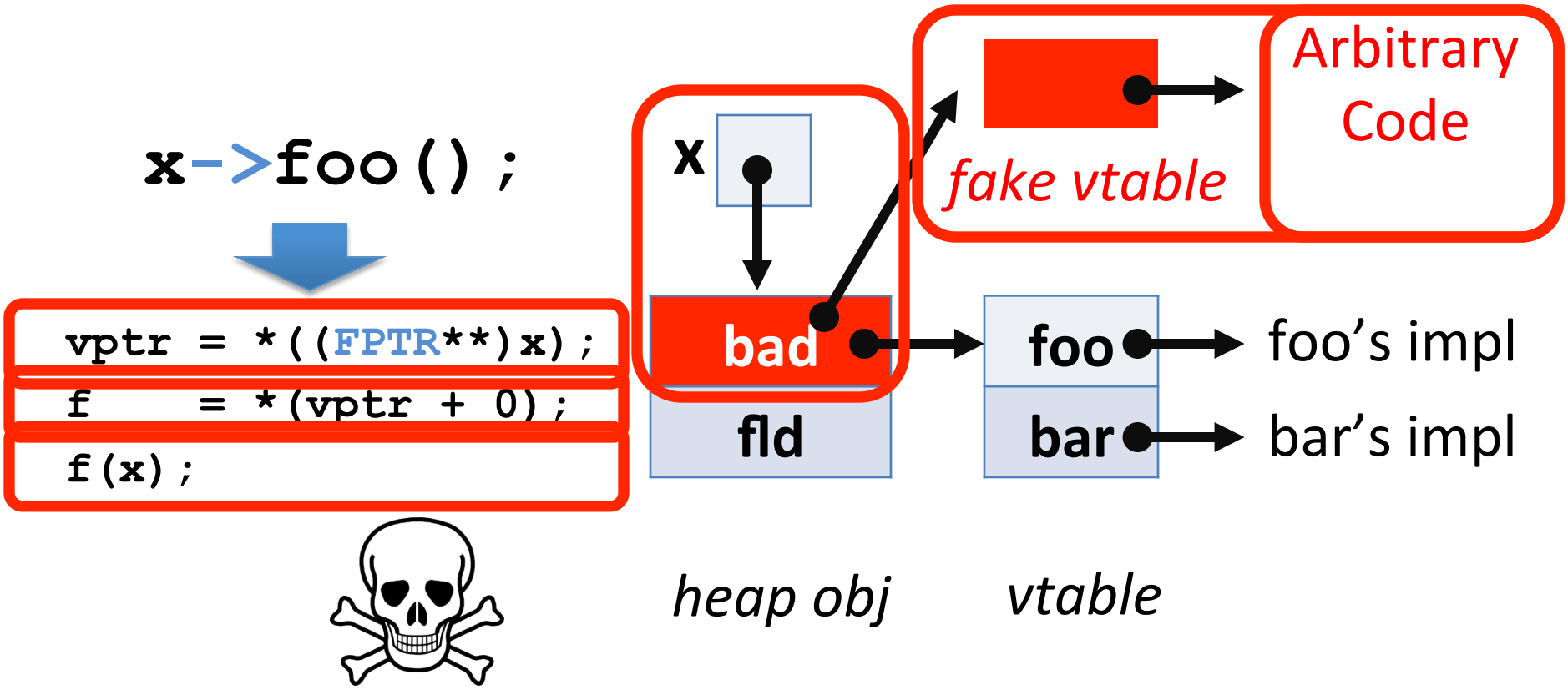


Vtable Pointers

Virtual Call : 2-Step Dereferencing for Callee



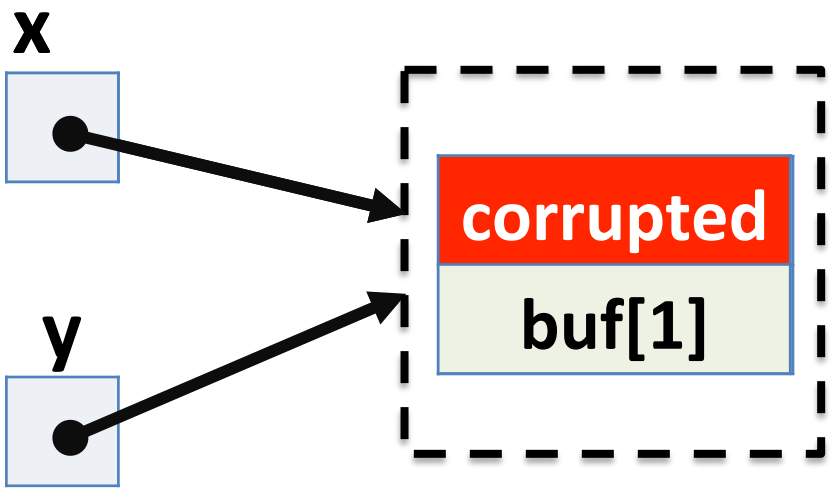
Vtable Hijacking



Vtable Hijacking via Use-after-Free

Use Corrupted Data for x's vptr

```
C *x = new C();
x->foo();
delete x;
// forget x = NULL;
...
D *y = new D();
y->buf[0] = input();
...
x->foo();
```



candidate for reallocation

Vtable Hijacking: Real Case

Vtable Hijacking of Chrome via Use-after-Free

Pinkie Pie's demonstration at Pwn2Own

Used to trigger ROP for sandbox escaping of Chrome

Found in IE, Firefox, Chrome

How to Prevent Vtable Hijacking?

With Accuracy & Low Overhead?

Code Instrumentation

```
C *x = ...
Check(x);
x->foo();
```

Code Instrumentation

```
C *x = ...
ASSERT(VPTR(x) ∈ Valid(C));
x->foo();
```

Valid(C) = { vptr of C or C's subclasses }

Obtained by class hierarchy analysis (CHA)

Code Instrumentation

```
C *x = ...
ASSERT(VPTR(x) ∈ Valid(C));
x->foo();
```

Simple Implementation Can Be Slow

Involved data structure lookup/function calls

Inlining Optimization

```
C *x = ...
ASSERT(VPTR(x) ∈ Valid(C));
x->foo();
vptr = *((FPTR**)x);
f    = *(vptr + 0);
f(x);
```

x->foo()

Inlining Optimization

```
C *x = ...
// ASSERT(VPTR(x) ∈ Valid(C));
vptr = *((FPTR**)x);
ASSERT(vptr ∈ Valid(C));
f    = *(vptr + 0);
f(x);
```

Inlining Optimization

```
C *x = ...
// ASSERT(VPTR(x) ∈ Valid(C));
vptr = *((FPTR**)x);
// ASSERT(vptr ∈ Valid(C));
ASSERT(vptr ∈ {C::vptr, D::vptr});
f    = *(vptr + 0);
f(x);
```

Say that C has only one subclass D

→ Specialization of Checks

Inlining Optimization

```
C *x = ...
// ASSERT(VPTR(x) ∈ Valid(C));
vptr = *((FPTR**)x);
// ASSERT(vptr ∈ Valid(C));
// ASSERT(vptr ∈ {C::vptr, D::vptr});
f    = *(vptr + 0);
f(x);
```

SAFE:

Inlining Optimization

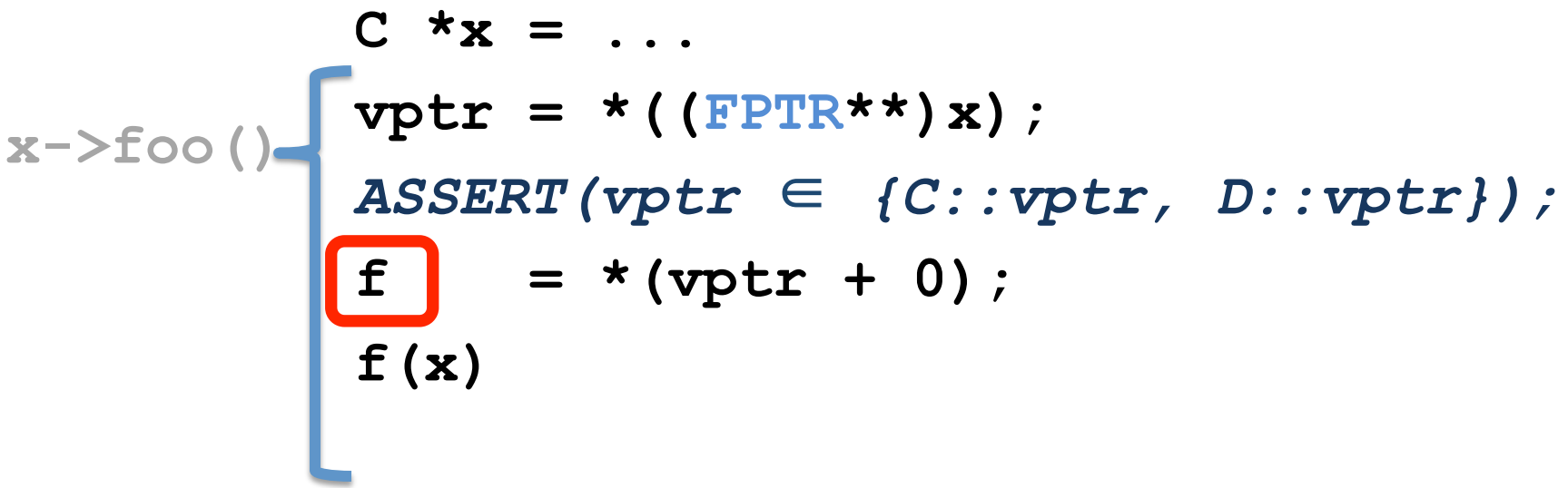
```
        C *x = ...
    // ASSERT(VPTR(x) ∈ Valid(C));
        vptr = *((FPTR**)x);
    // ASSERT(vptr ∈ Valid(C));
    // ASSERT(vptr ∈ {C::vptr, D::vptr});
        if (vptr == C::vptr) goto SAFE;
        if (vptr == D::vptr) goto SAFE;
        exit(-1);
SAFE:   f    = *(vptr + 0);
        f(x);
```

Inlining Optimization

How to Order Inlined Checked?
→ Profile-guided Inlining

```
        if (vptr == C::vptr) goto SAFE;
        if (vptr == D::vptr) goto SAFE;
        exit(-1);
SAFE:   f     = *(vptr + 0);
        f(x);
```

Method Pointer Checking



Method Pointer Checking

```
C *x = ...
vptr = *((FPTR**)x);
//ASSERT(vptr ∈ {C::vptr, D::vptr});
f    = *(vptr + 0);
ASSERT(f ∈ ValidM(C,foo));
f(x)
```

Checking Callee Before It Is Called

Provides same security as vtable checking

Method Pointer Checking

```
C *x = ...
vptr = *((FPTR**)x);
//ASSERT(vptr ∈ {C::vptr, D::vptr});
f    = *(vptr + 0);
//ASSERT(f ∈ ValidM(C,foo));
ASSERT(f ∈ {C::foo});
f(x)
```

Save Checks for Shared Methods

Member Pointers in C++

```
A *x = ...
// m: index into a vtable
// x->*m can be any methods of A
(x->*m)()
```

Say that A has 1000 methods

Vtable Checking Can Be Faster

Method Pointer Checking

Fewer Checks for Usual Virtual Calls

More Checks for Member Pointer Calls

Hybrid Checking

Method Checking for Usual Virtual Calls

Vtable Checking for Member Pointer Calls

Tamper Resistance

Inserted Checks in Read-Only Memory

Checking Data in Read-Only Memory

Performance: Benchmark

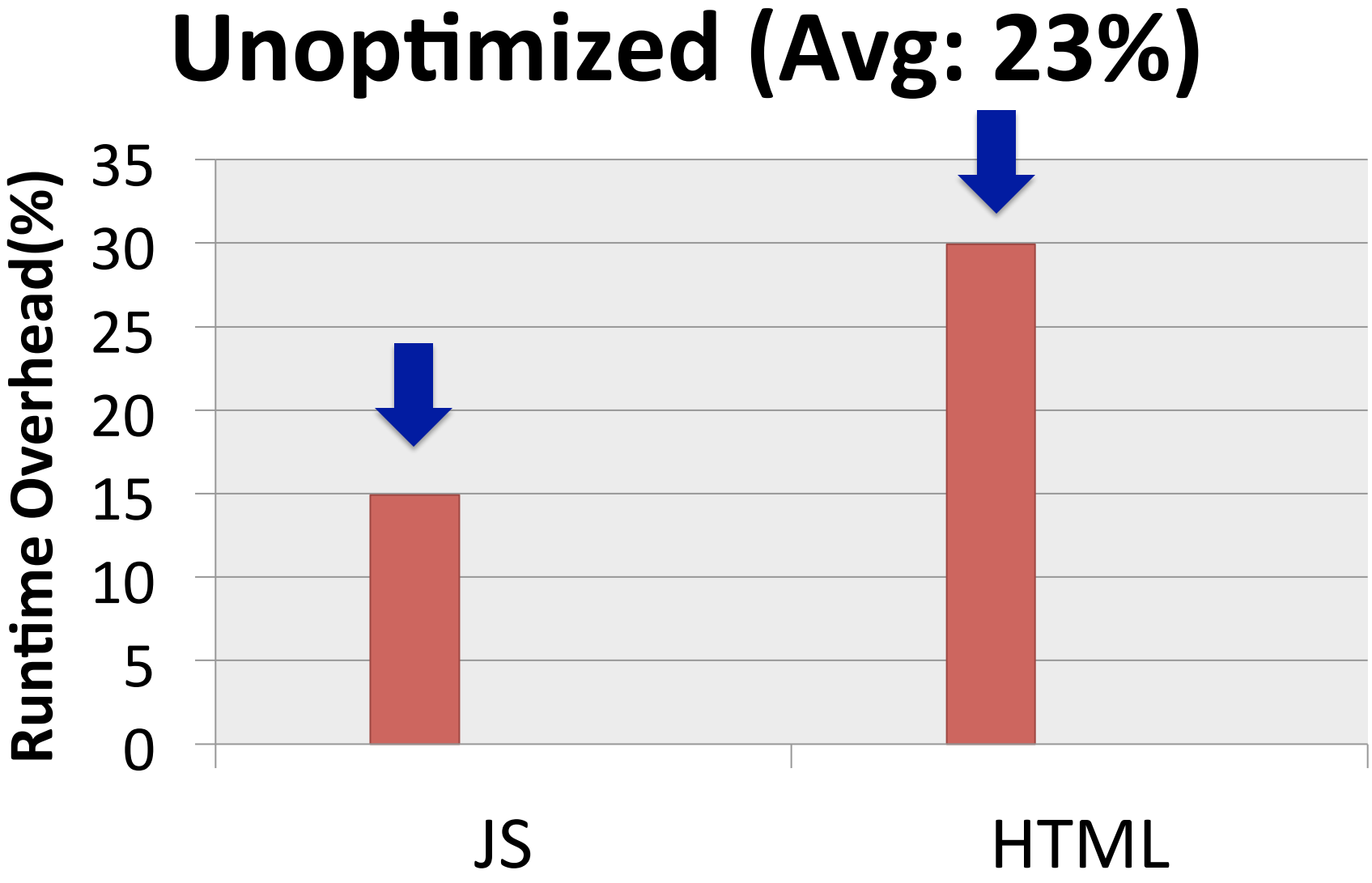
Chromium Browser

Realistic : ≈ 3 millions of C++/C LOC

Popular target of vtable hijacking

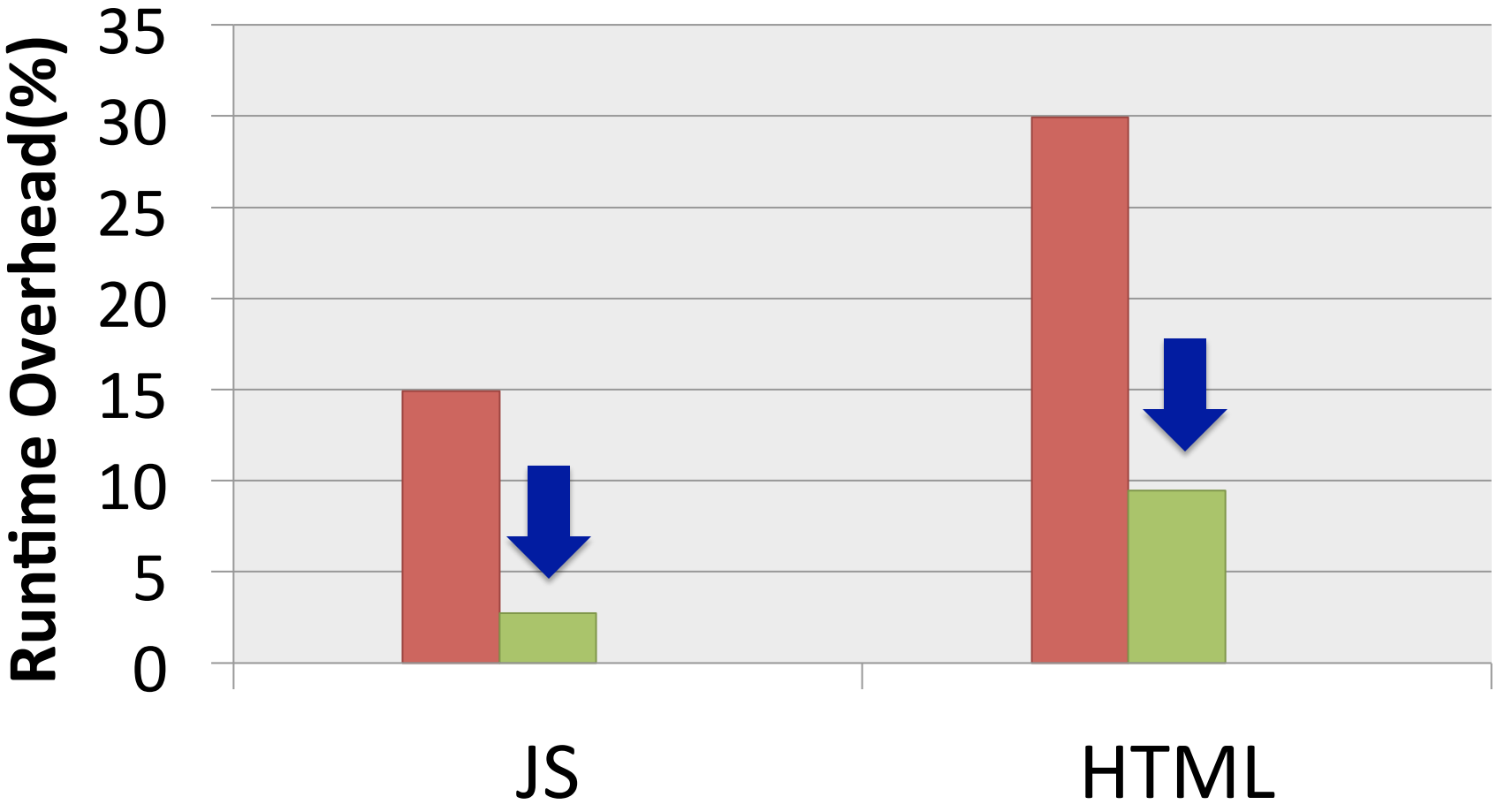
Running On JS, HTML5 Benchmark

Performance



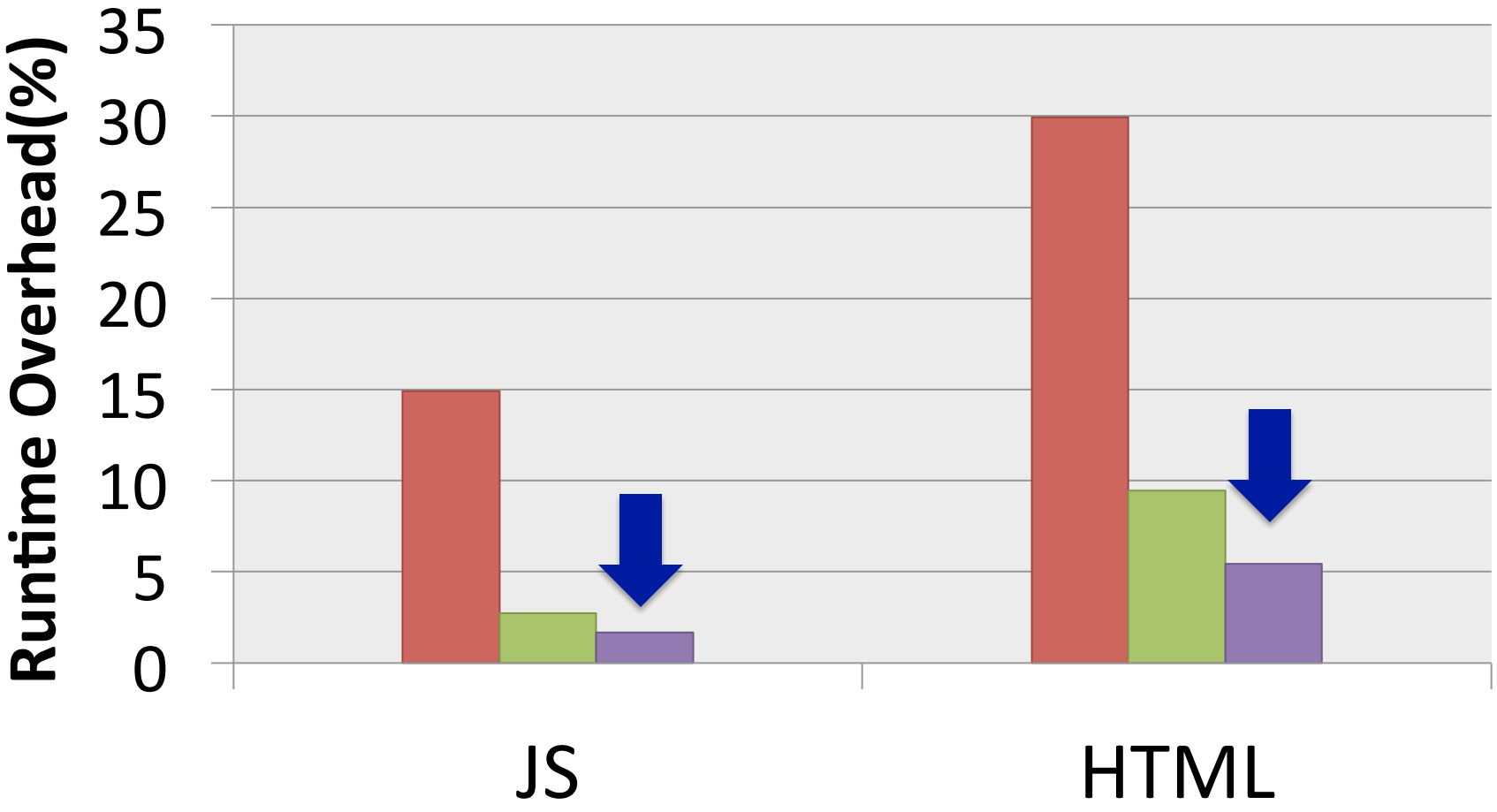
Performance

Profile-Guided Inlining (Avg: 6%)



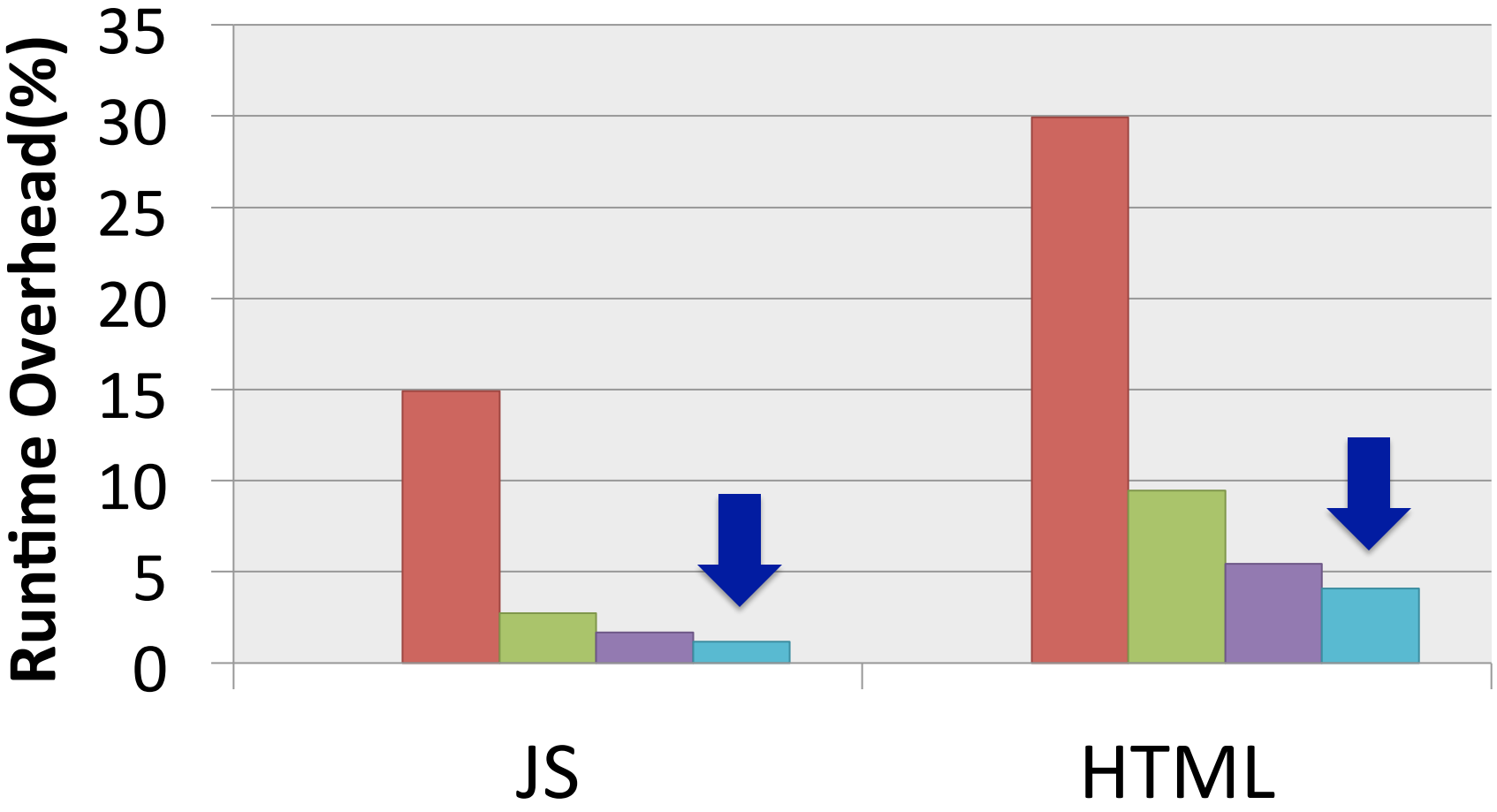
Performance

Inlined Method Ptr Checking (3%)



Performance

Hybrid Checking (Avg: 2%)



Code Size Overhead

7% Code Size Increase

8.3 MB out of 119 MB

Checking Data + Inlined Checks

Future Work

Separate Compilation

Link-time CHA / inlining

Dynamic Link Library

Runtime update of checking data

Summary

Vtable Hijacking

Often happening in web browsers

Compiler-based Approach

Code Instrumentation / static Analysis

Realistic Overhead

Careful compiler optimizations

Thank you!

http://goto.ucsd.edu/safedispatch