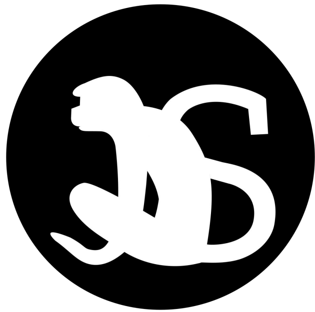
FaJITa:
verifying JIT-optimized programs

David Thien, Evan Johnson, Michael Smith, Sorin Lerner,
Fraser Brown, Hovav Shacham, Deian Stefan

Modern JavaScript JITs

- Need to start fast
- Need to produce fast code



Closed Bug 1607443 (CVE-2019-17026) Opened 1 year ago Closed 1 y

In-the-wild 0-day reported by Qihoo 360

```
[turbofan] Fix NumberConstant used with Word32 rep in ISel

Bug: chromium:1304658

Change-Id: I6a82603a7c5de5ae8f5a895990c1a904bbdd39b2
Reviewed-on: https://chromium-review.googlesource.com/c/v8/v8/+/3532263
Auto-Submit: Nico Hartmann <nicohartmann@chromium.org>
Reviewed-by: Tobias Tebbi <tebbi@chromium.org>
Commit-Queue: Tobias Tebbi <tebbi@chromium.org>
Cr-Commit-Position: refs/heads/main@{#79526}
```

Tuesday, September 1, 2020

JITSploitation I: A JIT Bug

By Samuel Groß, Project Zero

Issue 2020: JSC: JIT: In
Reported by saelo@google.c

The DFG and FTL JIT compi ... his can then be exploited to cau...

This three-part series highlights the technical challenges involved in finding and exploiting JavaScript engine vulnerabilities in modern web browsers and evaluates current exploit mitigation technologies. The exploited vulnerability, CVE-2020-9802, was i... mitigation bypasses, CVE-2020-9870 and CVE-2020-9910, were

On February 10, Threat Analysis Group discovered two distinct North Korean government-backed attacker groups exploiting a remote code execution vulnerability in Chrome, CVE-2022-0609. These groups' activity has been publicly tracked as *Operation Dream Job* and *Operation AppleJeus*.

```
[compiler][x64] Fix bug in InstructionSelector::ChangeInt32ToInt

Bug: chromium:1196683
Change-Id: Ib4ea738b47b64edc81450583be4c80a41698c3d1
Reviewed-on: https://chromium-review.googlesource.com/c/v8/v8/+/2820971
Commit-Queue: Georg Neis <neis@chromium.org>
Reviewed-by: Nico Hartmann <nicohartmann@chromium.org>
Cr-Commit-Position: refs/heads/master@{#73903}
```

Impact: Processing maliciously crafted web content may lead to code execution

Description: A memory corruption issue was addressed with improved state management.

WebKit Bugzilla: 238178

CVE-2022-26700: ryuzaki

We can do better

- More sane architecture (inline caches)
- Make bug harder to exploit (sandboxing)
- Verifying small independent parts (range analysis)
- Fuzz testing (Fuzzilli)

What should we do?

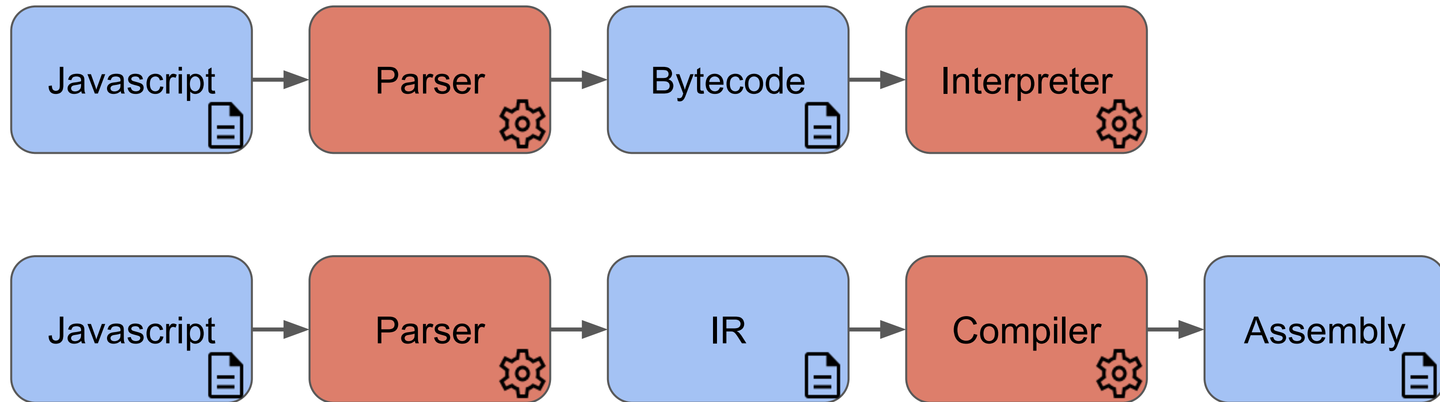
Ideal:

- We’d like to verify everything
- End-to-end verification for browser is intractable

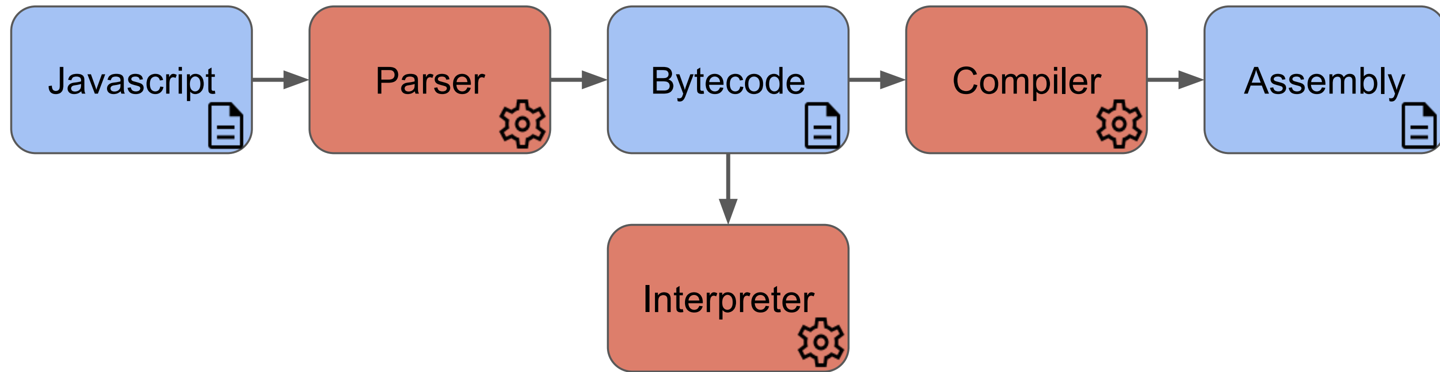
Practical:

- Focus verification on dangerous components
- Use domain-specific details for verification

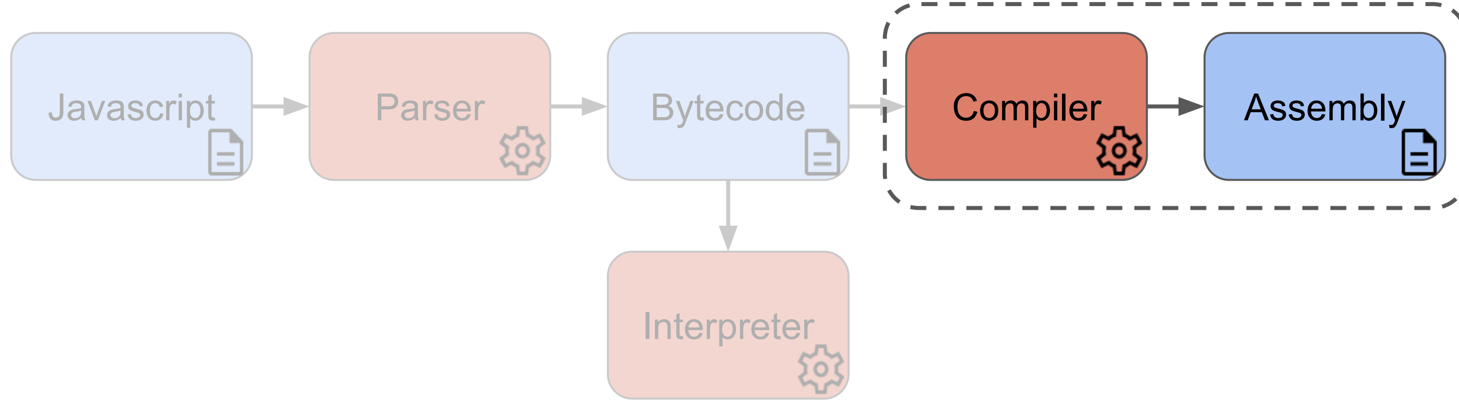
Traditional interpreter/compiler



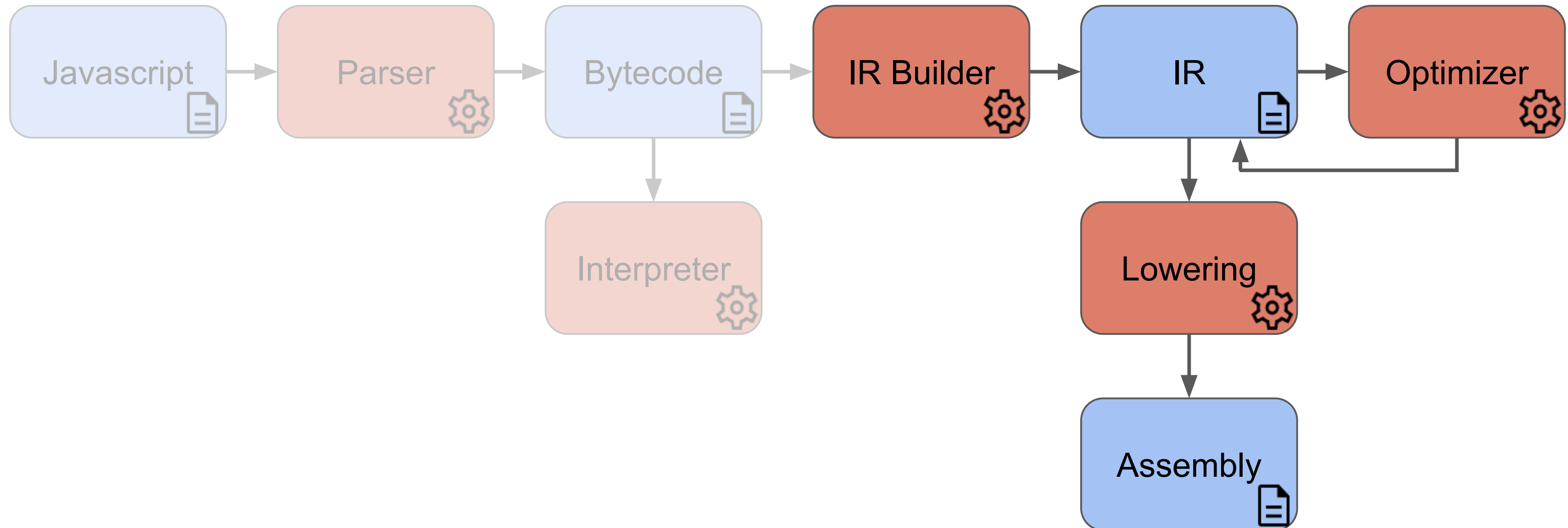
JIT compiler



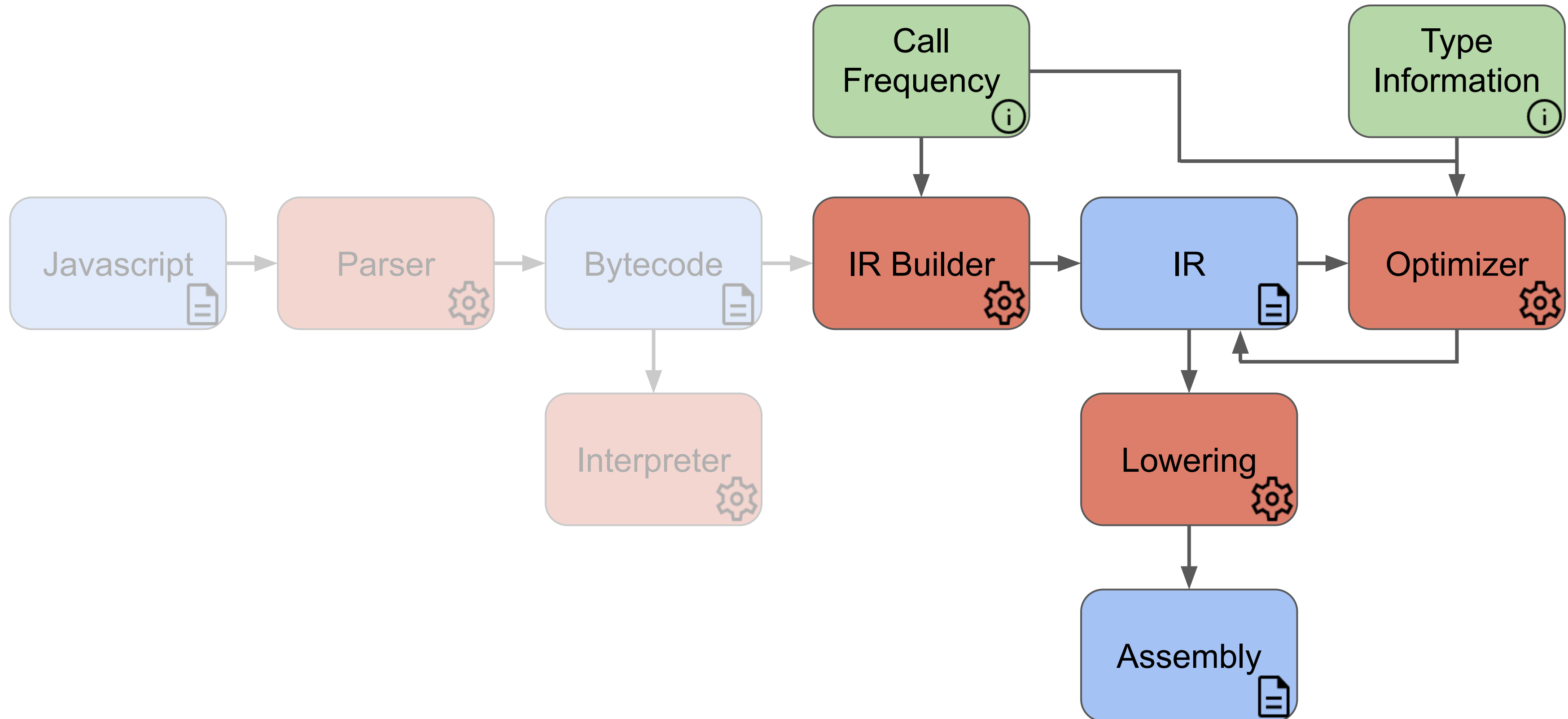
JIT compiler



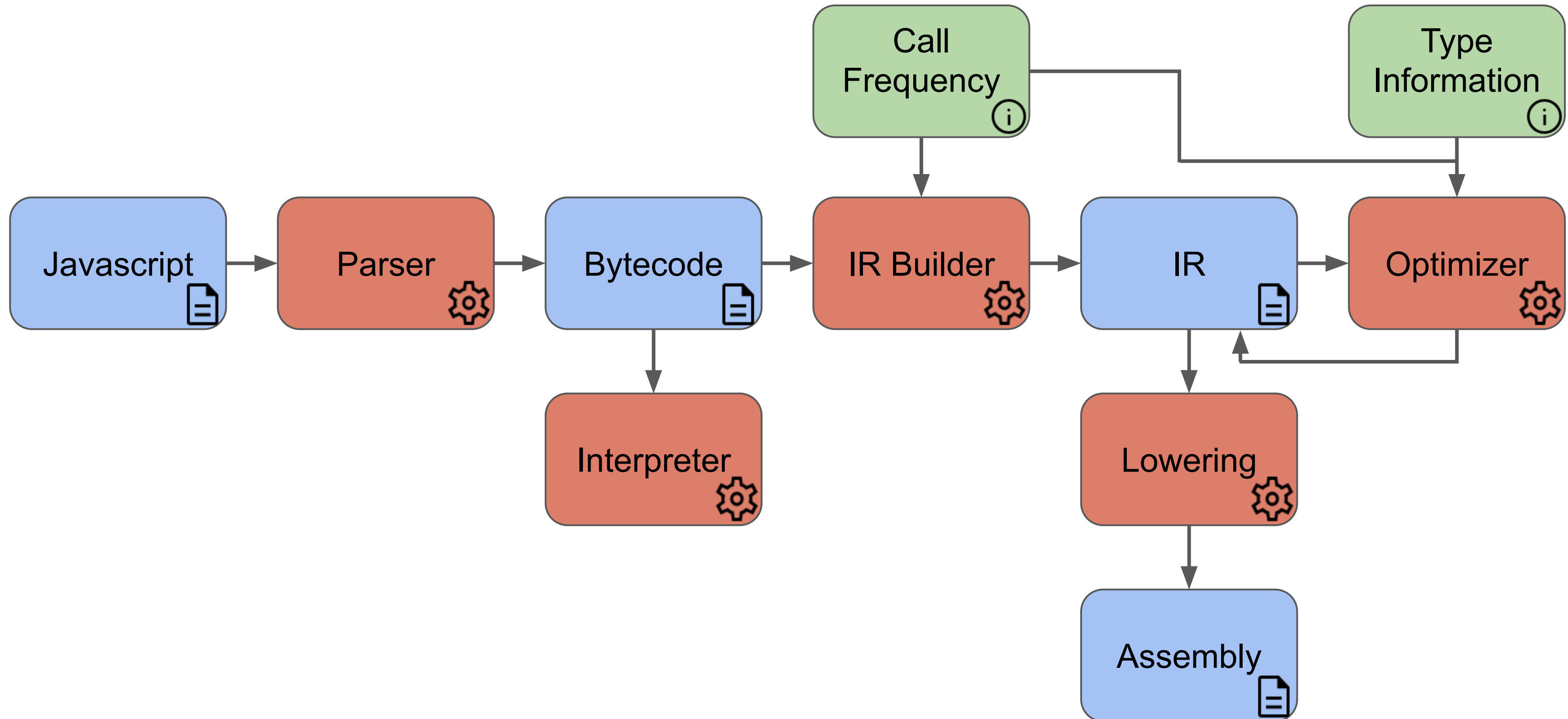
JIT compiler



JIT compiler with speculative optimizations



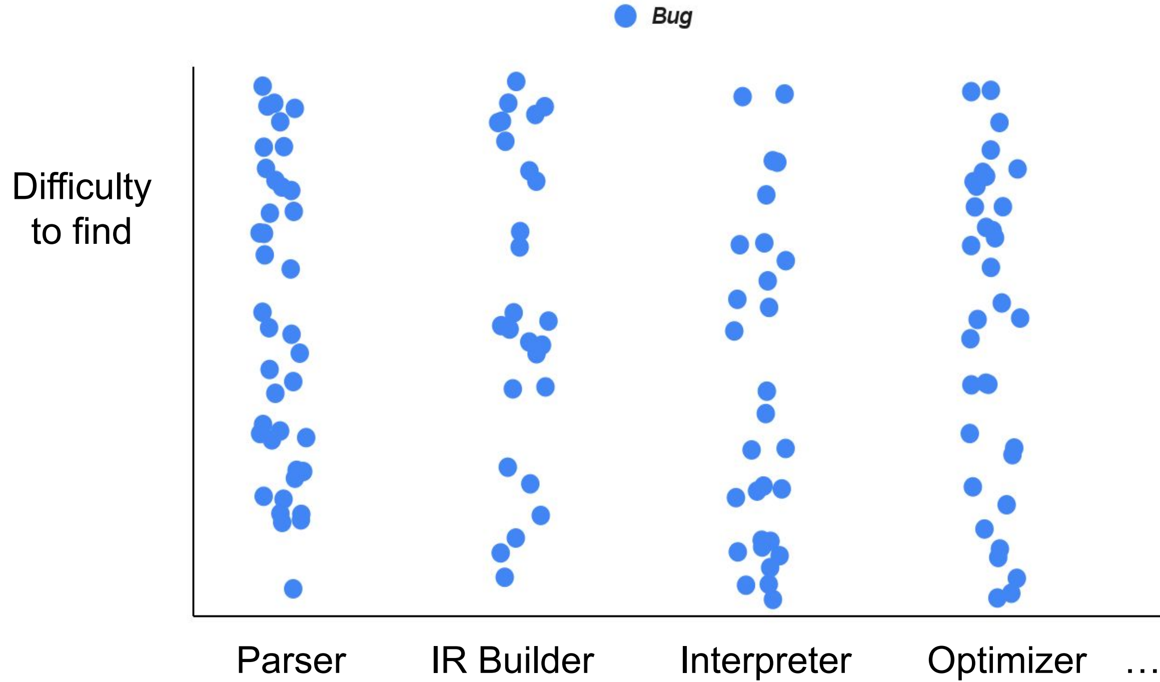
JIT compiler with speculative optimizations



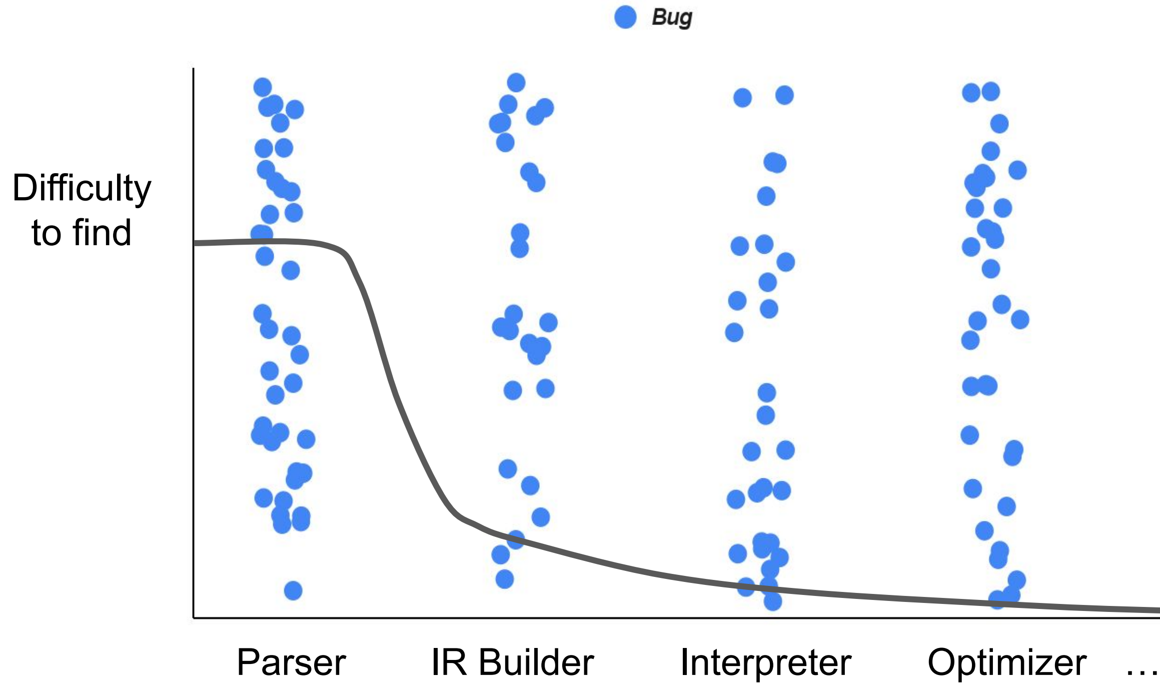
What are people looking at now

- Bugs in all parts of the JIT compiler and engine
- Search keeps finding new bugs
- Like to precisely target important components

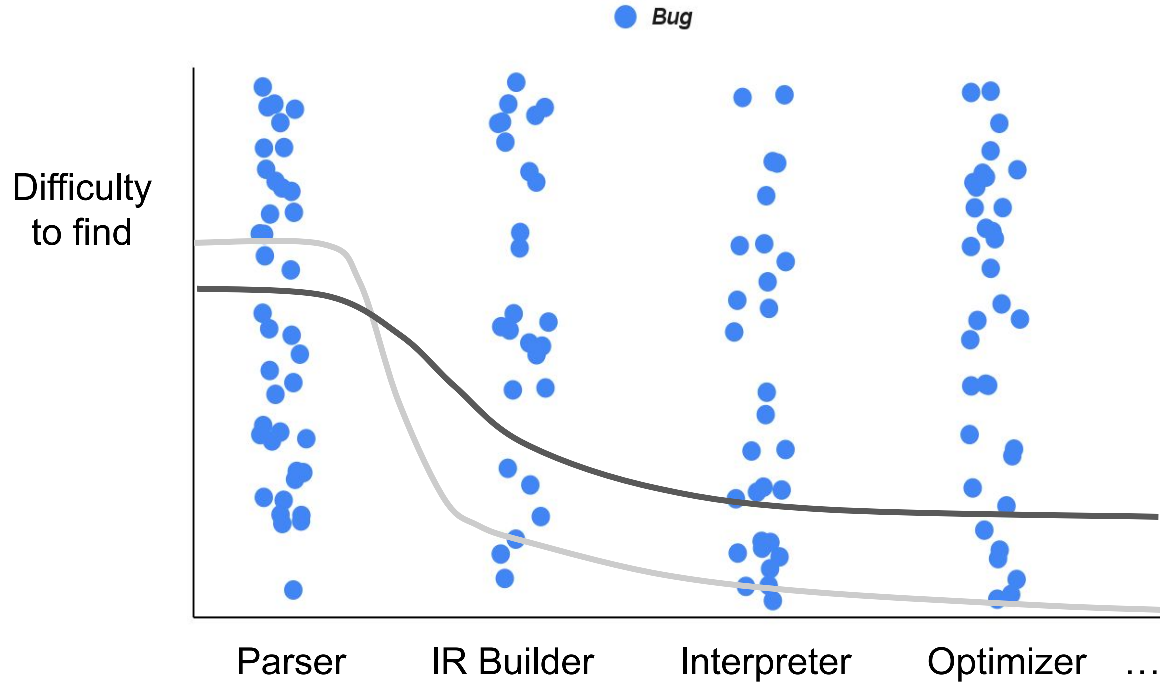
Finding bugs



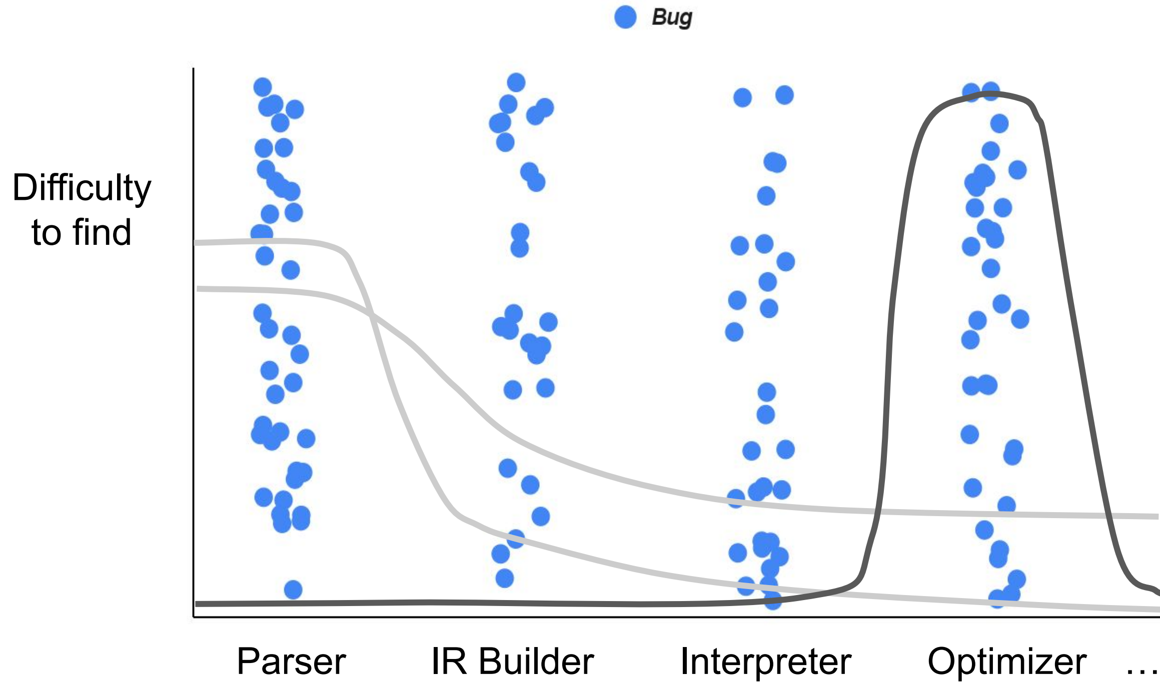
Finding bugs - naive fuzzing



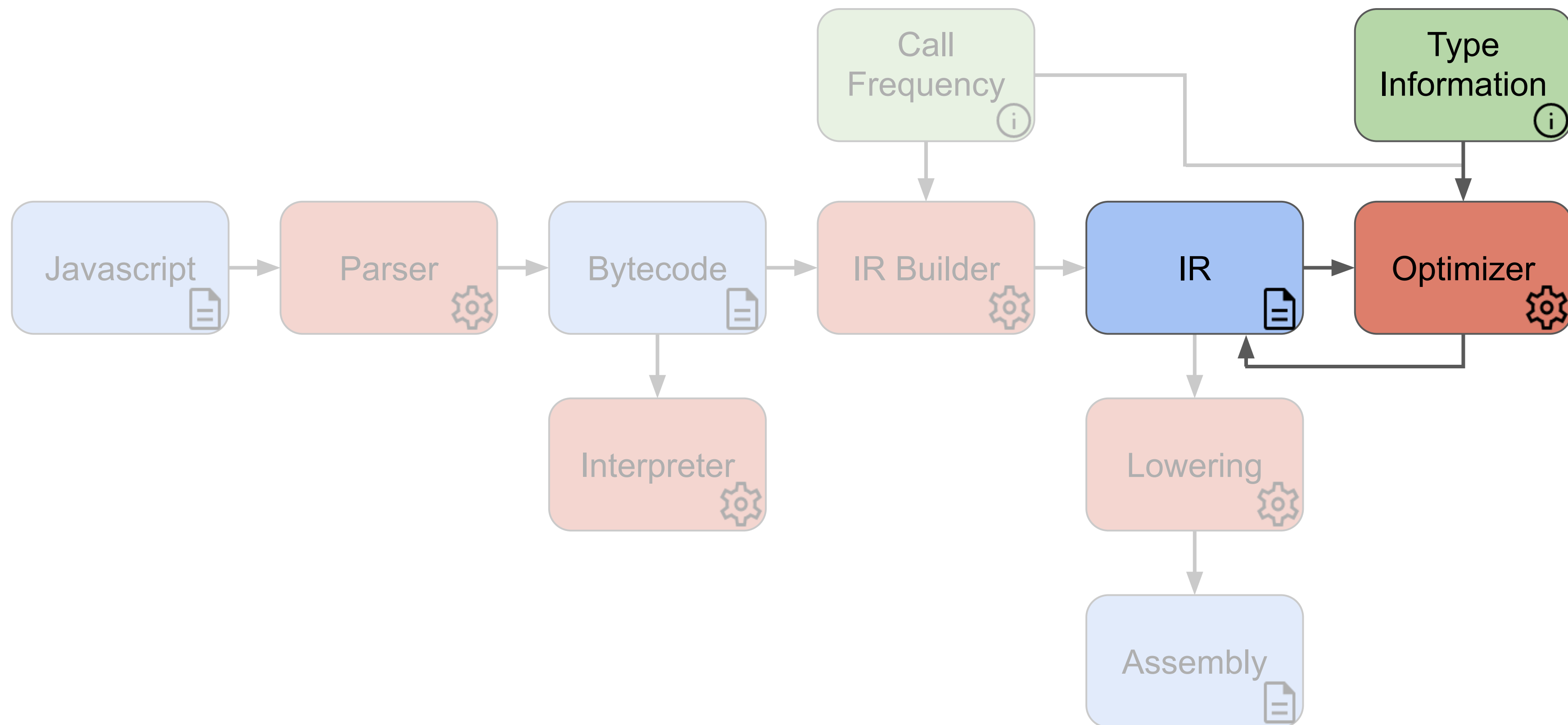
Finding bugs - semantics-aware fuzzing



Finding bugs - our target



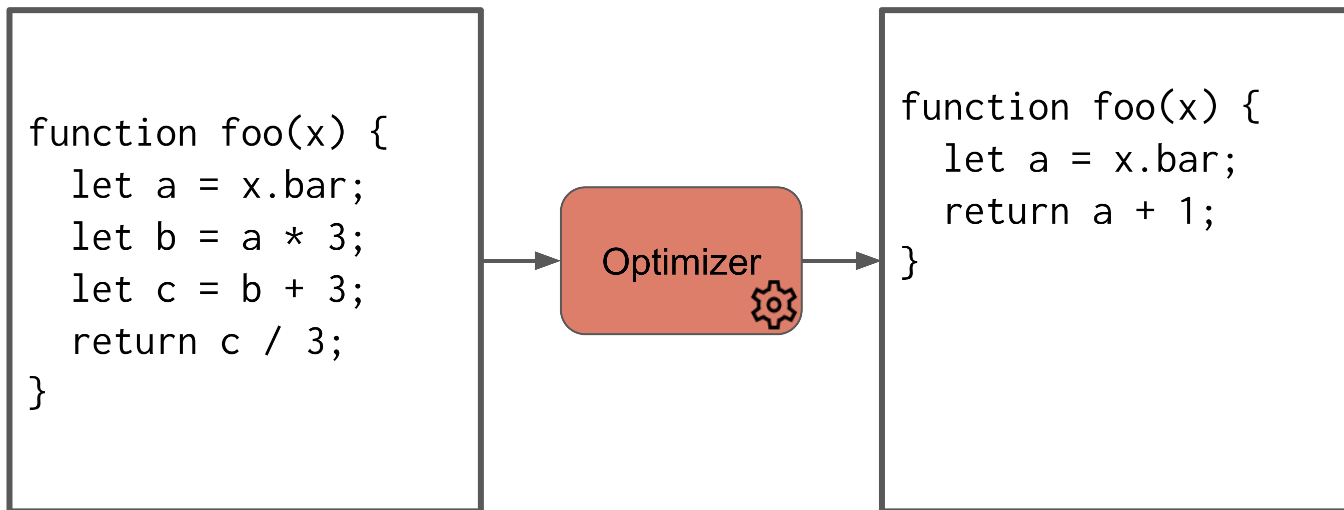
Focus on JIT-specific parts



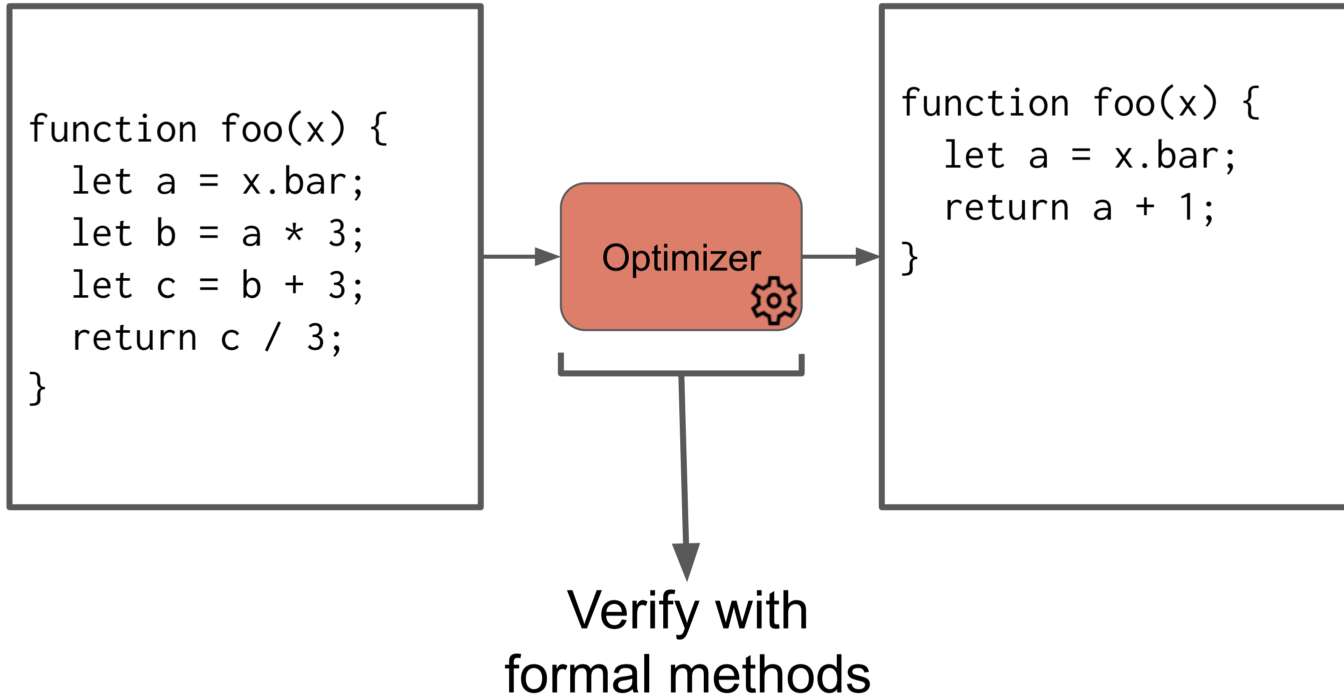
Overview

- **FaJITa translation validation**
- Domain-specific optimizations
- Using FaJITa

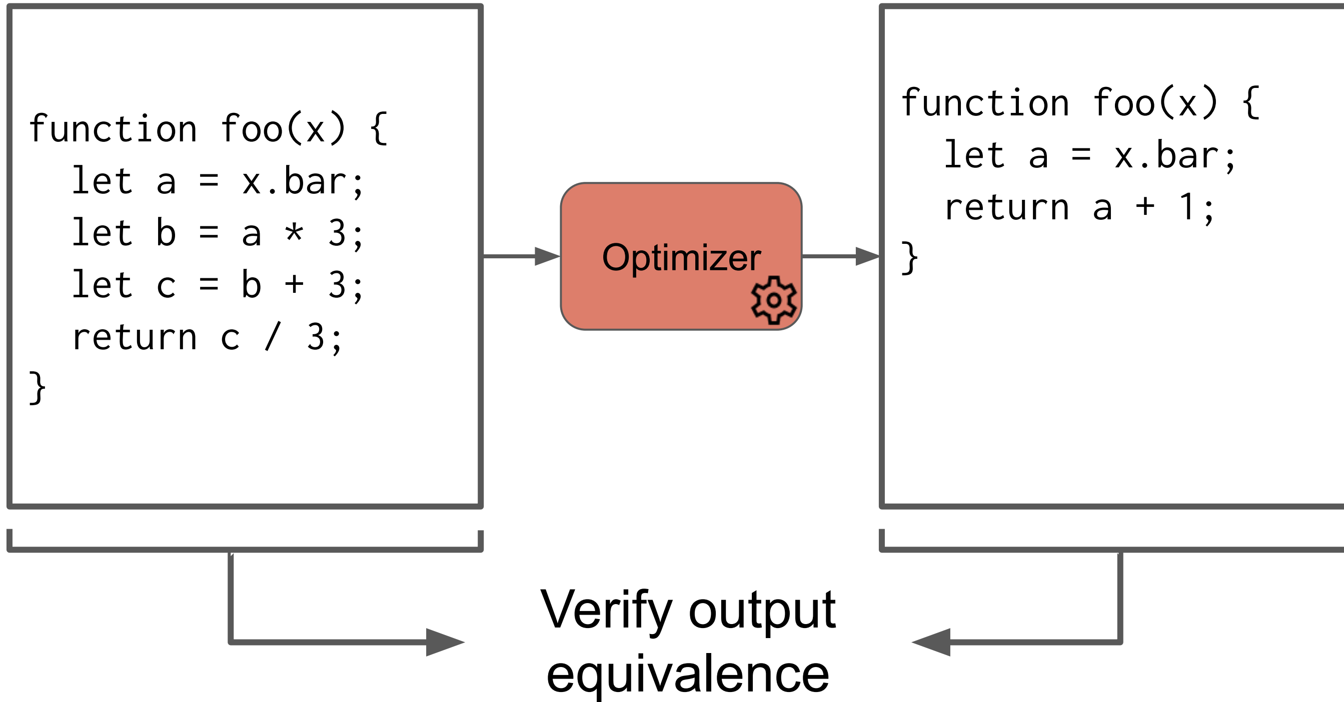
How should we verify the optimizer



How should we verify the optimizer - ideal



How should we verify the optimizer - practical



FaJITa

A translation validation-based verifier for JavaScript JIT optimizations

Using symbolic execution (symex)

- Express program values as set of constraints
- Query SMT solver for properties (verification conditions)

Proving semantic equivalence

JS Program

```
function foo(x) {
  let a = x.bar;
  let b = a * 3;
  let c = b + 3;
  return c / 3;
}
```

```
function foo(x) {
  let a = x.bar;
  return a + 1;
}
```

Proving semantic equivalence

JS Program

```
function foo(x) {
  let a = x.bar;
  let b = a * 3;
  let c = b + 3;
  return c / 3;
}
```

SMT Program Constraints

(assert (a_1 = x_1.bar))
(assert (b_1 = a_1 * 3))
(assert (c_1 = b_1 + 3))
(assert (ret_1 = c_1 / 3))

```
function foo(x) {
  let a = x.bar;
  return a + 1;
}
```

(assert (a_2 = x_2.bar))
(assert (ret_2 = a_2 + 1))

Proving semantic equivalence

JS Program

```
function foo(x) {
  let a = x.bar;
  let b = a * 3;
  let c = b + 3;
  return c / 3;
}
```

SMT Program Constraints

(assert (a_1 = x_1.bar))
(assert (b_1 = a_1 * 3))
(assert (c_1 = b_1 + 3))
(assert (ret_1 = c_1 / 3))

```
function foo(x) {
  let a = x.bar;
  return a + 1;
}
```

(assert (a_2 = x_2.bar))
(assert (ret_2 = a_2 + 1))

Equal Initial Constraints

(assert (x_1 = x_2))
(assert (mem_1 = mem_2))

Proving semantic equivalence

JS Program

```
function foo(x) {
  let a = x.bar;
  let b = a * 3;
  let c = b + 3;
  return c / 3;
}
```

SMT Program Constraints

(assert (a_1 = x_1.bar))
(assert (b_1 = a_1 * 3))
(assert (c_1 = b_1 + 3))
(assert (ret_1 = c_1 / 3))

Equal Initial Constraints

(assert (x_1 = x_2))
(assert (mem_1 = mem_2))

```
function foo(x) {
  let a = x.bar;
  return a + 1;
}
```

(assert (a_2 = x_2.bar))
(assert (ret_2 = a_2 + 1))

Verification Conditions

(assert (ret_1 = ret_2))
(assert (mem_1' = mem_2'))

Overview

- FaJITa
- **Domain-specific optimizations**
- Using FaJITa

The problem with symex

- Optimizations (and our symex) operates on MIR
- 100s of MIR instructions
- Complex semantics w/ JavaScript types
- Interactions with browser

```
function foo(x) {
  let a = x.bar;
  return a + 1;
}
```

```
foo(x)
  a = GetProperty(x, “bar”);
  tmp2 = IntAdd(a, 1);
  Return(tmp2);
```

Translation validation optimizations

- Model subset of semantics we need
- Specialize memory model to domain
- Verify loops with differential assertion checking
- Track allocation scheme
- Model on-stack replacement for multiple function entry points
- …

Translation validation optimizations

- **Model subset of semantics we need**
- **Specialize memory model to domain**
- Verify loops with differential assertion checking
- Track allocation scheme
- Model on-stack replacement for multiple function entry points
- …

Verifying MIR: approximate most semantics

```
function foo(x) {
  let a = x.bar;
  let b = a * 3;
  let c = b + 3;
  return c / 3;
}
```

```
function foo(x) {
  let a = x.bar;
  return a + 1;
}
```

Verifying MIR: approximate most semantics

```
function foo(x) {
  let a = x.bar;
  let b = a * 3;
  let c = b + 3;
  return c / 3;
}
```

```
function foo(x) {
  let a = x.bar;
  return a + 1;
}
```

```
foo(x)
  tmp1 = NurseryObject();
  a = GetProperty(x, “bar”);
  b = IntMultiply(a, 3);
  c = IntAdd(b, 3);
  tmp2 = IntDivide(c, 3);
  Return(tmp2);
```

```
foo(x)
  tmp1 = NurseryObject();
  a = GetProperty(x, “bar”);
  tmp2 = IntAdd(a, 1);
  Return(tmp2);
```

Verifying MIR: approximate most semantics

```
function foo(x) {
  let a = x.bar;
  let b = a * 3;
  let c = b + 3;
  return c / 3;
}
```

```
foo(x)
  tmp1 = NurseryObject();
  a = GetProperty(x, “bar”);
  b = IntMultiply(a, 3);
  c = IntAdd(b, 3);
  tmp2 = IntDivide(c, 3);
  Return(tmp2);
```

```
function foo(x) {
  let a = x.bar;
  return a + 1;
}
```

```
foo(x)
  tmp1 = NurseryObject();
  a = GetProperty(x, “bar”);
  tmp2 = IntAdd(a, 1);
  Return(tmp2);
```

Verifying MIR: approximate most semantics

```
function foo(x) {
  let a = x.bar;
  let b = a * 3;
  let c = b + 3;
  return c / 3;
}
```

```
foo(x)
  tmp1 = NurseryObject();
  a = GetProperty(x, “bar”);
  b = IntMultiply(a, 3);
  c = IntAdd(b, 3);
  tmp2 = IntDivide(c, 3);
  Return(tmp2);
```

```
function foo(x) {
  let a = x.bar;
  return a + 1;
}
```

```
foo(x)
  tmp1 = NurseryObject();
  a = GetProperty(x, “bar”);
  tmp2 = IntAdd(a, 1);
  Return(tmp2);
```

Verifying MIR: approximate most semantics

```
function foo(x) {
  l
  l
  l
  r
}

foo
  tr
  a
  b
  c = IntAdd(b, 3);
  tmp2 = IntDivide(c, 3);
  Return(tmp2);
```

```
function foo(x) {

  Return(tmp2);
```

What does `NurseryObject` do?

- Might modify memory
- Produce some value
- Does the same thing for the same inputs and memory state

Verifying MIR: approximate most semantics

```
function foo(x) {
  let a = x.bar;
  let b = a * 3;
  let c = b + 3;
  return c / 3;
}
```

```
function foo(x) {
  let a = x.bar;
  return a + 1;
}
```

```
foo(x)
  tmp1 = NurseryObject();
  a = GetProperty(x, “bar”);
  b = IntMultiply(a, 3);
  c = IntAdd(b, 3);
  tmp2 = IntDivide(c, 3);
  Return(tmp2);
```

```
foo(x)
  tmp1 = NurseryObject();
  a = GetProperty(x, “bar”);
  tmp2 = IntAdd(a, 1);
  Return(tmp2);
```

Verifying MIR: approximate most semantics

```
function foo(x) {
  let a = x.bar;
  let b = a * 3;
  let c = b + 3;
  return c / 3;
}
```

```
foo(x)
  tmp1 = NurseryObject();
  a = GetProperty(x, “bar”);
  b = IntMultiply(a, 3);
  c = IntAdd(b, 3);
  tmp2 = IntDivide(c, 3);
  Return(tmp2);
```

```
function foo(x) {
  let a = x.bar;
  return a + 1;
}
```

```
foo(x)
  tmp1 = NurseryObject();
  a = GetProperty(x, “bar”);
  tmp2 = IntAdd(a, 1);
  Return(tmp2);
```

(assert ($tmp2_1$ = $tmp2_2$))
(assert (mem_1’ = mem_2’))

Benefits to approximating the semantics

- Get sound approximations of 100s of instructions for free
- Easy to refine where we need more precise semantics
- Allows the SMT solver to handle reordering automatically

Optimizing the memory model for symex

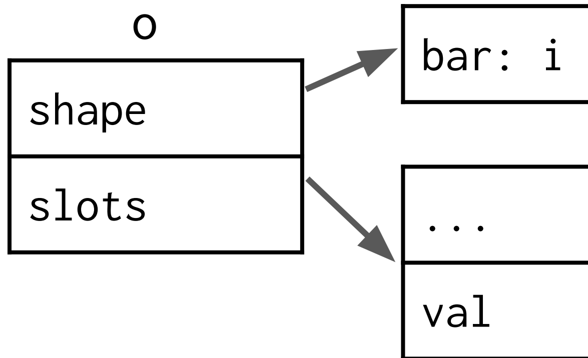
- Historically issue for symex
- Many layers of indirection
- Take advantage of problem structure
- Try to do things statically rather than symbolically

Memory model

```
var o = {... bar: val, ...};
```

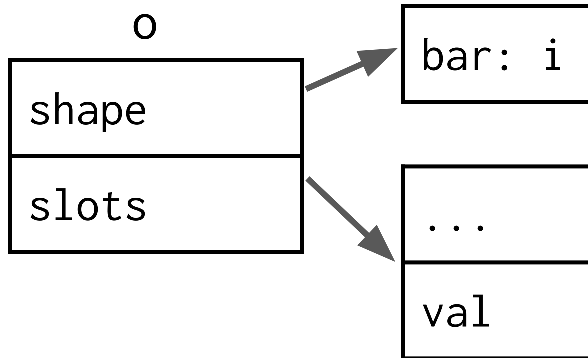
Memory model

```
var o = {... bar: val, ...};
```



Memory model

```
var o = {... bar: val, ...};
```



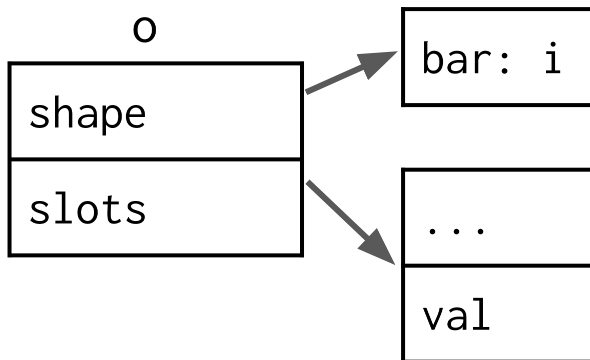
Traditional models:

- Symbolic obj location
- Symbolic field offsets
- Symbolic field values

Timeout after 1 hour to verify

Memory model

```
var o = {... bar: val, ...};
```



Traditional models:

- **Statically known** obj location
- **Statically known** field offsets
- Symbolic field values

<1 second to verify

Optimizations make verification tractable

- Can verify all programs in Firefox’s test suite
- Most take well under a second
- Low false positive rate (<.01% of programs);
- Detect when we need more precise semantics (<.01% of programs)

Overview

- FaJITa
- Domain-specific optimizations
- **Using FaJITa**

FaJITa as bug finding tool

- Use as additional test for fuzzing programs
- Collect all versions of a function during execution
- Use metrics for programs to guide fuzzing

Guided fuzzing

- Existing fuzzers don’t search our domain well
- Try to guide search to interesting optimizations
- Exploring different metrics
- Talk to me if you have ideas for precise fuzzing

FaJITa for online eventual verification

- To slow for pre-execution verification
- Verify programs off-thread instead
- Built on top of Prefect - designed for this style of runtime verification

Takeaway

Large-scale real-world browser component verification is feasible

by taking advantage of domain-specific optimizations