Towards a Verified Range Analysis for JavaScript JITs

Steve Gustaman

Original paper by Brown et al.
PLDI 2020

Motivation

Motivation

- **Browser JITs compile and execute web JS code**

Motivation

- **Browser JITs compile and execute web JS code**
  - Crucial for browser to be fast

Motivation

- **Browser JITs compile and execute web JS code**
    - Crucial for browser to be fast
    - Fast = more complex

Motivation

- **Browser JITs compile and execute web JS code**
  - Crucial for browser to be fast
  - Fast = more complex
  - More complex = more potential bugs

Motivation

- **Browser JITs compile and execute web JS code**
  - Crucial for browser to be fast
  - Fast = more complex
  - More complex = more potential bugs → **miscompilation**

Motivation

- **Browser JITs mis compile and execute web JS code**
  - Crucial for browser to be fast
  - Fast = more complex
  - More complex = more potential bugs → **miscompilation**

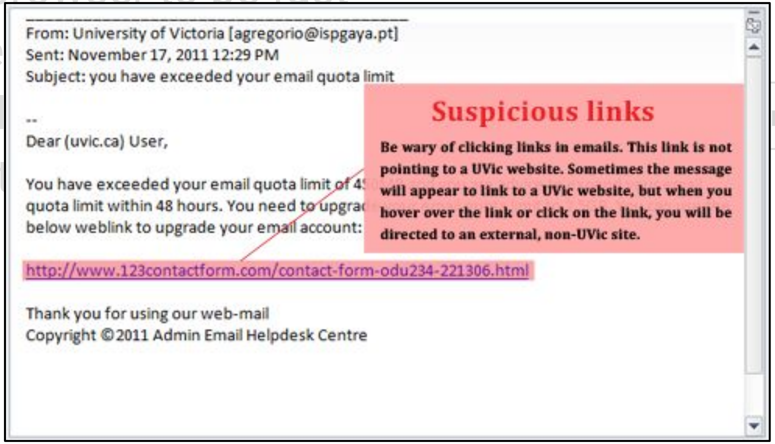
Motivation

- **Browser JITs <u>mis</u>compile and execute web JS code**
  - Crucial for browser to be fast
  - Fast = more complex
  - More complex = more potential bugs → **miscompilation**
  - More potential bugs = **<u>more security problems</u>**

Motivation

- **Browser JITs miscompile and execute web JS code**
  - Crucial for browser to be fast
  - Fast = more
  - More compl
  - More potent

mpilation



From: University of Victoria [agregorio@ispgaya.pt]
Sent: November 17, 2011 12:29 PM
Subject: you have exceeded your email quota limit

--
Dear (uvic.ca) User,

You have exceeded your email quota limit of 4
quota limit within 48 hours. You need to upgrad
below weblink to upgrade your email account:

http://www.123contactform.com/contact-form-odu234-221306.html

Thank you for using our web-mail
Copyright ©2011 Admin Email Helpdesk Centre

https://www.uvic.ca/systems/status/features/phishing.php

Motivation

- **Browser JITs mis compile and execute web JS code**
  - Crucial for browser to be fast
  - Fast = more complex
  - More complex = more potential bugs → **miscompilation**
  - More potential bugs = **more security problems**

Motivation

- **Browser JITs can be forced to miscompile and execute malicious web JS code**
    - Crucial for browser to be fast
    - Fast = more complex
    - More complex = more potential bugs → miscompilation
    - More potential bugs = **more security problems**

Motivation

- Browser JITs can be forced to miscompile and execute malicious web JS code
- **Every JIT compilation is security critical**

Motivation

- Browser JITs can be forced to miscompile and execute malicious web JS code
- **Every JIT compilation is security critical**
    - Buggy JIT compilation

Motivation

- Browser JITs can be forced to miscompile and execute malicious web JS code
- **Every JIT compilation is security critical**
    - Buggy JIT compilation → maliciously crafted JS

Motivation

- Browser JITs can be forced to miscompile and execute malicious web JS code
- **Every JIT compilation is security critical**
  - Buggy JIT compilation → maliciously crafted JS → security problem

Motivation

- Bro… …JS code
- Eve…
  - … …em

SECURITY

This article is more than **1 year old**

Anatomy of an attack: How Coinbase was targeted with emails booby-trapped with Firefox zero-days

32

Elaborate browser break-out betrayed by unusual behavior

Thomas Claburn — Fri 9 Aug 2019 // 23:56 UTC

Coinbase chief information security officer Philip Martin this week published an incident report covering the recent attack on the cryptocurrency exchange, revealing a phishing campaign of surprising sophistication.

The thwarted attack began with email messages on May 30 to more than a dozen Coinbase employees that appeared to be from Gregory Harris, a research grant administrator at the University of Cambridge in the UK.

At some point prior to that, the attackers – a group known to Coinbase as CRYPTO-3 or sometimes HYDSEVEN – compromised or created two email accounts at Cambridge.

https://www.theregister.com/2019/08/09/coinbase_pwned/

Motivation

- Browser JITs can be forced to miscompile and execute malicious web JS code
- **Every** …
  - B… …blem

code itself was well-structured, as might be expected from experienced malware authors.

Using those two vulnerabilities to achieve arbitrary code execution, the attacker's shellcode issued a curl command to download and run the stage-one implant, a Netwire variant. Used for reconnaissance and credential theft on victims' machines, the malicious code was detected by Coinbase at this point based on unusual behavior, specifically Firefox spawning a shell.

The stage-one payload then transitioned to a stage two implant, identified by Martin as a variant of the Mokes malware family. It's a remote access trojan (RAT) and was operated under direct human control. Martin speculates that the attackers moved to stage two when they believed they had compromised a target of value.

Once aware of the hack, Coinbase's security team collected data artifacts related to the break-in, revoked affected credentials, and contacted Mozilla's security team, which

https://www.theregister.com/2019/08/09/coinbase_pwned/

Motivation

- Browser JITs can be forced to miscompile and execute malicious web JS code
- **Every** ...
  - B... ...blem

code itself was well-structured, as might be expected from experienced malware authors.

Using those two vulnerabilities to achieve arbitrary code execution, the attacker's shellcode issued a curl command to download and run the stage-one implant, a Netwire variant. Used for reconnaissance and credential theft on victims' machines, the malicious code was detected by Coinbase at this point based on unusual behavior, specifically Firefox spawning a shell.

The stage-one payload then transitioned to a stage two implant, identified by Martin as a variant of the Mokes malware family. It's a remote access trojan (RAT) and was operated under direct human control. Martin speculates that the attackers moved to stage two when they believed they had compromised a target of value.

Once aware of the hack, Coinbase's security team collected data artifacts related to the break-in, revoked affected credentials, and contacted Mozilla's security team, which

https://www.theregister.com/2019/08/09/coinbase_pwned/

Motivation

- Browser JITs can be forced to miscompile and execute malicious web JS code
- **Every JIT compilation is security critical**

Motivation

- Browser JITs can be forced to miscompile and execute malicious web JS code
- **Every JIT compilation is security critical**

How do JIT bugs happen?

JIT Bugs

JIT Bugs

- Javascript is memory-safe language

JIT Bugs

- Javascript is memory-safe language
    - Array bounds checking on every array access

JIT Bugs

- Javascript is memory-safe language
  - Array bounds checking on every array access

```
x = ...
x = x+1
arr = [0,1,2,3,4]


return arr[x]
```

JIT Bugs

- Javascript is memory-safe language
  - Array bounds checking on every **array access**

```
x = ...
x = x+1
arr = [0,1,2,3,4]


return arr[x]
```

JIT Bugs

- Javascript is memory-safe language
  - Array **bounds checking** on every array access

```
x = ...
x = x+1
arr = [0,1,2,3,4]
if (x >= arr.len)
  return undefined
return arr[x]
```

JIT Bugs

- Javascript is memory-safe language
    - Array bounds checking on **<u>every</u>** array access

```
x = ...
x = x+1
arr = [0,1,2,3,4]
if (x >= arr.len)
  return undefined
return arr[x]
```

JIT Bugs

- Javascript is memory-safe language
  - Array bounds checking on **every** array access → SLOW

```
x = ...
x = x+1
arr = [0,1,2,3,4]
if (x >= arr.len)
  return undefined
return arr[x]
```

JIT Bugs

- Javascript is memory-safe language
  - Array bounds checking on **every** array access → SLOW
  - Use **range analysis** to eliminate bounds check

```
x = ...
x = x+1
arr = [0,1,2,3,4]
if (x >= arr.len)
  return undefined
return arr[x]
```

JIT Bugs

- Javascript is memory-safe language
    - Array bounds checking on **<u>every</u>** array access → SLOW
    - Use **<u>range analysis</u>** to eliminate bounds check

x=[0,3]

```
x = ...
x = x+1
arr = [0,1,2,3,4]
if (x >= arr.len)
  return undefined
return arr[x]
```

JIT Bugs

- Javascript is memory-safe language
  - Array bounds checking on **every** array access → SLOW
  - Use **range analysis** to eliminate bounds check

x=[0,3]

x=[1,4]

```
x = ...
x = x+1
arr = [0,1,2,3,4]
if (x >= arr.len)
  return undefined
return arr[x]
```

JIT Bugs

- Javascript is memory-safe language
  - Array bounds checking on **every** array access → SLOW
  - Use range analysis to **eliminate bounds check**

x=[0,3]

x=[1,4]

```
x = ...
x = x+1
arr = [0,1,2,3,4]
if (x >= arr.len)
  return undefined
return arr[x]
```

JIT Bugs

- Javascript is memory-safe language
  - Array bounds checking on **every** array access → SLOW
  - Use range analysis to **eliminate bounds check**

x=[0,3]

x=[1,4]

***JIT: this is never OOB**

```
x = ...
x = x+1
arr = [0,1,2,3,4]
if (x >= arr.len)
  return undefined
return arr[x]
```

JIT Bugs

- Javascript is memory-safe language
  - Array bounds checking on **every** array access → **SLOW**
  - Use range analysis to **eliminate bounds check**

x=[0,3]

x=[1,4]

***JIT: this is never OOB**

```
x = ...
x = x+1
arr = [0,1,2,3,4]
if (x >= arr.len)
  return undefined
return arr[x]
```

JIT Bugs

- Javascript is memory-safe language
  - Array bounds checking on **every** array access → SLOW
  - Use range analysis to **eliminate bounds check**
- **What will happen if JIT engine range analysis is wrong?**

```
x = ...
x = x+1
arr = [0,1,2,3,4]
if (x >= arr.len)
  return undefined
return arr[x]
```

JIT Bugs

- Javascript is memory-safe language
  - Array bounds checking on **every** array access → SLOW
  - Use range analysis to **eliminate bounds check**
- If range analysis is wrong, can access OOB

```
x = ...
x = x+1
arr = [0,1,2,3,4]
if (x >= arr.len)
  return undefined
return arr[x]
```

JIT Bugs

- Javascript is memory-safe language
    - Array bounds checking on **every** array access → SLOW
    - Use range analysis to **eliminate bounds check**
- If range analysis is wrong, can access OOB

actual: x = 4

```
x = ...
x = x+1
arr = [0,1,2,3,4]
if (x >= arr.len)
  return undefined
return arr[x]
```

JIT Bugs

- Javascript is memory-safe language
  - Array bounds checking on **<u>every</u>** array access → SLOW
  - Use range analysis to **<u>eliminate bounds check</u>**
- If range analysis is wrong, can access OOB

x=[0,3]

*buggy range analysis

actual: x = 4

```
x = ...
x = x+1
arr = [0,1,2,3,4]
if (x >= arr.len)
  return undefined
return arr[x]
```

JIT Bugs

- Javascript is memory-safe language
  - Array bounds checking on **every** array access → **SLOW**
  - Use range analysis to **eliminate bounds check**
- If range analysis is wrong, can access OOB

x=[0,3]

x=[1,4]

***buggy range analysis**

```
x = ...
x = x+1
arr = [0,1,2,3,4]
if (x >= arr.len)
  return undefined
return arr[x]
```

actual: x = 5

JIT Bugs

- Javascript is memory-safe language
  - Array bounds checking on **every** array access → SLOW
  - Use range analysis to **eliminate bounds check**
- If range analysis is wrong, can access OOB

x=[0,3]

x=[1,4]

***buggy JIT: this is never OOB**

```
x = ...
x = x+1
arr = [0,1,2,3,4]
if (x >= arr.len)
  return undefined
return arr[x]
```

actual: x = 5

JIT Bugs

- Javascript is memory-safe language
  - Array bounds checking on **every** array access → SLOW
  - Use range analysis to **eliminate bounds check**
- If range analysis is wrong, can access OOB

x=[0,3]

x=[1,4]

***buggy JIT: this is never OOB**

```
x = ...
x = x+1
arr = [0,1,2,3,4]
if (x >= arr.len)
  return undefined
return arr[x]
```

actual: x = 5

JIT Bugs

- Javascript is memory-safe language
  - Array bounds checking on **<u>every</u>** array access → SLOW
  - Use range analysis to **<u>eliminate bounds check</u>**
- If range analysis is wrong, can access OOB

x=[0,3]

x=[1,4]

***buggy JIT: this is never OOB**

x = ...
x = x+1 → actual: x = 5
arr = [0,1,2,3,4]
~~if (x >= arr.len)~~
~~return undefined~~
return **<u>arr[x]</u>**

Motivation

- Browser JITs can be forced to miscompile and execute malicious web JS code
- **Every JIT compilation is security critical**

Motivation

- Browser JITs can be forced to miscompile and execute malicious web JS code
- **JIT engine range analysis' correctness is critical**

Motivation

- Browser JITs can be forced to miscompile and execute malicious web JS code
- **JIT engine range analysis’ correctness is critical**

How to verify JIT engine
range analysis’ correctness?

VeRA:

a system for verifying the range analysis pass in browser JIT compilers

VeRA Key Ideas

- **Goal**: Verify correctness of JIT engine range analysis

VeRA Key Ideas

- **Goal**: In actual JS semantics, is it possible to get a value outside of JIT engine's range?

VeRA Key Ideas

- **Goal**: In actual JS semantics, is it possible to get a value outside of JIT engine’s range?
- **Key Ideas**:

VeRA Key Ideas

- **Goal**: In actual JS semantics, is it possible to get a value outside of JIT engine's range?
- **Key Ideas**:
  1. **Encode actual JS semantics in SMT**

VeRA Key Ideas

- **Goal**: In actual JS semantics, is it possible to get a value outside of JIT engine's range?
- **Key Ideas**:
  1. Encode actual JS semantics in SMT
  2. **Encode JIT engine range analysis routine in SMT**

VeRA Key Ideas

- **Goal**: In actual JS semantics, is it possible to get a value outside of JIT engine's range?
- **Key Ideas**:
  1. Encode actual JS semantics in SMT
  2. Encode JIT engine range analysis routine in SMT
  3. **Ask SMT solver: is it possible to get value from (1) outside of (2)?**

VeRA Key Ideas

utside of JIT

tside of (2)?

```
// exponent computation have to be over-estimations of the actual result. O
// the Int32 this over approximation is rectified.

MOZ_INIT_OUTSIDE_CTOR int32_t lower_;
MOZ_INIT_OUTSIDE_CTOR int32_t upper_;

MOZ_INIT_OUTSIDE_CTOR bool hasInt32LowerBound_;
MOZ_INIT_OUTSIDE_CTOR bool hasInt32UpperBound_;

MOZ_INIT_OUTSIDE_CTOR FractionalPartFlag canHaveFractionalPart_ : 1;
MOZ_INIT_OUTSIDE_CTOR NegativeZeroFlag canBeNegativeZero_ : 1;
MOZ_INIT_OUTSIDE_CTOR uint16_t max_exponent_;

// Any symbolic lower or upper bound computed for this term.
const SymbolicBound* symbolicLower_;
const SymbolicBound* symbolicUpper_;
```

`integer, float, NaN, inf, 0, -0, ...`

Firefox JIT Range Analysis

```
// exponent computation have to be over-estimations of the actual result. O
// the Int32 this over approximation is rectified.

MOZ_INIT_OUTSIDE_CTOR int32_t lower_;
MOZ_INIT_OUTSIDE_CTOR int32_t upper_;

MOZ_INIT_OUTSIDE_CTOR bool hasInt32LowerBound_;
MOZ_INIT_OUTSIDE_CTOR bool hasInt32UpperBound_;

MOZ_INIT_OUTSIDE_CTOR FractionalPartFlag canHaveFractionalPart_ : 1;
MOZ_INIT_OUTSIDE_CTOR NegativeZeroFlag canBeNegativeZero_ : 1;
MOZ_INIT_OUTSIDE_CTOR uint16_t max_exponent_;

// Any symbolic lower or upper bound computed for this term.
const SymbolicBound* symbolicLower_;
const SymbolicBound* symbolicUpper_;
```

x=[lower,upper]

integer, float, NaN, inf, 0, -0, ...

Firefox JIT Range Analysis

VeRA Key Ideas

```
// exponent computation have to be over-estimations of the actual result. O
// the Int32 this over approximation is rectified.

MOZ_INIT_OUTSIDE_CTOR int32_t lower_;
MOZ_INIT_OUTSIDE_CTOR int32_t upper_;

MOZ_INIT_OUTSIDE_CTOR bool hasInt32LowerBound_;
MOZ_INIT_OUTSIDE_CTOR bool hasInt32UpperBound_;

MOZ_INIT_OUTSIDE_CTOR FractionalPartFlag canHaveFractionalPart_ : 1;
MOZ_INIT_OUTSIDE_CTOR NegativeZeroFlag canBeNegativeZero_ : 1;
MOZ_INIT_OUTSIDE_CTOR uint16_t max_exponent_;

// Any symbolic lower or upper bound computed for this term.
const SymbolicBound* symbolicLower_;
const SymbolicBound* symbolicUpper_;
```

utside of JIT

tside of (2)?

```
integer, float, NaN, inf, 0, -0, ...
```

```cpp
Range* Range::abs(TempAllocator& alloc, const Range* op) {
  int32_t l = op->lower_;
  int32_t u = op->upper_;
  FractionalPartFlag canHaveFractionalPart = op->canHaveFractionalPart_;

  // Abs never produces a negative zero.
  NegativeZeroFlag canBeNegativeZero = ExcludesNegativeZero;

  return new (alloc) Range(
      std::max(std::max(int32_t(0), l), u == INT32_MIN ? INT32_MAX : -u), true,
      std::max(std::max(int32_t(0), u), l == INT32_MIN ? INT32_MAX : -l),
      op->hasInt32Bounds() && l != INT32_MIN, canHaveFractionalPart,
      canBeNegativeZero, op->max_exponent_);
}
```

flow function for:

<u>**`x = abs(y)`**</u>

Firefox JIT Range Analysis

VeRA Key Ideas

```
// the Int32 this over approximation is rectified.

MOZ_INIT_OUTSIDE_CTOR int32_t lower_;
MOZ_INIT_OUTSIDE_CTOR int32_t upper_;

MOZ_INIT_OUTSIDE_CTOR
MOZ_INIT_OUTSIDE_CTOR

MOZ_INIT_OUTSIDE_CTOR
MOZ_INIT_OUTSIDE_CTOR
MOZ_INIT_OUTSIDE_CTOR

// Any symbolic lower
const SymbolicBound*
```

alue outside of JIT

integer, floa

```
Range* Range::abs(TempAllocator& alloc, const Range* op) {
  int32_t l = op->lower_;
  int32_t u = op->upper_;
  FractionalPartFlag canHaveFractionalPart = op->canHaveFractionalPart_;

  // Abs never produces a negative zero.
  NegativeZeroFlag canBeNegativeZero = ExcludesNegativeZero;

  return new (alloc) Range(
      std::max(std::max(int32_t(0), l), u == INT32_MIN ? INT32_MAX : -u), true,
      std::max(std::max(int32_t(0), u), l == INT32_MIN ? INT32_MAX : -l),
      op->hasInt32Bounds() && l != INT32_MIN, canHaveFractionalPart,
      canBeNegativeZero, op->max_exponent_);
}
```

y Range

flow function for:

<u>**x = abs(y)**</u>

Firefox JIT Range Analysis

```
Range* Range::abs(TempAllocator& alloc, const Range* op) {
  int32_t l = op->lower_;
  int32_t u = op->upper_;
  FractionalPartFlag canHaveFractionalPart = op->canHaveFractionalPart_;

  // Abs never produces a negative zero.
  NegativeZeroFlag canBeNegativeZero = ExcludesNegativeZero;

  return new (alloc) Range(
      std::max(std::max(int32_t(0), l), u == INT32_MIN ? INT32_MAX : -u), true,
      std::max(std::max(int32_t(0), u), l == INT32_MIN ? INT32_MAX : -l),
      op->hasInt32Bounds() && l != INT32_MIN, canHaveFractionalPart,
      canBeNegativeZero, op->max_exponent_);
}
```

x Range

flow function for:

x = abs(y)

Firefox JIT Range Analysis

VeRA Key Ideas

```
// exponent computation have to be over-estimations of the actual result. O
// the Int32 this over approximation is rectified.

MOZ_INIT_OUTSIDE_CTOR int32_t lower_;
MOZ_INIT_OUTSIDE_CTOR int32_t upper_;

MOZ_INIT_OUTSIDE_CTOR
MOZ_INIT_OUTSIDE_CTOR

MOZ_INIT_OUTSIDE_CTOR
MOZ_INIT_OUTSIDE_CTOR
MOZ_INIT_OUTSIDE_CTOR

// Any symbolic lower
const SymbolicBound* s
```

alue outside of JIT

integer, floa

```
Range* Range::abs(TempAllocator& alloc, const Range* op) {
  int32_t l = op->lower_;
  int32_t u = op->upper_;
  FractionalPartFlag canHaveFractionalPart = op->canHaveFractionalPart_;

  // Abs never produces a negative zero.
  NegativeZeroFlag canBeNegativeZero = ExcludesNegativeZero;

  return new (alloc) Range(
      std::max(std::max(int32_t(0), l), u == INT32_MIN ? INT32_MAX : -u), true,
      std::max(std::max(int32_t(0), u), l == INT32_MIN ? INT32_MAX : -l),
      op->hasInt32Bounds() && l != INT32_MIN, canHaveFractionalPart,
      canBeNegativeZero, op->max_exponent_);
}
```

flow function for:

`x = abs(y)`

VeRA Key Ideas

- **Goal**: In actual JS semantics, is it possible to get a value outside of JIT engine’s range?
- **Key Ideas**:
  1. Encode actual JS semantics in SMT
  2. Encode JIT engine range analysis routine in SMT
  3. Ask SMT solver: is it possible to get value from (1) outside of (2)?

VeRA Key Ideas

- **Goal**: In actual JS semantics, is it possible to get a value outside of JIT engine's range?
- **Key Ideas**:
  1. Encode actual JS semantics in SMT
  2. Encode JIT engine range analysis routine in SMT → **COMPLEX**
  3. Ask SMT solver: is it possible to get value from (1) outside of (2)?

VeRA Key Ideas

- **<u>Goal</u>**: In actual JS semantics, is it possible to get a value outside of JIT engine's range?
- **<u>Key Ideas</u>**:
  1. Encode actual JS semantics in SMT
  2. ~~Encode JIT engine range analysis routine in SMT~~ → COMPLEX
  3. Ask SMT solver: is it possible to get value from (1) outside of (2)?

VeRA Key Ideas

- **Goal**: In actual JS semantics, is it possible to get a value outside of JIT engine's range?
- **Key Ideas**:
  1. Encode actual JS semantics in SMT
  2. Automatically generate JIT engine range analysis routine in SMT
  3. Ask SMT solver: is it possible to get value from (1) outside of (2)?

VeRA in Action – `ceil`

VeRA in Action – `ceil`

round up value to
nearest integer

VeRA in Action – `ceil`

1. Encode actual JS semantics in SMT
2. Automatically generate JIT engine range analysis routine in SMT
3. Ask SMT solver: is it possible to get value from (1) outside of (2)?

VeRA in Action – `ceil`

1. **Encode actual JS semantics in SMT**
2. Automatically generate JIT engine range analysis routine in SMT
3. Ask SMT solver: is it possible to get value from (1) outside of (2)?

VeRA in Action – `ceil`

1. **Encode actual JS semantics in SMT**
   - Trivial: use SMT ceil operator
2. Automatically generate JIT engine range analysis routine in SMT
3. Ask SMT solver: is it possible to get value from (1) outside of (2)?

VeRA in Action – `ceil`

1. Encode actual JS semantics in SMT
    - Trivial: use SMT ceil operator
2. **Automatically generate JIT engine range analysis routine in SMT**
3. Ask SMT solver: is it possible to get value from (1) outside of (2)?

VeRA in Action – `ceil`

1. Encode actual JS semantics in SMT
    - Trivial: use SMT ceil operator
2. **Automatically generate JIT engine range analysis routine in SMT**
    - Rewrite JIT engine range analysis logic in **VeRA C++**
3. Ask SMT solver: is it possible to get value from (1) outside of (2)?

VeRA in Action – `ceil`

1. Encode actual JS semantics in SMT
    - Trivial: use SMT ceil operator
2. **Automatically generate JIT engine range analysis routine in SMT**
    - Rewrite JIT engine range analysis logic in **VeRA C++**
        - Subset of C++ specific for this problem
3. Ask SMT solver: is it possible to get value from (1) outside of (2)?

VeRA in Action – `ceil`

1. Encode actual JS semantics in SMT
    - Trivial: use SMT ceil operator
2. **Automatically generate JIT engine range analysis routine in SMT**
    - Rewrite JIT engine range analysis logic in **VeRA C++**
        - Subset of C++ specific for this problem
3. Ask SMT solver: is it possible to get value from (1) outside of (2)?

disallows several construct e.g. loops

VeRA in Action – `ceil`

1. Encode actual JS semantics in SMT
    - Trivial: use SMT ceil operator
2. **Automatically generate JIT engine range analysis routine in SMT**
    - Rewrite JIT engine range analysis logic in **VeRA C++**
        - Subset of C++ specific for this problem
3. Ask SMT solver: is it possible to get value from (1) outside of (2)?

`ceil` range flow function

```cpp
Range*
Range::ceil(TempAllocator& alloc, const Range* op)
{
    Range* copy = new(alloc) Range(*op);

    // We need to refine max_exponent_ because ceil may have incremente
    // If we have got int32 bounds defined, just deduce it using the de
    // Else we can just increment its value,
    // as we are looking to maintain an over estimation.
    if (copy->hasInt32Bounds())
        copy->max_exponent_ = copy->exponentImpliedByInt32Bounds();
    else if (copy->max_exponent_ < MaxFiniteExponent)
        copy->max_exponent_++;

    copy->canHaveFractionalPart_ = ExcludesFractionalParts;
    copy->assertInvariants();
    return copy;
}
```

Firefox original implementation (C++)

e analysis routine in SMT

ic in VeRA C++

oblem

e from (1) outside of (2)?

`ceil` range flow function

```cpp
Range*
Range::ceil(TempAllocator& alloc, const Range* op)
{
    Range* copy = new(alloc) Range(*op);

    // We need to refine max_exponent_ because ceil may have incre
    // If we have got int32 bounds defined, just deduce it using t
    // Else we can just increment its value,
    // as we are looking to maintain an over estimation.
    if (copy->hasInt32Bounds())
        copy->max_exponent_ = copy->exponentImpliedByInt32Bounds()
    else if (copy->max_exponent_ < MaxFiniteExponent)
        copy->max_exponent_++;

    copy->canHaveFractionalPart_ = ExcludesFractionalParts;
    copy->assertInvariants();
    return copy;
}
```

Firefox original implementation (C++)

```cpp
range ceil(range const& op) {
  range copy = op;

  // missing fract check

  if (hasInt32Bounds(copy)) {
    copy.maxExponent = exponentImpliedByInt32Bounds(copy);
  } else if (copy.maxExponent < maxFiniteExponentS) {
    copy.maxExponent += (uint16_t) 1;
  }

  copy.canHaveFractionalPart = excludesFractionalPartsS;
  return copy;
}
```

Rewritten (VeRA C++)

VeRA in Action – `ceil`

1. Encode actual JS semantics in SMT
    - Trivial: use SMT ceil operator
2. **Automatically generate JIT engine range analysis routine in SMT**
    - Rewrite JIT engine range analysis logic in **VeRA C++**
        - Subset of C++ specific for this problem
3. Ask SMT solver: is it possible to get value from (1) outside of (2)?

VeRA in Action – `ceil`

1. Encode actual JS semantics in SMT
   - Trivial: use SMT ceil operator
2. **Automatically generate JIT engine range analysis routine in SMT**
   - Rewrite JIT engine range analysis logic in **VeRA C++**
     - Subset of C++ specific for this problem
   - Vera C++
3. Ask SMT solver: is it possible to get value from (1) outside of (2)?

VeRA in Action – `ceil`

1. Encode actual JS semantics in SMT
    - Trivial: use SMT ceil operator
2. **Automatically generate JIT engine range analysis routine in SMT**
    - Rewrite JIT engine range analysis logic in **VeRA C++**
        - Subset of C++ specific for this problem
    - Vera C++ → IR (SMT compatible SSA)
3. Ask SMT solver: is it possible to get value from (1) outside of (2)?

VeRA in Action – `ceil`

1. Encode actual JS semantics in SMT
    - Trivial: use SMT ceil operator
2. **Automatically generate JIT engine range analysis routine in SMT**
    - Rewrite JIT engine range analysis logic in **VeRA C++**
        - Subset of C++ specific for this problem
    - Vera C++ → IR (SMT compatible SSA) → **SMT**
3. Ask SMT solver: is it possible to get value from (1) outside of (2)?

VeRA in Action – `ceil`

1. Encode actual JS semantics in SMT ✔
    - Trivial: use SMT ceil operator
2. Automatically generate JIT engine range analysis routine in SMT ✔
    - Rewrite JIT engine range analysis logic in **VeRA C++**
        - Subset of C++ specific for this problem
    - Vera C++ → IR (SSA) → **SMT**
3. Ask SMT solver: is it possible to get value from (1) outside of (2)?

VeRA in Action – `ceil`

1. Encode actual JS semantics in SMT ✔
   - Trivial: use SMT ceil operator
2. Automatically generate JIT engine range analysis routine in SMT ✔
   - Rewrite JIT engine range analysis logic in **VeRA C++**
     - Subset of C++ specific for this problem
   - Vera C++ → IR (SSA) → **SMT**
3. **Ask SMT solver: is it possible to get value from (1) outside of (2)?**

VeRA in Action – `ceil`

1. Encode actual JS semantics in SMT
    - Trivial: use SMT ceil operator
2. Automatically generate JIT engine range analysis routine in SMT
    - Rewrite JIT engine range analysis logic in **VeRA C++**
        - Subset of C++ specific for this problem
    - Vera C++ → IR (SSA) → **SMT**
3. **Ask SMT solver: is it possible to get value from (1) outside of (2)?**
    - **`inRange(value, range)`** predicate

VeRA in Action – `ceil`

1. Encode actual JS semantics in SMT
    - Trivial: use SMT ceil operator
2. Automatically generate JIT engine range analysis routine in SMT
    - Rewrite JIT engine range analysis logic in **VeRA C++**
        - Subset of C++ specific for this problem
    - Vera C++ → IR (SSA) → **SMT**
3. **Ask SMT solver: is it possible to get value from (1) outside of (2)?**
    - **`inRange(value, range)`** predicate
        - `value = 5`
        - `range.lower = 1; range.upper = 10;`

VeRA in Action – `ceil`

1. Encode actual JS semantics in SMT
    - Trivial: use SMT ceil operator
2. Automatically generate JIT engine range analysis routine in SMT
    - Rewrite JIT engine range analysis logic in **VeRA C++**
        - Subset of C++ specific for this problem
    - Vera C++ → IR (SSA) → **SMT**
3. **Ask SMT solver: is it possible to get value from (1) outside of (2)?**
    - **`inRange(value, range)`** predicate
        - `value = 5`
        - `range.lower = 1; range.upper = 10;`

`inRange = true`

VeRA in Action – `ceil`

1. Encode actual JS semantics in SMT
    - Trivial: use SMT ceil operator
2. Automatically generate JIT engine range analysis routine in SMT
    - Rewrite JIT engine range analysis logic in **VeRA C++**
        - Subset of C++ specific for this problem
    - Vera C++ → IR (SSA) → **SMT**
3. **Ask SMT solver: is it possible to get value from (1) outside of (2)?**
    - **`inRange(value, range)`** predicate
        - `value = -0`
        - `range.canBeNegativeZero = false`

VeRA in Action – `ceil`

1. Encode actual JS semantics in SMT
    - Trivial: use SMT ceil operator
2. Automatically generate JIT engine range analysis routine in SMT
    - Rewrite JIT engine range analysis logic in **VeRA C++**
        - Subset of C++ specific for this problem
    - Vera C++ → IR (SSA) → **SMT**
3. **Ask SMT solver: is it possible to get value from (1) outside of (2)?**
    - **`inRange(value, range)`** predicate
        - `value = -0`
        - `range.canBeNegativeZero = false`

`inRange = false`

VeRA in Action – `ceil`

1. Encode actual JS semantics in SMT
    - Trivial: use SMT ceil operator
2. Automatically generate JIT engine range analysis routine in SMT
    - Rewrite JIT engine range analysis logic in **VeRA C++**
        - Subset of C++ specific for this problem
    - Vera C++ → IR (SSA) → **SMT**
3. **Ask SMT solver: is it possible to get value from (1) outside of (2)?**
    - **`inRange(value, range)`** predicate
        - `value = 0.5`
        - `range.canHaveFractionalPart = false`

VeRA in Action – **`ceil`**

1. Encode actual JS semantics in SMT
    - Trivial: use SMT ceil operator
2. Automatically generate JIT engine range analysis routine in SMT
    - Rewrite JIT engine range analysis logic in **VeRA C++**
        - Subset of C++ specific for this problem
    - Vera C++ → IR (SSA) → **SMT**
3. **Ask SMT solver: is it possible to get value from (1) outside of (2)?**
    - **`inRange(value, range)`** predicate
        - `value = 0.5`
        - `range.canHaveFractionalPart = false`

`inRange = false`

$$
\begin{aligned}
\mathtt{inRange}(R, v) \triangleq\ & \\
& R.\mathtt{exp} < \mathtt{e\_INF} \implies \neg\mathtt{isInf}(v) \\
& \land R.\mathtt{exp} \neq \mathtt{e\_INF\_OR\_NAN} \implies \neg\mathtt{isNaN}(v) \\
& \land \neg R.\mathtt{canBeNegZero} \implies v \neq -0.0 \\
& \land \neg R.\mathtt{canHaveFraction} \implies \mathtt{round}(v) = v \\
& \land R.\mathtt{hasInt32LowerBound} \implies (\mathtt{isNaN}(v) \lor v \geq R.\mathtt{lower}) \\
& \land R.\mathtt{hasInt32UpperBound} \implies (\mathtt{isNaN}(v) \lor v \leq R.\mathtt{upper}) \\
& \land R.\mathtt{exp} \geq \mathtt{expOf}(v)
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{wellFormed}(R) \triangleq \\
&\quad R.\texttt{lower} \geq \texttt{JS\_INT\_MIN} \land R.\texttt{lower} \leq \texttt{JS\_INT\_MAX} \\
&\land R.\texttt{upper} \geq \texttt{JS\_INT\_MIN} \land R.\texttt{upper} \leq \texttt{JS\_INT\_MAX} \\
&\land \neg R.\texttt{hasInt32LowerBound} \implies R.\texttt{lower} = \texttt{JS\_INT\_MIN} \\
&\land \neg R.\texttt{hasInt32UpperBound} \implies R.\texttt{upper} = \texttt{JS\_INT\_MAX} \\
&\land R.\texttt{canBeNegZero} \implies \texttt{contains}(0, R) \\
&\land (R.\texttt{exp} = \texttt{e\_INF} \lor R.\texttt{exp} = \texttt{e\_INF\_OR\_NAN} \lor R.\texttt{exp} \leq 1023) \\
&\land (R.\texttt{hasInt32LowerBound} \land R.\texttt{hasInt32UpperBound}) \\
&\qquad \implies R.\texttt{exp} = \texttt{expOf}(\texttt{max}(|R.\texttt{lower}|, |R.\texttt{upper}|)) \\
&\land R.\texttt{hasInt32LowerBound} \implies R.\texttt{exp} \geq \texttt{expOf}(R.\texttt{lower}) \\
&\land R.\texttt{hasInt32UpperBound} \implies R.\texttt{exp} \geq \texttt{expOf}(R.\texttt{upper})
\end{aligned}
$$

VeRA in Action – `ceil`

1. Encode actual **JS semantics in SMT**
    - Trivial: use SMT ceil operator
2. Automatically generate **JIT engine range analysis routine in SMT**
    - Rewrite JIT engine range analysis logic in VeRA C++
        - Subset of C++ specific for this problem
    - Vera C++ → IR (SSA) → SMT
3. Ask SMT solver: is it possible to get value from (1) outside of (2)?
    - **`inRange(value, range)`** predicate

VeRA in Action – `ceil`

- JS semantics in SMT
- JIT engine range analysis routine in SMT
- `inRange(value, range)`

VeRA in Action – `ceil`

- JS semantics in SMT
- JIT engine range analysis routine in SMT
- `inRange(value, range)`

Verification SMT

VeRA in Action – `ceil`

- **JS semantics in SMT**
- JIT engine range analysis routine in SMT
- `inRange(value, range)`

Verification SMT

```
endVal = jsCeil(startVal)
```

VeRA in Action – `ceil`

- JS semantics in SMT
- **JIT engine range analysis routine in SMT**
- `inRange(value, range)`

Verification SMT

```
endVal = jsCeil(startVal)

endRange = ceilRa(startRange)
```

VeRA in Action – **`ceil`**

- JS semantics in SMT
- JIT engine range analysis routine in SMT
- **`inRange(value, range)`**

Verification SMT

```
endVal = jsCeil(startVal)

endRange = ceilRa(startRange)

inRange(startVal, startRange)
```

VeRA in Action – `ceil`

- JS semantics in SMT
- JIT engine range analysis routine in SMT
- `inRange(value, range)`

Verification SMT

```
endVal = jsCeil(startVal)

endRange = ceilRa(startRange)

inRange(startVal, startRange)
```

assume previous range analysis was correct

VeRA in Action – **`ceil`**

- JS semantics in SMT
- JIT engine range analysis routine in SMT
- `inRange(value, range)`

Verification SMT

```
endVal = jsCeil(startVal)

endRange = ceilRa(startRange)

inRange(startVal, startRange)

NOT inRange(endVal, endRange) ??
```

VeRA in Action – `ceil`

- JS semantics in SMT
- JIT engine range analysis routine in SMT
- `inRange(value, range)`

general

Verification SMT

```
endVal = jsCeil(startVal)

endRange = ceilRa(startRange)

inRange(startVal, startRange)

NOT inRange(endVal, endRange) ??
```

Implementation

Implementation

- Test VeRA-rewritten flow functions with Firefox’s test suites
  - Passed all ~147k tests

Implementation

- Test VeRA-rewritten flow functions with Firefox’s test suites
  - Passed all ~147k tests
- Verify 21 top-level Firefox range analysis flow functions using VeRA

Evaluation

Operation	R1	R2	R3	R4	R5.i32	R5.double	R6.i32	R6.double	R7	W1	W2	W3	W4	Undef
add	15	2	5	386	2	∞	2	∞	21	2	1	2	80	1
sub	13	2	11	445	8	∞	5	∞	14	2	2	2	78	1
and*	-	-	-	-	2	-	1	-	-	-	-	-	-	1
or*	-	-	-	-	2	-	2	-	-	-	-	-	-	1
xor*	-	-	-	-	2	-	2	-	-	-	-	-	-	1
not*	-	-	-	-	2	-	1	-	-	-	-	-	-	1
mul	92	65	22	362	∞	∞	∞	∞	94	4	4	4	∞	11
lsh*	-	-	-	-	1	-	1	-	-	-	-	-	-	1
rsh*	-	-	-	-	1	-	1	-	-	-	-	-	-	1
ursh*	-	-	-	-	**X**	-	**X**	-	-	-	-	-	-	1
lsh'*	-	-	-	-	1	-	1	-	-	-	-	-	-	1
rsh'*	-	-	-	-	1	-	1	-	-	-	-	-	-	1
ursh'*	-	-	-	-	**X**	-	**X**	-	-	-	-	-	-	1
abs	1	1	1	5	1	∞	1	224	1	1	1	4	∞	1
min	2	20	2	2	5	224	2	∞	3	2	1	2	∞	1
max	3	17	2	2	15	∞	2	∞	4	3	2	12	∞	1
floor	4	2	1	5	1	146	1	9	54	1	1	5	∞	1
ceil	∞	1	**X**	8	1	5	1	9	266	1	1	∞	∞	1
sign	1	1	1	1	1	2	1	2	1	1	1	1	2	1

Evaluation

Operation	R1	R2	R3	R4	R5.i32	R5.double	R6.i32	R6.double	R7	W1	W2	W3	W4	Undef
add	15	2	5	386	2	∞	2	∞	21	2	1	2	80	1
sub	13	2	11	445	8	∞	5	∞	14	2	2	2	78	1
and∗	-	-	-	-	2	-	1	-	-	-	-	-	-	1
or∗	-	-	-	-	2	-	2	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
xor∗	-	-	-	-	2	-	2	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
not∗	-	-	-	-	2	-	1	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
mul	92	65	22	362	∞	∞	∞	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
lsh∗	-	-	-	-	1	-	1	-	-	-	-	-	-	1
rsh∗	-	-	-	-	1	-	1	-	-	-	-	-	-	1
ursh∗	-	-	-	-	X	-	X	-	-	-	-	-	-	1
lsh'∗	-	-	-	-	1	-	1	-	-	-	-	-	-	1
rsh'∗	-	-	-	-	1	-	1	-	-	-	-	-	-	1
ursh'∗	-	-	-	-	X	-	X	-	-	-	-	-	-	1
abs	1	1	1	5	1	∞	1	224	1	1	1	4	∞	1
min	2	20	2	2	5	224	2	∞	3	2	1	2	∞	1
max	3	17	2	2	15	∞	2	∞	4	3	2	12	∞	1
floor	4	2	1	5	1	146	1	9	54	1	1	5	∞	1
ceil	∞	1	X	8	1	5	1	9	266	1	1	∞	∞	1
sign	1	1	1	1	1	2	1	2	1	1	1	1	2	1

```
// ursh's left operand is uint32, not int32, but for range
// analysis we currently approximate it as int32. We assume
// here that the range has already been adjusted...
```

Evaluation

Operation	R1	R2	R3	R4	R5.i32	R5.double	R6.i32	R6.double	R7	W1	W2	W3	W4	Undef
`add`	15	2	5	386	2	∞	2	∞	21	2	1	2	80	1
`sub`	13	2	11	445	8	∞	5	∞	14	2	2	2	78	1
`and*`	-	-	-	-	2	-	1	-	-	-	-	-	-	1
`or*`	-	-	-	-	2	-	2	-	-	-	-	-	-	1
`xor*`	-	-	-	-	2	-	2	-	-	-	-	-	-	1
`not*`	-	-	-	-	2	-	1	-	-	-	-	-	-	1
`mul`	92	65	22	362	∞	∞	∞	∞	94	4	4	4	∞	11
`lsh*`	-	-	-	-	1	-	1	-	-	-	-	-	-	1
`rsh*`	-	-	-	-	1	-	1	-	-	-	-	-	-	1
`ursh*`	-	-	-	-	**X**	-	**X**	-	-	-	-	-	-	1
`lsh'*`	-	-	-	-	1	-	1	-	-	-	-	-	-	1
`rsh'*`	-	-	-	-	1	-	1	-	-	-	-	-	-	1
`ursh'*`	-	-	-	-	**X**	-	**X**	-	-	-	-	-	-	1
`abs`	1	1	1	5	1	∞	1	224	1	1	1	4	∞	1
`min`	2	20	2	2	5	224	2	∞	3	2	1	2	∞	1
`max`	3	17	2	2	15	∞	2	∞	4	3	2	12	∞	1
`floor`	4	2	1	5	1	146	1	9	54	1	1	5	∞	1
`ceil`	∞	1	**X**	8	1	5	1	9	266	1	1	∞	∞	1
`sign`	1	1	1	1	1	2	1	2	1	1	1	1	2	1

actual bug

Evaluation

- Bug in Firefox JIT range analysis flow function for **`ceil`**

Evaluation

- Bug in Firefox JIT range analysis flow function for **`ceil`**
- $\mathtt{inRange}(R, v) \triangleq$

$$
\begin{aligned}
& R.\mathtt{exp} < \mathtt{e\_INF} \implies \neg\mathtt{isInf}(v) && \text{(R1)} \\
\wedge & R.\mathtt{exp} \neq \mathtt{e\_INF\_OR\_NAN} \implies \neg\mathtt{isNaN}(v) && \text{(R2)} \\
& \boxed{\wedge\neg R.\mathtt{canBeNegZero} \implies v \neq -0.0} && \text{(R3)} \\
\wedge & \neg R.\mathtt{canHaveFraction} \implies \mathtt{round}(v) = v && \text{(R4)} \\
\wedge & R.\mathtt{hasInt32LowerBound} \implies (\mathtt{isNaN}(v) \vee v \geq R.\mathtt{lower}) && \text{(R5)} \\
\wedge & R.\mathtt{hasInt32UpperBound} \implies (\mathtt{isNaN}(v) \vee v \leq R.\mathtt{upper}) && \text{(R6)} \\
\wedge & R.\mathtt{exp} \geq \mathtt{expOf}(v) && \text{(R7)}
\end{aligned}
$$

Evaluation

- Bug in Firefox JIT range analysis flow function for **`ceil`**
- $\mathtt{inRange}(R, v) \triangleq$

$$
\begin{aligned}
& R.\mathtt{exp} < \mathtt{e\_INF} \implies \neg\mathtt{isInf}(v) && \text{(R1)} \\
& \land R.\mathtt{exp} \neq \mathtt{e\_INF\_OR\_NAN} \implies \neg\mathtt{isNaN}(v) && \text{(R2)} \\
& \boxed{\land\neg R.\mathtt{canBeNegZero} \implies v \neq -0.0} && \text{(R3)} \\
& \land\neg R.\mathtt{canHaveFraction} \implies \mathtt{round}(v) = v && \text{(R4)} \\
& \land R.\mathtt{hasInt32LowerBound} \implies (\mathtt{isNaN}(v) \lor v \geq R.\mathtt{lower}) && \text{(R5)} \\
& \land R.\mathtt{hasInt32UpperBound} \implies (\mathtt{isNaN}(v) \lor v \leq R.\mathtt{upper}) && \text{(R6)} \\
& \land R.\mathtt{exp} \geq \mathtt{expOf}(v) && \text{(R7)}
\end{aligned}
$$

ceil(-0.5)

Evaluation

- Bug in Firefox JIT range analysis flow function for **`ceil`**
- $\mathtt{inRange}(R, v) \triangleq$

$$
\begin{aligned}
& R.\mathtt{exp} < \mathtt{e\_INF} \implies \neg\mathtt{isInf}(v) && \text{(R1)} \\
& \land R.\mathtt{exp} \neq \mathtt{e\_INF\_OR\_NAN} \implies \neg\mathtt{isNaN}(v) && \text{(R2)} \\
& \boxed{\land\neg R.\mathtt{canBeNegZero} \implies v \neq -0.0} && \text{(R3)} \\
& \land\neg R.\mathtt{canHaveFraction} \implies \mathtt{round}(v) = v && \text{(R4)} \\
& \land R.\mathtt{hasInt32LowerBound} \implies (\mathtt{isNaN}(v) \lor v \geq R.\mathtt{lower}) && \text{(R5)} \\
& \land R.\mathtt{hasInt32UpperBound} \implies (\mathtt{isNaN}(v) \lor v \leq R.\mathtt{upper}) && \text{(R6)} \\
& \land R.\mathtt{exp} \geq \mathtt{expOf}(v) && \text{(R7)}
\end{aligned}
$$

ceil(-0.5)
- actual: -0

Evaluation

- Bug in Firefox JIT range analysis flow function for **`ceil`**
- $\texttt{inRange}(R, v) \triangleq$

$$
\begin{aligned}
&R.\texttt{exp} < \texttt{e\_INF} \implies \neg\texttt{isInf}(v) && \text{(R1)}\\
&\wedge R.\texttt{exp} \neq \texttt{e\_INF\_OR\_NAN} \implies \neg\texttt{isNaN}(v) && \text{(R2)}\\
&\boxed{\wedge\neg R.\texttt{canBeNegZero} \implies v \neq -0.0} && \text{(R3)}\\
&\wedge\neg R.\texttt{canHaveFraction} \implies \texttt{round}(v) = v && \text{(R4)}\\
&\wedge R.\texttt{hasInt32LowerBound} \implies (\texttt{isNaN}(v) \vee v \geq R.\texttt{lower}) && \text{(R5)}\\
&\wedge R.\texttt{hasInt32UpperBound} \implies (\texttt{isNaN}(v) \vee v \leq R.\texttt{upper}) && \text{(R6)}\\
&\wedge R.\texttt{exp} \geq \texttt{expOf}(v) && \text{(R7)}
\end{aligned}
$$

ceil(-0.5)

- actual: -0
- Firefox JIT Range Analysis: `canBeNegZero = false`

Evaluation

- Bug in Firefox JIT range analysis flow function for `ceil`
- $\texttt{inRange}(R, v) \triangleq$

$R.\texttt{exp} < \texttt{e}$

$\wedge R.\texttt{exp} \neq \texttt{e}$

$\boxed{\wedge \neg R.\texttt{canBeN}}$

$\wedge \neg R.\texttt{canHav}$

$\wedge R.\texttt{hasInt3}$

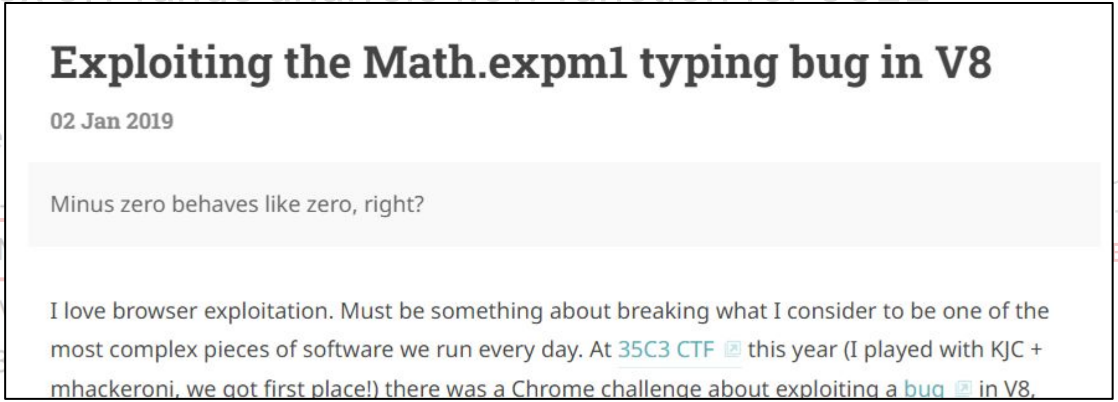
$\wedge R.\texttt{hasInt32UpperBound} \implies (\texttt{isNaN}(v) \vee v \leq R.\texttt{upper})$ (R6)

$\wedge R.\texttt{exp} \geq \texttt{expOf}(v)$ (R7)

ual: -0

fox JIT Range Analysis:

`BeNegZero = false`



Exploiting the Math.expm1 typing bug in V8

02 Jan 2019

Minus zero behaves like zero, right?

I love browser exploitation. Must be something about breaking what I consider to be one of the most complex pieces of software we run every day. At 35C3 CTF this year (I played with KJC + mhackeroni, we got first place!) there was a Chrome challenge about exploiting a bug in V8,

https://abiondo.me/2019/01/02/exploiting-math-expm1-v8/

Conclusion

- VeRA: a system for verifying the range analysis pass in browser JIT compilers
  - Rewrite range analysis flow function in VeRA C++
  - Verify by SMT solver
- Verified 21 top-level Firefox range analysis flow functions
  - Detected Firefox bug (existed for 6 years)

My Review

- **Strengths**
  - Targets critical part of browser
  - Presents end-to-end system: C++ to SMT
  - Easy to adapt to logic change
- **Weaknesses**
  - SMT solver could not verify everything (timeouts)
  - Current implementation is tightly coupled to Firefox
  - Hard to extend to other RA engine (e.g. Chrome)

Thank you

Steve Gustaman
stevegustaman@kaist.ac.kr

$$\texttt{inRange}(R, v) \triangleq$$

$$
\begin{aligned}
& R.\texttt{exp} < \texttt{e\_INF} \implies \neg\texttt{isInf}(v) && \text{(R1)}\\
\wedge & R.\texttt{exp} \neq \texttt{e\_INF\_OR\_NAN} \implies \neg\texttt{isNaN}(v) && \text{(R2)}\\
\wedge & \neg R.\texttt{canBeNegZero} \implies v \neq -0.0 && \text{(R3)}\\
\wedge & \neg R.\texttt{canHaveFraction} \implies \texttt{round}(v) = v && \text{(R4)}\\
\wedge & R.\texttt{hasInt32LowerBound} \implies (\texttt{isNaN}(v) \vee v \geq R.\texttt{lower}) && \text{(R5)}\\
\wedge & R.\texttt{hasInt32UpperBound} \implies (\texttt{isNaN}(v) \vee v \leq R.\texttt{upper}) && \text{(R6)}\\
\wedge & R.\texttt{exp} \geq \texttt{expOf}(v) && \text{(R7)}
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{wellFormed}(R) \triangleq \\
&\quad R.\texttt{lower} \geq \texttt{JS\_INT\_MIN} \land R.\texttt{lower} \leq \texttt{JS\_INT\_MAX} && \text{(W1)} \\
&\quad \land R.\texttt{upper} \geq \texttt{JS\_INT\_MIN} \land R.\texttt{upper} \leq \texttt{JS\_INT\_MAX} && \text{(W1)} \\
&\quad \land \neg R.\texttt{hasInt32LowerBound} \implies R.\texttt{lower} = \texttt{JS\_INT\_MIN} && \text{(W2)} \\
&\quad \land \neg R.\texttt{hasInt32UpperBound} \implies R.\texttt{upper} = \texttt{JS\_INT\_MAX} && \text{(W2)} \\
&\quad \land R.\texttt{canBeNegZero} \implies \texttt{contains}(0, R) \\
&\quad \land (R.\texttt{exp} = \texttt{e\_INF} \lor R.\texttt{exp} = \texttt{e\_INF\_OR\_NAN} \lor R.\texttt{exp} \leq 1023) && \text{(W3)} \\
&\quad \land (R.\texttt{hasInt32LowerBound} \land R.\texttt{hasInt32UpperBound}) \\
&\qquad\qquad \implies R.\texttt{exp} = \texttt{expOf}(\texttt{max}(|R.\texttt{lower}|, |R.\texttt{upper}|)) \\
&\quad \land R.\texttt{hasInt32LowerBound} \implies R.\texttt{exp} \geq \texttt{expOf}(R.\texttt{lower}) && \text{(W4)} \\
&\quad \land R.\texttt{hasInt32UpperBound} \implies R.\texttt{exp} \geq \texttt{expOf}(R.\texttt{upper}) && \text{(W4)}
\end{aligned}
$$

Evaluation

Operation	R1	R2	R3	R4	R5.i32	R5.double	R6.i32	R6.double	R7	W1	W2	W3	W4	Undef
add	15	2	5	386	2	∞	2	∞	21	2	1	2	80	1
sub	13	2	11	445	8	∞	5	∞	14	2	2	2	78	1
and*	-	-	-	-	2	-	1	-	-	-	-	-	-	1
or*	-	-	-	-	2	-	2	-	-	-	-	-	-	1
xor*	-	-	-	-	2	-	2	-	-	-	-	-	-	1
not*	-	-	-	-	2	-	1	-	-	-	-	-	-	1
mul	92	65	22	362	∞	∞	∞	∞	94	4	4	4	∞	11
lsh*	-	-	-	-	1	-	1	-	-	-	-	-	-	1
rsh*	-	-	-	-	1	-	1	-	-	-	-	-	-	1
ursh*	-	-	-	-	**X**	-	**X**	-	-	-	-	-	-	1
lsh'*	-	-	-	-	1	-	1	-	-	-	-	-	-	1
rsh'*	-	-	-	-	1	-	1	-	-	-	-	-	-	1
ursh'*	-	-	-	-	**X**	-	**X**	-	-	-	-	-	-	1
abs	1	1	1	5	1	∞	1	224	1	1	1	4	∞	1
min	2	20	2	2	5	224	2	∞	3	2	1	2	∞	1
max	3	17	2	2	15	∞	2	∞	4	3	2	12	∞	1
floor	4	2	1	5	1	146	1	9	54	1	1	5	∞	1
ceil	∞	1	**X**	8	1	5	1	9	266	1	1	∞	∞	1
sign	1	1	1	1	1	2	1	2	1	1	1	1	2	1

Evaluation - Time to Verify

We successfully prove or refute 137 conditions out of a possible 159, for a success rate of ≈86%; the shortest proofs complete in under a second, while the longest takes ≈ten minutes. The results suggest that **R5.double**, **R6.double**, and **W4** are particularly challenging to verify. **R5.double** and **R6.double** are more challenging than their integer counterparts because they involve reasoning about floating-point values, which is generally more expensive. **W4** is challenging because it involves proving a relationship between two properties of the range, both of which may be modified by the range analysis. Finally, **R1** and **W3** of `Math.ceil` may timeout because they involve bounding the size of an exponent, since `Math.ceil` involves extracting the exponent from the absolute value of the range bounds.