JSIR

Adversarial JavaScript Analysis with MLIR

Zhixun Tan (tzx@google.com)

2024-10-24

https://github.com/google/jsir

Malicious JavaScript appears everywhere

Web pages

Mobile apps

Browser extensions

Motivating Examples

Example 1: steganography

```
var imageData = ctx.getImageData(...);
var modMessage = someTransformFrom(imageData);
var message = "";
var charCode = 0;
var bitCount = 0;
var mask = Math.pow(2, codeUnitSize) - 1;
for (var i = 0; i < modMessage.length; i += 1) {
  charCode += modMessage[i] << bitCount;
  bitCount += t;
  if (bitCount >= codeUnitSize) {
    message += String.fromCharCode(charCode & mask);
    bitCount %= codeUnitSize;
    charCode = modMessage[i] >> (t-bitCount);
  }
}
if (charCode !== 0)
  message += String.fromCharCode(charCode & mask);
eval(message);
```

https://github.com/petereigenschink/steganography.js/blob/master/src/decode.js

Example 1: steganography

```
var imageData = ctx.getImageData(...);
var modMessage = someTransformFrom(imageData);
var message = "";
var charCode = 0;
var bitCount = 0;
var mask = Math.pow(2, codeUnitSize) - 1;
for (var i = 0; i < modMessage.length; i += 1) {
  charCode += modMessage[i] << bitCount;
  bitCount += t;
  if (bitCount >= codeUnitSize) {
    message += String.fromCharCode(charCode & mask);
    bitCount %= codeUnitSize;
    charCode = modMessage[i] >> (t-bitCount);
  }
}
if (charCode !== 0)
  message += String.fromCharCode(charCode & mask);
eval(message);
```

Getting data from an image.

Malicious behavior: steganography

- Hiding information in an image.
- There are automatic tools to encode and decode.

eval() is usually evil().

https://github.com/petereigenschink/steganography.js/blob/master/src/decode.js

Example 1: steganography

```
var imageData = ctx.getImageData(...);
var modMessage = someTransformFrom(imageData);
var message = "";
var charCode = 0;
var bitCount = 0;
var mask = Math.pow(2, codeUnitSize) - 1;
for (var i = 0; i < modMessage.length; i += 1) {
  charCode += modMessage[i] << bitCount;
  bitCount += t;
  if (bitCount >= codeUnitSize) {
    message += String.fromCharCode(charCode & mask);
    bitCount %= codeUnitSize;
    charCode = modMessage[i] >> (t-bitCount);
  }
}
if (charCode !== 0)
  message += String.fromCharCode(charCode & mask);
eval(message);
```

Solution: Taint Analysis

- Taint analysis: dataflow analysis to discover suspicious information flows.
- Example:
  - Source: getImageData(...)
  - Sink: eval(...)

https://github.com/petereigenschink/steganography.js/blob/master/src/decode.js

Example 1: steganography

```
var imageData = ctx.getImageData(...);
var modMessage = someTransformFrom(imageData);
var message = "";
var charCode = 0;
var bitCount = 0;
var mask = Math.pow(2, codeUnitSize) - 1;
for (var i = 0; i < modMessage.length; i += 1) {
  charCode += modMessage[i] << bitCount;
  bitCount += t;
  if (bitCount >= codeUnitSize) {
    message += String.fromCharCode(charCode & mask);
    bitCount %= codeUnitSize;
    charCode = modMessage[i] >> (t-bitCount);
  }
}
if (charCode !== 0)
  message += String.fromCharCode(charCode & mask);
eval(message);
```

Source: `getImageData(...)`

Sink: `eval(...)`

```
{}
{imageData}
{imageData, modMessage}
{imageData, modMessage}
{imageData, modMessage}
{imageData, modMessage}
{imageData, modMessage}
{imageData, modMessage}
{imageData, modMessage, charCode}
{imageData, modMessage, charCode}
{imageData, modMessage, charCode}
{imageData, modMessage, charCode, message}
{imageData, modMessage, charCode, message}
{imageData, modMessage, charCode, message}
{imageData, modMessage, charCode, message}
{imageData, modMessage, charCode, message}
{imageData, modMessage, charCode, message}
{imageData, modMessage, charCode, message}
{imageData, modMessage, charCode, message}
```

Variables tainted by source

Taint Analysis

A forward dataflow analysis that marks all variables tainted by the source.

Requirements: IR, CFG, (ideally) SSA

Example 2: obfuscation

Original source

```
function concat(a, b) {
  return a + b;
}
console.log(concat("hello, ", "world"));
```

Obfuscated source

```
function concat(_0x172308, _0x422cff) {
  return _0x172308 + _0x422cff;
}
console['log'](concat('hel' + 'lo,' + '\x20', 'wor' + 'ld'));
```

Malicious behavior: obfuscation

- Obfuscation: intentionally makes code more complex.

  This example: string splitting.
- There are automatic tools to obfuscate code.

https://obfuscator.io

Example 2: obfuscation

Original source

```
function concat(a, b) {
  return a + b;
}
console.log(concat("hello, ", "world"));
```

Obfuscated source

```
function concat(_0x172308, _0x422cff) {
  return _0x172308 + _0x422cff;
}
console['log'](concat('hel' + 'lo,' + '\x20', 'wor' + 'ld'));
```

Solution: deobfuscation

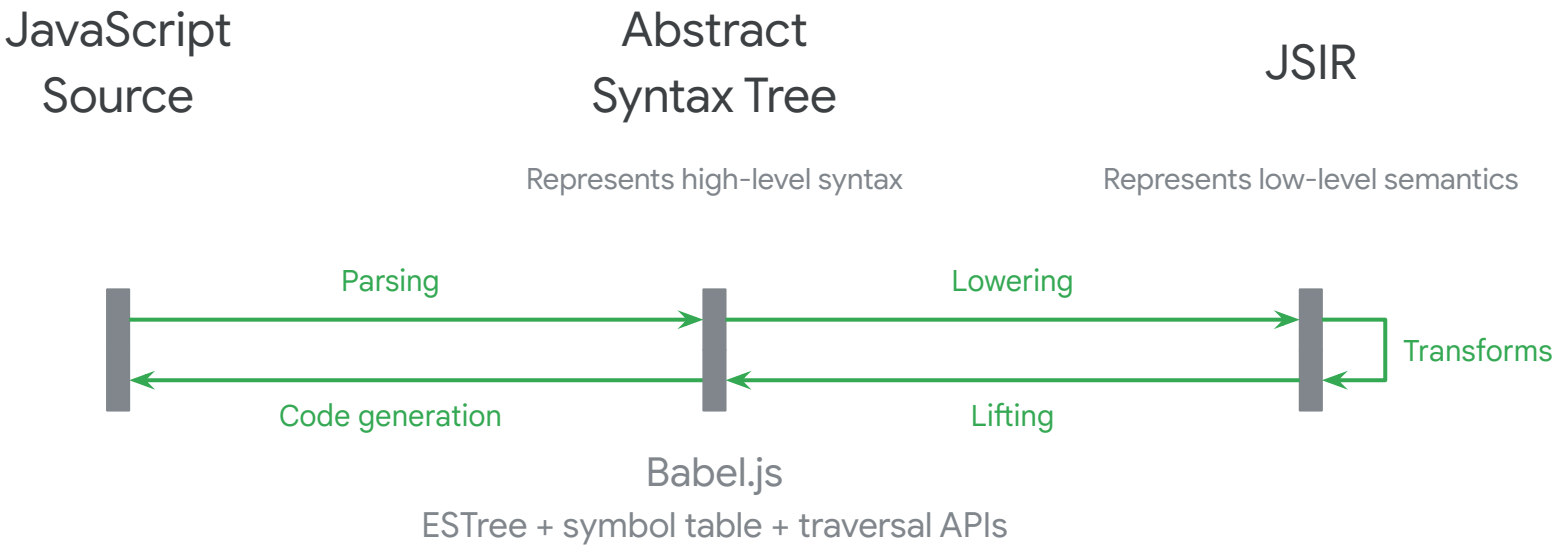
- Constant propagation - forward dataflow analysis
  - IR, CFG, (ideally) SSA
- We want source-to-source transformation - help reverse engineers
  - Convert transformed IR back to source; untransformed IR should revert to original source. Is this even possible?

https://obfuscator.io

How to balance conflicting requirements?

	Signal extraction	Code simplification
Used by ↑	Rule-based decision engines ML classifiers	Reverse engineers Analysts and manual reviewers
Requires ↓	IR + CFG for dataflow analysis SSA for better performance	IR + CFG for dataflow analysis No SSA - too low-level AST for code generation

JSIR: JavaScript IR that can be lifted back to the AST



JSIR Design Tour

Design issue 1: SSA values

Source

```
1 + 2 + 3;
```


IR

```
%0 = jsir.numeric_literal {1}
%1 = jsir.numeric_literal {2}
%2 = jsir.binary_expression {"+"} (%0, %1)
%3 = jsir.numeric_literal {3}
%4 = jsir.binary_expression {"+"} (%2, %3)
jsir.expression_statement (%4)
```

Reconstructed Source?

```
let r0 = 1;
let r1 = 2;
let r2 = r0 + r1;
let r3 = 3;
let r4 = r2 + r3;
r4;
```

Issue: we don't want each SSA value to be lifted to a variable.

Design issue 1: SSA values

Source

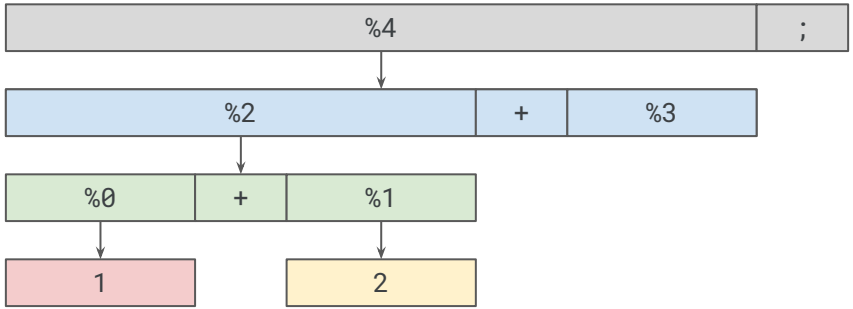
```
1 + 2 + 3;
```

IR

```
%0 = jsir.numeric_literal {1}
%1 = jsir.numeric_literal {2}
%2 = jsir.binary_expression {"+"} (%0, %1)
%3 = jsir.numeric_literal {3}
%4 = jsir.binary_expression {"+"} (%2, %3)
jsir.expression_statement (%4)
```

Reconstructed Source?

```
%4;
```



Solution: Recognize “statement” operations and recursively lift.

Design issue 1: SSA values

Source

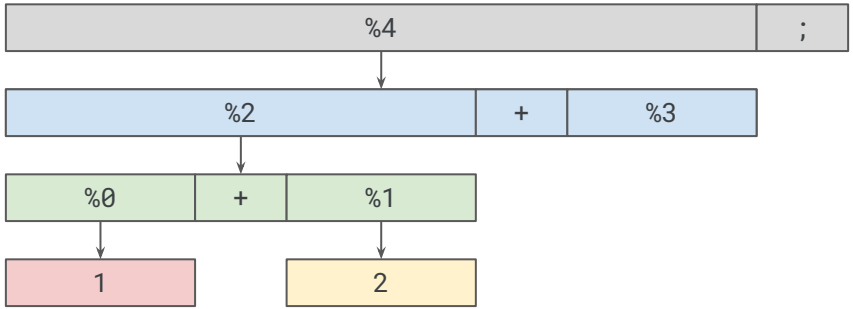
```
1 + 2 + 3;
```

IR

```
%0 = jsir.numeric_literal {1}
%1 = jsir.numeric_literal {2}
%2 = jsir.binary_expression {"+"} (%0, %1)
%3 = jsir.numeric_literal {3}
%4 = jsir.binary_expression {"+"} (%2, %3)
jsir.expression_statement (%4)
```

Reconstructed Source?

```
1 + 2 + 3;
```



Solution: Recognize “statement” operations and recursively lift.

Design issue 2: variables

Source

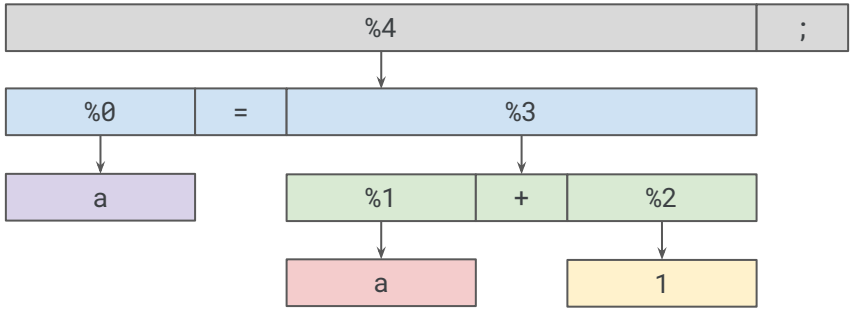
```
a = a + 1;
```

IR

```
%0 = jsir.identifier_ref {"a"}
%1 = jsir.identifier {"a"}
%2 = jsir.numeric_literal {1}
%3 = jsir.binary_expression {"+"} (%1, %2)
%4 = jsir.assignment_expression (%0, %3)
jsir.expression_statement (%4)
```

Reconstructed Source

```
%4;
```



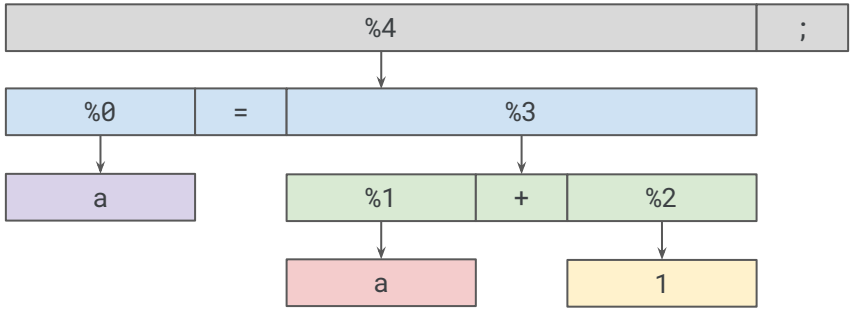
Design issue 2: variables

Source

```
a = a + 1;
```

IR

```
%0 = jsir.identifier_ref {"a"}
%1 = jsir.identifier {"a"}
%2 = jsir.numeric_literal {1}
%3 = jsir.binary_expression {"+"} (%1, %2)
%4 = jsir.assignment_expression (%0, %3)
jsir.expression_statement (%4)
```



Reconstructed Source

```
a = a + 1;
```

Design issue 3: control flow structures

Source

```
...
if (cond_1) {
  ...
} else {
  ...
  if (cond_2) {
    ...
  } else {
    ...
  }
  ...
}
```

Using regions

```
...
jshir.if (%cond_1) ({
  ...
}, {
  ...
  jshir.if (%cond_2) ({
    ...
  }, {
    ...
  })
})
...
```

Using CFG

```
  ...
  cf.cond_br (%cond_1) [^BB1, ^BB2]
^BB1:
  ...
  cf.br [^BB6]
^BB2:
  ...
  cf.cond_br (%cond_2) [^BB3, ^BB4]
^BB3:
  ...
  cf.br [^BB5]
^BB4:
  ...
  cf.br [^BB5]
^BB5:
  ...
  cf.br [^BB6]
^BB6:
  ...
```

CFG is useful for dataflow analysis, but how can we lift this back to AST?

Design issue 3: control flow structures

Source

```
...
if (cond_1) {
  ...
} else {
  ...
  if (cond_2) {
    ...
  } else {
    ...
  }
  ...
}
```

Using CFG

```
  ...
  cf.cond_br (%cond_1) [^outer_if_true, ^outer_if_false]
```

```
^outer_if_true:
  ...
  cf.br [^outer_if_end]
```

```
^outer_if_false:
  ...
  cf.br [^outer_if_end]
```

```
^outer_if_end:
  ...
```

Design issue 3: control flow structures

Source

```
...
if (cond_1) {
  ...
} else {
  ...
  if (cond_2) {
    ...
  } else {
    ...
  }
  ...
}
```

Using CFG

```
  ...
  cf.cond_br (%cond_1) [^outer_if_true, ^outer_if_false]
```

```
^outer_if_true:
  ...
  cf.br [^outer_if_end]
```

```
^outer_if_false:
  ...
  cf.cond_br (%cond_2) [^inner_if_true, ^inner_if_false]
```

```
^inner_if_true:
  ...
  cf.br [^inner_if_end]
```

```
^inner_if_false:
  ...
  cf.br [^inner_if_end]
```

```
^inner_if_end:
  ...
  cf.br [^outer_if_end]
```

```
^outer_if_end:
  ...
```

Design issue 3: control flow structures

Create a token for each control flow structure

Source

```
...
if (cond_1) {
  ...
} else {
  ...
  if (cond_2) {
    ...
  } else {
    ...
  }
  ...
}
```

Using tokens

```
  ...
  %if_1 = jslir.if
  cf.cond_br (%cond_1) [^outer_if_true, ^outer_if_false]
```

```
^outer_if_true:
  jslir.if_true (%if_1)
  ...
  cf.br [^outer_if_end]
```

```
^outer_if_false:
  jslir.if_false (%if_1)
  ...
  %if_2 = jslir.if
  cf.cond_br (%cond_2) [^inner_if_true, ^inner_if_false]
```

```
^inner_if_true:
  jslir.if_true (%if_2)
  ...
  cf.br [^inner_if_end]
```

```
^inner_if_false:
  jslir.if_false (%if_2)
  ...
  cf.br [^inner_if_end]
```

```
^inner_if_end:
  jslir.if_end (%if_2)
  ...
  cf.br [^outer_if_end]
```

```
^outer_if_end:
  jslir.if_end (%if_1)
  ...
```

Create an annotation no-op for each block

We can traverse the CFG “recursively” as if traversing an AST.

Evaluation

> 5B

Real JavaScript samples tested

> 99.9%

Succeeded in source ⇄ AST ⇄ IR roundtrip with same source

Evaluation

~ 70%

React Native bytecode decompiled

Design issue 3: control flow structures

Source

```
...
if (cond_1) {
  ...
} else {
  ...
  if (cond_2) {
    ...
  } else {
    ...
  }
  ...
}
```

Using regions

```
...
jshir.if (%cond_1) ({
  ...
}, {
  ...
  jshir.if (%cond_2) ({
    ...
  }, {
    ...
  })
})
...
```

Using CFG

```
  ...
  cf.cond_br (%cond_1) [^BB1, ^BB2]
^BB1:
  ...
  cf.br [^BB6]
^BB2:
  ...
  cf.cond_br (%cond_2) [^BB3, ^BB4]
^BB3:
  ...
  cf.br [^BB5]
^BB4:
  ...
  cf.br [^BB5]
^BB5:
  ...
  cf.br [^BB6]
^BB6:
  ...
```

Can we just do region-based dataflow analysis?
Key issue: support “break” and “continue”

Efficient Data-Flow Analysis on Region-Based Control Flow in MLIR
https://www.youtube.com/watch?v=vvVR3FyU9TE

[RFC] Region-based control-flow with early exits in MLIR
https://discourse.llvm.org/t/rfc-region-based-control-flow-with-early-exits-in-mlir

Takeaways

Malicious JavaScript

Malicious JavaScript is a prevalent problem

MLIR to Represent General Purpose Languages

MLIR has much more potential than representing ML programs

“Reversible” IR Design

It is possible to design an IR that can convert back to AST

Long-term goals / ideas / visions

- Could a high-level IR completely replace the AST?
  Seems like Mojo is doing something like this
  (https://discourse.llvm.org/t/rfc-region-based-control-flow-with-early-exits-in-mlir/76998/11)

- A JavaScript IR standard?
  In other words, it's like ESTree but an IR instead of AST

- An IR-based JavaScript tooling framework?
  In other words, it's like Babel but IR-based

- Check it out! https://github.com/google/jsir