Fuzzing for complex bugs across languages in JS Engines

Creating deep interactions - POC 2024

Carl Smith - V8 Security - Google

README.md

Fuzzilli

A (coverage-)guided fuzzer for dynamic language interpreters based on a custom intermediate language ("FuzzIL") which can be mutated and translated to JavaScript.

fuzzil.li

Fuzzilli Recap

```
v0 <- BeginPlainFunction -> v1
    v2 <- CreateArray [v1, v1, v1]
    v3 <- LoadInt ‘1’
    v4 <- CallMethod ‘slice’, v2, [v3]
    Return v4
EndPlainFunction
v5 <- LoadFloat ‘13.37’
v6 <- CallFunction v0, [v5]
```

Fuzzilli Recap

```
v0 <- BeginPlainFunction -> v1
    v2 <- CreateArray [v1, v1, v1]
    v3 <- LoadInt ‘1’
    v4 <- CallMethod ‘slice’, v2, [v3]
    Return v4
EndPlainFunction
v5 <- LoadFloat ‘13.37’
v6 <- CallFunction v0, [v5]
```

Mutate

```
v0 <- BeginPlainFunction -> v1
    v2 <- CreateArray [v1, v1, v1]
    v4 <- LoadInt ‘100’
    SetProperty v2, ‘length’, v4
    v5 <- CallMethod ‘slice’, v2, [v1]
    Return v5
EndPlainFunction
v6 <- LoadFloat ‘42.0’
v7 <- CallFunction v0, [v5]
```

Splicing

Program 1

```
...
v21 <- BeginPlainFunction -> v22, v23
  ...

EndPlainFunction
...
```

Program 2

```
v0 <- BeginPlainFunction -> v1
    v2 <- CreateArray [v1, v1, v1]
    v3 <- LoadInt ‘1’
    v4 <- CallMethod ‘slice’, v2, [v3]
    Return v4
EndPlainFunction
v5 <- LoadFloat ‘13.37’
V6 <- CallFunction v0, [v5]
```

Splicing

Program 1

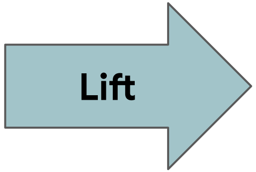
```
...
v21 <- BeginPlainFunction -> v22, v23
  ...
  v35 <- CreateArray [v23, v23, v23]
  v36 <- LoadInt ‘1’
  v37 <- CallMethod ‘slice’ v35, [v36]
  Return v37
EndPlainFunction
...
```

Program 2

```
v0 <- BeginPlainFunction -> v1
    v2 <- CreateArray [v1, v1, v1]
    v3 <- LoadInt ‘1’
    v4 <- CallMethod ‘slice’, v2, [v3]
    Return v4
EndPlainFunction
v5 <- LoadFloat ‘13.37’
V6 <- CallFunction v0, [v5]
```

Fuzzilli Recap

```
v0 <- BeginPlainFunction -> v1
    v2 <- CreateArray [v1, v1, v1]
    v3 <- LoadInt ‘1’
    v4 <- CallMethod ‘slice’, v2, [v3]
    Return v4
EndPlainFunction
v5 <- LoadFloat ‘13.37’
v6 <- CallFunction v0, [v5]
```




```
function f0(v1) {
    const v2 = [v1, v1, v1];
    return v2.slice(1);
}
f0(13.37);
```

This has worked well for JavaScript but we have a bunch of bugs in WebAssembly…

What do we want to actually fuzz?

- WebAssembly is highly integrated with JavaScript
- WebAssembly itself is a simple virtual machine; it cannot do a lot on its own
- It needs to call back into JavaScript to interact with the world
- It is statically typed and allows engines to produce fast code

=> Good test cases are highly structural

Motivating Example: crbug.com/338908243

```
function CreateWasmObjects() {
  let builder = new WasmModuleBuilder();
  let struct_type = builder.addStruct([makeField(kWasmI32, true)]);
  builder.addFunction('MakeStruct',
    makeSig([], [kWasmExternRef])).exportFunc().addBody(
        [kExprI32Const, 42, kGCPrefix, kExprStructNew,
         struct_type, kGCPrefix, kExprExternConvertAny]);
  let instance = builder.instantiate();
  return instance.exports.MakeStruct();
}
let struct = CreateWasmObjects();
Array.prototype.__proto__ = struct;
print([1].concat());
```

Motivating Example: crbug.com/338908243

```
function CreateWasmObjects() {
  let builder = new WasmModuleBuilder();
  let struct_type = builder.addStruct([makeField(kWasmI32, true)]);
  builder.addFunction('MakeStruct',
    makeSig([], [kWasmExternRef])).exportFunc().addBody(
        [kExprI32Const, 42, kGCPrefix, kExprStructNew,
         struct_type, kGCPrefix, kExprExternConvertAny]);
  let instance = builder.instantiate();
  return instance.exports.MakeStruct();
}
let struct = CreateWasmObjects();
Array.prototype.__proto__ = struct;
print([1].concat());
```

“old” code never expected a WasmObject here!

Motivating Example: crbug.com/338908243

```
function CreateWasmObjects() {
  let builder = new WasmModuleBuilder();
  let struct_type = builder.addStruct([makeField(kWasmI32, true)]);
  builder.addFunction('MakeStruct',
    makeSig([], [kWasmExternRef])).exportFunc().addBody(
        [kExprI32Const, 42, kGCPrefix, kExprStructNew,
         struct_type, kGCPrefix, kExprExternConvertAny]);
  let instance = builder.instantiate();
  return instance.exports.MakeStruct();
}
let struct = CreateWasmObjects();
Array.prototype.__proto__ = struct;
print([1].concat());
```

Type confusion here!
JSObject - Wasm struct

Motivating Example: crbug.com/338908243

```
function CreateWasmObjects() {
  let builder = new WasmModuleBuilder();
  let struct_type = builder.addStruct([makeField(kWasmI32, true)]);
  builder.addFunction('MakeStruct',
    makeSig([], [kWasmExternRef])).exportFunc().addBody(
        [kExprI32Const, 42, kGCPrefix, kExprStructNew,
         struct_type, kGCPrefix, kExprExternConvertAny]);
  let instance = builder.instantiate();
  return instance.exports.MakeStruct();
}
let struct = CreateWasmObjects();
Array.prototype.__proto__ = struct;
print([1].concat());
```

Motivating Example: crbug.com/338908243

```
function CreateWasmObjects() {
  let builder = new WasmModuleBuilder();
  let struct_type = builder.addStruct([makeField(kWasmI32, true)]);
  builder.addFunction('MakeStruct',
    makeSig([], [kWasmExternRef])).exportFunc().addBody(
        [kExprI32Const, 42, kGCPrefix, kExprStructNew,
         struct_type, kGCPrefix, kExprExternConvertAny]);
  let instance = builder.instantiate();
  return instance.exports.MakeStruct();
}
```

```
let struct = CreateWasmObjects();
Array.prototype.__proto__ = struct;
print([1].concat());
```

What should “good” test cases look like?

What do we want to actually fuzz?

```
let v0 = new WebAssembly.Table({ element: "externref", initial: 10, maximum: 20 });
let v1 = new WebAssembly.Global({ value: "i64", mutable: true }, 1337n);
const o4 = {
    "a": 41,
    "b": 42,
};

v0.set(1, o4);

const v12 = new WebAssembly.Instance(new WebAssembly.Module(new Uint8Array([
      .............
])),
{ imports: {
    import_0_v0: v0,
    import_1_v1: v1,
} });

v12.exports.w0();                    // w0 is an exported function.
```

What do we want to actually fuzz?

```
let v0 = new WebAssembly.Table({ element: "externref", initial: 10, maximum: 20 });
let v1 = new WebAssembly.Global({ value: "i64", mutable: true }, 1337n);
const o4 = {
    "a": 41,
    "b": 42,
};

v0.set(1, o4);

const v12 = new WebAssembly.Instance(new WebAssembly.Module(new Uint8Array([
     What do we do here?
])),
{ imports: {
    import_0_v0: v0,
    import_1_v1: v1,
} });

v12.exports.w0();                    // w0 is an exported function.
```

What do we want to actually fuzz?

```
let v0 = new WebAssembly.Table({ element: "externref", initial: 10, maximum: 20 });
let v1 = new WebAssembly.Global({ value: "i64", mutable: true }, 1337n);
const o4 = {
    "a": 41,
    "b": 42,
};

v0.set(1, o4);

const v12 = new WebAssembly.Instance(new WebAssembly.Module(new Uint8Array([
     What do we do here?
])),
{ imports: {
    import_0_v0: v0,
    import_1_v1: v1,
} });

v12.exports.w0();                          // w0 is an exported function.
```

Making use of the powerful IL!

- Fuzzilli uses the FuzzIL to model JavaScript
- Can we extend the FuzzIL to model WebAssembly?
  - Well, yes, we actually can!
- We can design the IL to suit our needs and model what we want to fuzz
- We need to make some trade offs though, we can't have everything!

Making use of the powerful IL!

```
v0 <- CreateWasmGlobal i64: 1337, mutable
BeginWasmModule
    v1 <- WasmDefineGlobal wasmi64(1337)
    BeginWasmFunction ([] => wasmi64)
        v2 <- Consti64(42)
        v3 <- WasmLoadGlobal v0
        v4 <- WasmLoadGlobal v1
        v5 <- WasmI64BinOp v3 Add v4
        v6 <- WasmI64BinOp v5 Sub v2
        WasmReturn v6
    v7 <- EndWasmFunction
v8 <- EndWasmModule
v9 <- GetProperty 'exports', v8
v10 <- CallMethod 'w0', v9
```

Making use of the powerful IL!

```
v0 <- CreateWasmGlobal i64: 1337, mutable
BeginWasmModule
    v1 <- WasmDefineGlobal wasmi64(1337)
    BeginWasmFunction ([] => wasmi64)
        v2 <- Consti64(42)
        v3 <- WasmLoadGlobal v0
        v4 <- WasmLoadGlobal v1
        v5 <- WasmI64BinOp v3 Add v4
        v6 <- WasmI64BinOp v5 Sub v2
        WasmReturn v6
    v7 <- EndWasmFunction
v8 <- EndWasmModule
v9 <- GetProperty 'exports', v8
v10 <- CallMethod 'w0', v9
```

JavaScript context

Making use of the powerful IL!

```
v0 <- CreateWasmGlobal i64: 1337, mutable
BeginWasmModule
    v1 <- WasmDefineGlobal wasmi64(1337)
    BeginWasmFunction ([] => wasmi64)
        v2 <- Consti64(42)
        v3 <- WasmLoadGlobal v0
        v4 <- WasmLoadGlobal v1
        v5 <- WasmI64BinOp v3 Add v4
        v6 <- WasmI64BinOp v5 Sub v2
        WasmReturn v6
    v7 <- EndWasmFunction
v8 <- EndWasmModule
v9 <- GetProperty 'exports', v8
v10 <- CallMethod 'w0', v9
```

WebAssembly Module context

Making use of the powerful IL!

```
v0 <- CreateWasmGlobal i64: 1337, mutable
BeginWasmModule
    v1 <- WasmDefineGlobal wasmi64(1337)
    BeginWasmFunction ([] => wasmi64)
        v2 <- Consti64(42)
        v3 <- WasmLoadGlobal v0
        v4 <- WasmLoadGlobal v1
        v5 <- WasmI64BinOp v3 Add v4
        v6 <- WasmI64BinOp v5 Sub v2
        WasmReturn v6
    v7 <- EndWasmFunction
v8 <- EndWasmModule
v9 <- GetProperty 'exports', v8
v10 <- CallMethod 'w0', v9
```

WebAssembly Function context

Making use of the powerful IL!

- How do we pass information between the two target languages?
    - We have a type inference system that works across JavaScript and Wasm!

Type inference is a strong feature!

```
v0 <- CreateWasmGlobal i64: 1337, mutable  // v0 = object("WasmGlobal", i64)
BeginWasmModule
    v1 <- WasmDefineGlobal i64(1337)       // v1 = object("WasmGlobal", i64)
    BeginWasmFunction ([] => i64)
        v2 <- Consti64(42)                 // v2 = i64
        v3 <- WasmLoadGlobal v0            // v3 = i64
        v4 <- WasmLoadGlobal v1            // v4 = i64
        v5 <- WasmI64BinOp v3 Add v4       // v5 = i64
        v6 <- WasmI64BinOp v5 Sub v2       // v6 = i64
        WasmReturn v6
    v7 <- EndWasmFunction                  // v7 = method([] => i64)
v8 <- EndWasmModule                        // v8 = object("WasmModule",
                                               properties: "exports")
v9 <- GetProperty 'exports', v8            // v9 = object("WasmModuleExports",
                                               methods: [] => wasmi64,
                                               properties: [wasmi64, wasmi64])
v10 <- CallMethod 'w0', v9                 // v10 = wasmi64
```

The life of a generated program

```
1.                                        // javascript context
```

Detour: CodeGenerators

```
CodeGenerator("WasmGlobalGenerator", inContext: .javascript) { b in
    let wasmGlobal: WasmGlobal = withEqualProbability({
        return .wasmf32(Float32(b.randomFloat()))
    }, {
        return .wasmf64(b.randomFloat())
    }, {
        return .wasmi32(Int32(truncatingIfNeeded: b.randomInt()))
    }, {
        return .wasmi64(b.randomInt())
    })
    b.createWasmGlobal(value: wasmGlobal, isMutable: probability(0.5))
}
```

FuzzIL: CreateWasmGlobal i64: 1337, mutable

JavaScript: new WebAssembly.Global({ value: "i64", mutable: true }, 1337);

Detour Macro Instructions

```
new WebAssembly.Global({ value: "i64", mutable: true }, 1337);
```

- This sequence of JavaScript can be lifted with Fuzzilli’s regular instructions
- Any mutation could easily invalidate the return value

Macro Instructions in the IL can help us!

- CreateWasmGlobal(type, mutability, value)

Detour Macro Instructions

```
new WebAssembly.Global({ value: "i64", mutable: true }, 1337);
```

- This sequence of JavaScript can be lifted with Fuzzilli’s regular instructions
- Any mutation could easily invalidate the return value

Macro Instructions in the IL can help us!

- `CreateWasmGlobal(type, mutability, value)`

Mutate this!

Detour Macro Instructions

```
new WebAssembly.Global({ value: "i64", mutable: true }, 1337);
```

- This sequence of JavaScript can be lifted with Fuzzilli's regular instructions
- Any mutation could easily invalidate the return value

Macro Instructions in the IL can help us!

- CreateWasmGlobal(type, mutability, value)

Mutate these!

Detour Macro Instructions

```
new WebAssembly.Global({ value: "i64", mutable: true }, 1337);
```

- This sequence of JavaScript can be lifted with Fuzzilli’s regular instructions
- Any mutation could easily invalidate the return value

Macro Instructions in the IL can help us!

- CreateWasmGlobal(type, mutability, value)

This also informs our type inference system!

=> object(“WasmGlobal”, WasmGlobal(mutable, i64))

The life of a generated program

```
1. v0 <- CreateWasmGlobal i64: 1337, mutable
2.                                            // javascript context
```

Detour: CodeGenerators (2)

```
RecursiveCodeGenerator("WasmModuleGenerator", inContext: .javascript) { b in
    let m = b.buildWasmModule { m in
        b.buildRecursive()
    }
}
```

The life of a generated program

```
1. v0 <- CreateWasmGlobal i64: 1337, mutable
2. BeginWasmModule
3.                                          // Wasm context
```

The life of a generated program

```
v0 <- CreateWasmGlobal i64: 1337, mutable
BeginWasmModule
    v1 <- WasmDefineGlobal wasmi64(1337)
    BeginWasmFunction ([] => wasmi64)
                                            // WasmFunction context
```

The life of a generated program

```
v0 <- CreateWasmGlobal i64: 1337, mutable
BeginWasmModule
    v1 <- WasmDefineGlobal wasmi64(1337)
    BeginWasmFunction ([] => wasmi64)
        v2 <- Consti64(42)
        v3 <- WasmLoadGlobal v0
                                              // WasmFunction context
```

The life of a generated program

```
1.  v0 <- CreateWasmGlobal i64: 1337, mutable
2.  BeginWasmModule
3.      v1 <- WasmDefineGlobal wasmi64(1337)
4.      BeginWasmFunction ([] => wasmi64)
5.          v2 <- Consti64(42)
6.          v3 <- WasmLoadGlobal v0
7.          v4 <- WasmLoadGlobal v1
8.          v5 <- WasmI64BinOp v3 Add v4       // WasmBinOpGenerator
9.          v6 <- WasmI64BinOp v5 Sub v2       // WasmBinOpGenerator
10.         WasmReturn v6                      // WasmFunctionGenerator
                                                  fills in the return
11.     v7 <- EndWasmFunction
12. v8 <- EndWasmModule
13. v9 <- GetProperty 'exports', v8            // WasmModuleCallGenerator
14. v10 <- CallMethod 'w0', v9
```

Ok, now we just need a Wasm compiler...

- So the IL is cool, and we can describe Wasm Control- and Dataflow
- We built a custom compiler to compile our IL to Wasm
- Our IL is a “register machine” and Wasm is a “stack machine”
- The compiler will automatically wire up imports correctly!
- Why not compile to the WasmBuilder?

Compiler details

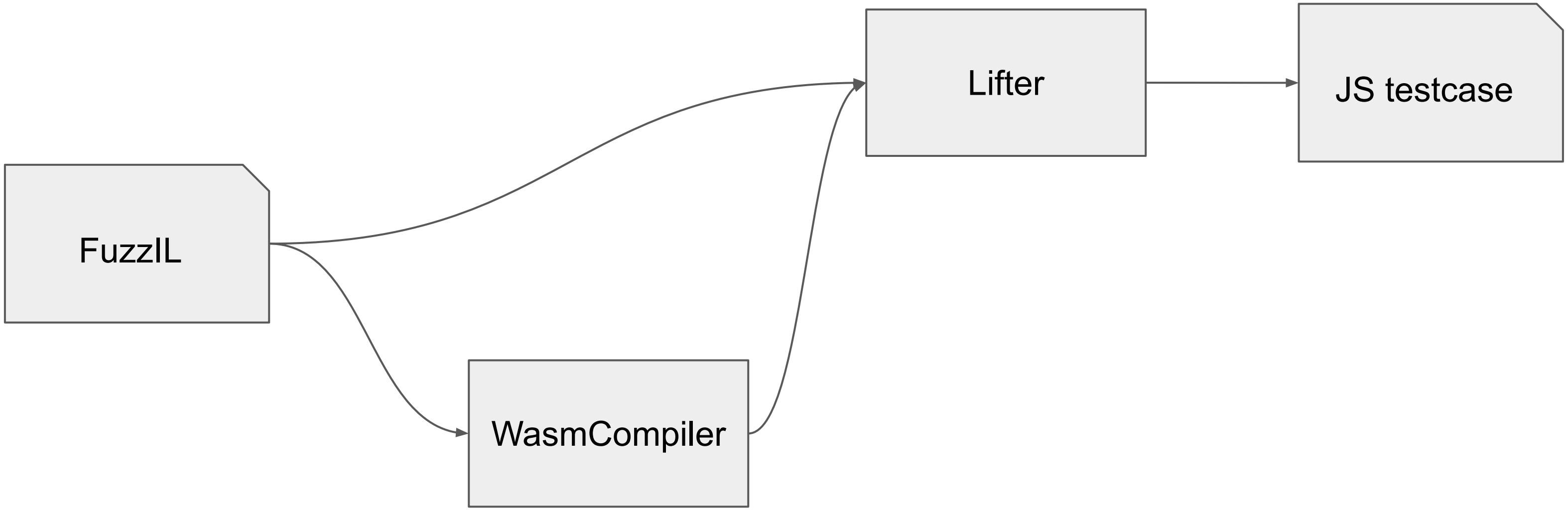
```
v0 <- CreateWasmGlobal i64: 1337, mutable
BeginWasmModule
    v1 <- WasmDefineGlobal wasmi64(1337)
    BeginWasmFunction ([] => wasmi64)
        ...
        v3 <- WasmLoadGlobal v0
        v4 <- WasmLoadGlobal v1
        ...

```

Seeing v0 here means we need to import it from JS!

Compiler details

```
let v0 = new WebAssembly.Global({ value: "i64", mutable: true }, 1337n);
const v8 = new WebAssembly.Instance(new WebAssembly.Module(new Uint8Array([
    0x00, 0x61, 0x73, 0x6D, 0x01, 0x00, 0x00, 0x00, 0x01, 0x05,
    0x01, 0x60, 0x00, 0x01, 0x7E, 0x02, 0x18, 0x01, 0x07, 0x69,
    0x6D, 0x70, 0x6F, 0x72, 0x74, 0x73, 0x0B, 0x69, 0x6D, 0x70,
    0x6F, 0x72, 0x74, 0x5F, 0x30, 0x5F, 0x76, 0x30, 0x03, 0x7E,
    0x01, 0x03, 0x02, 0x01, 0x00, 0x04, 0x01, 0x00, 0x05, 0x01,
    0x00, 0x06, 0x07, 0x01, 0x7E, 0x00, 0x42, 0xB9, 0x0A, 0x0B,
    0x07, 0x12, 0x03, 0x02, 0x77, 0x30, 0x00, 0x00, 0x03, 0x77,
    0x67, 0x31, 0x03, 0x00, 0x03, 0x77, 0x67, 0x30, 0x03, 0x01,
    0x0A, 0x2B, 0x01, 0x29, 0x05, 0x01, 0x7E, 0x01, 0x7E, 0x01,
    0x7E, 0x01, 0x7E, 0x01, 0x7E, 0x42, 0x2A, 0x21, 0x00, 0x23,
    0x00, 0x21, 0x01, 0x23, 0x01, 0x21, 0x02, 0x20, 0x01, 0x20,
    0x02, 0x7C, 0x21, 0x03, 0x20, 0x03, 0x20, 0x00, 0x7D, 0x21,
    0x04, 0x20, 0x04, 0x0F, 0x0B,
])),
{ imports: {
    import_0_v0: v0,
} });
v8.exports.w0();
```

FuzzIL Lifting / Compiling pipeline



The full sample

```
v0 <- CreateWasmGlobal i64: 1337, mutable
BeginWasmModule
    v1 <- WasmDefineGlobal wasmi64(1337)
    BeginWasmFunction ([] => wasmi64)
        v2 <- Consti64(42)
        v3 <- WasmLoadGlobal v0
        v4 <- WasmLoadGlobal v1
        v5 <- WasmI64BinOp v3 Add v4
        v6 <- WasmI64BinOp v5 Sub v2
        WasmReturn v6
    v7 <- EndWasmFunction
v8 <- EndWasmModule
v9 <- GetProperty 'exports', v8
v10 <- CallMethod 'w0', v9
```

```
let v0 = new WebAssembly.Global({ value: "i64", mutable: true }, 1337n);
const v8 = new WebAssembly.Instance(new WebAssembly.Module(new Uint8Array([
    0x00, 0x61, 0x73, 0x6D, 0x01, 0x00, 0x00, 0x00, 0x01, 0x05,
    0x01, 0x60, 0x00, 0x01, 0x7E, 0x02, 0x18, 0x01, 0x07, 0x69,
    0x6D, 0x70, 0x6F, 0x72, 0x74, 0x73, 0x0B, 0x69, 0x6D, 0x70,
    0x6F, 0x72, 0x74, 0x5F, 0x30, 0x5F, 0x76, 0x30, 0x03, 0x7E,
    0x01, 0x03, 0x02, 0x01, 0x00, 0x04, 0x01, 0x00, 0x05, 0x01,
    0x00, 0x06, 0x07, 0x01, 0x7E, 0x00, 0x42, 0xB9, 0x0A, 0x0B,
    0x07, 0x12, 0x03, 0x02, 0x77, 0x30, 0x00, 0x00, 0x03, 0x77,
    0x67, 0x31, 0x03, 0x00, 0x03, 0x77, 0x67, 0x30, 0x03, 0x01,
    0x0A, 0x2B, 0x01, 0x29, 0x05, 0x01, 0x7E, 0x01, 0x7E, 0x01,
    0x7E, 0x01, 0x7E, 0x01, 0x7E, 0x42, 0x2A, 0x21, 0x00, 0x23,
    0x00, 0x21, 0x01, 0x23, 0x01, 0x21, 0x02, 0x20, 0x01, 0x20,
    0x02, 0x7C, 0x21, 0x03, 0x20, 0x03, 0x20, 0x00, 0x7D, 0x21,
    0x04, 0x20, 0x04, 0x0F, 0x0B,
])),
{ imports: {
    import_0_v0: v0,
} });
v8.exports.w0();
```

Trade-offs

- We need to think about what we actually want to model
- We don't model the sections in the binary format at all
    - Automatically generated by the compiler
- We model control flow and data flow only, the rest is derived from it
- This limits some things we can fuzz for now
- The IL does not know about the Wasm value stack
    - We don't fuzz things there
- Being correct at compile time is easy with our type inference system
- Being correct at runtime is hard!

Mutating JS is easy, mutating Wasm is hard

```
v0 <- BeginPlainFunction ->
    v1 <- LoadInteger '42'
    v2 <- LoadString 'aaa'
    v3 <- LoadInteger '43'
    v4 <- BinaryOperation v1, '+', v3
    Return v4
EndPlainFunction
v5 <- CallFunction v0, []
```

```
v0 <- BeginPlainFunction ->
    v1 <- LoadInteger '42'
    v2 <- LoadString 'aaa'
    v3 <- LoadInteger '43'
    v4 <- BinaryOperation v1, '+', v2
    Return v4
EndPlainFunction
v5 <- CallFunction v0, []
```

Mutating JS is easy, mutating Wasm is hard

```
BeginWasmModule
    BeginWasmFunction [] ([] => .wasmi32)
        v0 <- Consti32 '42'
        v1 <- Consti32 '43'
        v2 <- Consti64 '44'
        v3 <- Wasmi32BinOp v0 Add v1
        WasmReturn v3
    v4 <- EndWasmFunction
v5 <- EndWasmModule
v6 <- GetProperty v5, 'exports'
v7 <- CallMethod v6, 'w0', []
```

```
BeginWasmModule
    BeginWasmFunction [] ([] => .wasmi32)
        v0 <- Consti32 '42'
        v1 <- Consti32 '43'
        v2 <- Consti64 '44'
        v3 <- Wasmi32BinOp v0 Add v2
        WasmReturn v3
    v4 <- EndWasmFunction
v5 <- EndWasmModule
v6 <- GetProperty v5, 'exports'
v7 <- CallMethod v6, 'w0', []
```

Case Study: JSPI

- JavaScript Promise Integration
- Support async function calls from WebAssembly
- Think WebAPI functions that return promises
- We need to wrap imports and exports
  - Somewhat complex dataflow

Case Study: JSPI

- Imports can be wrapped with WebAssembly.Suspending
  - Marks the function as a function that might suspend execution
- Exports can be wrapped with WebAssembly.promising
  - Marks the exported Wasm function as a function that returns a promise
- We can use Fuzzilli's ProgramTemplates to fuzz this interaction!

Case Study: JSPI

```
ProgramTemplate("JSPI") { b in
    b.buildPrefix()
    b.build(n: 20)

    let f = b.buildAsyncFunction(with: b.randomParameters()) { _ in
        b.build(n: 10)
    }

    let signature = b.type(of: f).signature ?? Signature.forUnknownFunction
    var wasmSignature = b.convertJsSignatureToWasmSignature(signature)
    let wrapped = b.wrapSuspending(function: f)

    let m = b.buildWasmModule { mod in
        mod.addWasmFunction(with: [] => .nothing) { fbuilder, _  in
            b.build(n: 20)
            let args = b.randomWasmArguments(forWasmSignature: wasmSignature)
            if let args {
                fbuilder.wasmJsCall(function: wrapped, withArgs: args, withWasmSignature: wasmSignature)
            }
        }
    }

    let exportedMethod = b.wrapPromising(function: b.getProperty("w0", of: m.loadExports()))

    b.callFunction(exportedMethod)

    b.build(n: 5)
}
```

Case Study: JSPI

```
ProgramTemplate("JSPI") { b in
    b.buildPrefix()
    b.build(n: 20)

    let f = b.buildAsyncFunction(with: b.randomParameters()) { _ in
        b.build(n: 10)
    }

    let signature = b.type(of: f).signature ?? Signature.forUnknownFunction
    var wasmSignature = b.convertJsSignatureToWasmSignature(signature)
    let wrapped = b.wrapSuspending(function: f)

    let m = b.buildWasmModule { mod in
        mod.addWasmFunction(with: [] => .nothing) { fbuilder, _  in
            b.build(n: 20)
            let args = b.randomWasmArguments(forWasmSignature: wasmSignature)
            if let args {
                fbuilder.wasmJsCall(function: wrapped, withArgs: args, withWasmSignature: wasmSignature)
            }
        }
    }

    let exportedMethod = b.wrapPromising(function: b.getProperty("w0", of: m.loadExports()))

    b.callFunction(exportedMethod)

    b.build(n: 5)
}
```

Case Study: JSPI

```
ProgramTemplate("JSPI") { b in
    b.buildPrefix()
    b.build(n: 20)

    let f = b.buildAsyncFunction(with: b.randomParameters()) { _ in
        b.build(n: 10)
    }

    let signature = b.type(of: f).signature ?? Signature.forUnknownFunction
    var wasmSignature = b.convertJsSignatureToWasmSignature(signature)
    let wrapped = b.wrapSuspending(function: f)

    let m = b.buildWasmModule { mod in
        mod.addWasmFunction(with: [] => .nothing) { fbuilder, _  in
            b.build(n: 20)
            let args = b.randomWasmArguments(forWasmSignature: wasmSignature)
            if let args {
                fbuilder.wasmJsCall(function: wrapped, withArgs: args, withWasmSignature: wasmSignature)
            }
        }
    }

    let exportedMethod = b.wrapPromising(function: b.getProperty("w0", of: m.loadExports()))

    b.callFunction(exportedMethod)

    b.build(n: 5)
}
```

Case Study: JSPI

```
ProgramTemplate("JSPI") { b in
    b.buildPrefix()
    b.build(n: 20)

    let f = b.buildAsyncFunction(with: b.randomParameters()) { _ in
        b.build(n: 10)
    }

    let signature = b.type(of: f).signature ?? Signature.forUnknownFunction
    var wasmSignature = b.convertJsSignatureToWasmSignature(signature)
    let wrapped = b.wrapSuspending(function: f)

    let m = b.buildWasmModule { mod in
        mod.addWasmFunction(with: [] => .nothing) { fbuilder, _  in
            b.build(n: 20)
            let args = b.randomWasmArguments(forWasmSignature: wasmSignature)
            if let args {
                fbuilder.wasmJsCall(function: wrapped, withArgs: args, withWasmSignature: wasmSignature)
            }
        }
    }

    let exportedMethod = b.wrapPromising(function: b.getProperty("w0", of: m.loadExports()))

    b.callFunction(exportedMethod)

    b.build(n: 5)
}
```

Bugs! crbug.com/338122900

```
function f0() {
  // Contains an empty WebAssembly function.
  const v8 = new WebAssembly.Instance(new WebAssembly.Module(new Uint8Array([
     ...
  ])));
  let v11 = WebAssembly.promising(v8.exports.w0);
  v11(f0, f0);

  // Contains a function that returns a float32
  const v57 = new WebAssembly.Instance(new WebAssembly.Module(new Uint8Array([
     ...
  ])));
  v57.exports.w0();
}

// Contains a call to f0
const v104 = new WebAssembly.Instance(new WebAssembly.Module(new Uint8Array([
    ...
])),
{ imports: {
    import_0_v0: f0,
} });

let v107 = WebAssembly.promising(v104.exports.w0);
v107();
```

Bugs! crbug.com/338122900

```
Received signal 11 SEGV_ACCERR 5652fa6e4a68

==== C stack trace ===============================

 [0x5652c0e7c2d3]
 [0x5652c0e7c222]
 [0x7f69dd8521a0]
 [0x5652fa6e4a68]
[end of stack trace]
```

```
gef➤  i r $pc
pc             0x555c858332d8      0x555c858332d8
gef➤  xinfo $pc
──────────────────────────────────────────── xinfo: 0x555c858332d8 ─
Page: 0x00555c85799000  →  0x00555c85872000 (size=0xd9000)
Permissions: rw-
Pathname: [heap]
Offset (from page): 0x9a2d8
Inode: 0
gef➤
```

The future: WasmGC

- The future of Wasm is WasmGC, a version of language that adds support for garbage collected languages with more complex types.
- The WasmGC type system is supposed to be very generalized and abstract to support a lot of underlying type systems from different source languages!
- We need to model these types in Fuzzilli’s type inference system
- We need to add instructions to model the operations on these types
- Build compiler support to emit the correct signatures

Takeaways

- Fuzzing two languages is cool if you have a unifying IL and typesystem
- Still hard to produce good test cases
- WebAssembly, its features and interactions are complex
  - We are still catching up!
  - This means it will have bugs!

Questions?