Fuzzilli

(Guided-)fuzzing for JavaScript engines

Samuel Groß (saelo@google.com)

Motivation

Cool bugs in JS engine runtime implementations, JIT compilers, etc.

```
var a = [1, 2, 3, 4, 5];
var i = {};
i.valueOf = function() {
    a.length = 1;
    return 5;
}
a.slice(0, i);
```

CVE-2016-4622

```
function hax(o) {
    o.a;
    Object.create(o);
    return o.b;
}

for (let i = 0; i < 100000; i++) {
    let o = {a: 42};
    o.b = 43;
    hax(o);
}
```

CVE-2018-17463

How to fuzz JavaScript Engines?

How to fuzz JavaScript Engines?

```
./js_shell < /dev/urandom
```

How to fuzz JavaScript Engines?

```
./js_shell < /dev/urandom
```

:/

Requirements

1. Valid syntax of produced samples

Syntactical Correctness

- Possible to achieve with grammar-based generative fuzzing
  - Example: [domato]
- Basic idea: formulate JavaScript language as context-free grammar
- Then apply random production rules

A.3 Statements

Statement[Yield, Await, Return] :
  BlockStatement[?Yield, ?Await, ?Return]
  VariableStatement[?Yield, ?Await]
  EmptyStatement
  ExpressionStatement[?Yield, ?Await]
  IfStatement[?Yield, ?Await, ?Return]
  BreakableStatement[?Yield, ?Await, ?Return]
  ContinueStatement[?Yield, ?Await]
  BreakStatement[?Yield, ?Await]
  [+Return] *ReturnStatement*[?Yield, ?Await]
  WithStatement[?Yield, ?Await, ?Return]
  LabelledStatement[?Yield, ?Await, ?Return]
  ThrowStatement[?Yield, ?Await]
  TryStatement[?Yield, ?Await, ?Return]
  DebuggerStatement

Excerpt from the ECMAScript grammar

Grammar-based Fuzzing

```
...;
var v4 = new Array(42, v3, “foobar”);
for (var v5 = 0; v5 < 1000; v5++) {
    v4 = v5 * 7;
    var v6 = v4.slice(v1, v1, v2);
}
...;
```

Script generated by
grammar-based fuzzer

Grammar-based Fuzzing

```
...;
var v4 = new Array(42, v3, “foobar”);
for (var v5 = 0; v5 < 1000; v5++) {
    v4 = v5 * 7;
    var v6 = v4.slice(v1, v1, v2);
}
...;
```

Exception: TypeError: v4.slice is not a function.

Grammar-based Fuzzing

```
...;
var v4 = new Array(42, v3, “foobar”);
for (var v5 = 0; v5 < 1000; v5++) {
    v4 = v5 * 7;
    var v6 = v4.slice(v1, v1, v2);
}
...;
```

Exception: TypeError: v4.slice is not a function.

Following code is never executed...

Solution: Try-Catch ?

```
...;
var v4 = new Array(42, v3, “foobar”);
for(var v5 = 0; v5 < 1000; v5++) {
    v4 = v5 * 7;
    var v6 = v4.slice(v1, v1, v2);
}
...;
```

```
...;
try {
    var v4 = new Array(42, v3, “foobar”);
} catch(e) {}
for (var v5 = 0; v5 < 1000; v5++) {
    try {
        v4 = v5 * 7;
    } catch(e) {}
    try {
        var v6 = v4.slice(v1, v1, v2);
    } catch(e) {}
}
...;
```

Solution: Try-Catch ?

```
...;
var v4 = new Array(42, v3, "foobar");
for(var v5 = 0; v5 < 1000; v5++) {
    v4 = v5 * 7;
    var v6 = v4.slice(v1, v1, v2);
}
...;
```

```
...;
try {
    var v4 = new Array(42, v3, "foobar");
} catch(e) {}
for (var v5 = 0; v5 < 1000; v5++) {
    try {
        v4 = v5 * 7;
    } catch(e) {}
    try {
        var v6 = v4.slice(v1, v1, v2);
    } catch(e) {}
}
...;
```

Two pretty different things for a JIT compiler...

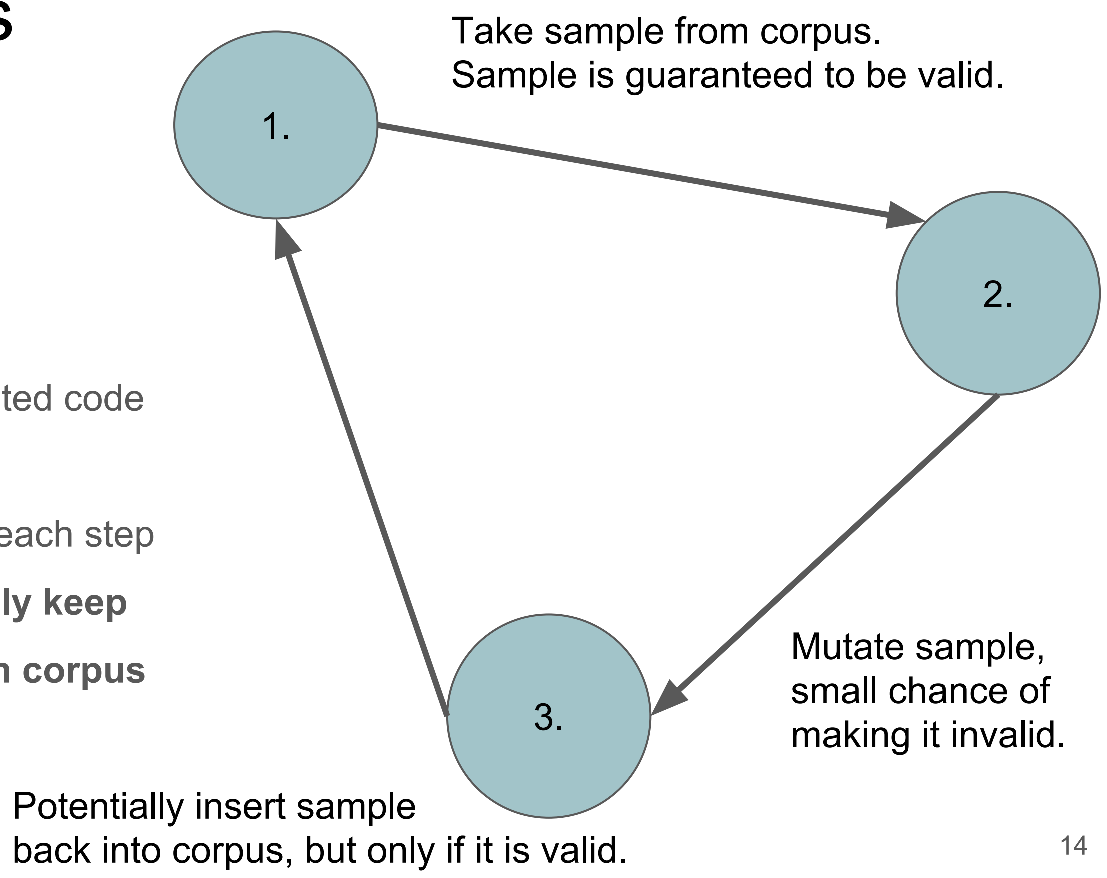
Requirements

1. Valid syntax of produced samples

2. High degree of semantic correctness

Semantic correctness

- Harder to achieve than syntactical correctness
- Multiple options:
  a. Precise type tracking in generated code
  b. Generate JavaScript code "step-by-step", validating after each step
  c. Use mutational approach, only keep semantically valid samples in corpus
  d. … ?



Requirements

1. Valid syntax of produced samples

2. High degree of semantic correctness

3. Definition of sensible mutations of JavaScript code

Mutating Programs

- Mutations possible at different “levels”:
- Observation: relevant are mostly control and data flow of the programs
- Syntactic representations are largely irrelevant for execution

=> Mutate at “bytecode” level

Source Code

Bit and byte mutations
String insertions, replacements, ...

Syntax Tree (AST)

Mutations of literals and identifiers
Subtree insertions, replacements, ...

Bytecode

Mutations of operands and “registers”
Instruction insertions, replacements, ...

FuzzIL

- Define custom intermediate language: “FuzzIL”
- Captures control and data flow
- Define mutations on the IL
- Translate IL to JavaScript for execution

```
; Example FuzzIL program
v0 <- LoadInt '0'
v1 <- LoadInt '10'
v2 <- LoadInt '1'
v3 <- Phi v0
BeginFor v0, '<', v1, '+', v2 -> v4
   v6 <- BinaryOperation v3, '+', v4
   Copy v3, v6
EndFor
v7 <- LoadString 'Result: '
v8 <- BinaryOperation v7, '+', v3
v9 <- LoadGlobal 'console'
v10 <- CallMethod v9, 'log', [v8]
```

FuzzIL - Lifting

```
v0 <- LoadInt '0'
v1 <- LoadInt '10'
v2 <- LoadInt '1'
v3 <- Phi v0
BeginFor v0, '<', v1, '+', v2 -> v4
   v6 <- BinaryOperation v3, '+', v4
   Copy v3, v6
EndFor
v7 <- LoadString 'Result: '
v8 <- BinaryOperation v7, '+', v3
v9 <- LoadGlobal 'console'
v10 <- CallMethod v9, 'log', [v8]
```

FuzzIL - Lifting

```
v0 <- LoadInt ‘0’
v1 <- LoadInt ‘10’
v2 <- LoadInt ‘1’
v3 <- Phi v0
BeginFor v0, ‘<’, v1, ‘+’, v2 -> v4
   v6 <- BinaryOperation v3, ‘+’, v4
   Copy v3, v6
EndFor
v7 <- LoadString ‘Result: ‘
v8 <- BinaryOperation v7, ‘+’, v3
v9 <- LoadGlobal ‘console’
v10 <- CallMethod v9, ‘log’, [v8]
```

```
// Trivial lifting
const v0 = 0;
const v1 = 10;
const v2 = 1;
let v3 = v0;
for (let v4 = v0; v4 < v1; v4 = v4 + v2) {
    const v6 = v3 + v4;
    v3 = v6;
}
const v7 = ”Result:”;
const v8 = v7 + v3;
const v9 = console;
const v10 = v9.log(v8);
```

FuzzIL - Lifting

```
v0 <- LoadInt '0'
v1 <- LoadInt '10'
v2 <- LoadInt '1'
v3 <- Phi v0
BeginFor v0, '<', v1, '+', v2 -> v4
   v6 <- BinaryOperation v3, '+', v4
   Copy v3, v6
EndFor
v7 <- LoadString 'Result: '
v8 <- BinaryOperation v7, '+', v3
v9 <- LoadGlobal 'console'
v10 <- CallMethod v9, 'log', [v8]
```

```
// Trivial lifting
const v0 = 0;
const v1 = 10;
const v2 = 1;
let v3 = v0;
for (let v4 = v0; v4 < v1; v4 = v4 + v2) {
    const v6 = v3 + v4;
    v3 = v6;
}
const v7 = "Result:";
const v8 = v7 + v3;
const v9 = console;
const v10 = v9.log(v8);
```

```
// Lifting with expression inlining
let v3 = 0;
for (let v4 = 0; v4 < 10; v4++) {
    v3 = v3 + v4;
}
console.log("Result:" + v3);
```

Mutating FuzzIL

```
v0 <- LoadGlobal  'print'
v1 <- LoadString  'Hello World'
v2 <- CallFunction v0, v1
```

Mutating FuzzIL

```
v0 <- LoadGlobal  ‘print’
v1 <- LoadString  ‘Hello World’
v2 <- CallFunction v0,  v0
```

Input Mutator

```
v0 <- LoadGlobal  ‘print’
v1 <- LoadString  ‘Hello World’
v2 <- CallFunction v0, v1
```

Mutating FuzzIL

```
v0 <- LoadGlobal  'print'
v1 <- LoadString  'Hello World'
v2 <- CallFunction v0, v0
```

Input Mutator

```
v0 <- LoadGlobal  'print'
v1 <- LoadString  'Hello World'
v2 <- CallFunction v0, v1
```

Operation Mutator

```
v0 <- LoadGlobal  'encodeURI'
v1 <- LoadString  'Hello World'
v2 <- CallFunction v0, v1
```

Mutating FuzzIL

```
v0 <- LoadGlobal  'print'
v1 <- LoadString  'Hello World'
v2 <- CallFunction v0, v1
```

Input Mutator

```
v0 <- LoadGlobal  'print'
v1 <- LoadString  'Hello World'
v2 <- CallFunction v0, v0
```

Operation Mutator

```
v0 <- LoadGlobal  'encodeURI'
v1 <- LoadString  'Hello World'
v2 <- CallFunction v0, v1
```

Insertion Mutator
(Generates new code)

```
v0 <- LoadGlobal  'print'
v1 <- LoadString  'Hello World'
v2 <- LoadProperty v0, 'foo'
v3 <- CallFunction v0, v1
```

Mutating FuzzIL

```
v0 <- LoadGlobal 'print'
v1 <- LoadString 'Hello World'
v2 <- CallFunction v0, v1
```

Input Mutator

```
v0 <- LoadGlobal 'print'
v1 <- LoadString 'Hello World'
v2 <- CallFunction v0, v0
```

Operation Mutator

```
v0 <- LoadGlobal 'encodeURI'
v1 <- LoadString 'Hello World'
v2 <- CallFunction v0, v1
```

Splice Mutator
(Inserts existing code)

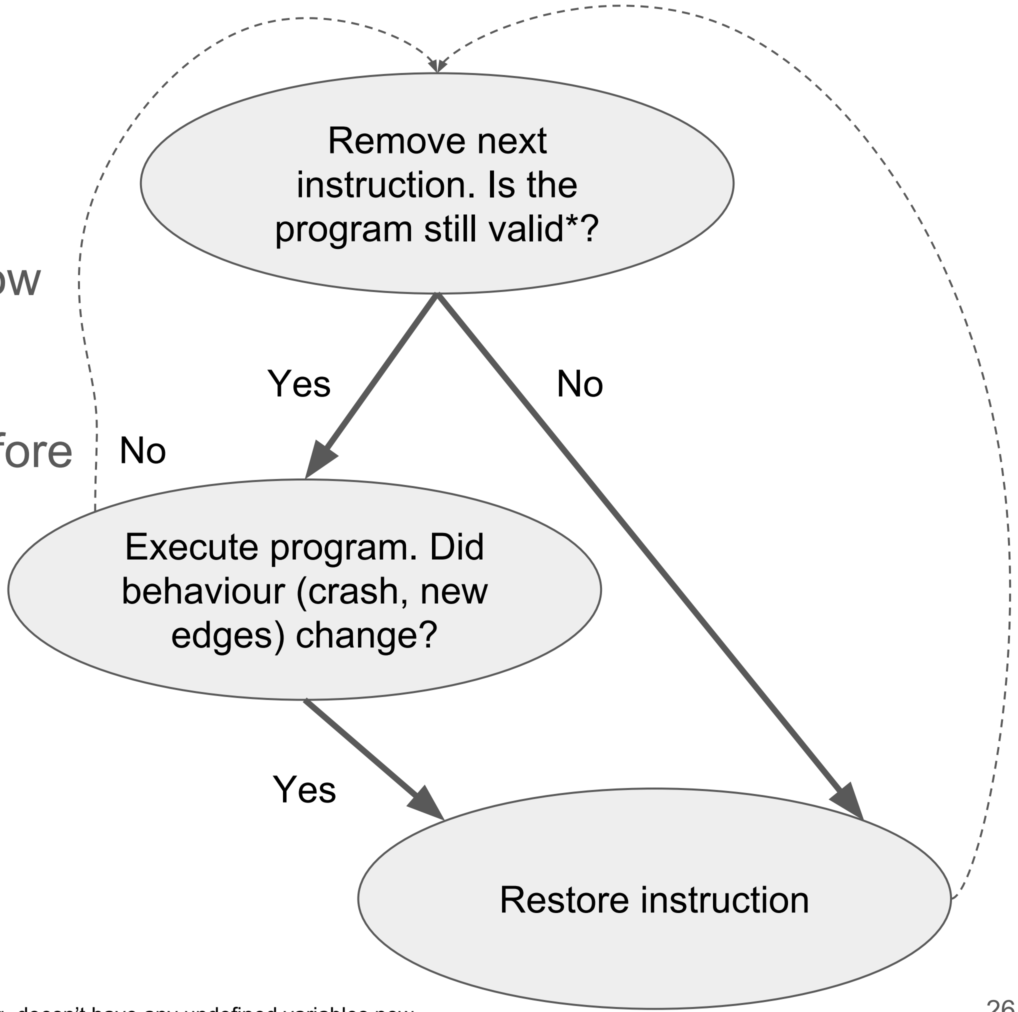
```
v0 <- LoadGlobal 'print'
v1 <- LoadString 'Hello World'
v2 <- LoadGlobal 'print'
v3 <- CallFunction v0, v1
```

Insertion Mutator
(Generates new code)

```
v0 <- LoadGlobal 'print'
v1 <- LoadString 'Hello World'
v2 <- LoadProperty v0, 'foo'
v3 <- CallFunction v0, v1
```

Minimization

- Problem: mutations can only grow a program in size
- Solution: minimize programs before inserting then into the corpus
- Simple algorithm: remove one instruction (starting at end) and check if behaviour changed
- But very expensive…
    - Room for improvement here!



* E.g. doesn’t have any undefined variables now

Guided Fuzzing

- Have mutation-based fuzzer

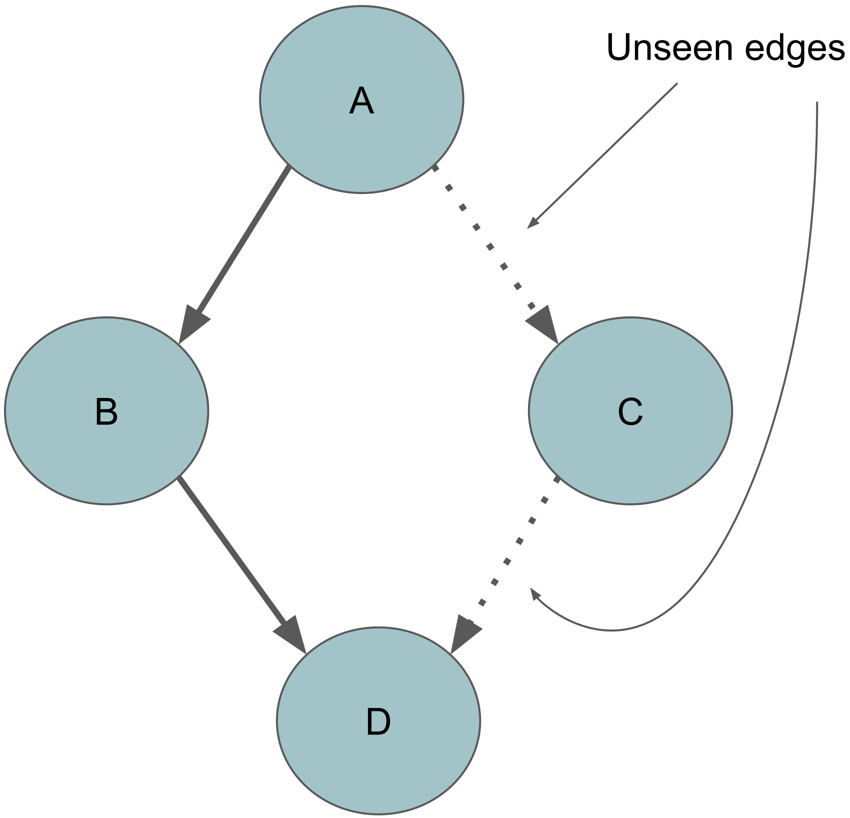
  => Plug in a feedback system and keep "interesting" programs for future mutations

- Currently implemented:

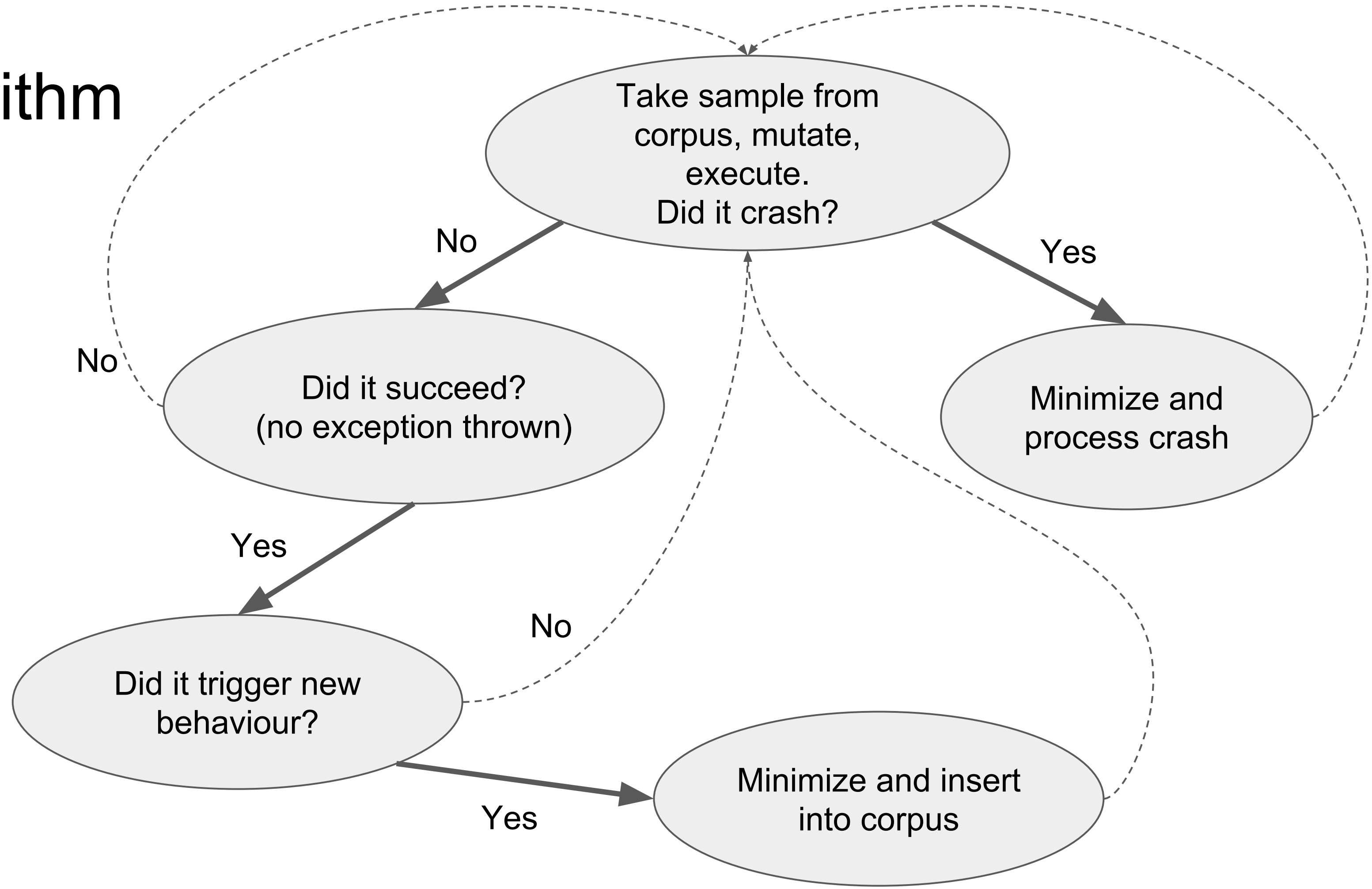
  edge-coverage, similar to afl

  - For JIT, only coverage in the compiler though!

- Easily replaced by different metrics
  - Ideas anyone?

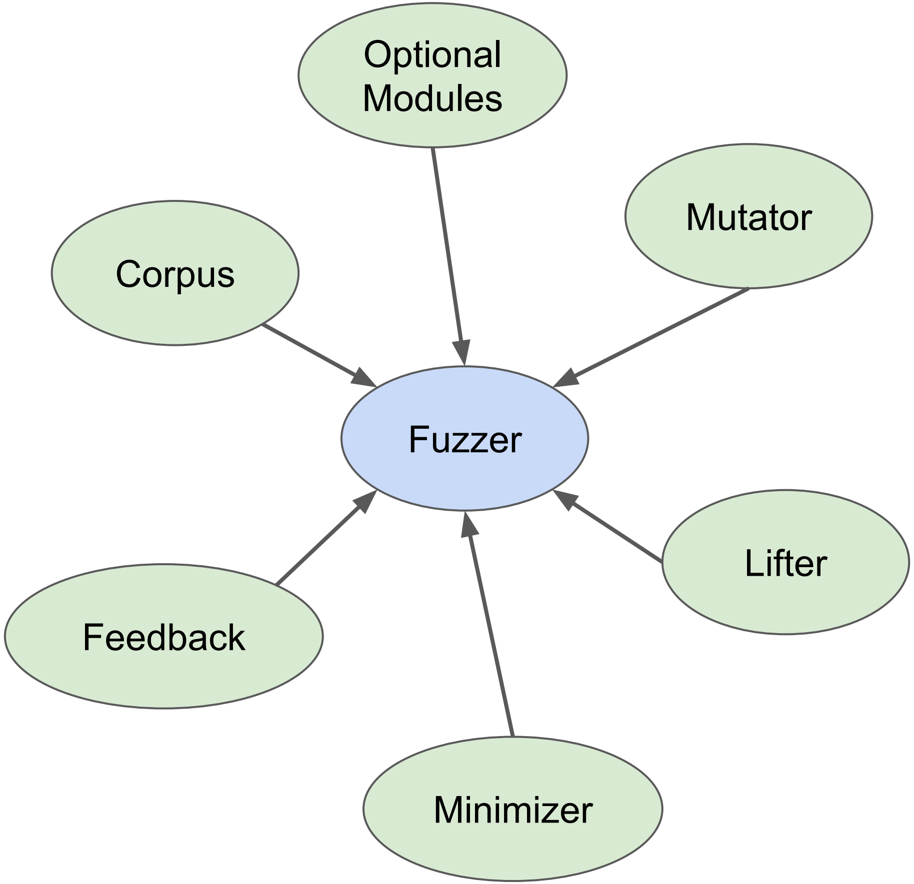


Algorithm

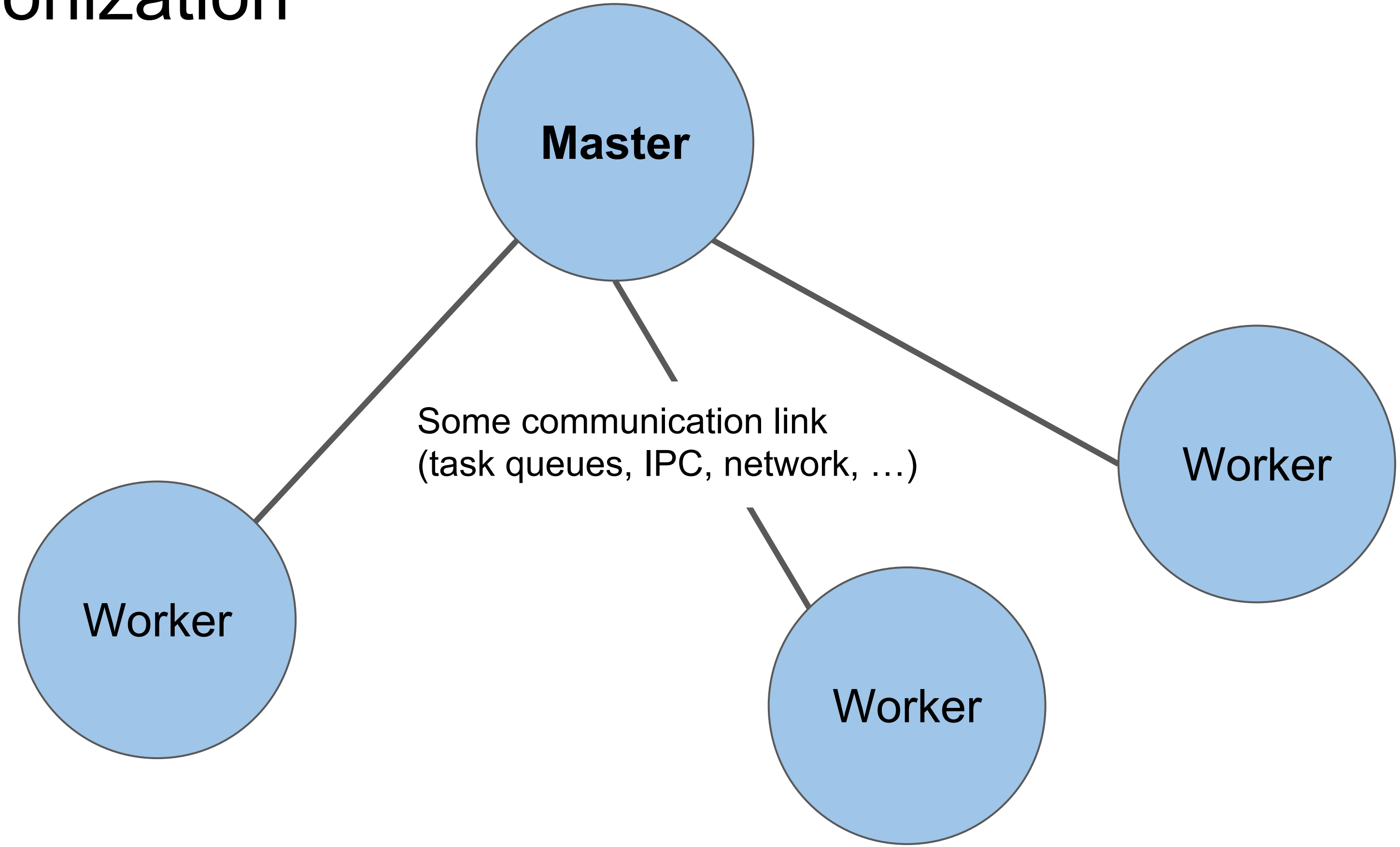

Take sample from corpus, mutate, execute. Did it crash?
No
Yes
Did it succeed? (no exception thrown)
Minimize and process crash
No
Yes
No
Did it trigger new behaviour?
Yes
Minimize and insert into corpus

Architecture

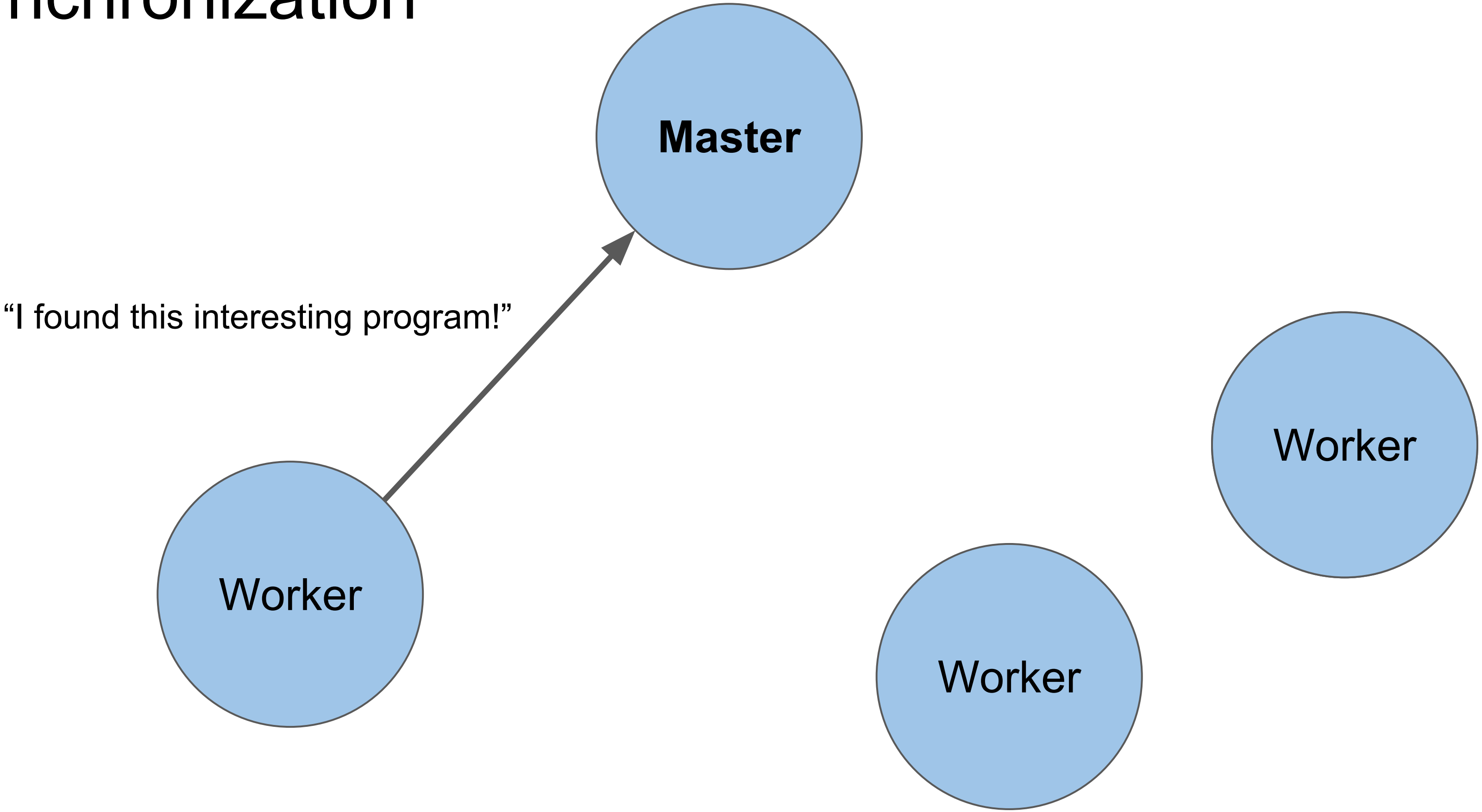
- 1 fuzzer instance per target process
  - No locking of e.g. corpus required
  - Simplifies code because program execution is synchronous
- Synchronization over IPC/network
- Programs can be imported from another instance, will be executed and compared against local corpus



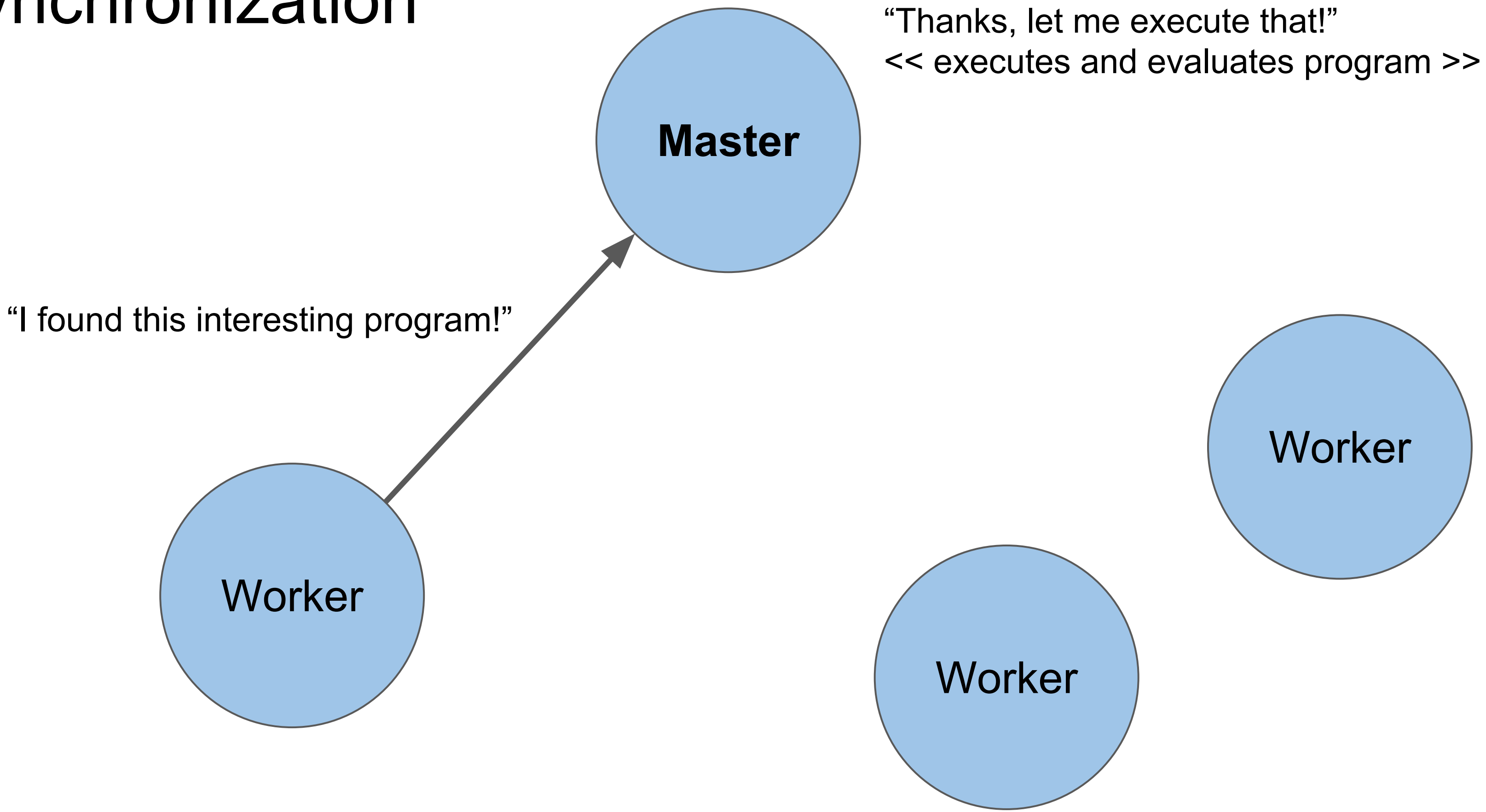
Synchronization



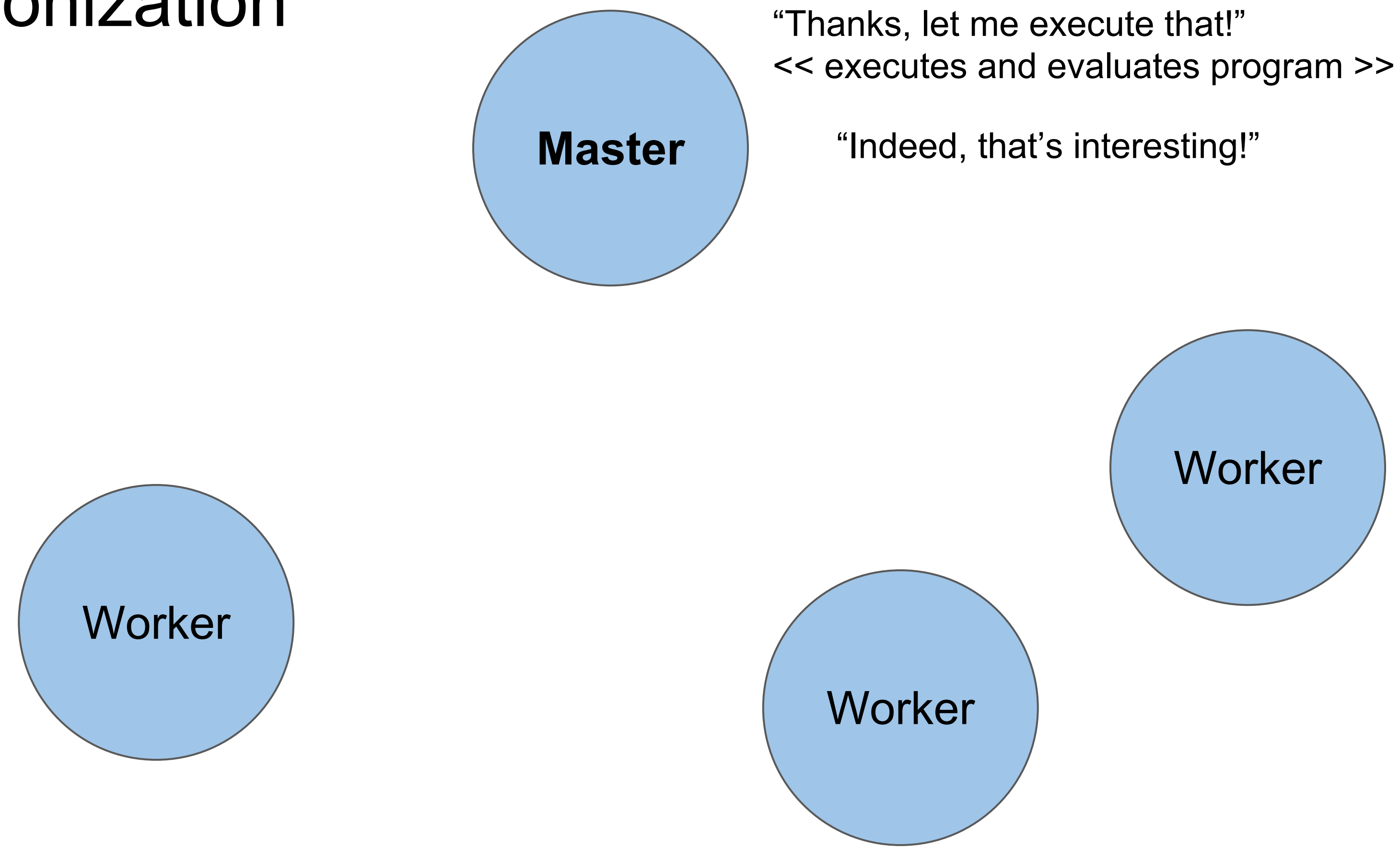
Synchronization



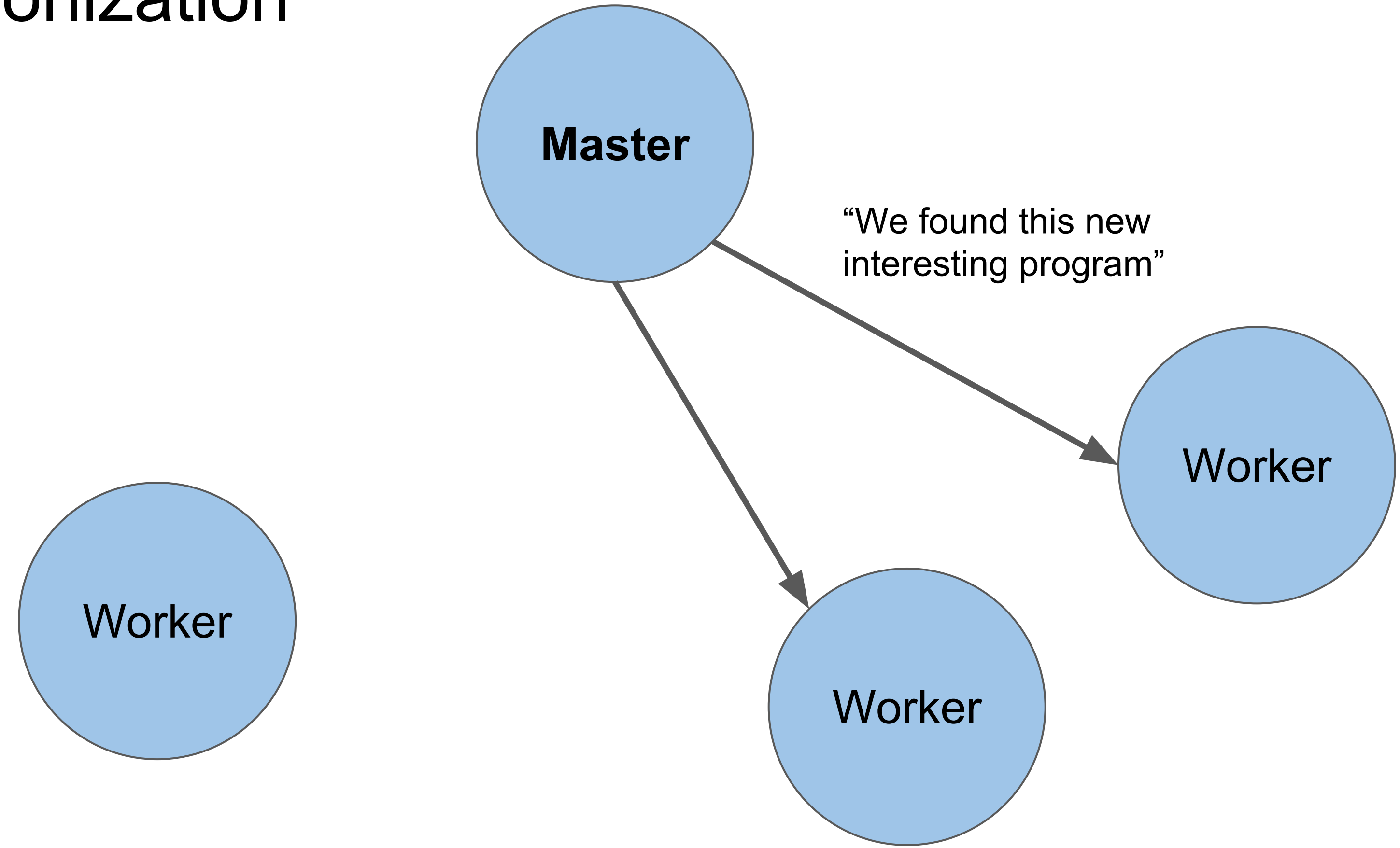
Synchronization



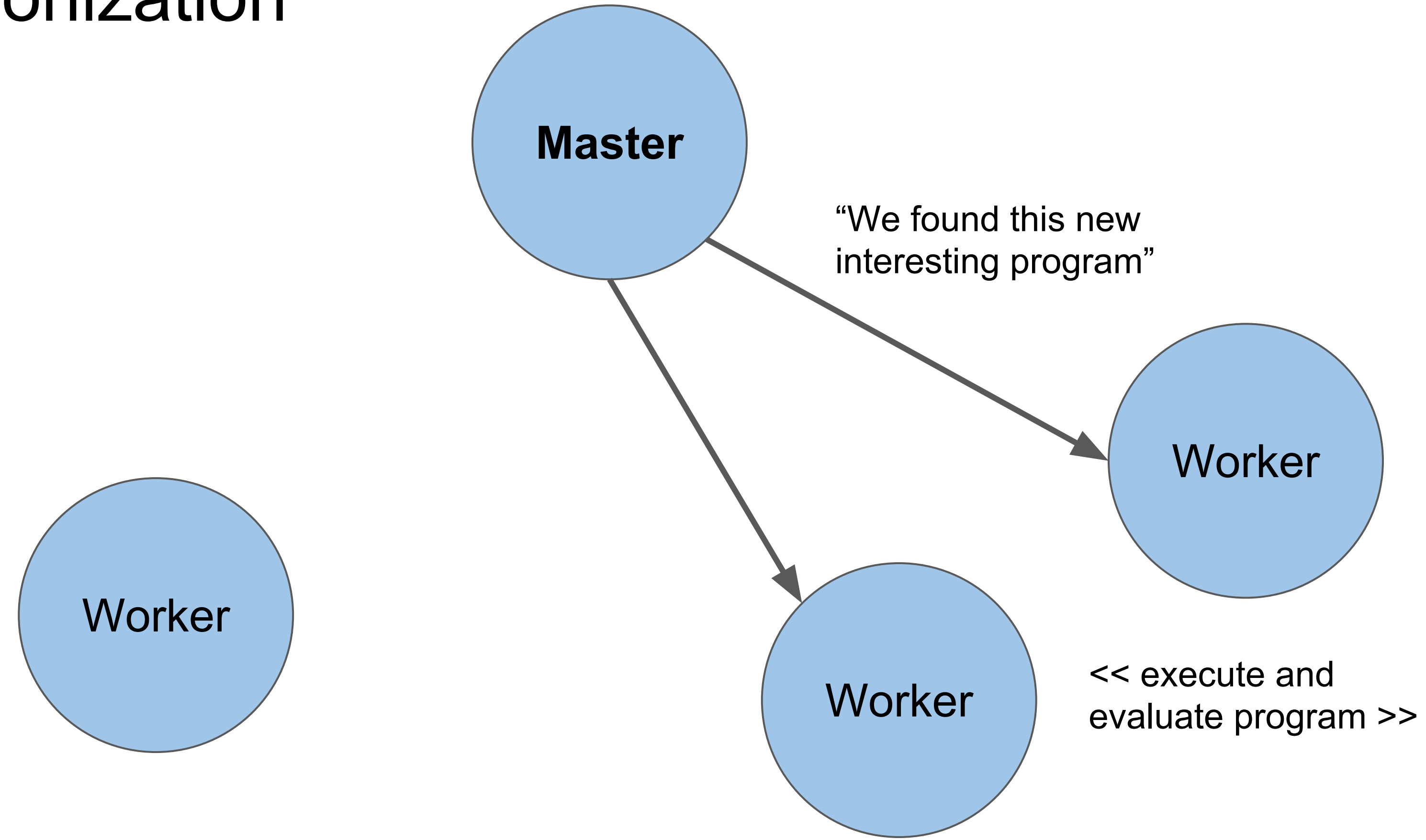
Synchronization



Synchronization

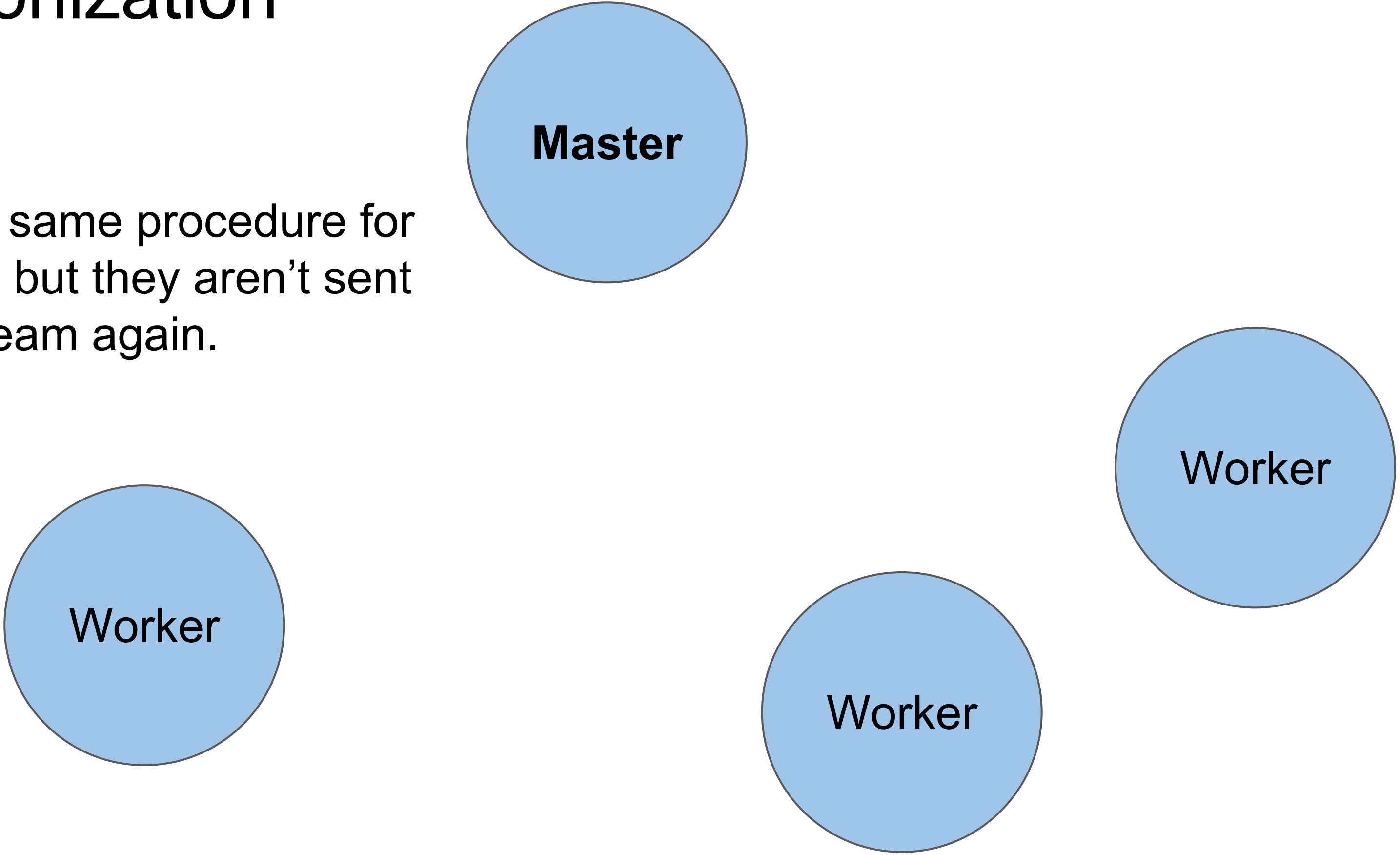


Synchronization



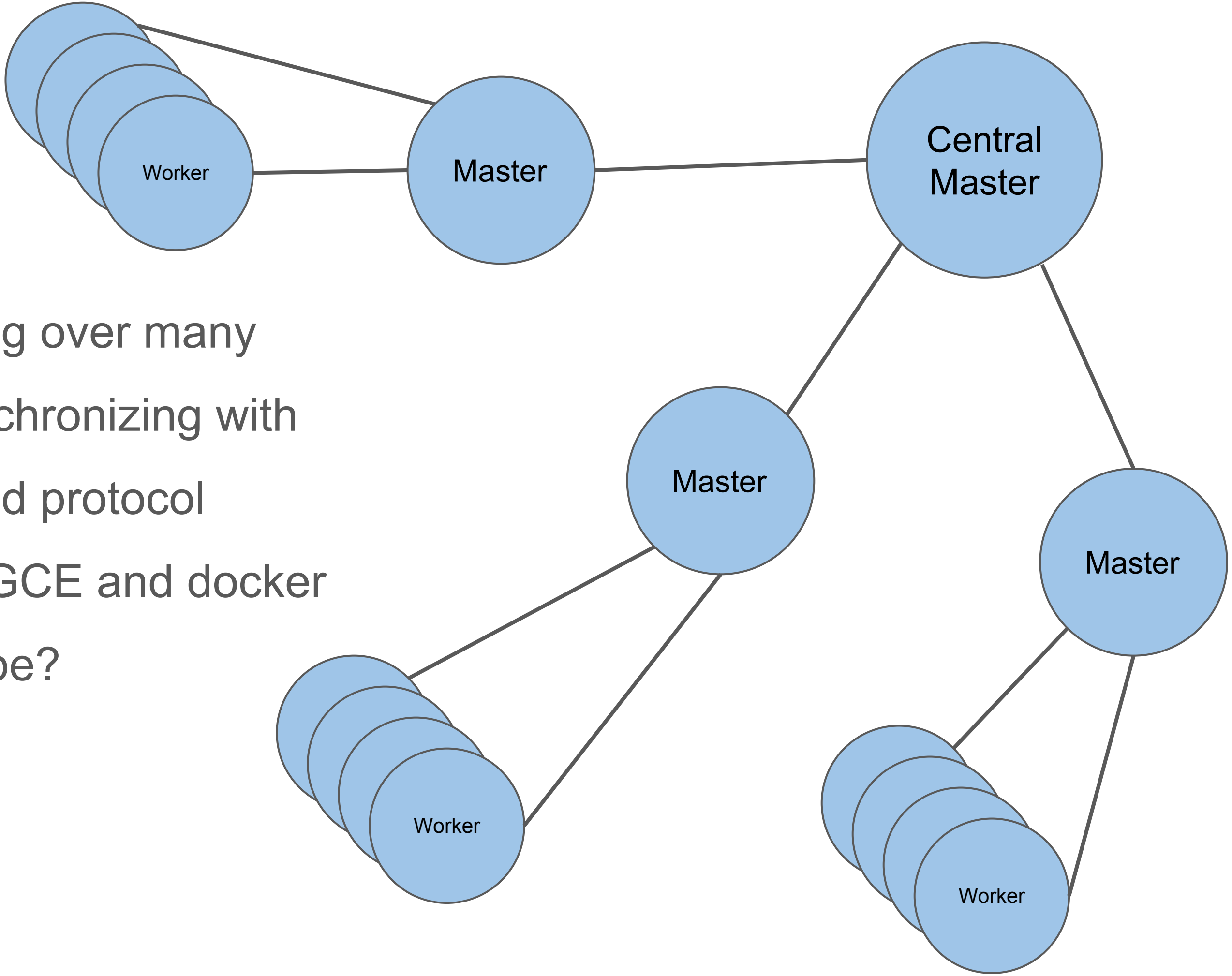
Synchronization

Roughly same procedure for crashes, but they aren't sent downstream again.



Scaling...

- Distributed fuzzing over many machines by synchronizing with simple TCP-based protocol
- Easy setup with GCE and docker
- Kubernetes maybe?



Results

- Currently supported: JavaScriptCore, Spidermonkey, v8
- Some results from last year:
    - Numerous unique crashes (>50 or >100 or so…)
        - Many assertion failures in debug builds, misbehaviour but no security impact, nullptr derefs, crashes in HEAD but not (yet) RELEASE, etc. Analysis often tedious...
    - 2 CVEs in JavaScriptCore (CVE-2018-4299, CVE-2018-4359)
    - 1 CVE in Spidermonkey (CVE-2018-12386)
        - Cool register allocation bug, used in Hack2Win competition =)
- Now running on > 1 server
    - ...

Roadmap

- Next few weeks:
  - Clean up code
  - Put into review for release
  - Wait for current bugs to be fixed, probably ...
- Open source release!
- Afterwards:
  - Implement “compiler” JavaScript -> FuzzIL
  - Extend FuzzIL language features
  - Experiment with more generative approaches (“Hybrid-fuzzing”?)
  - Better type tracking/prediction
  - Play with different instrumentations, also custom ones
  - Much much more ...

Wrap-up

Summary:

- Guided fuzzing of JS engines by mutating a custom IL
- Fairly generic code mutation engine

Watch this space: https://github.com/googleprojectzero/fuzzilli

Looking for collaborators! :)