Webassembly Security

Related Work

1. JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification, ICSE 2021
2. Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer, USENIX Security 2020
3. Fuzzing JavaScript Engines with Aspect-preserving Mutation, IEEE S&P 2020
4. JVM Fuzzing for JIT-Induced Side-Channel Detection, ICSE 2020
5. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines, NDSS 2019
6. FuzzIL: Coverage Guided Fuzzing for JavaScript Engines 2018
7. Random Testing for C and C++ Compilers with YARPGen 2020
8. Differential Fuzzing the WebAssembly 2020

01

- ❏ Difference between JIT fuzzing and normal fuzzing
- ❏ Paper Environment setup
- ❏ Basic knowledge
  - ❏ Compilation principle
  - ❏ Javascript
  - ❏ AFL
  - ❏ Linux
  - ❏ C++
  - ❏ JS engine

JIT in JSC (old)

Js source code →

Lexer: lexical analysis to Tokens →

Parser : tokens to AST → traverse AST to bytecode

Bytecode is the high-level intermediate representation

Now jsc doesn't need js source code anymore, all operations relay on bytecode

LLInt : execute bytecode

→ low performance?

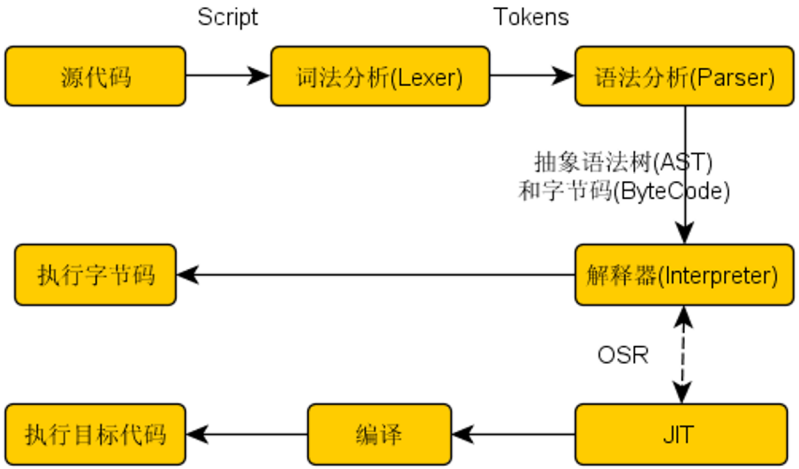
Using Baseline JIT, compile bytecode to machine code

→ still low performance?

Using DFG JIT, re-compile bytecode and optimize to machine code

→ still low performance?

Virtual machine (LLVM), compile IR code of DFG to optimized machine code



JIT in JSC

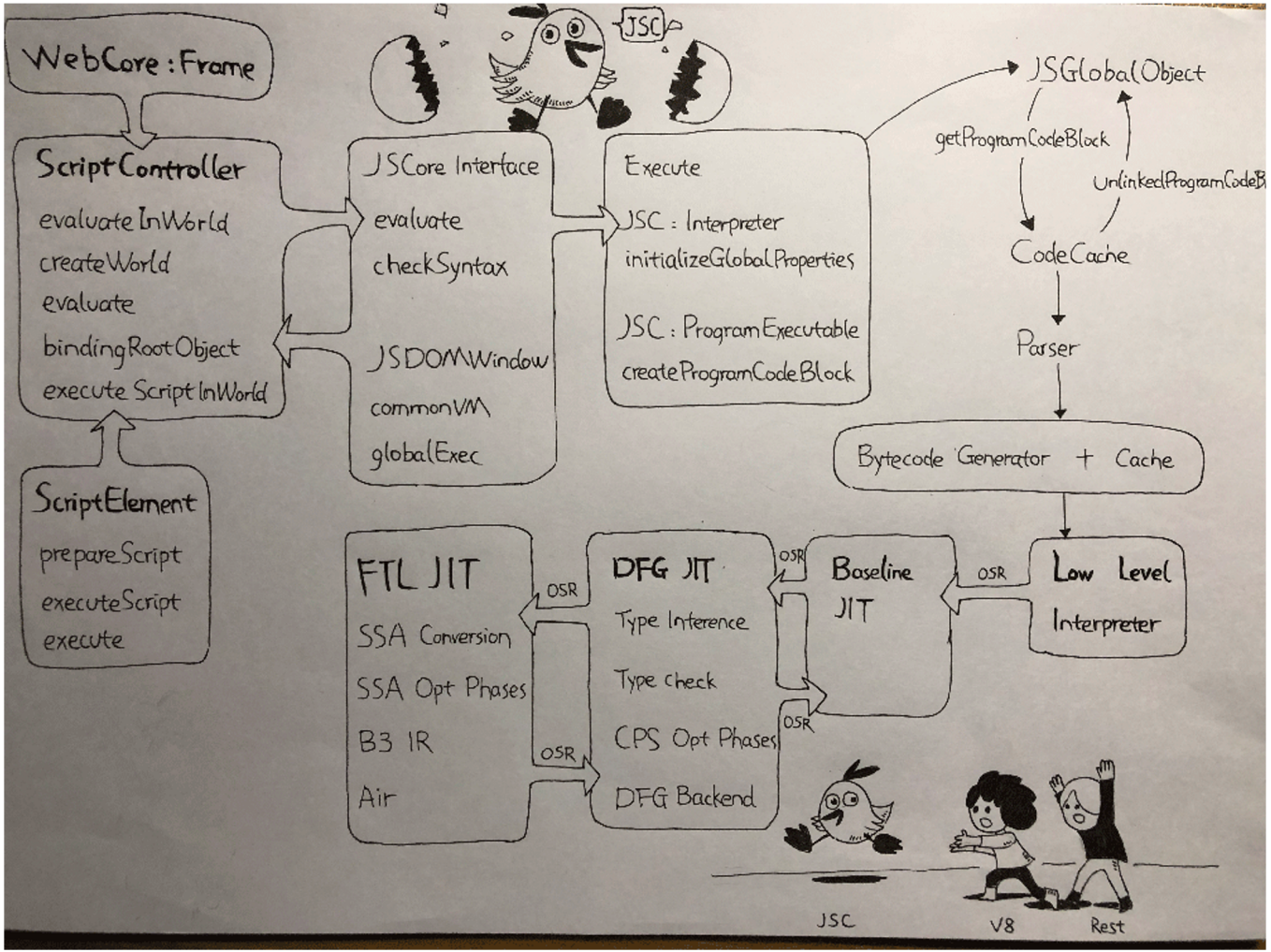
Baseline JIT: real-time compilation. Triggered when the functinois called 6 times or a certain code is looped more than 100 times

DFG JIT: low latency optimization. Triggered when the function is called 60 times or a certain code is looped more than 1000 times.

Will collect parameters from LLint and Baseline JIT, to do type judgment. If type prediction fails, DFG will cancel the optimization, also called OSR exit.

If exit to Baseline JIT for several times, it will re-optimize, collect more info and then call DFG.

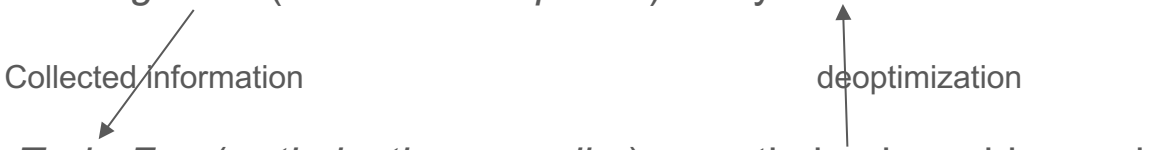
FTL JIT: B3 tailored LLVM. Using B3 instead of LLVM to reduce memory overhead



JIT in V8

The JIT process in V8 is similar to JSC.

Js source code → *parser* → AST → *Ignition* (*baseline interpreter*) → bytecode → execute



Difference between JIT fuzzing and normal fuzzing

In JIT, code will run and be compiled at the same times.

JIT engine will compile the hot code based on the run-time information, and sometimes will deoptimize.

This process is much more complicated than just using compiler or interpreter, which means fuzzing should consider more cases to trigger these feature and not stop by normal errors and exception before triggering JIT.

· Test cases cannot be called only one time ---- optimization compiler will not be triggered.

· need to call functions for different times --- to trigger different JIT strategies

· need to call the same function with different type parameters --- to trigger deoptimization

Environment setup

1. Montage -- NNL model

   It is tested on a machine running Ubuntu 20.04 with GTX Titan XP GPUs. Python 3.8 and PyTorch 1.4.0 with CUDA are required to run Montage.

1. Favocado -- fuzzing binding code

   only contain core parts. you may need to implement new binding objects

Environment setup

3. Superion : Grammar-Aware Greybox Fuzzing

    on Ubuntu 16.04 with gcc-5.4.0 and clang-3.8.

    Successfully setup, but having bugs when try to fuzz WebKit:

> To fuzz it using AFL or Superion, we first need to instrument the executable.

```
export CC=~/path_to_Superion/afl-clang-fast
export CXX=~/path_to_Superion/afl-clang-fast++
export AFL_HARDEN=1
./Tools/Scripts/build-jsc --jsc-only --j14
```

It seems that the ICU version is too low but still not working after the upgrade.

Environment setup

4. Die : Aspect - preserving mutation

Tested on Ubuntu 18.04 with Python 3.6.10, npm v6.14.6, n v6.7.0.

Missing /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor, should use cpufrequtils to set GOVERNOR=”performance”

Seems that it is a kernel bug

Successfully setup

Make Corpus Directory using make_initial_corpus.py ----- need to modify *n_cpu* (*multiprocessing.cpu_count() -4* to *1*)

No error information but the nothing happens after run the .sh file

AFL

Setup original AFL2.52b on Ubuntu 18.04

Use AFL to do some simple fuzzing

todo

AFL -- projects of papers are all based on AFL

JIT -- details of JIT in JS cores

Basic -- LLVM, js, linux

02

- ❏ V8 engine details
- ❏ AFL details
- ❏ Method to improve jit-fuzzing

V8 -- AST

AST is the key stage of JS engine, V8 engine used to directly compile AST rather than using bytecode.

The paper “aspect mutation” also focused on AST stage, trying to reserve type and structure from JS source code to AST.

The NNL paper tries to use neural network to learn rules of effective input case with serialized AST.

Seems that using AST to produce valid corpus is a effective way to fuzzing JIT

V8 -- AST

V8 AST process:

Source JS code

→ lexical analysis scanner : tokens, to identify the structure, type and value of the source code. Using automate machine

→ syntax analysis parser : transfer tokens to objects,

V8 uses its own syntax standard to increase compile speed.

V8 -- bytecode

V8 will translate AST to bytecode using BytecodeGenerator class with Ignition engine.

Ignition code directly execute bytecode, which means it is a virtual machine like JVM, actually JVM and V8 are both written by Lars Bak.

Is bytecode a kind of assembly language?

V8 -- optimization

?

Will the optimization and deoptimization influence the improvement of jit fuzzing?

AFL

1. Coverage measurements

   Insert instrumentation in the compiled program, to catch the branch and the branch-taken hit counts.

   Record the code route as tuples.

1. Detecting new behaviors

   Map all tuple to find the input case which causes new tuple

   Put this input case into input queue

AFL

3. Evolving the input queue

Mutated test cases, which will cause new tuples, will be add to input queue as the start point of next fuzz.

AFL will use some strategies to randomly mutate the seed, which means most of the mutation are not valid, or will only cause simple crash.

And that is the reason why DIE and NNL tries to modify AFL to produce effective corpus.

AFL

4. Culling the corpus

The test cases in the input queue are iteratively mutated by the former cases, which means the coverage of the new cases are the superset of the former ones.

Therefore, AFL will periodically re-evaluates the input queue to choose a small queue that could still coverage all tuples.

Improve jit fuzzing

How to fuzz effectively?

1. Optimize the strategies to produce high-quality input cases (AST).

   DIE, NNL

   Reserving the feature of the existed seed.

1. Try to optimize corpus culling?
2. Try to optimize bytecode generating rather than AST?

todo

JIT

AFL

Practice

Compilation principle

Virtual machine

How to fuzz jit?

03

- ❏ Automate program generator
- ❏ Differential testing
- ❏ Environment setup

- ❏ Fuzzilli

Program generator

- **Generative Fuzzing**

In generative fuzzing, each input file is generated from scratch, often following a set of predefined rules. Commonly, grammar-based approaches are used, in which the set of all inputs is first defined through a context-free grammar [GKL08]. The generation of new inputs is then achieved by taking random production rules starting from an initial ”root” production.

- pros
  - the resulting samples will always be syntactically valid if the grammar is correct.
  - high degree of control over the produced samples by the grammar engineer, easy to tune
- Cons
  - hard to achieve semantic correctness without some form of code emulation
    - runtime exception would stop further processing of the generated code
    - ✓ Common solution: wrap each generated statement or expression into a try-catch block so that execution continues even in the event of a runtime exception.

[GKL08] Godefroid, Patrice, Adam Kiezun and Michael Y Levin: Grammar- based whitebox fuzzing. In ACM Sigplan Notices, volume 43, pages 206–215. ACM, 2008.

Goals of mutation

- all generated programs have to be syntactically correct under all circumstances.
  - syntactically invalid programs will most of the time fail to reach past the parsing stage of the engine.
  - Parser is a small and simple part of the whole javascript engine,
- define sensible mutations to preserve most of the interesting features of existing samples
  - necessary to define mutations that are able to change the control and data flow of a program
  - Goal of coverage-guided fuzz, goal of JIT compiler
- high percentage of semantically valid samples
  - Complete semantic correctness is not desirable, cause vulnerabilities often occur due to unexpected internal exceptions being raised

Fuzzilli

- define a custom intermediate language which we call FuzzIL.
  - enables to define new mutation strategies that could not easily be implemented as AST mutations

- mutate on a ”bytecode” level which is closer to the engines internal representation of the code.
  - Instead of mutating syntactic constructs like the AST or even the textual source code representation
  - bytecode level resembles a control and data flow graph, which is what is ultimately required to reach the majority of the attack surface of a scripting engine
  - syntactic information, such as the AST, is largely discarded during the parsing step.

Fuzzilli

To achieve 3 requirements

- ❏ syntactical correctness -- guarantee that the conversion from our IL to JavaScript is always possible

- ❏ Define mutations to control and data flow of a program -- define new mutation strategies on “bytecode” level

- ❏ high percentage of semantically valid -- all mutations are required to obey to a set of basic semantic correctness rules of the IL,

Fuzzilli

Mutation on different levels:

- Source code: bit and byte mutation, String insertion, replacement…
- AST : mutations of literals and identifiers, subtree insertion, replacement…
- Bytecode : mutations of operands and registers, instruction insertion, replacement…

jackiequ@ubuntu:~/work/fuzzilli$ swift run -c release -Xlinker='-lrt' FuzzilliCli --profile=v8 --storagePath=./output ~/v8/v8/out/fuzzbuild/d8

[0/0] Build complete!

[Coverage] Initialized, 692962 edges

[JavaScriptEnvironment] initialized static JS environment model

[JavaScriptEnvironment] Have 56 available builtins: ["parseFloat", "JSON", "Set", "Int8Array", "TypeError", "Int32Array", "Promise", "SyntaxError", "String", "Error", "ReferenceError", "EvalError", "Int16Array", "undefined", "Boolean", "isFinite", "PrepareFunctionForOptimization", "Map", "this", "OptimizeFunctionOnNextCall", "BigInt", "DeoptimizeNow", "WeakMap", "Reflect", "WeakSet", "print", "Float32Array", "RangeError", "Uint8ClampedArray", "RegExp", "Symbol", "Proxy", "Float64Array", "parseInt", "ArrayBuffer", "isNaN", "Uint16Array", "Object", "Infinity", "DataView", "Date", "placeholder", "Function", "eval", "DeoptimizeFunction", "OptimizeOsr", "Uint32Array", "Array", "gc", "Uint8Array", "NeverOptimizeFunction", "Math", "NaN", "AggregateError", "arguments", "Number"]

[JavaScriptEnvironment] Have 211 available method names: ["setUTCSeconds", "trimStart", "random", "pow", "abs", "setUTCMinutes", "has", "test", "getUTCDate", "exec", "trimRight", "filter", "for", "trunc", "reverse", "every", "seal", "toJSON", "isSafeInteger", "setInt32", "getUint32", "setUTCFullYear", "lastIndexOf", "toUTCString", "sign", "defineProperty", "findIndex", "getInt8", "toGMTString", "localeCompare", "getUTCFullYear", "setMinutes", "getFullYear", "fromCodePoint", "values", "toTimeString", "setUTCHours", "setUTCDate", "set", "getUTCMonth", "getFloat64", "asUintN", "includes", "sinh", "cbrt", "forEach", "isArray", "call", "create", "isSealed", "allSettled", "matchAll", "race", "min", "of", "setUint8", "delete", "push", "fround", "indexOf", "toISOString", "floor", "getMonth", "charCodeAt", "getInt32", "isInteger", "getMilliseconds", "sqrt", "match", "imul", "log1p", "setMilliseconds", "p", "setTime", "isView", "ownKeys", "deleteProperty", "entries", "now", "from", "get", "toString", "repeat", "max", "toUpperCase", "finally", "shift", "getPrototypeOf", "search", "bind", "setDate", "all", "trimLeft", "getOwnPropertySymbols", "getUTCMilliseconds", "asin", "apply", "raw", "setPrototypeOf", "getTimezoneOffset", "getUint8", "getUTCSeconds", "splice", "getTime", "reject", "toDateString", "compile", "add", "expm1", "pop", "cosh", "log10", "getMinutes", "toLowerCase", "isNaN", "preventExtensions", "trim", "log", "getUTCHours", "stringify", "setFloat64", "getOwnPropertyDescriptor", "asIntN", "setHours", "getUTCDay", "fromCharCode", "resolve", "toLocaleString", "substring", "getHours", "log2", "endsWith", "freeze", "codePointAt", "o", "cos", "getInt16", "getOwnPropertyNames", "setFloat32", "slice", "trimEnd", "getUint16", "then", "hypot", "acosh", "UTC", "construct", "m", "tan", "setUint16", "catch", "getUTCMinutes", "parse", "flatMap", "isExtensible", "getOwnPropertyDescriptors", "fromEntries", "sin", "atan2", "setUint32", "setInt8", "keyFor", "charAt", "some", "exp", "round", "setUTCMilliseconds", "setMonth", "copyWithin", "acos", "atan", "getDay", "split", "setSeconds", "getFloat32", "padStart", "concat", "tanh", "setYear", "startsWith", "map", "is", "unshift", "clz32", "setInt16", "asinh", "atanh", "getYear", "setFullYear", "flat", "fill", "replaceAll", "n", "assign", "clear", "setUTCMonth", "keys", "reduce", "getDate", "join", "padEnd", "sort", "ceil", "isFinite", "find", "defineProperties", "reduceRight", "getSeconds", "subarray", "replace", "isFrozen"]

[JavaScriptEnvironment] Have 51 property names that are available for read access: ["__proto__", "global", "match", "isConcatSpreadable", "message", "PI", "NEGATIVE_INFINITY", "length", "NaN", "name", "MIN_SAFE_INTEGER", "c", "dotAll", "source", "a", "sticky", "search", "E", "iterator", "e", "byteLength", "byteOffset", "d", "description", "MAX_SAFE_INTEGER", "matchAll", "unicode", "EPSILON", "asyncIterator", "split", "multiline", "unscopable", "constructor", "b", "prototype", "POSITIVE_INFINITY", "replace", "caller", "hasInstance", "MIN_VALUE", "species", "arguments", "MAX_VALUE", "size", "toPrimitive", "toString", "toStringTag", "buffer", "flags", "ignoreCase", "valueOf"]

[JavaScriptEnvironment] Have 10 property names that are available for write access: ["constructor", "b", "length", "c", "e", "valueOf", "a", "__proto__", "d", "toString"]

[JavaScriptEnvironment] Have 5 custom property names: ["d", "e", "a", "c", "b"]

[JavaScriptEnvironment] Have 4 custom method names: ["n", "p", "o", "m"]

[Fuzzer] Initialized

[Fuzzer] Recommended timeout: at least 300ms. Current timeout: 250ms

[Fuzzer] Startup tests finished successfully

[Fuzzer] Let's go!

Fuzzer Statistics --v8

Total Samples: 475

Interesting Samples Found: 193

Valid Samples Found: 338

Corpus Size: 194

Correctness Rate: 71.16%

Timeout Rate: 2.74%

Crashes Found: 0

Timeouts Hit: 13

Coverage: 5.10%

Avg. program size: 144.53

Connected workers: 0

Execs / Second: 27.64

Fuzzer Overhead: 4.74%

Total Execs: 20649

[Fuzzer] Mutator correctness rates: CodeGenMutator: 69.66%, InputMutator: 52.50%, OperationMutator: 65.67%, CombineMutator: 100.00%, JITStressMutator: 89.55%

[Corpus] Corpus cleanup finished: 395 -> 395

[Fuzzer] Mutator correctness rates: CodeGenMutator: 67.45%, InputMutator: 56.71%, OperationMutator: 65.45%, CombineMutator: 98.24%, JITStressMutator: 86.91%

Fuzzer Statistics --v8

Total Samples: 1430

Interesting Samples Found: 394

Valid Samples Found: 1015

Corpus Size: 395

Correctness Rate: 70.98%

Timeout Rate: 2.52%

Crashes Found: 0

Timeouts Hit: 36

Coverage: 7.34%

Avg. program size: 141.78

Connected workers: 0

Execs / Second: 21.84

Fuzzer Overhead: 3.51%

Total Execs: 43374

Corpus seed

https://source.chromium.org/chromium/chromium/src/+/master:v8/test/mjsunit/

https://github.com/chakra-core/ChakraCore/tree/master/test

https://github.com/v8/v8/tree/master/test

https://github.com/WebKit/webkit/tree/main/JSTests

https://github.com/mozilla/spidernode/tree/master/test

https://github.com/tunz/js-vuln-db

04

- ❏ Automate program generator
- ❏ Web assembly
- ❏ Environment setup

JS random code generator

Differential testing need to be used for randomly generated code .

Some projects have tried to generate random test case to fuzz C and C++ compilers.

But seems that there is few random code generator for JavaScript

Random program generation

Mutation-based fuzzing : pros and cons

The primary advantage of mutation is that since it builds on existing test cases, the expressiveness of the generated code is limited only by the expressiveness of the test cases that are used as the basis for mutation.

The main disadvantage is that the generator is not stand-alone and cannot easily guarantee the absence of undefined behavior in test cases.

mutation-based and generation-based approaches to fuzzing are sufficiently different that they cannot be compared directly. Moreover, neither approach is likely to subsume the other: in practice, we need to employ both kinds of fuzzing.

webassembly

WebAssembly needs to communicate with JavaScript. And that is the task of JS engine.

WebAssembly can only communicate in number type. JavaScript supports other types. Even the number types are different.

The difference between these types is not only the name, their values are also stored in memory in different ways.

Jsc -- fuzzilli

[Corpus] Corpus cleanup finished: 242 -> 242

[Fuzzer] Mutator correctness rates: CodeGenMutator: 71.60%, InputMutator: 69.23%, OperationMutator: 68.52%, CombineMutator: 96.77%, JITStressMutator: 91.38%

Fuzzer Statistics

Total Samples: 460

Interesting Samples Found: 241

Valid Samples Found: 352

Corpus Size: 242

Correctness Rate: 76.52%

Timeout Rate: 0.87%

Crashes Found: 0

Timeouts Hit: 4

Coverage: 7.17%

Avg. program size: 397.23

Connected workers: 0

Execs / Second: 28.71

Fuzzer Overhead: 5.08%

Total Execs: 55508

Jsc -- fuzzilli

[Corpus] Corpus cleanup finished: 1337 -> 1337

[Fuzzer] Mutator correctness rates: CodeGenMutator: 17.45%, InputMutator: 18.51%, OperationMutator: 27.73%, CombineMutator: 25.37%, JITStressMutator: 21.62%

Fuzzer Statistics

Total Samples: 24235

Interesting Samples Found: 1336

Valid Samples Found: 4969

Corpus Size: 1337

Correctness Rate: 20.50%

Timeout Rate: 0.44%

Crashes Found: 0

Timeouts Hit: 106

Coverage: 15.57%

Avg. program size: 302.87

Connected workers: 0

Execs / Second: 21.19

Fuzzer Overhead: 7.10%

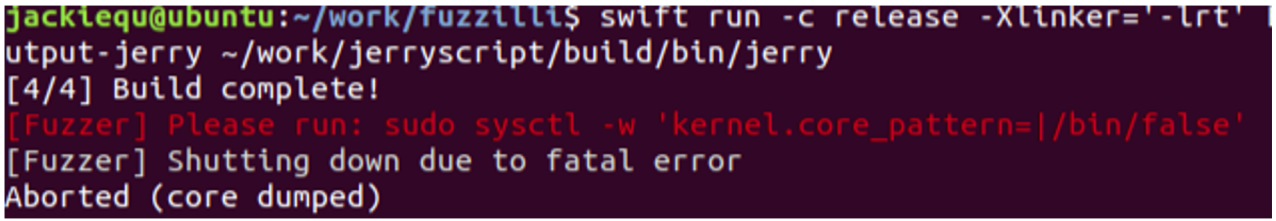
Total Execs: 323253

Jerryscript

swift run -c release -Xlinker='-lrt' FuzzilliCli --profile=jerryscript --storagePath=./output-jerry ~/work/jerryscript/build/bin/jerry

```
Fuzzer Statistics
-----------------
Total Samples:              46500
Interesting Samples Found:    958
Valid Samples Found:        32956
Corpus Size:                  959
Correctness Rate:           70.87%
Timeout Rate:                1.56%
Crashes Found:                  0
Timeouts Hit:                 725
Coverage:                   42.87%
Avg. program size:          49.28
Connected workers:              0
Execs / Second:            191.60
Fuzzer Overhead:            58.10%
Total Execs:                90758

########## Unique Crash Found ##########
function placeholder(){}
function main() {
var v2 = Array(1337);
var v4 = Promise.race(v2);
}
main();
// STDERR:
```



Fuzzer Statistics

Total Samples: 119450
Interesting Samples Found: 1244
Valid Samples Found: 83729
Corpus Size: 1245
Correctness Rate: 70.10%
Timeout Rate: 1.36%
Crashes Found: 1
Timeouts Hit: 1625
Coverage: 47.37%
Avg. program size: 51.45
Connected workers: 0
Execs / Second: 242.54
Fuzzer Overhead: 58.78%
Total Execs: 176587

########## Unique Crash Found ##########
```
function placeholder(){}
function main() {
function v0(v1,v2,v3,v4) {
    var v5 = new v0();
}
var v6 = v0();
}
main();
// STDERR:
```

[Fuzzer] Mutator correctness rates: CodeGenMutator: 55.09%, InputMutator: 73.03%, OperationMutator: 70.79%, CombineMutator: 99.88%, JITStressMutator: 80.48%

Fuzzer Statistics

Total Samples: 154985
Interesting Samples Found: 1334
Valid Samples Found: 109220
Corpus Size: 1335
Correctness Rate: 70.47%
Timeout Rate: 1.32%
Crashes Found: 2
Timeouts Hit: 2042
Coverage: 48.58%
Avg. program size: 54.17
Connected workers: 0
Execs / Second: 213.86
Fuzzer Overhead: 56.00%
Total Execs: 216542

########## Unique Crash Found ##########
```
function placeholder(){}
function main() {
var v2 = new Promise(RegExp);
do {
    var v5 = v2.then();
} while (-3662418045 < 8);
}
main();
// STDERR:
```

```
[REPRL] Script execution failed: Child unexpectedly terminated with signal 11 between executions. Retrying in 1 second...
########## Unique Crash Found ##########
function placeholder(){}
function main() {
var v1 = "function";
var v4 = [13.37,13.37];
var v6 = [1337,1337];
var v7 = [1337,-1024,13.37];
var v8 = {__proto__:v7,a:-1024,c:AggregateError,e:1337,valueOf:-1024};
var v9 = {__proto__:1337,constructor:v7,length:1337,valueOf:-1024};
var v10 = 3217819641;
var v11 = v10++;
var v13 = 44994;
var v14 = v13++;
var v21 = [13.37,-290921985,13.37];
var v22 = [];
var v29 = [13.37,13.37,13.37,Uint8ClampedArray,13.37];
var v31 = [13.37];
async function v32(v33,v34,v35,v36) {
    var v37 = v32(v22,v36,v36,v36);
    var v40 = new Uint8ClampedArray(39449);
    var v41 = v40.copyWithin(2231875224,v40);
}
var v42 = [13.37,"-4294967297",v29,"-4294967297",2231875224,v31,"QvTV1zxcVp","QvTV1zxcVp",RangeError];
var v43 = {a:v21,constructor:v32};
var v44 = v32(1337,v43,RegExp,v42);
var v46 = [13.37,v11,13.37];
var v47 = [];
var v52 = [13.37,13.37,13.37,Uint8ClampedArray,13.37];
var v54 = [13.37];
async function v55(v56,v57,v58,v59) {
    var v60 = v55(v47,v59,v59,v59);
    var v63 = new Uint8ClampedArray(39449);
    var v64 = v58.slice(2231875224,v63);
}
var v65 = [13.37,"-4294967297",v52,"-4294967297",v14,v54,"QvTV1zxcVp","QvTV1zxcVp",RangeError];
var v66 = {a:v46,constructor:v55};
var v67 = v55(1337,v66,RegExp,v65);
}
main();
// STDERR:
```

Fuzzer Statistics

Total Samples: 4027850

Interesting Samples Found: 2507

Valid Samples Found: 1519197

Corpus Size: 1029

Correctness Rate: 37.72%

Timeout Rate: 0.39%

Crashes Found: 2082

Timeouts Hit: 15571

Coverage: 61.67%

Avg. program size: 62.81

Connected workers: 0

Execs / Second: 13.86

Fuzzer Overhead: 28.47%

Total Execs: 4873900

05

- Set up
  - Wasm runtime
    - https://github.com/appcypher/awesome-wasm-runtimes
    - emscripten
  - Random program generator
    - Yarpgen
    - Csmith

wasm

Using Emscripten to compile c/c++ source code to .wasm code for wasm runtime to execute.

Randomly generating programs

Design Goals

Csmith:

- be well formed and have a single meaning according to the C standard.
- maximize expressiveness.

  (support many language features and combinations of features)

YarpGen:

- to avoid, or at least delay, saturation.

  (diversity, expressiveness, target specific optimization pass)

Randomly generating programs

Advantage

Csmith:

• cover a large subset of C, avoiding the undefined and unspecified behaviors. (to ensuring single interpretation.)

• C features supporting: complex control flow, data structures, pointers, arrays, structs.

(cost: analysis and dynamic checks increase Csmith code size. 40k lines)

YarpGen:

• generating programs without using dynamic checks.

• Using generation policies to target different parts of an optimizer.

• Automated tools for compiler fuzzing. (Harness)

• Do not support function call.

Randomly generating programs

Csmith:

- governed by a grammar for a subset of C.
- maintains: a global environment, a local environment (for safety check)
- top-down recursive generation with 6 steps.

YarpGen:

- Top-down recursive generation.
- No function call. Top level is main block.
- Lowering IR to the target language.

How to avoid Undefined and Unspecified Behaviors (UBs)

Csmith:

• Easy UBs: can be avoided structurally by generating programs in such a way that problems never arise.

• Hard UBs: using static analysis and adding run-time checks to the generated code.

    • Static: to avoid use variables without initialization, initialize variables close to where they are declared.

    • Run-time check: null pointer checks.

YarpGen:

• easy UBs: same with Csmith

• Hard UBs: It interleaves analysis and code generation. While generating, convert operations that trigger undefined behavior into similar operations that are safe.

06

- Setup
  - Using csmith to generate random C program
  - Using Emscripten to compile C source code to .wasm file
  - Differential testing on wasmer, wasmtime, wasm3, wavm (most popular runtimes)
- Some issues
  - Csmith may generate programs with infinite loops -- using timeout to limit execution time.
  - Emscripten may fail to compile codes generated by csmith -- try to use yarpgen.
- Todo
  - add other compiler than Emscripten
  - use optimize options of the wasm runtimes
  - add other random code generators
  - add other language other than c/c++

- ❏ Wasm Runtime coverage
- ❏ Compile error handle

coverage

Libfuzzer for wasmer

Kcov -- for rust: wasmer & wasmtime

Gcov -- for c: wasm3

Honggfuzz-rs, afl-rs, cargo-fuzz

wasm-runtimes-fuzzing

收集动态执行trace, 从trace获取basic block执行信息 -- coverage info

Binary tool -- pin, qemu, valgrind

Script-- capture error info,

- Coverage info
  - Using Gcov to collect coverage info of wasm3
  - Using Kcov to collect coverage info of wasmer, wasmtime
  - Using pin to collect instruction execution times -- slow

- todo
  - Clean up test case files regularly -- the number of files will significant influence execution efficiency.
  - Integrate Kcov and Gcov to collect coverage info in real-time
  - Further test the timeout test case

07

Pintool :

collect number of total executed instructions, basic blocks, and modules, with addresses. Output the detail cov-info of every test, and a brief summary.

Issue

- the numbers of instructions and basic blocks running at different time with the same input are always different. The difference is not huge, but it may influence the analysis.
- If we want to compare basic block addresses executed at two different times, we need to load binary at the same base address.
- using binary BI could simply get the basic block info, but we do not know the coverage ratio.
- since Csmith is a random program generator, the test cases generated sequentially may have no relationship. Therefore, it is hard to do mutation like other coverage-guide testing tools.

➢ Seems that wasmtime runs mush slower with pintool than other WebAssembly runtimes. And sometimes will thow “compiling function run over the stack limit” error

TODO

- a way to analyze the coverage info. -- json may be suitable.
- need to feed the coverage info back to the baseline.
    - feed back what?
    - feedback , then what? -- mutate like AFL?

08

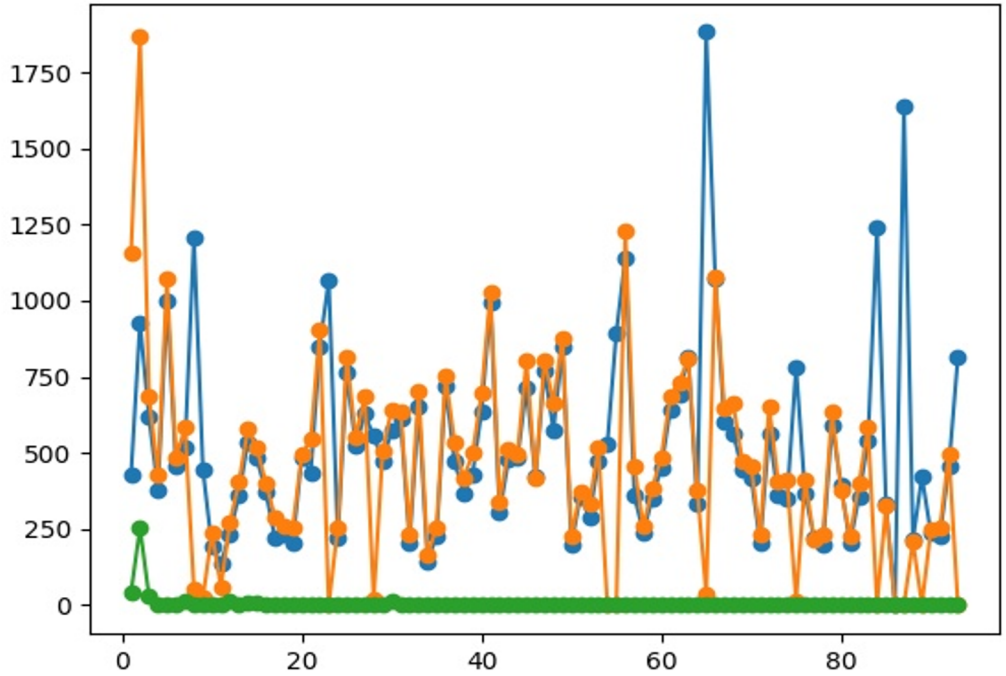
Use RTN to ban dynamic BBL

Disable ASLR to load the test at the same address

Implement a statistic script to show the new coverage of each test

Part of the new coverage of each test, -1 means there is no data in the cov-*-*.log(wasm running error)

Blue: wasmer, orange: wasmtime, green: wasm3



todo

Does RTN api really works?

Feedback to generator

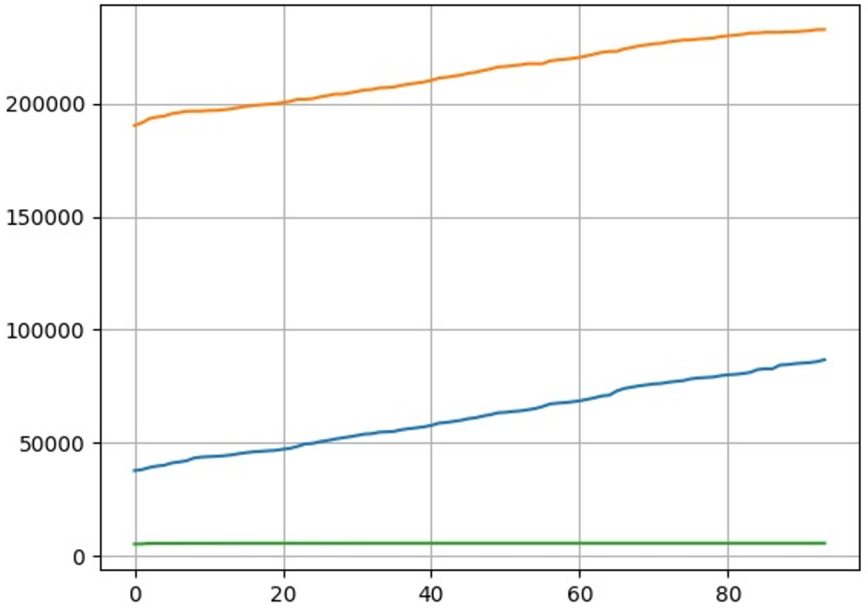
09

Update the .sh script

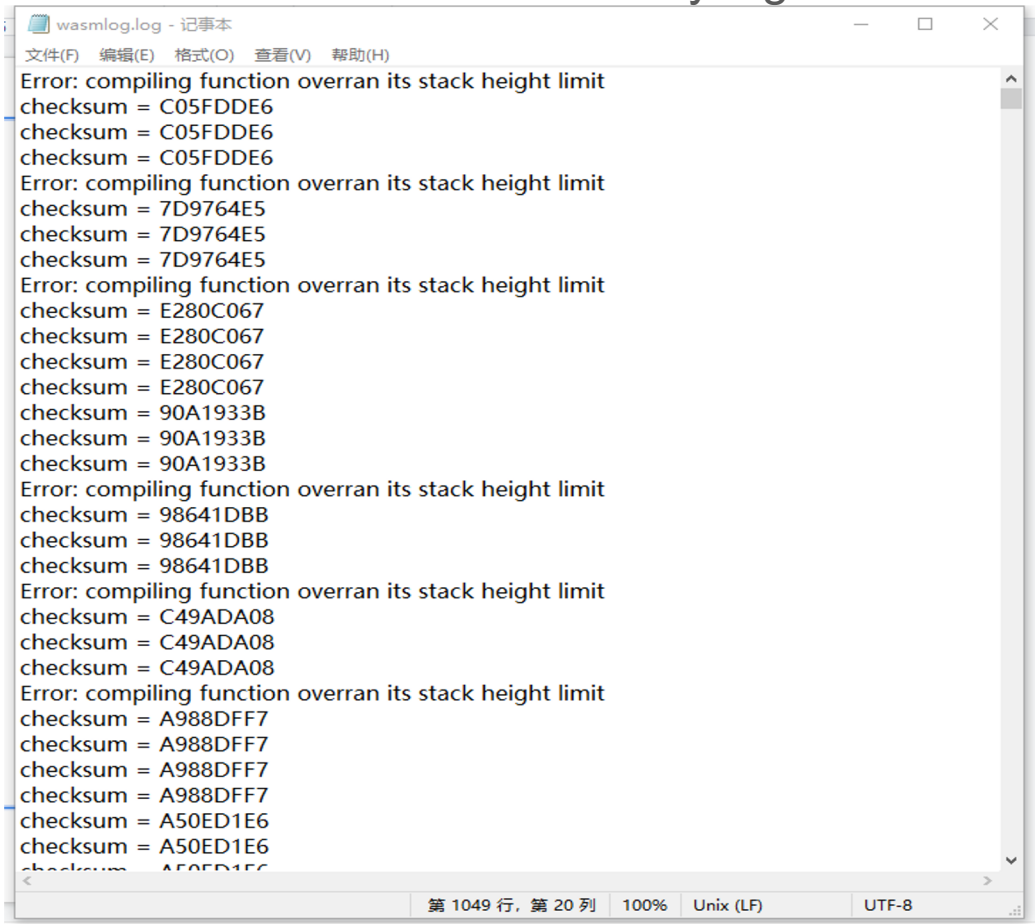
Show result in cumulative graph

Use no-pie to find the address range:

Seems that the addresses filtered by pintool are all in the range



Record all the info in a summary.log:



Todo:

Connect BBL with source code

Update pintool

10

wasmer, wasmtime are multithreaded programs

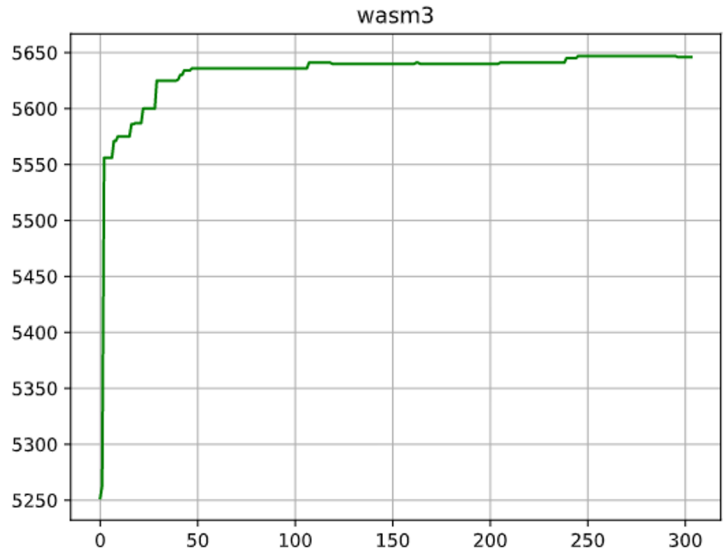
```
dorafan    84948  0.1  0.0  16920   644 pts/0    S    11:51   0:00 timeout 60s ../../../pin -t obj-intel64/coverageTest_filter.so -support_jit_api
dorafan    84951 30.4  3.4 640068 136344 pts/0   Rl   11:51   0:08 wasmer /home/dorafan/work/wasmtest/csmith/generated/random-306.wasm
dorafan    84951 57.3  3.4 640092 136344 pts/0   Sl   11:52   0:05 wasmer /home/dorafan/work/wasmtest/csmith/generated/random-306.wasm
dorafan    84951 56.9  3.4 640068 136344 pts/0   Sl   11:52   0:05 wasmer /home/dorafan/work/wasmtest/csmith/generated/random-306.wasm
```

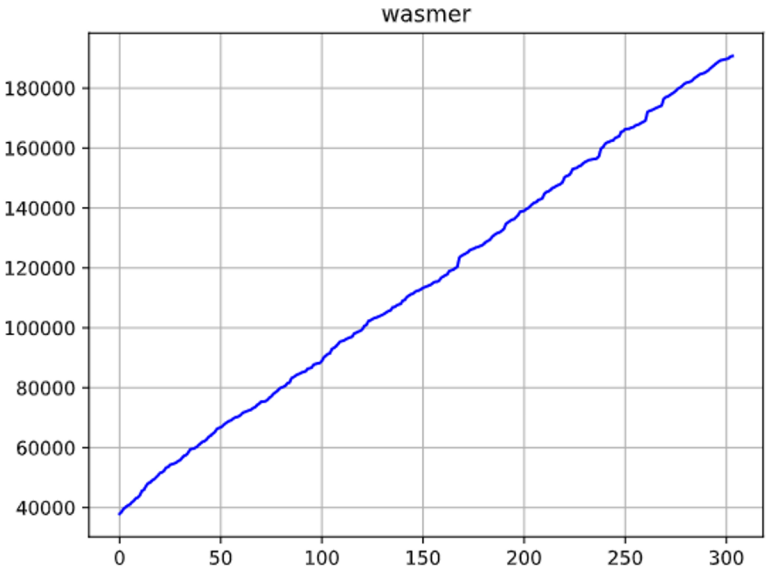
```
dorafan    84772  0.0  0.0  16920   656 pts/0    S    11:19   0:00 timeout 120s ../../../pin -t obj-intel64/coverageTest_filter.so -support_jit_api
dorafan    84774  3.7 12.6 1475240 504988 pts/0  tl   11:19   0:10 wasmtime /home/dorafan/work/wasmtest/csmith/generated/random-305.wasm
dorafan    84774  0.0 12.6 1475240 504988 pts/0  SNl  11:19   0:00 wasmtime /home/dorafan/work/wasmtest/csmith/generated/random-305.wasm
dorafan    84774 31.3 12.6 1475240 504988 pts/0  Sl   11:19   1:28 wasmtime /home/dorafan/work/wasmtest/csmith/generated/random-305.wasm
dorafan    84774 26.5 12.6 1475240 504988 pts/0  Sl   11:19   1:14 wasmtime /home/dorafan/work/wasmtest/csmith/generated/random-305.wasm
```

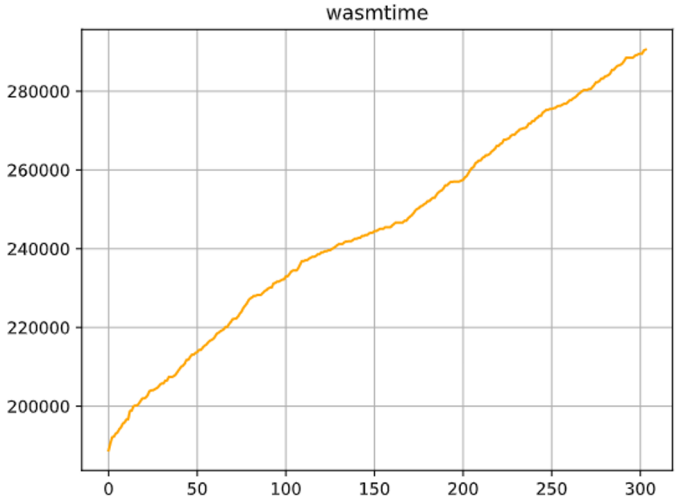
Some errors:

```
checksum = 3D2EC3C9
C: [tid:62062] Tool (or Pin) caused signal 11 at PC 0x7ffff5d06460
timeout: the monitored command dumped core
checksum = 3D2EC3C9
```

Wasmtime once output a 40GB coverage log


wasm3
5650
5600
5550
5500
5450
5400
5350
5300
5250
0
50
100
150
200
250
300


wasmer
180000
160000
140000
120000
100000
80000
60000
40000
0
50
100
150
200
250
300


wasmtime
280000
260000
240000
220000
200000
0
50
100
150
200
250
300

11

Update coverage script : add thread-safe structure

Add new coverage script : the old one could not identify dynamic created code?

Need more tests

Update baseline script

todo

1. Find interesting test case:

   If a test case could cause more new basic blocks in one runtime than other test cases, or it could cause more new basic blocks in one runtime than in other runtimes, we think it is an “INTERESTING” test case. (just like what coverage-guide fuzzing does)

1. Need to update script:

   The old pintool couldn’t collect correct basic blocks -- it couldn’t ignore dynamic created code. And that could explain why the former results are positively related (runtime created new JIT basic blocks for every test case)

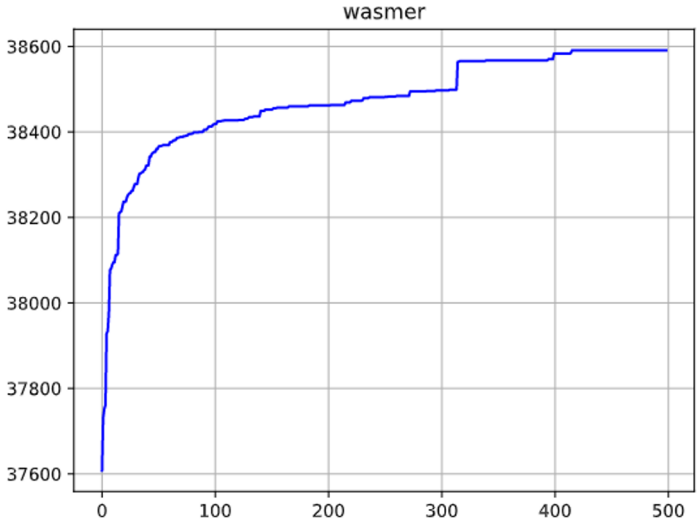
12

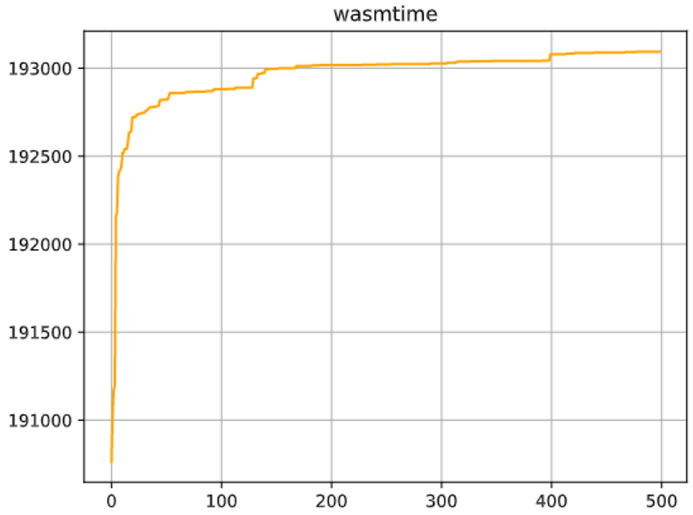
Update pintool -- use new api other than RTN_isDynamic (watch the same question on StackOverflow)

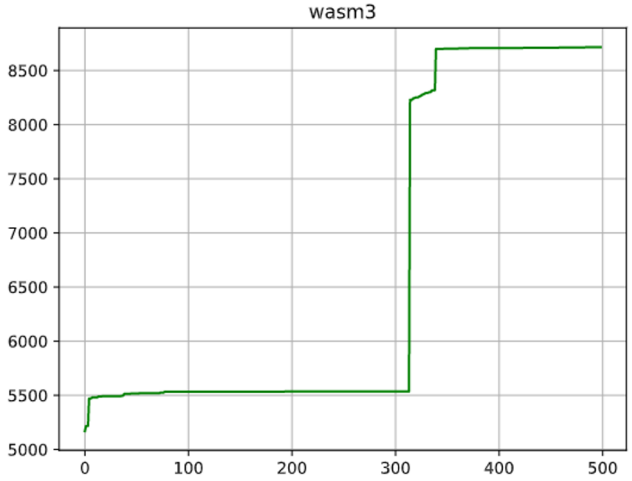
Update statistic script, record the test case to find the one that create more bbls

Upload the baseline script on the server

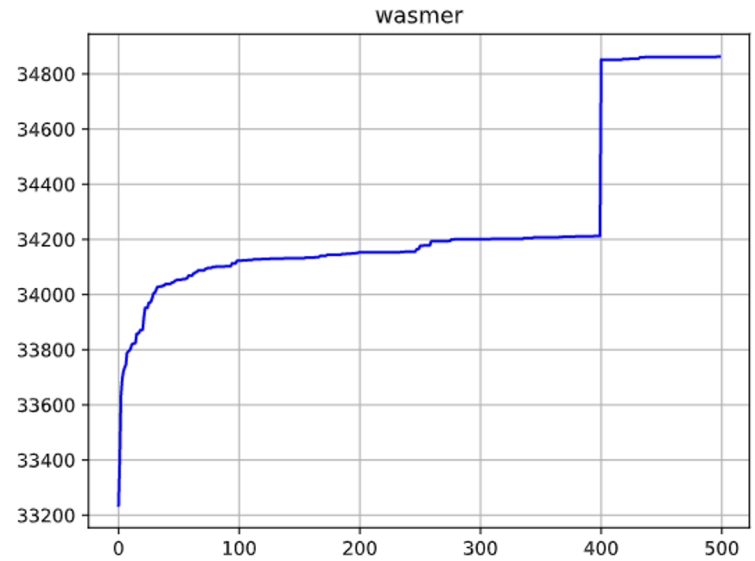
Using isValid():

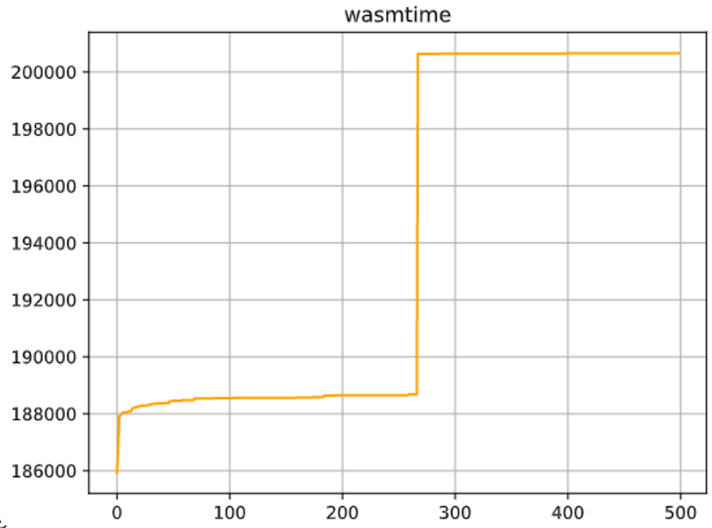


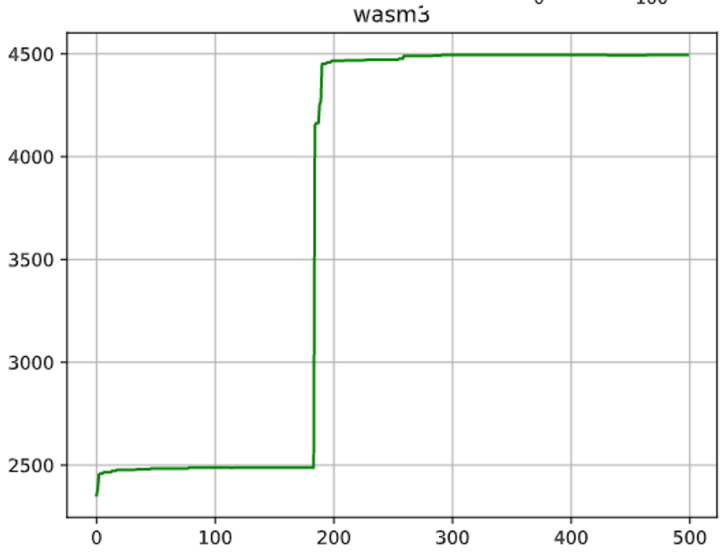




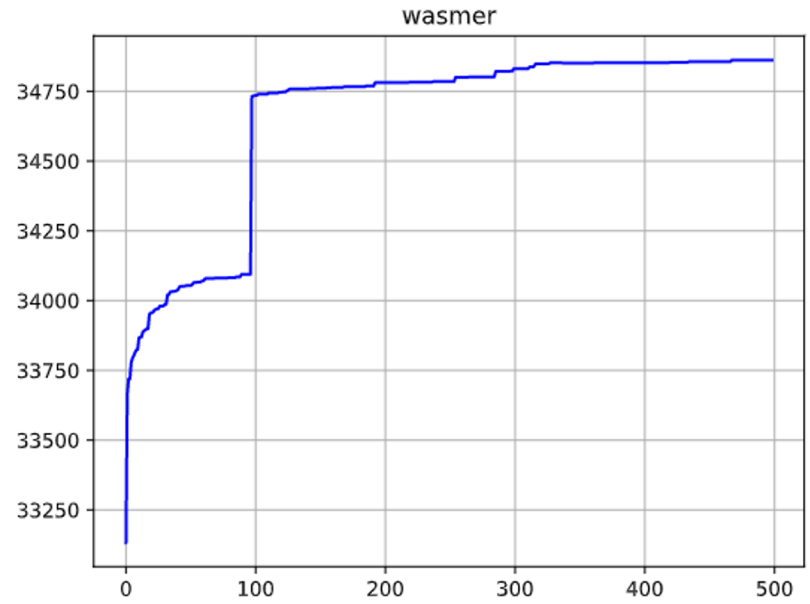
Using address boundary :

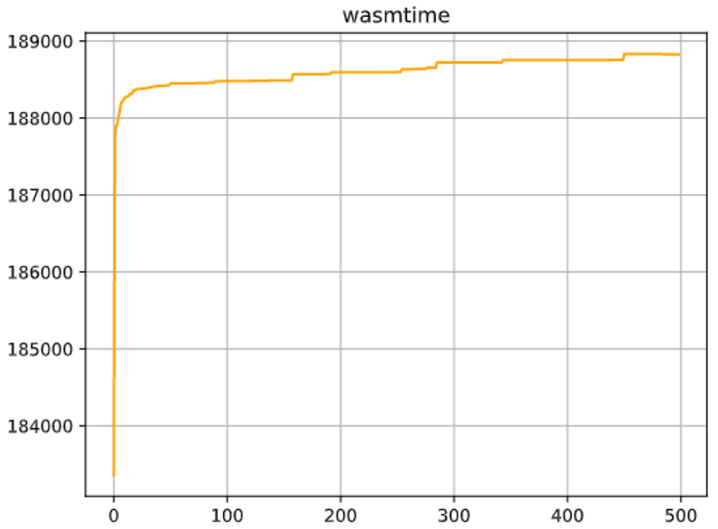


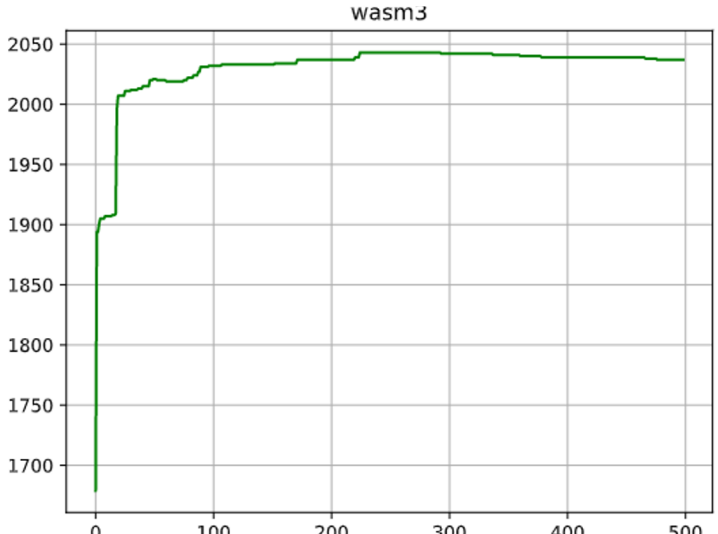




Using address boundary 2:







issues:

Why in address boundary 2, results in wasm3 decreases?

-- if there is no coverage data, the statistic script (calculate new bbls) will return -1, and wasm3 always has some issues like “overrun its stack height limit”

Run pintool on Wasmtime always need lots of time, and sometimes will output a very large record file, or just get stuck.

-- maybe pintool cause deadlock or endless loop when running on wasmtime.

-- why is timeout command useful?

todo:

Double check the “interesting” test cases.

Update script -- build a integrated tool to connect statistic data, bbls, results and test case source code.

The baseline running on the server is slow.

13

Compare source code:

Haven't find obvious differences

Csmith statistic info

1. Structure info: structure depth, number of union variables
2. Bitfield info: number of bit-field defined in structure(zero, non-zero, const, volatile), structures with bit-field
3. Address info: number of pointers, times of reference and dereference, depth of pointers, address compare
4. Volatile info: volatile & non-volatile operation times
5. Expression and block info: depth of expression and block, bit-field & reference on LHS/RHS

Source code generated by csmith

1. Define structures (with or without bit-field), or not
2. Define variables
3. Define functions: func_1 is the test function, others are subfunctions
    a. Set values to variables and structures
    b. Do reference and dereference operation on the variables and structures
    c. Call functions nested
4. Calculate checksums of all values in all variables

Wasm runtime architecture

Wasmtime:

1. Compile a module:
    a. First compilation walks over the WebAssembly module validating everything except function bodies.
    b. all functions within a module are validated and compiled
    c. Put the compilation information into a structure (module info, compiled jit code, and other info)
    d. place all code into a form that's ready to get executed.(new memory mapping is allocated and the JIT code is copied into this memory mapping.)(jit code is executed at this point)

Wasm3:

Using a self-made interpreter, called M3.

Like "threaded code" http://www.complang.tuwien.ac.at/forth/threaded-code.html

baseline

```
jackie@lever:~/work/wasmtest/csmith/diffs$ cat result-14815.txt
error: failed to run `/home/jackie/work/wasmtest/csmith/generated/random-14815.wasm`
|   1: module instantiation failed (engine: universal, compiler: cranelift)
└─> 2: Validation error: locals exceed maximum (at offset 716)
Error: failed to run main module `/home/jackie/work/wasmtest/csmith/generated/random-14815.wasm`

Caused by:
    0: WebAssembly failed to compile
    1: WebAssembly translation error
    2: Invalid input WebAssembly code at offset 716: locals exceed maximum
checksum = 64486176
Error: compiling function overran its stack height limit
```

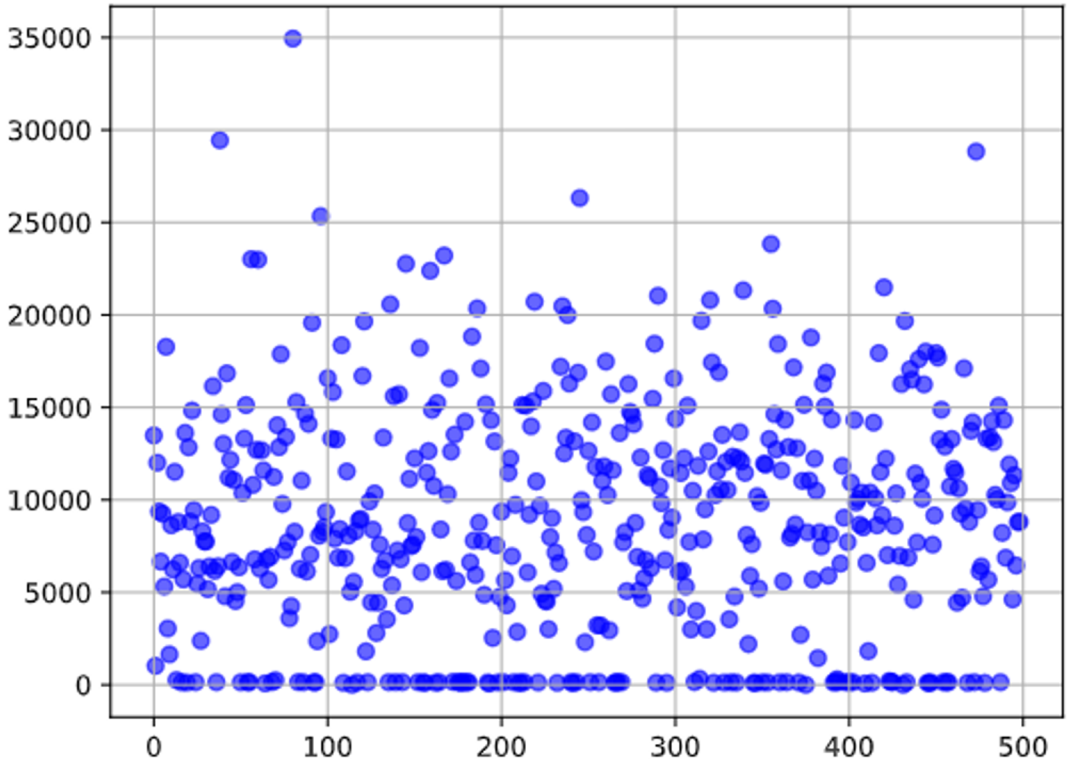
```
jackie@lever:~/work/wasmtest/csmith/diffs$ cat result-14518.txt
error: failed to run `/home/jackie/work/wasmtest/csmith/generated/random-14518.wasm`
|   1: module instantiation failed (engine: universal, compiler: cranelift)
└─> 2: Validation error: locals exceed maximum (at offset 687)
Error: failed to run main module `/home/jackie/work/wasmtest/csmith/generated/random-14518.wasm`

Caused by:
    0: WebAssembly failed to compile
    1: WebAssembly translation error
    2: Invalid input WebAssembly code at offset 687: locals exceed maximum
Error: compiling function overran its stack height limit
```

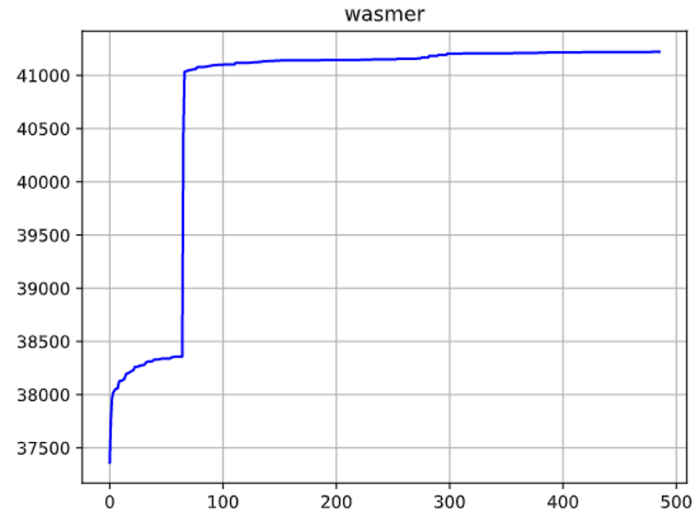
2021.08.12

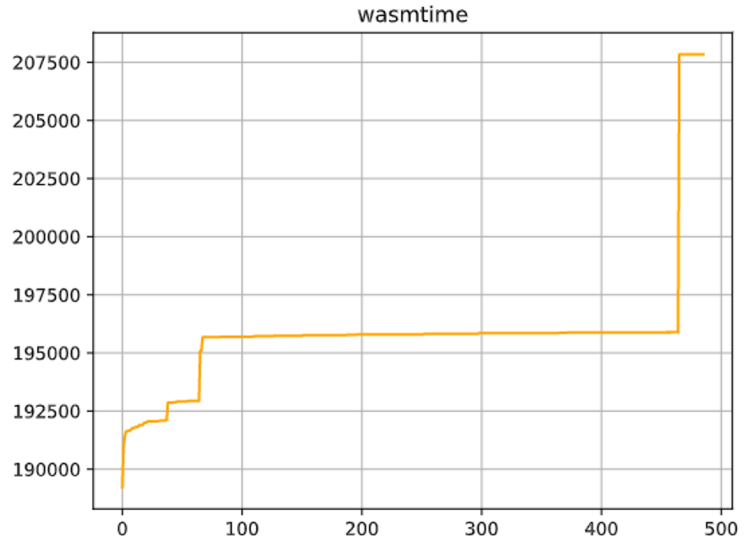
Calculate the distance of csmith statistic info:

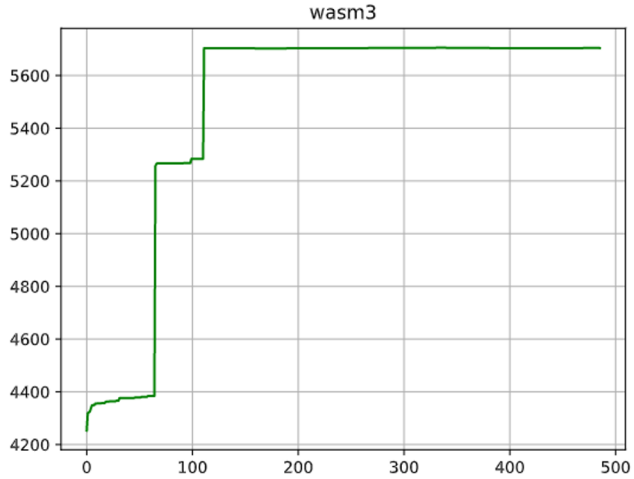
Didn’t find any relevant between source code info and interesting test case. (400, 268, 185)

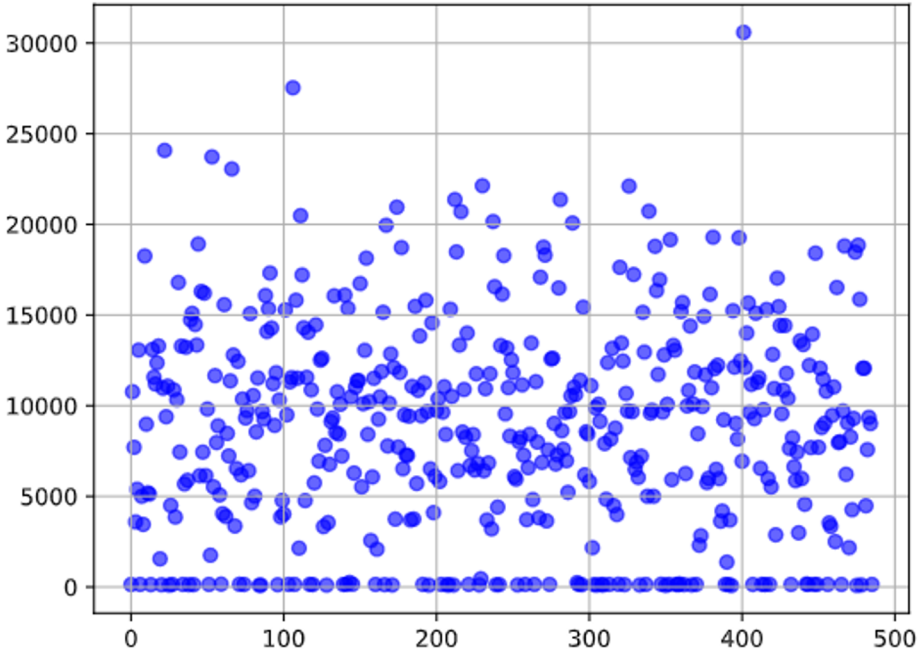


isValid()


wasmer
41000
40500
40000
39500
39000
38500
38000
37500
0
100
200
300
400
500


wasmtime
207500
205000
202500
200000
197500
195000
192500
190000
0
100
200
300
400
500


wasm3
5600
5400
5200
5000
4800
4600
4400
4200
0
100
200
300
400
500


30000
25000
20000
15000
10000
5000
0
0
100
200
300
400
500

Interesting cases

random-14815.c

error: failed to run `/home/jackie/work/wasmtest/csmith/generated/random-14815.wasm`

│ 1: module instantiation failed (engine: universal, compiler: cranelift)

└─> 2: Validation error: locals exceed maximum (at offset 716)

Error: failed to run main module `/home/jackie/work/wasmtest/csmith/generated/random-14815.wasm`

wasmer

Caused by:wasmtime

0: WebAssembly failed to compile

1: WebAssembly translation error

2: Invalid input WebAssembly code at offset 716: locals exceed maximum

checksum = 64486176 wavm

Error: compiling function overran its stack height limit wasm3

Gcc could execute successfully.

No runtime could execute in double-check.

But there is a result in the log -- wavm

About random-14518.c:
Seems that the source code is incorrect -- a.out generated by gcc cannot execute successfully.

Verify coverage data -- shared lib & multi-thread

Re-calculate distance -- random-14815

Debug information -- find related source code from coverage address

Find bug in 14815

Strength of differential testing (rather than normal fuzzing) -- find semantic error

14

Segment fault in wasm3: stack height limit -- should set stack height limit to a appropriate value, can’ t be too large

If you're generating artificial/pathological code, then it's probable you'll encounter a "function stack overflow" condition. But, there should be a constant large enough for d_m3MaxFunctionStackHeight that satisfies the Wasm code you're interpreting. Just keep doubling it. If there really isn't a solution, then this is probably a bug. Post the .wasm file for us.

Seems that the max value of stack height is 8000 -- a related value (stackheight * 2) is a unsigned short int type. Once the unsigned conversion from ‘int’ to ‘short unsigned int’ happens, there will be a segment fault.

objdump

Collect the number of times the unique basic block occurs in 500 test cases with its address. Compared with objdump result.

```
('555555574951', 4)
('555555574a04', 4)
('555555574a54', 4)
('555555574a57', 4)
('555555574a60', 4)
('555555574a74', 4)
('55555556643b', 6)
('555555574627', 6)
('5555555746da', 6)
('555555574728', 6)
('55555557472b', 6)
('555555574734', 6)
('555555574748', 6)
('555555574d7a', 6)
('555555574d83', 6)
('555555574d97', 6)
('55555556f64b', 7)
('555555574c77', 7)
('555555574d2a', 7)
('555555574d77', 7)
('555555566d6e', 11)
('55555556f4f1', 13)
('555555575dc2', 13)
('555555576056', 13)
('55555557a657', 13)
```

```
('555555579719', 476)
('55555557971f', 476)
('55555557976c', 476)
('55555557977f', 476)
('555555579788', 476)
('55555557979a', 476)
('55555557979e', 476)
('5555555797b7', 476)
('5555555797bb', 476)
('5555555797c1', 476)
('5555555797d3', 476)
('5555555797eb', 476)
('555555579803', 476)
('55555557ada5', 476)
('55555557adbe', 476)
('55555557adca', 476)
('55555555fd8f', 480)
('55555555fbd2', 482)
('555555571f8c', 484)
('555555571f95', 484)
('555555571fa9', 484)
('555555571eaa', 486)
('555555571f37', 486)
('55555556fbd5', 488)
('5555555712e1', 488)
```

```
Disassembly of section .text:

0000000000007860 <_start>:
    7860:   f3 0f 1e fa             endbr64
    7864:   31 ed                   xor    %ebp,%ebp
    7866:   49 89 d1                mov    %rdx,%r9
    7869:   5e                      pop    %rsi
    786a:   48 89 e2                mov    %rsp,%rdx
    786d:   48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
    7871:   50                      push   %rax
    7872:   54                      push   %rsp
    7873:   4c 8d 05 36 71 02 00            lea    0x27136(%rip),%r8        # 2e9b0 <__libc_csu_fini>
    787a:   48 8d 0d bf 70 02 00            lea    0x270bf(%rip),%rcx       # 2e940 <__libc_csu_init>
    7881:   48 8d 3d c4 65 02 00            lea    0x265c4(%rip),%rdi       # 2de4c <main>
    7888:   ff 15 4a 67 03 00       callq  *0x3674a(%rip)        # 3dfd8 <__libc_start_main@GLIBC_2.2.5>
    788e:   f4                      hlt
    788f:   90                      nop

0000000000007890 <deregister_tm_clones>:
    7890:   48 8d 3d 21 6a 03 00            lea    0x36a21(%rip),%rdi       # 3e2b8 <__TMC_END__>
    7897:   48 8d 05 1a 6a 03 00            lea    0x36a1a(%rip),%rax       # 3e2b8 <__TMC_END__>
    789e:   48 39 f8                cmp    %rdi,%rax
    78a1:   74 15                   je     78b8 <deregister_tm_clones+0x28>
    78a3:   48 8b 05 26 67 03 00            mov    0x36726(%rip),%rax       # 3dfd0 <_ITM_deregisterTMCloneTab
    78aa:   48 85 c0                test   %rax,%rax
    78ad:   74 09                   je     78b8 <deregister_tm_clones+0x28>
    78af:   ff e0                   jmpq   *%rax
    78b1:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)
```

Dynamo rio

Pintool -- debug, collect info

Verify coverage data -- shared lib & multi-thread

Re-calculate distance -- random-14815 (not work)

Debug information -- find related source code from coverage address -- using pintool (rtn)

Find bug in 14815

Strength of differential testing (rather than normal fuzzing) -- find semantic error

Add gcc to script -- verify result

todo

Has used pintool to collect each bbl’s corresponding function -- analyze which part or stage of the runtime is more unstable. And that means, we could generate test case in a targeted manner by adjust the strategy of csmith(random test case generator), to get “interesting test case” that could occur bugs in runtimes in a more efficient way.

Interesting test cases found -- random-61519, random-14815, random-78992

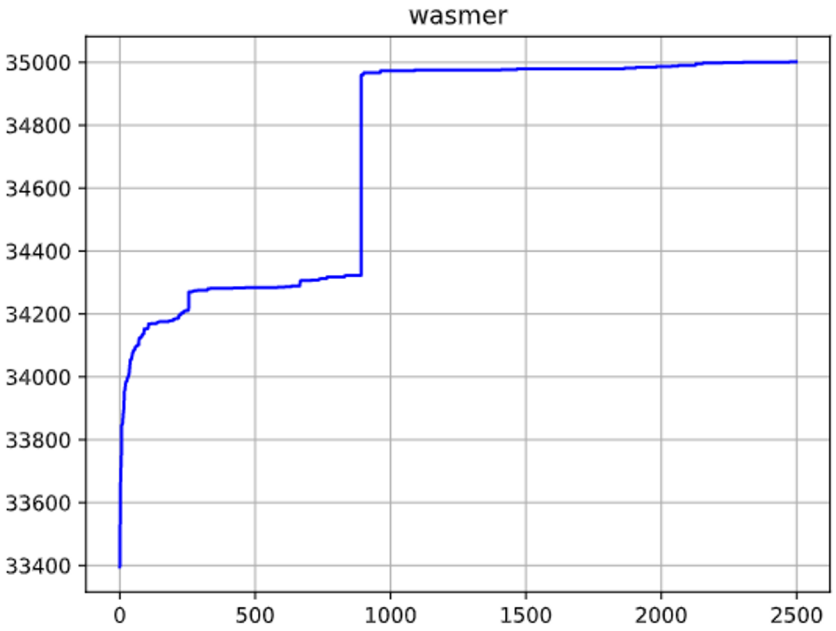
Strength of differential testing (rather than normal fuzzing) -- find semantic error, normal fuzzing focuses on crashes in one target rather than semantic error like producing wrong result. By using differential testing, we could find if there is semantic errors comparing results of different runtimes.

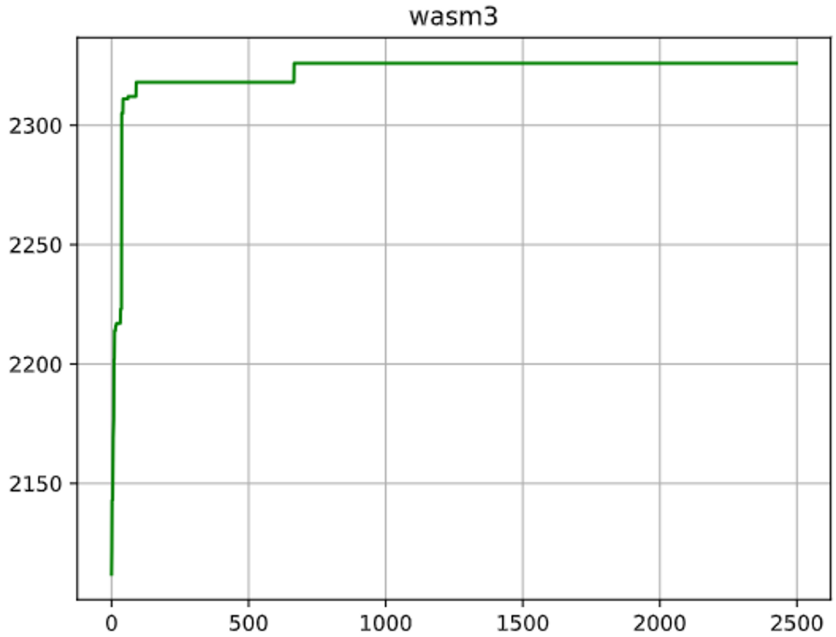
Wasmer: no.893 : 638 new bbls (case 256: 58 new bbls)(find similarity and difference between 2 cases -- to find individual feature or common feature)

wasm3: no. 667: 8 new bbls ， no.91:6

Find functions related to these new bbls -- change script

Double check interesting cases.





Interesting test case

Random-78992:

```
jackie@lever:~/work/wasmtest/csmith/diffs$ cat result-78992.txt
timeout: the monitored command dumped core
checksum = FB47E98C
checksum = FB47E98C
Error: compiling function overran its stack height limit
```

Random-61569:

```
jackie@lever:~/work/wasmtest/csmith/diffs$ cat result-61569.txt
error: failed to run `/home/jackie/work/wasmtest/csmith/generated/random-61569.wasm`
|   1: module instantiation failed (engine: universal, compiler: cranelift)
└─> 2: Validation error: locals exceed maximum (at offset 693)
Error: failed to run main module `/home/jackie/work/wasmtest/csmith/generated/random-61569.wasm`

Caused by:
    0: WebAssembly failed to compile
    1: WebAssembly translation error
    2: Invalid input WebAssembly code at offset 693: locals exceed maximum
checksum = 3C388430
Error: compiling function overran its stack height limit
```

progress

Update statistic script -- could collect new bbls and related function names with their executed times in interesting test cases

Number of new bbl in this test case

The related function name of all new bbls

& sum of all executed times of bbls

Details : function name and its related new bbls

Check the related feature of these bbls

& if the rtn is executed in former test cases.

progress

Interesting cases:

```
jackie@lever:~/work/wasmtest/csmith/diffs$ cat result-89828.txt
error: failed to run `/home/jackie/work/wasmtest/csmith/generated/random-89828.wasm`
|   1: module instantiation failed (engine: universal, compiler: cranelift)
└─> 2: Validation error: locals exceed maximum (at offset 675)
Error: failed to run main module `/home/jackie/work/wasmtest/csmith/generated/random-89828.wasm`

Caused by:
    0: WebAssembly failed to compile
    1: WebAssembly translation error
    2: Invalid input WebAssembly code at offset 675: locals exceed maximum
checksum = 5C6EBB88
Error: compiling function overran its stack height limit
```

```
jackie@lever:~/work/wasmtest/csmith/diffs$ cat result-80633.txt
error: failed to run `/home/jackie/work/wasmtest/csmith/generated/random-80633.wasm`
|   1: module instantiation failed (engine: universal, compiler: cranelift)
└─> 2: Validation error: locals exceed maximum (at offset 671)
Error: failed to run main module `/home/jackie/work/wasmtest/csmith/generated/random-80633.wasm`

Caused by:
    0: WebAssembly failed to compile
    1: WebAssembly translation error
    2: Invalid input WebAssembly code at offset 671: locals exceed maximum
checksum = BC4705EE
Error: compiling function overran its stack height limit
```

++

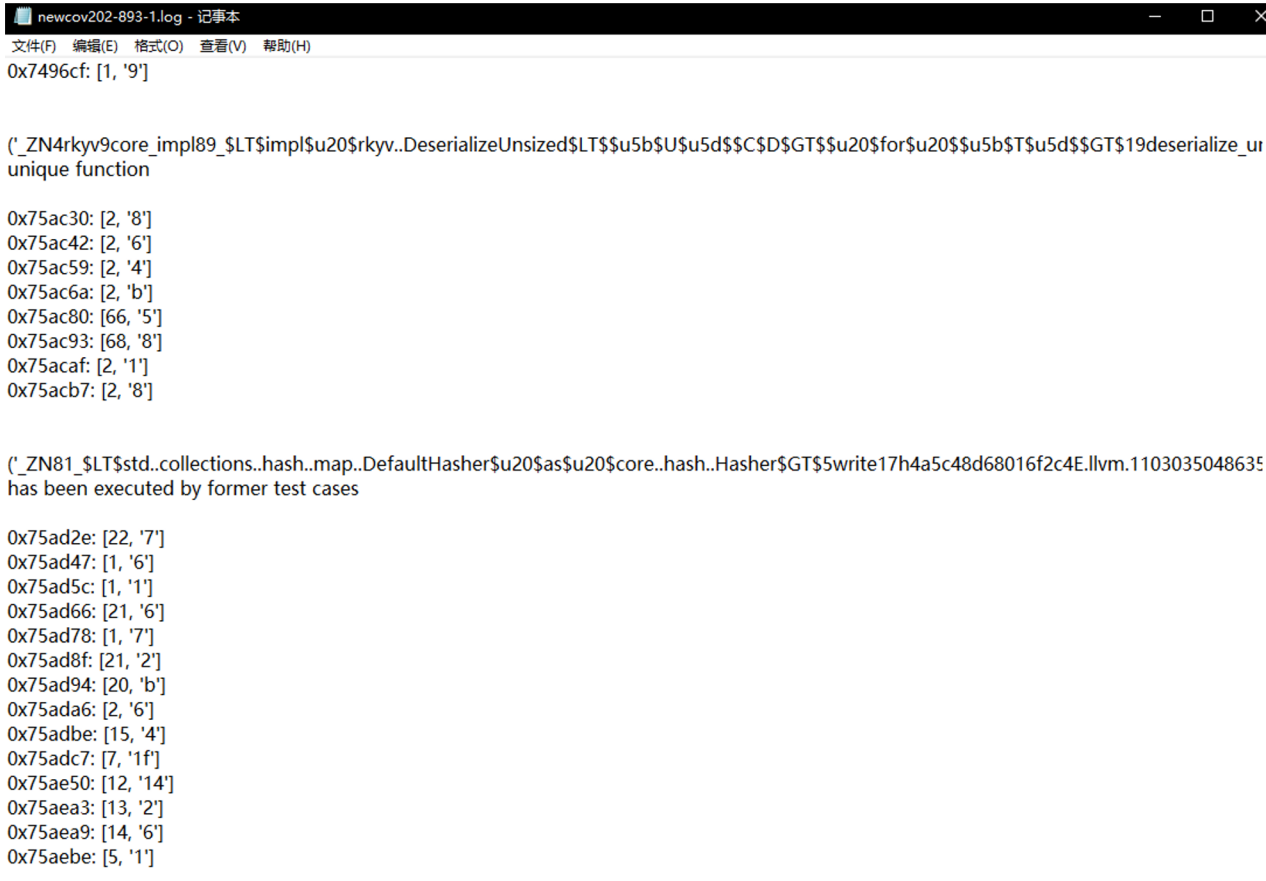
```
jackie@lever:~/work/wasmtest/csmith/diffs$ cat result-79675.txt
error: failed to run `/home/jackie/work/wasmtest/csmith/generated/random-79675.wasm`
|   1: module instantiation failed (engine: universal, compiler: cranelift)
└─> 2: Validation error: locals exceed maximum (at offset 798)
Error: failed to run main module `/home/jackie/work/wasmtest/csmith/generated/random-79675.wasm`

Caused by:
    0: WebAssembly failed to compile
    1: WebAssembly translation error
    2: Invalid input WebAssembly code at offset 798: locals exceed maximum
checksum = 102A4BA9
Error: compiling function overran its stack height limit
```

```
result-14815.txt
result-1.txt
result-61569.txt
result-78992.txt
result-79675.txt
result-80633.txt
result-89828.txt
```

todo

The rtn name collected by pintool in runtimes written by rust has garbled characters -- how to fix it.



wasm3:no667

details:
('PreserveArgsAndLocals', 33777)
has been executed by former test cases

0x24e9a: [9, '4']
0x24eaa: [9, '3']
0x24eb9: [6750, '6']
0x24ecd: [6750, '7']
0x24ee8: [6750, '3']
0x24ef7: [6750, '3']
0x24f75: [6750, '4']
0x24f79: [9, '3']

```
M3Result  PreserveArgsAndLocals  (IM3Compilation o)
{
    M3Result result = m3Err_none;

    if (o->stackIndex > o->stackFirstDynamicIndex)
    {
        u32 numArgsAndLocals = GetFunctionNumArgsAndLocals (o->function);

        for (u32 i = 0; i < numArgsAndLocals; ++i)
        {
            u16 slot = GetSlotForStackIndex (o, i);

            u16 preservedSlotNumber;
            (FindReferencedLocalWithinCurrentBlock (o, & preservedSlotNumber, slot));

            if (preservedSlotNumber != slot)
            {
                u8 type = GetStackTypeFromBottom (o, i);                    d_m3Assert (type != c_m3Type_none)
                IM3Operation op = Is64BitType (type) ? op_CopySlot_64 : op_CopySlot_32;

                EmitOp          (o, op);
                EmitSlotOffset  (o, preservedSlotNumber);
                EmitSlotOffset  (o, slot);
            }
        }
    }

    _catch:
    return result;
}
```

The source code of the new bbls in the interesting cases

Wams3: no667

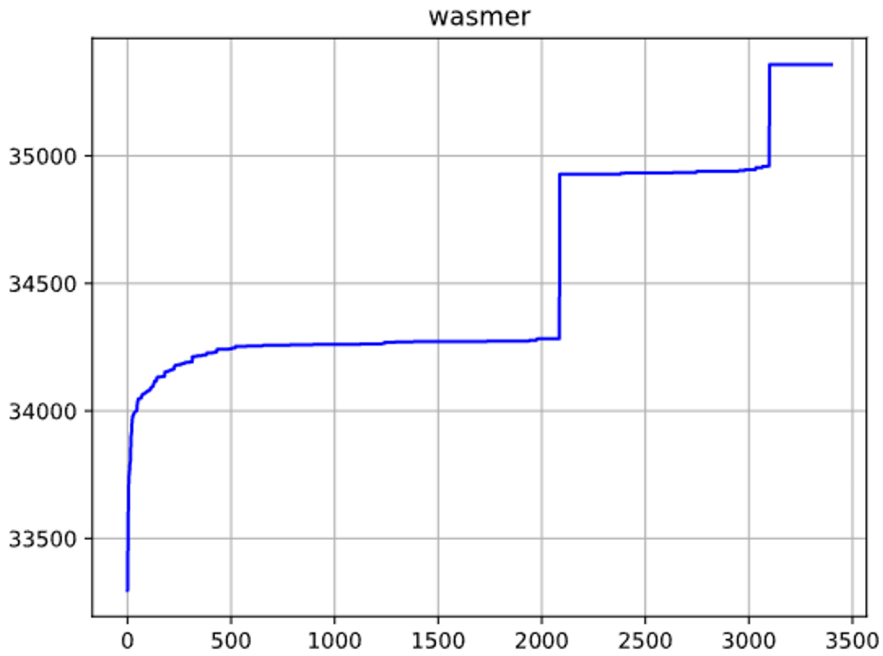
There is a endless loop in the related source code. If we delete the loop, then there will be no “new bbls”.

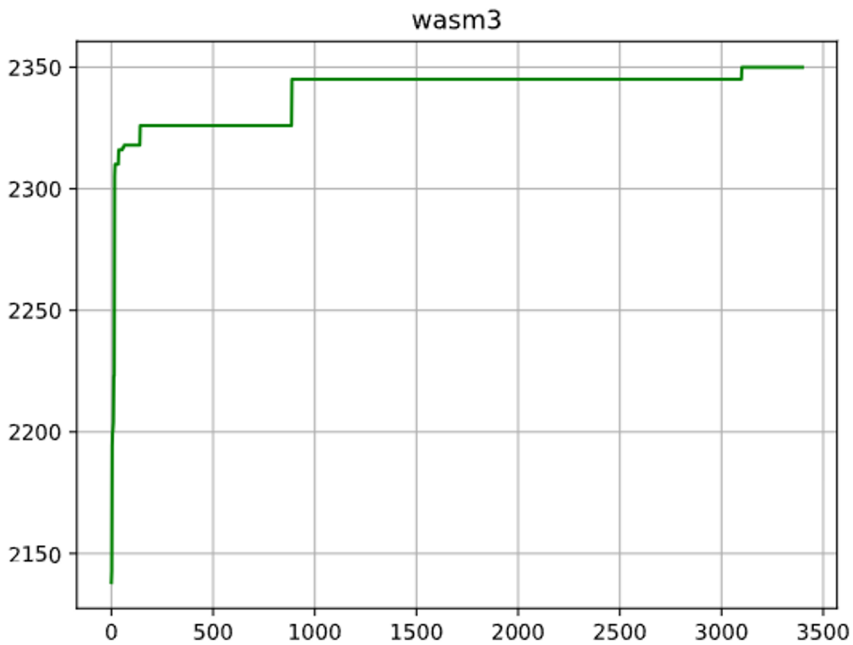
Some new interesting cases(generating new bbls)

Wasm3: no. 142, no. 888 (seems that they all have endless loop), no. 3100(throw error)

Wasmer: no. 2087(execute successfully) no. 3100(fail to execute and throw error)

Wasm3:





Try to find feature related to “preserveargsandlocals”

In test case 667: change loop condition

Const uint_64 65535: [761,760]

Const uint_32 65535: [759,758]

Const uint_16 65535: [759,758]

After changed type of loop condition, loop number, or whether it is const:

Times or type cannot influence number of “preserveargsandlocals”, the only reason is whether it is const. ()

Try to find feature related to “preserveargsandlocals”

But “const” cannot cause “preserve” itself,

Try to make some simple loop structure with a const number as its condition,

Cannot cause “preserve”.

So there should be another reason.

Try to find feature related to “preserveargsandlocals”

Endless loop with a const as its loop condition

Nested function with bit operation?

Delta debugging

Clang plug-in

Llvm

--transformation

Todo list

1. One feature (code coverage) could be triggered by different type of test cases. A test cases will be considered as “interesting case” if it has a new code coverage. Follow-up test cases with the same coverage will not be considered as “interesting” even if the coverage might be triggered by different source code feature.

   Therefore, we need to double check if a coverage is triggered be feature we have already analyzed and collect, to avoid missing new feature for further fuzzing.

Todo list

2. After analyzing and collecting features, we need to use these feature to reproduce new test cases for fuzzing (with higher code coverage). Theoretically, a reproduced test code block could trigger related coverage all the time.

However, in practice, some simple reproduced test code block could trigger coverage in a simple test cases. But it cannot trigger coverage in a more complicated test cases.

This issue needs further study.

Using AST to analyze and reproduce coverage-related feature

After using pintool to identify the location of test code which could trigger new coverage, we need to analyze and find the exact feature.

In summary, we need to recursively remove code from the identified code block to see if the rest code could still trigger the coverage, so we could locate the exact statement.

After that, simplify the statement to find the minimal structure. We need to do some mutation to check if a code node is redundant, and then remove the redundant node to reduce the statement or block.

Using AST to analyze and reproduce coverage-related feature

This progress will be easier to implement by using AST operation rather than directly modifying source code.

Clang is a front-end compiler, providing rich details of AST so that we could write our own plugin to do source-to-source transformation based on AST.

Pseudo algorithm -- analyzing coverage-related feature

Preparation: using pintool and wasm text file to identify the approximately location of the coverage-related code.

At this stage, we might get a statement or a code block containing the code we want.

1. If get a statement, start simplification.
2. If get a code block, need to remove redundant statements.
    a. Recursively remove the sub-block or statements to check if the rest code could still trigger coverage.
    b. Repeat (a) to get minimal block, this block contains feature that could trigger coverage
3. After get the exact block, we need to do simplification

Pseudo algorithm -- analyzing coverage-related feature

Simplification: the statement or block we find may contains lots of redundant structure irrelevant to the coverage. Remove them and get a minimal structure so that we could reproduce test cases.

This progress should base on AST.

1. Start at the innermost layer of ast, try to change the node (type, value)
2. If (1) could trigger coverage, move this layer while avoiding syntax error
3. If (2) the rest code could still trigger coverage, then the removed layer might be redundant.
4. If (1)(2) or (3) causing no coverage triggered, then the original node or layer might be the feature we want to find. Preserve it, and repeat (123) on the outer layer or other node on the same layer.

Pseudo algorithm -- analyzing coverage-related feature

Double check: after simplification, we should get a minimal structure that could trigger coverage. We need to double check if we miss some feature when simplifying.

1. Everytime we find a related layer or node in simplification, we could remove it and then add the removed redundant code back to the statement or block to see if coverage could be triggered. If the added-back statement cannot trigger the coverage, we could say that the removed code is actually redundant and the related layer or node is the key feature.
2. Or we can add redundant code to the minimal structure we have simplified to check if the feature we found could always trigger the coverage

Pseudo algorithm -- analyzing coverage-related feature

The thinking of simplification is like delta debugging (both need to pinpoint the issue).

If we could collect enough features related to new coverages, we could then feedback these feature to test case generator to produce higher code-coverage test cases, which could help us find error and bugs more efficient that just do brute force fuzzing without any guide. (actually this is very similar to coverage-guided fuzzing, but more targeted)

Source-to-source transformation (using clang AST)

Todo list

2-d feature-runtime map

Improve the algorithm

Combine clang plugin and pintool

Different from delta debugging

Missing features (multiple features)

How to feedback

Baseline -- use different compile option or runtime option

How to use clang libtools to do automated feature extraction

2 issues:

(1) Recursively access Stmt node from outer layer to inner layer, and remove it. Record the accessed node and coverage result.
   (a) Possible issues: syntax error during remove
   (b) Try to locate stmt without remove -- instrument log stmt?
      (i) Will log insertion influence runtime behaviour?
      (ii) Or directly use wasm text file to locate

(2) Automated mutation strategy on AST node from the inner layer to the outer layer. Preserve potential features.

Coverage features feedback to code generator strategy

Make the code generator tendentiously produces test cases with certain structures(features).

The features are extracted from “interesting test cases” which have a significant increases in new coverage by clang plugin.

Combine pintool and clang plugin

To build a automated feature analysis tool:

1. Use pintool to identify the approximate location of stmt that have potential feature.
2. After located exact location of stmt, clang plugin will use pintool to verify whether the coverage is influenced by each mutation operation on the identified stmt.

A simple way is to just watch on whether there is coverage after mutation.

Or record the times of coverage to do a finer analysis.