M9 - Interpreter-Guided Testing

Guillermo Polito

guillermo.polito@inria.fr
@guillep

Goals

- See an advanced use case of compiler testing
- Interpreters as executable language specifications
- Guided fuzzing based on concrete-symbolic analysis
- Differential testing

@PLDI'22

Link: https://hal.inria.fr/hal-03607939v1

PLDI'22

Interpreter-Guided Differential JIT Compiler Unit Testing

Guillermo Polito
Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille
France
guillermo.polito@univ-lille.fr

Stéphane Ducasse
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL
France
stephane.ducasse@inria.fr

Pablo Tesone
Pharo Consortium
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL
France
pablo.tesone@inria.fr

Abstract

Modern language implementations using Virtual Machines feature diverse execution engines such as byte-code interpreters and machine-code dynamic translators, a.k.a. JIT compilers. Validating such engines requires not only validating each in isolation, but also that they are functionally

San Diego, CA, USA. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3519939.3523457

1 Introduction

Modern Virtual Machines support code generation for JIT

Interpreters

```
interpret
 [ true ] whileTrue: [
    currentBytecode := self fetchNextBytecode.
    self dispatch: currentBytecode ]
```

```
void interpret(){
  while (1){
    switch(nextInstruction){
      case push: …
      case pop: …
      case send: …
      …
    }
  }
}
```

Interpreter are Executable Semantics

Pharo VM Example

```
1  Interpreter >> bytecodePrimAdd
2    | rcvr arg result |
3    rcvr := self internalStackValue: 1.
4    arg := self internalStackValue: 0.
5    (objectMemory areIntegers: rcvr and: arg) ifTrue: [
6      result := (objectMemory integerValueOf: rcvr) + (
7    "Check for overflow"
8    (objectMemory isIntegerValue: result) ifTrue: [
9      self
10       internalPop: 2
11       thenPush: (objectMemory integerObjectOf: result).
12     ^ self fetchNextBytecode "success"]].
13   "Slow path, message send"
14   self normalSend
```

If both operands are integers

If their sum does not overflow

Else, slow path => message send

How can we exploit interpreters for **compiler testing**?

Interpreter-guided Fuzzing

Interpreter are Executable Semantics

Pharo VM Example

```
1  Interpreter >> bytecodePrimAdd
2    | rcvr arg result |
3    rcvr := self internalStackValue: 1.
4    arg := self internalStackValue: 0.
5    (objectMemory areIntegers: rcvr and: arg) ifTrue: [
6      result := (objectMemory integerValueOf: rcvr) + (
7    "Check for overflow"
8    (objectMemory isIntegerValue: result) ifTrue: [
9      self
10       internalPop: 2
11       thenPush: (objectMemory integerObjectOf: result).
12     ^ self fetchNextBytecode "success"]].
13 "Slow path, message send"
14 self normalSend
```

If both operands are integers

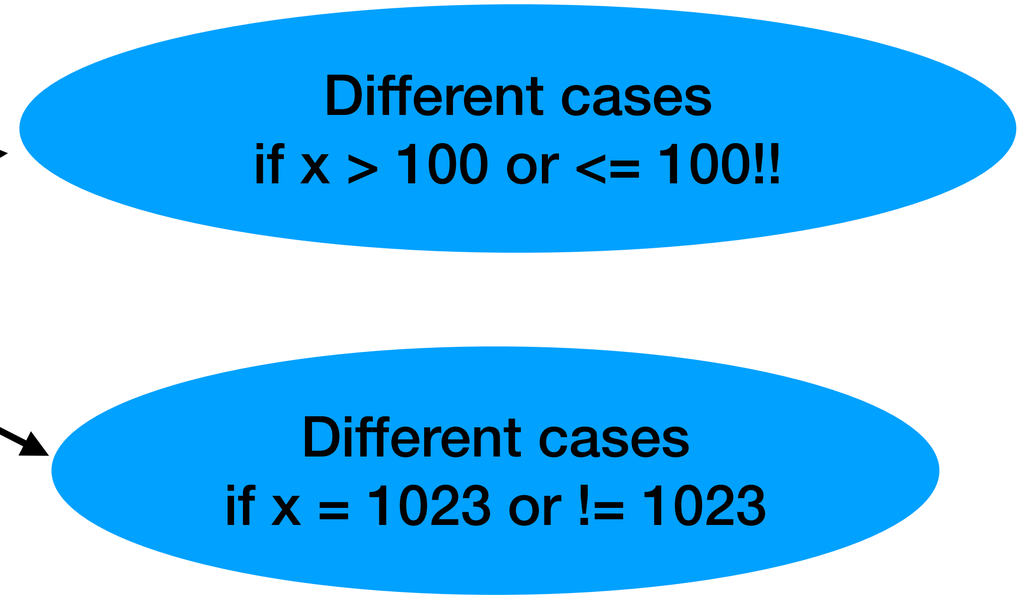
If their sum does not overflow

Else, slow path => message send

Intermezzo: Concolic Testing

- Idea: Guide test generation by looking at the implementation

```
int f(int x, int y){
 if (x > 100){
  if (y == 1023){
   segfault(!!)
  } } }
```



Godefroid et al. DART: Directed Automated Random Testing. PLDI' 05
Set et al. CUTE: a concolic unit testing engine for C. FSE'05

Concolic Testing by Example

- **Conc**rete + Symb**olic** execution
- Goal: automatically discover *all* execution *paths*

```
int f(int x, int y){
 if (x > 100){
  if (y == 1023){
   segfault(!!)
  } } }
```

x	y	constraints	next?

Godefroid et al. DART: Directed Automated Random Testing. PLDI' 05

Set et al. CUTE: a concolic unit testing engine for C. FSE'05

Concolic Testing by Example

- **Conc**rete + Symb**olic** execution
- Goal: automatically discover *all* execution *paths*

```
int f(int x, int y){
 if (x > 100){
  if (y == 1023){
   segfault(!!)
  } } }
```

x	y	constraints	next?
0	0	x <= 100	

Godefroid et al. DART: Directed Automated Random Testing. PLDI' 05

Set et al. CUTE: a concolic unit testing engine for C. FSE'05

Concolic Testing by Example

- **Conc**rete + Symb**olic** execution
- Goal: automatically discover *all* execution *paths*

```
int f(int x, int y){
 if (x > 100){
  if (y == 1023){
   segfault(!!)
  } } }
```

x	y	constraints	next?
0	0	x <= 100	x > 100

Godefroid et al. DART: Directed Automated Random Testing. PLDI' 05

Set et al. CUTE: a concolic unit testing engine for C. FSE'05

Concolic Testing by Example

- **Conc**rete + Symb**olic** execution
- Goal: automatically discover *all* execution *paths*

```
int f(int x, int y){
 if (x > 100){
  if (y == 1023){
   segfault(!!)
  } } }
```

x	y	constraints	next?
0	0	x <= 100	x > 100
101	0		

Godefroid et al. DART: Directed Automated Random Testing. PLDI' 05

Set et al. CUTE: a concolic unit testing engine for C. FSE'05

Concolic Testing by Example

- **Conc**rete + Symb**olic** execution
- Goal: automatically discover *all* execution *paths*

```
int f(int x, int y){
 if (x > 100){
  if (y == 1023){
   segfault(!!)
  } } }
```

x	y	constraints	next?
0	0	x <= 100	x > 100
101	0	x > 100, y != 1023	

Godefroid et al. DART: Directed Automated Random Testing. PLDI' 05

Set et al. CUTE: a concolic unit testing engine for C. FSE'05

Concolic Testing by Example

- **Conc**rete + Symb**olic** execution
- Goal: automatically discover *all* execution *paths*

```
int f(int x, int y){
 if (x > 100){
  if (y == 1023){
   segfault(!!)
  } } }
```

x	y	constraints	next?
0	0	x <= 100	x > 100
101	0	x > 100, y != 1023	x > 100, y == 1023

Godefroid et al. DART: Directed Automated Random Testing. PLDI' 05

Set et al. CUTE: a concolic unit testing engine for C. FSE'05

Concolic Testing by Example

- **Conc**rete + Symb**olic** execution
- Goal: automatically discover *all* execution *paths*

```
int f(int x, int y){
 if (x > 100){
  if (y == 1023){
   segfault(!!)
  } } }
```

x	y	constraints	next?
0	0	x <= 100	x > 100
101	0	x > 100, y != 1023	x > 100, y == 1023
101	1023		

Godefroid et al. DART: Directed Automated Random Testing. PLDI' 05

Set et al. CUTE: a concolic unit testing engine for C. FSE'05

Concolic Testing by Example

- **Conc**rete + Symb**olic** execution
- Goal: automatically discover *all* execution *paths*

```
int f(int x, int y){
 if (x > 100){
  if (y == 1023){
   segfault(!!)
  } } }
```

x	y	constraints	next?
0	0	x <= 100	x > 100
101	0	x > 100, y != 1023	x > 100, y == 1023
101	1023	x > 100, y == 1023	finished!

Godefroid et al. DART: Directed Automated Random Testing. PLDI' 05

Set et al. CUTE: a concolic unit testing engine for C. FSE'05

Concolic Meta-interpretation

```
1  Interpreter >> bytecodePrimAdd
2    | rcvr arg result |
3    rcvr := self internalStackValue: 1.
4    arg := self internalStackValue: 0.
5    (objectMemory areIntegers: rcvr and: arg) ifTrue: [
6      result := (objectMemory integerValueOf: rcvr) + (
         objectMemory integerValueOf: arg).
7      "Check for overflow"
8      (objectMemory isIntegerValue: result) ifTrue: [
9        self
10         internalPop: 2
11         thenPush: (objectMemory integerObjectOf: result).
12       ^ self fetchNextBytecode "success"]].
13   "Slow path, message send"
14   self normalSend
```

Interpreter Code

Argument 0 (type)	Argument 1(type)	Path
0 (integer)	0 (integer)	isInteger(arg0), isInteger(arg1), isInteger(arg0+arg1)
0xFFFFFFFF (integer)	1 (integer)	isInteger(arg0), isInteger(arg1), isNotInteger(arg0+arg1)
0 (integer)	object1 (object)	isInteger(arg0), isNotInteger(arg1)
object1 (object)	0 (integer)	isNotInteger(arg0), isInteger(arg1)
object1 (object)	object2 (object)	isNotInteger(arg0), isNotInteger(arg1)

Interpreter-based Differential Testing

Interpreter vs Compiled Code

Pharo VM Example

```
1  Interpreter >> bytecodePrimAdd
2    | rcvr arg result |
3    rcvr := self internalStackValue: 1.
4    arg := self internalStackValue: 0.
5    (objectMemory areIntegers: rcvr and: arg) ifTrue: [
6      result := (objectMemory integerValueOf: rcvr) + (
         objectMemory integerValueOf: arg).
7      "Check for overflow"
8      (objectMemory isIntegerValue: result) ifTrue: [
9        self
10         internalPop: 2
11         thenPush: (objectMemory integerObjectOf: result).
12       ^ self fetchNextBytecode "success"]].
13   "Slow path, message send"
14   self normalSend
```

Interpreter Code

```
1     ... # previous bytecode IR
2           checkSmallInteger t0
3           jumpzero notsmi
4           checkSmallInteger t1
5           jumpzero notsmi
6           t2 := t0 + t1
7           jumpIfNotOverflow continue
8   notsmi: #slow case first send
9           t2 := send #+ t0 t1
10  continue:
11    ... # following bytecode IR
```

Equivalent Compiler IR

Duplicated Semantics

Pharo VM Example

```
1  Interpreter >> bytecodePrimAdd
2    | rcvr arg result |
3    rcvr := self internalStackValue: 1.
4    arg := self internalStackValue: 0.
5    (objectMemory areIntegers: rcvr and: arg)
6      ifTrue: [result := (objectMemory integerValueOf: rcvr) + (
7    "Check for overflow"
8    (objectMemory isIntegerValue: result)
9      ifTrue: [
10       self
11         internalPop: 2
12         thenPush: (objectMemory integerObjectOf: result).
13     self fetchNextBytecode. "success"]].
14   "Slow path, message send"
     self normalSend
```

Interpreter Code

```
1    ... # previous bytecode IR
2    checkSmallInteger t0
3    jumpzero notsmi
4    checkSmallInteger t1
5    jumpzero notsmi
6    t2 := t0 + t1
7    jumpIfNotOverflow continue
8  notsmi: #slow case first send
9    t2 := send #+ t0 t1
10 continue:
11   ... # following bytecode IR
```

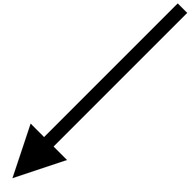
Equivalent Compiler IR

Example

Argument 0 (type)	Argument 1(type)	Path
0 (integer)	0 (integer)	isInteger(arg0), isInteger(arg1), isInteger(arg0+arg1)
0xFFFFFFFF (integer)	1 (integer)	isInteger(arg0), isInteger(arg1), isNotInteger(arg0+arg1)
0 (integer)	object1 (object)	isInteger(arg0), isNotInteger(arg1)
object1 (object)	0 (integer)	isNotInteger(arg0), isInteger(arg1)
object1 (object)	object2 (object)	isNotInteger(arg0), isNotInteger(arg1)

```
1  Interpreter >> bytecodePrimAdd
2    | rcvr arg result |
3    rcvr := self internalStackValue: 1.
4    arg := self internalStackValue: 0.
5    (objectMemory areIntegers: rcvr and: arg) ifTrue: [
6      result := (objectMemory integerValueOf: rcvr) + (
         objectMemory integerValueOf: arg).
7      "Check for overflow"
8      (objectMemory isIntegerValue: result) ifTrue: [
9        self
10         internalPop: 2
11         thenPush: (objectMemory integerObjectOf: result).
12       ^ self fetchNextBytecode "success"]].
13   "Slow path, message send"
14   self normalSend
```

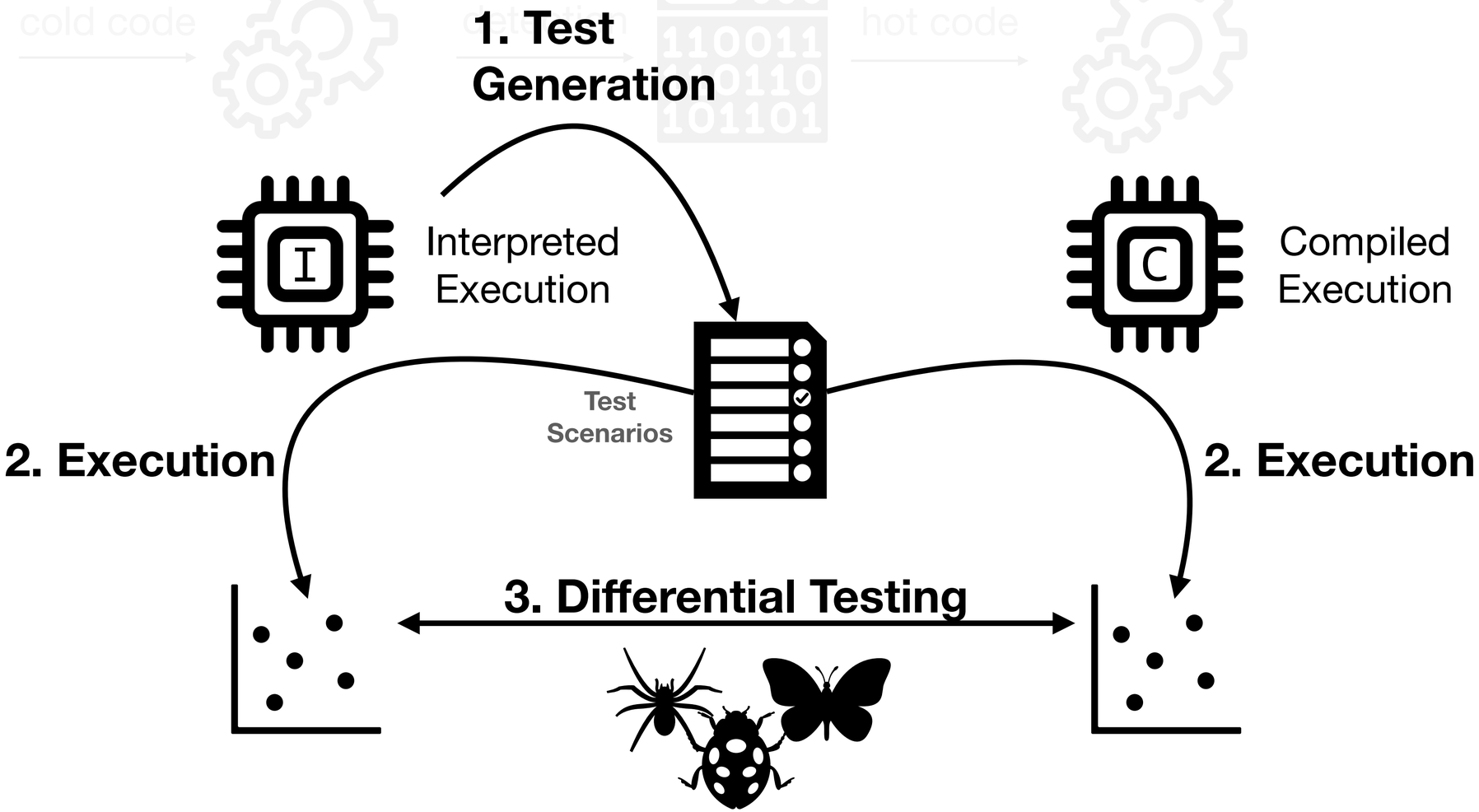
Listing 1. Excerpt of the byte-code interpretation implementing addition in the Pharo Virtual Machine.



```
1      ... # previous bytecode IR
2          checkSmallInteger t0
3          jumpzero notsmi
4          checkSmallInteger t1
5          jumpzero notsmi
6          t2 := t0 + t1
7          jumpIfNotOverflow continue
8    notsmi: #slow case first send
9          t2 := send #+ t0 t1
10   continue:
11     ... # following bytecode IR
```

Listing 2. Illustration of the Intermediate Representation instructions created when compiling the byte-code instruction in Listing 1.

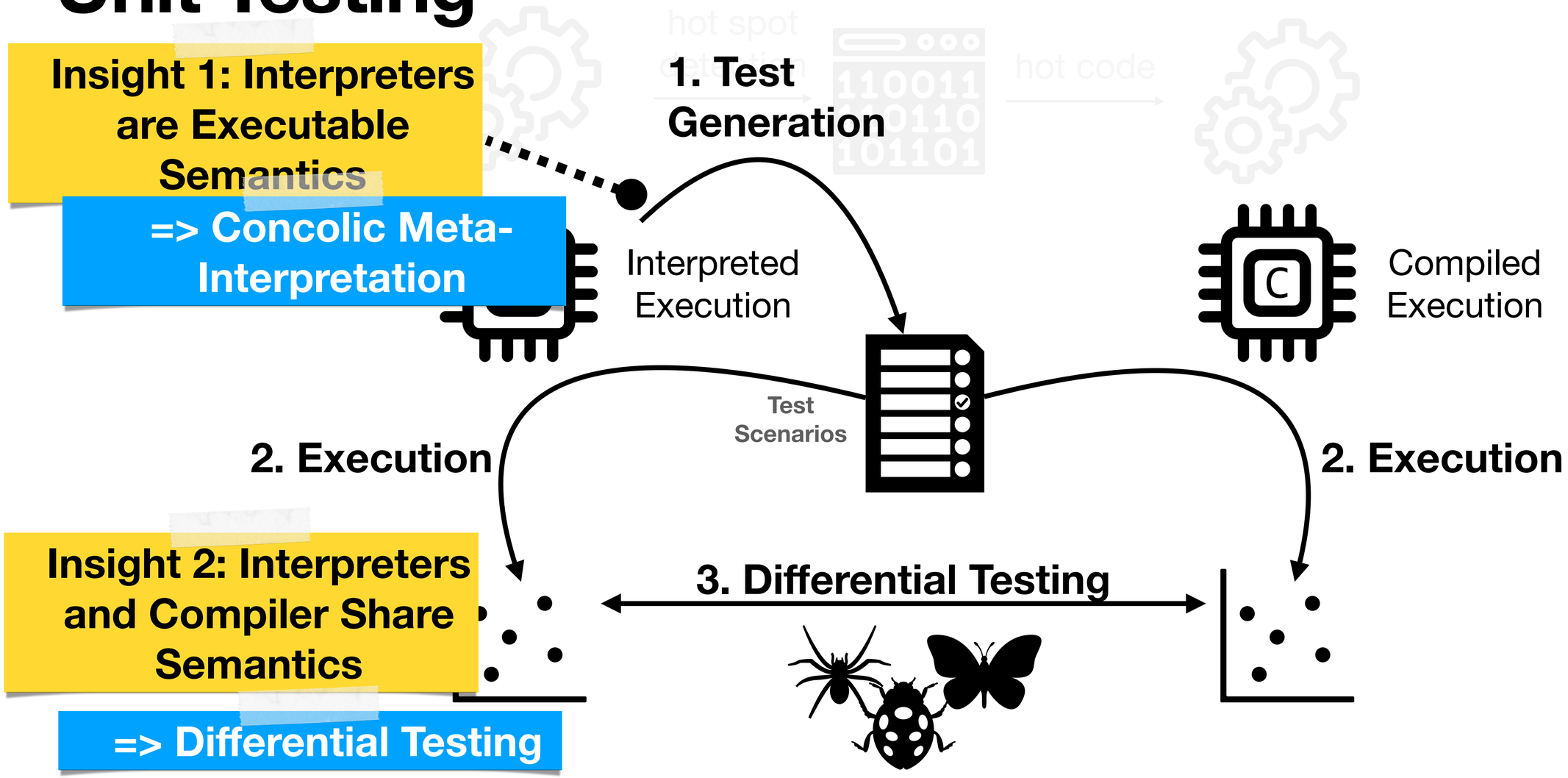
Interpreter-Guided Automatic JIT Compiler Unit Testing



Interpreter-Guided Automatic JIT Compiler Unit Testing

Insight 1: Interpreters are Executable Semantics

=> Concolic Meta-Interpretation



Insight 2: Interpreters and Compiler Share Semantics

=> Differential Testing

JIT + Interpreter Bugs!

- 3 bytecode compilers + 1 native method compiler
- 4928 tests generated
- **478 differences**

Compiler	# Tested Instructions	# Interpreter Paths	# Curated Paths	# Differences (%)
Native Methods (primitives)	112	2024	1520	440 (28,95%)
Simple Stack BC Compiler	175	1308	1136	18 (1,59%)
Stack-to-Register BC Compiler	175	1308	1136	10 (0,88%)
Linear-Scan Allocator BC Compiler	175	1308	1136	10 (0,88%)
Total	637	5948	4928	478 (9,7%)

Analysis of Differences Through Manual Inspection

- 91 causes, *6 different categories*
- Errors both in the interpreter AND the compilers
- 14 causes of ***segmentation faults***!

Family	# Cases
Missing interpreter type check	1
Missing compiled type check	13
Optimisation difference	10
Behavioral difference	5
Missing Functionality	60
Simulation Error	2

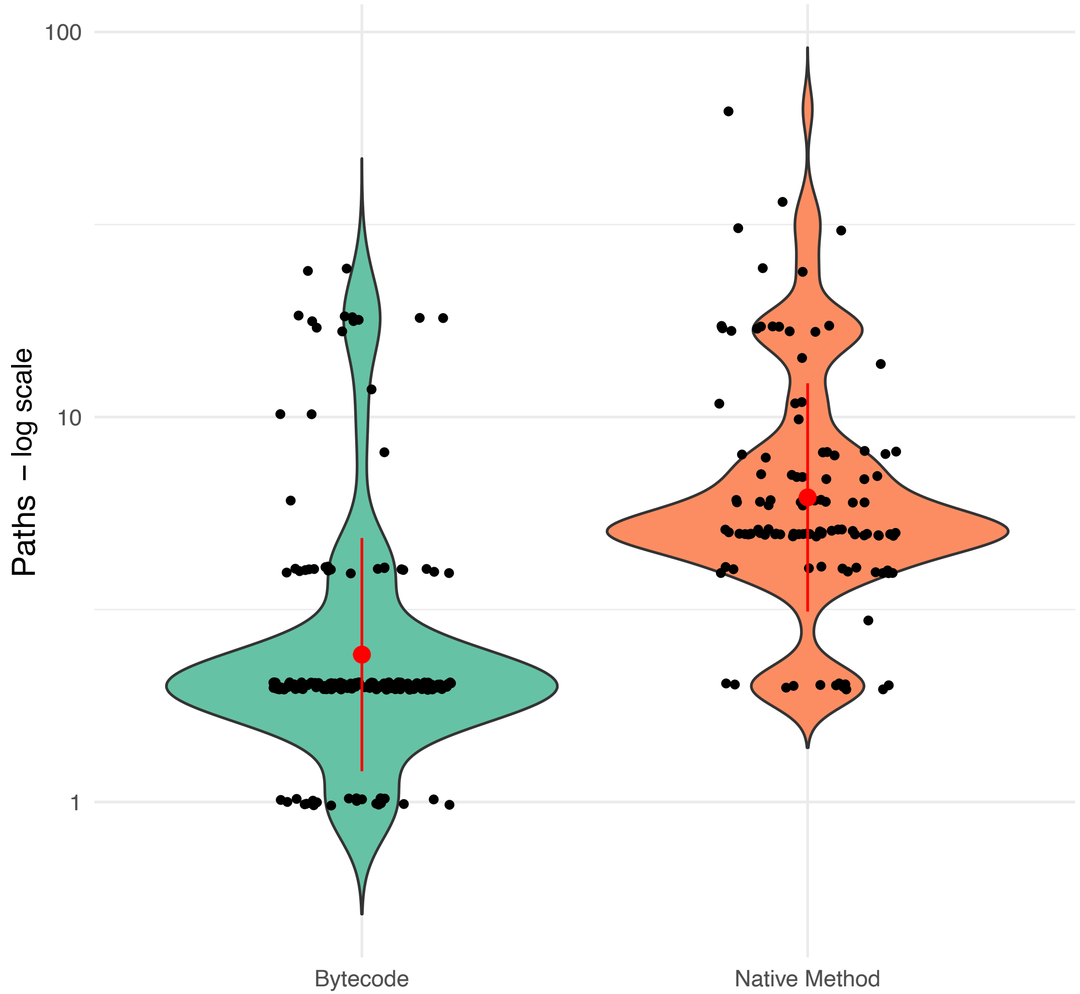
Characterising Concolic Execution

Paths per instruction

- Native methods present in average more paths than bytecode instructions

  => longer time to explore

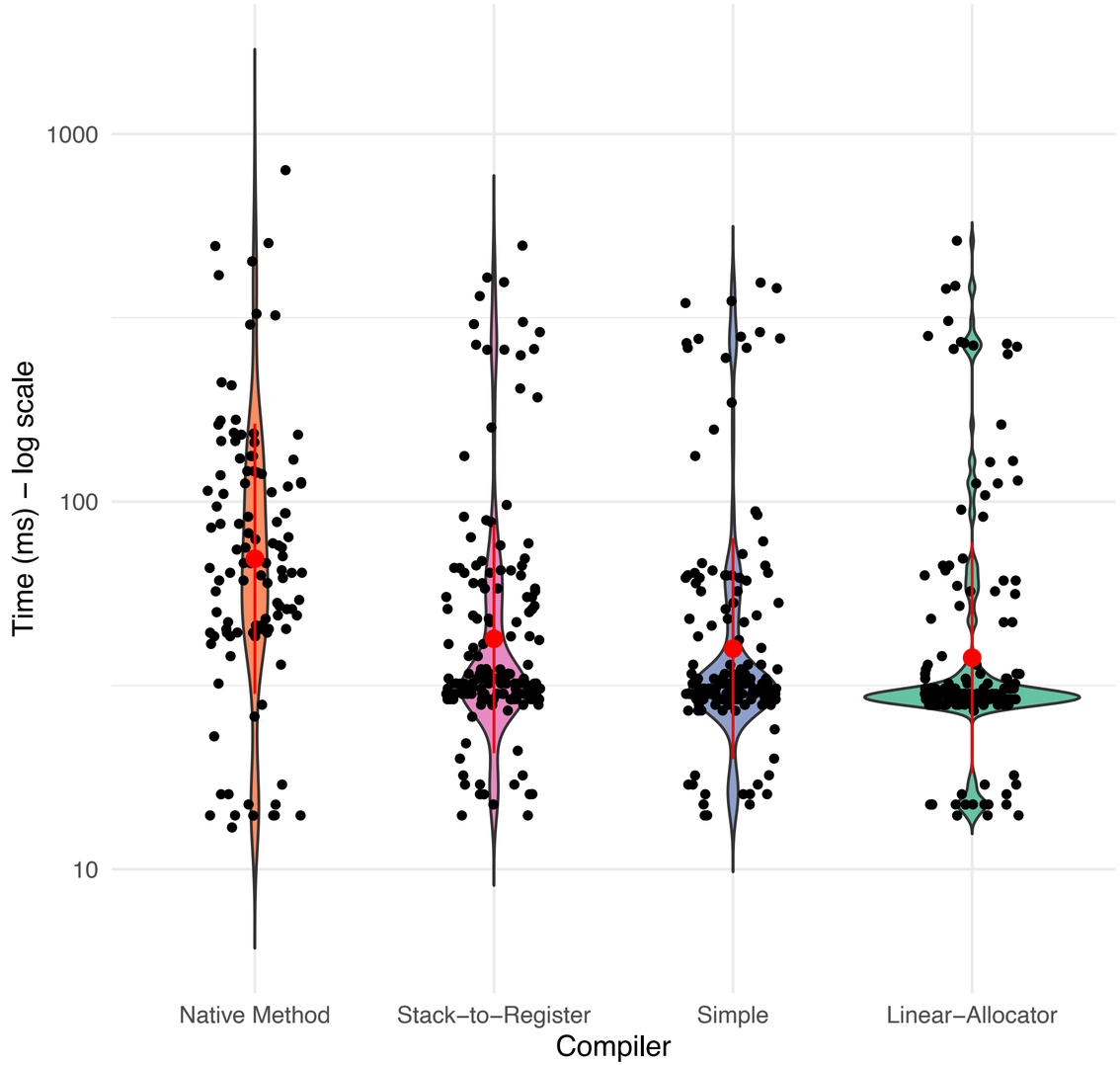
  => potentially more bugs



Paths per Type of Instruction

Practical and Cheap

- Test generation ~5 minutes
- Total run time of ~10 seconds
  - *Avg* 30ms per instruction



Takeaways

- Interpreters are executable language semantics
- Thus, they are useful to derive fuzzers and oracles

- Combining different techniques we can arrive to powerful testing

Material

- DART: Directed Automated Random Testing. PLDI’ 05
  Godefroid et al.
- CUTE: a concolic unit testing engine for C. FSE’05
  Set et al.
- Interpreter-Guided JIT Compiler Unit Testing. PLDI’22
  Polito et al.