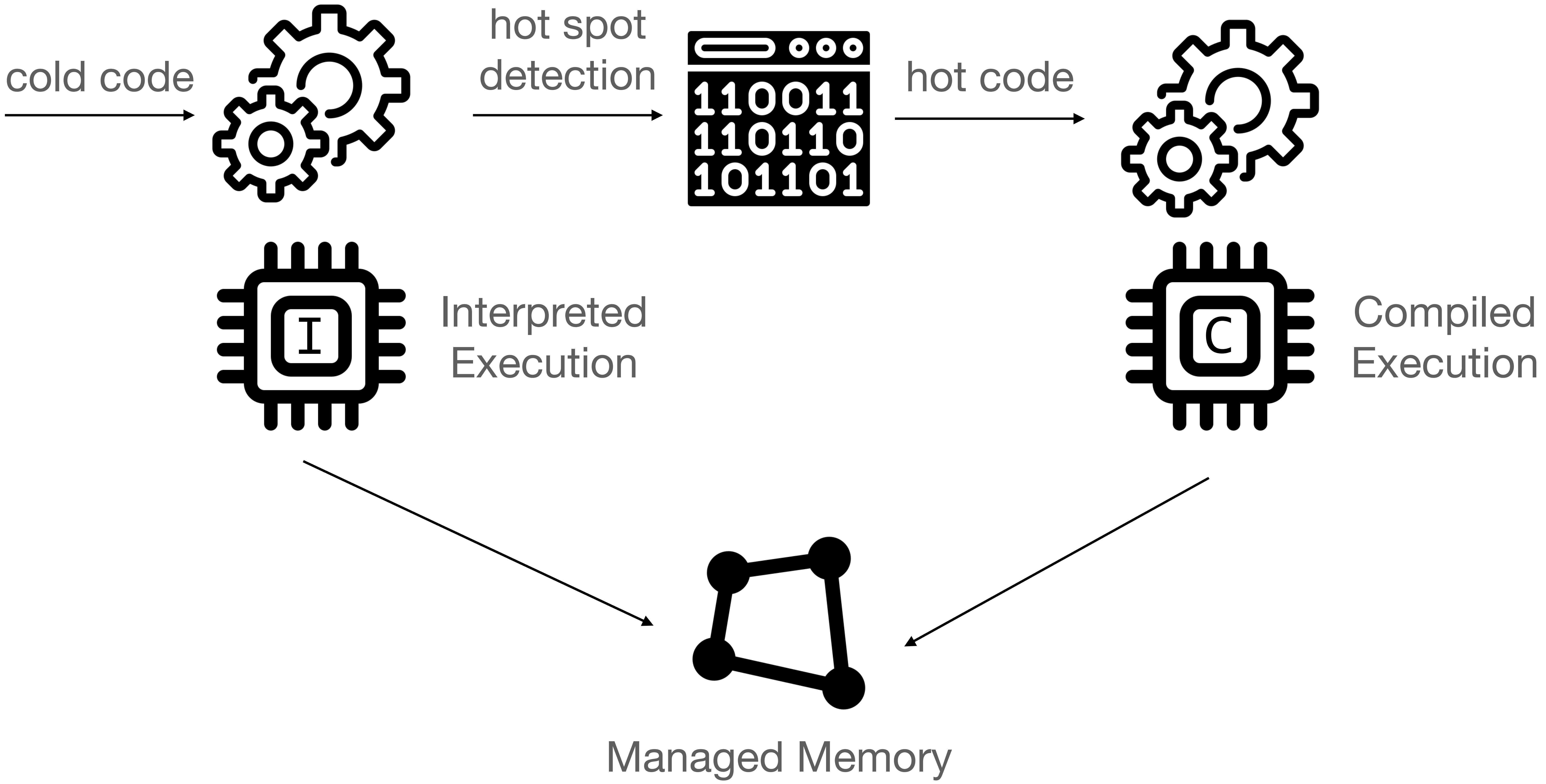
Interpreter-Guided JIT Compiler Test Generation

Validating the Pharo JIT compiler through **concolic** execution and **differential** testing

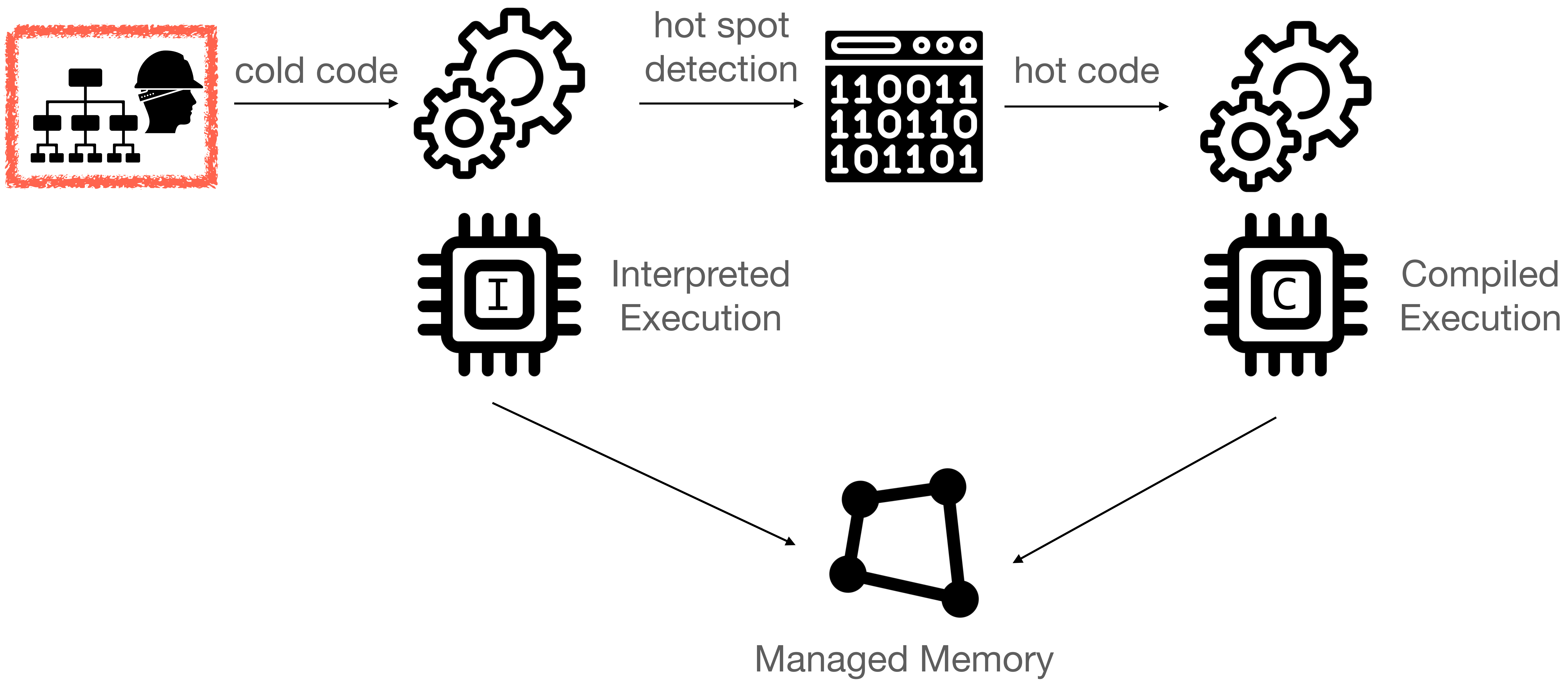
Guille Polito - Pablo Tesone - Stéphane Ducasse
guillermo.polito@univ-lille.fr
@guillep

PLDI'22 — San Diego

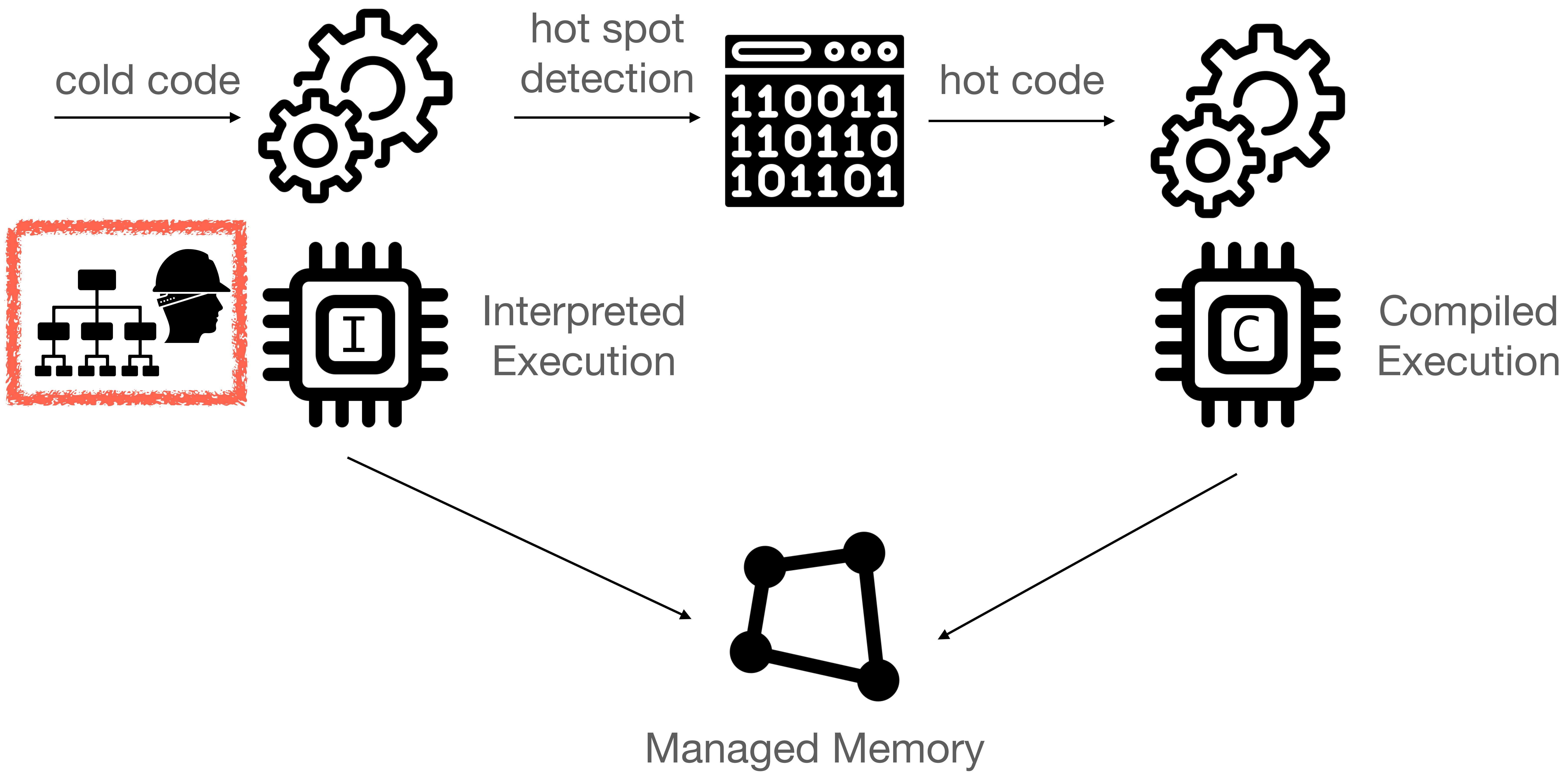
Virtual Machine Execution Engine



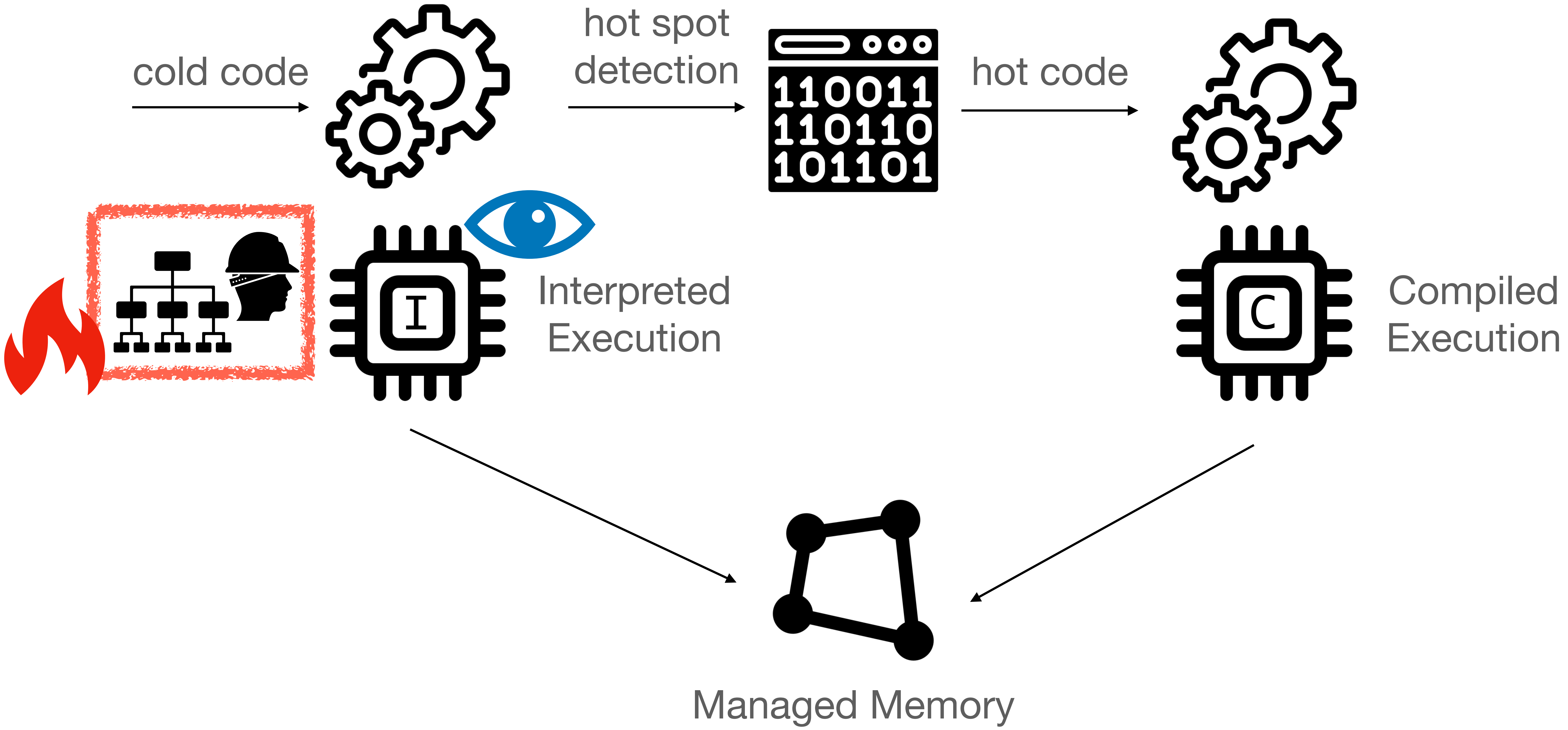
Virtual Machine Execution Engine



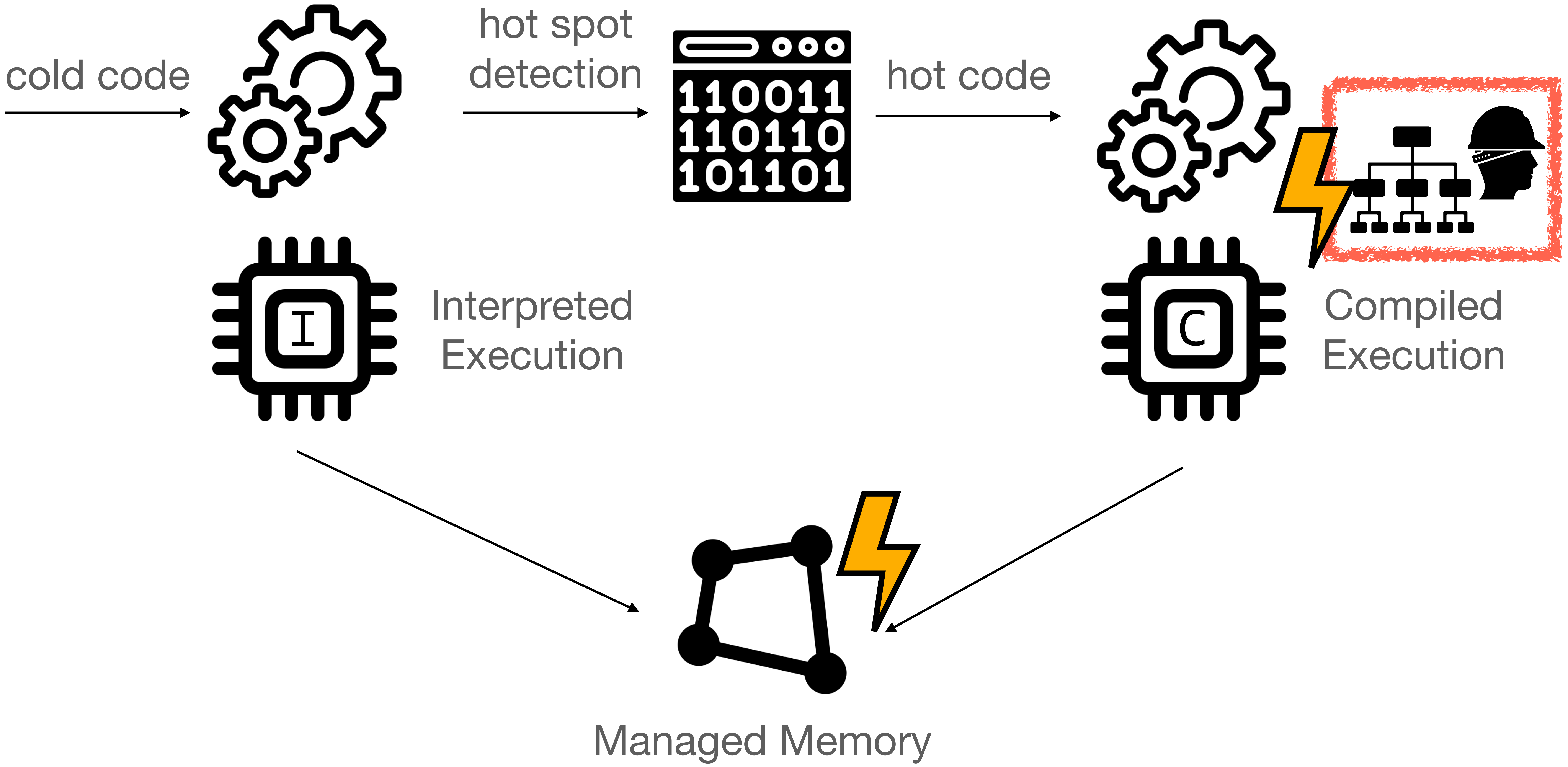
Virtual Machine Execution Engine



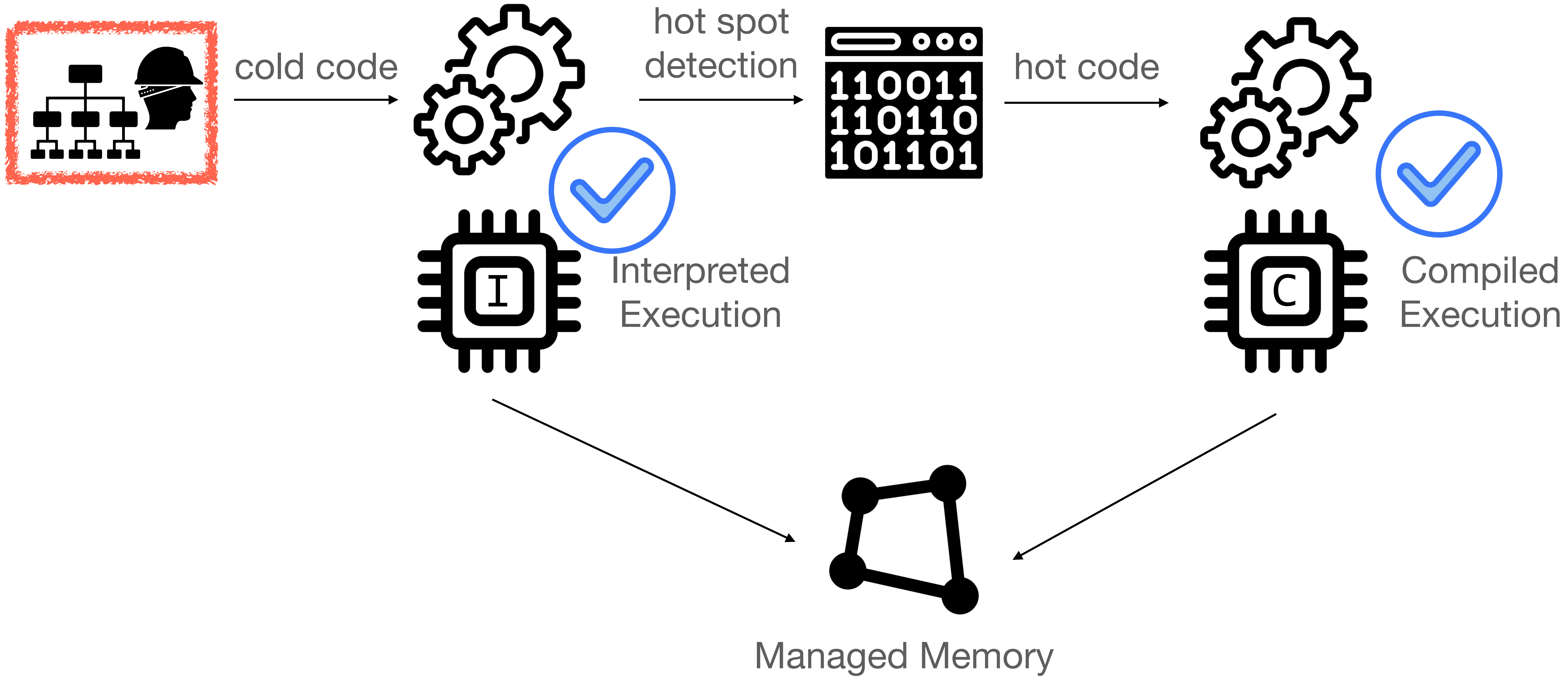
Virtual Machine Execution Engine



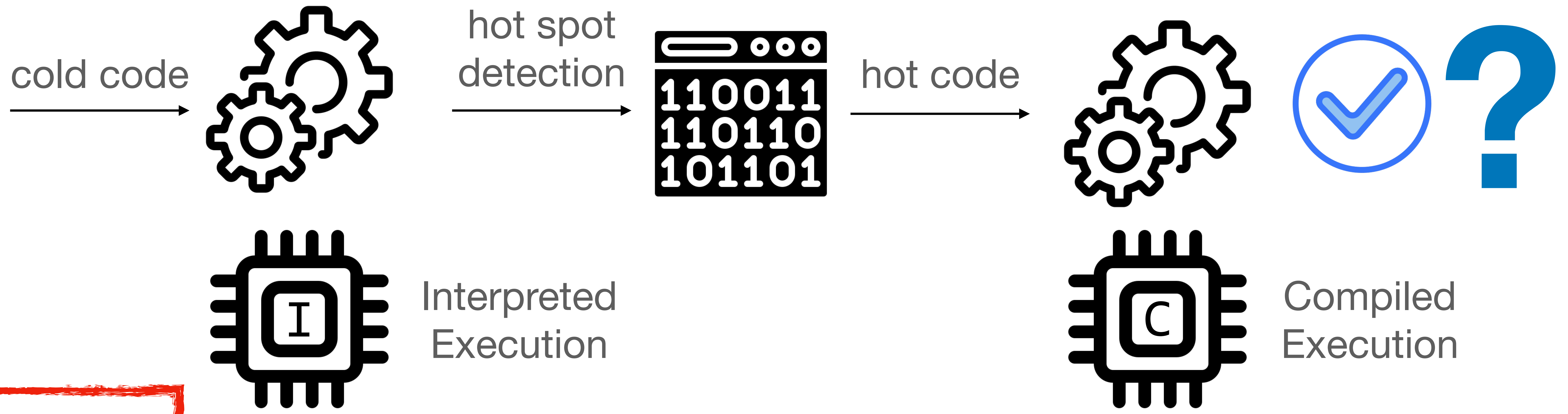
Virtual Machine Execution Engine



How can we automatically test VMs?



Challenges of VM Test Generation



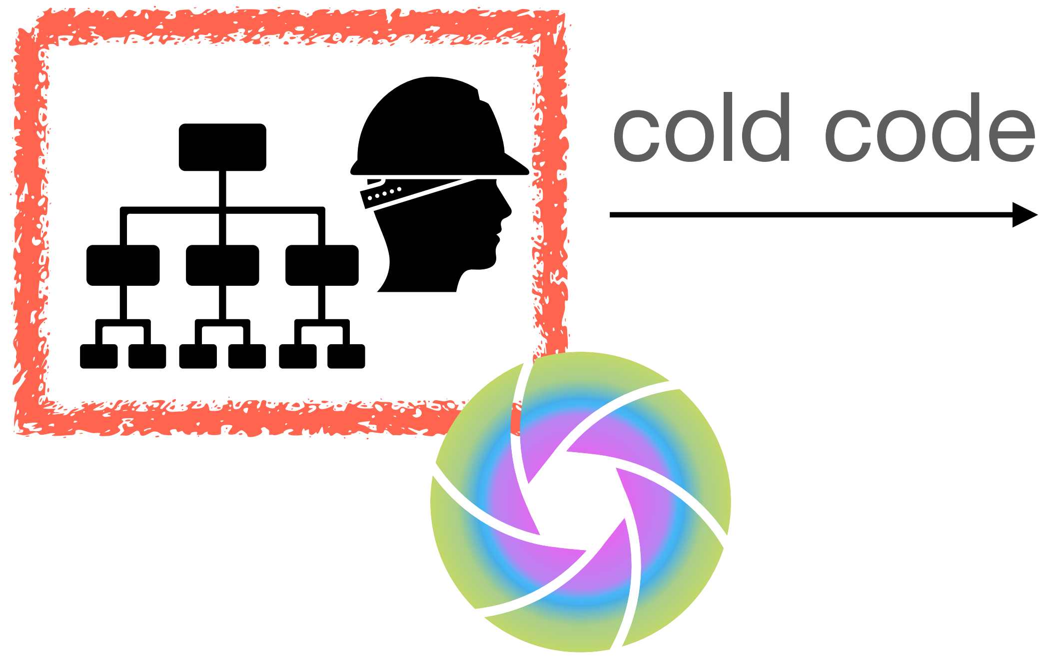
Challenge 1: Test Diversity

Do they cover different code ***regions/branches/paths***?

Challenge 2: Test Oracles

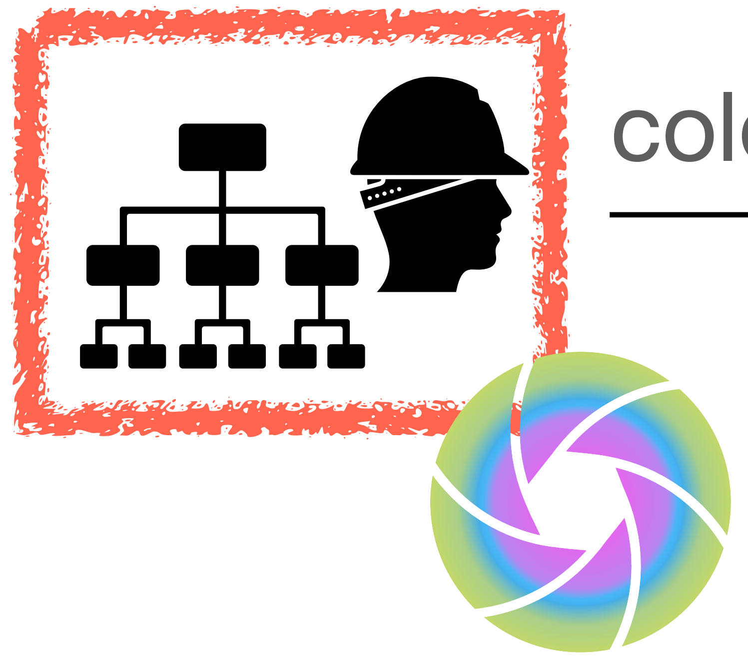
How do we determine what is the ***expected output*** of a generated test?

Black Box Testing + Fuzzing



[redacted]

Black Box Testing + Fuzzing

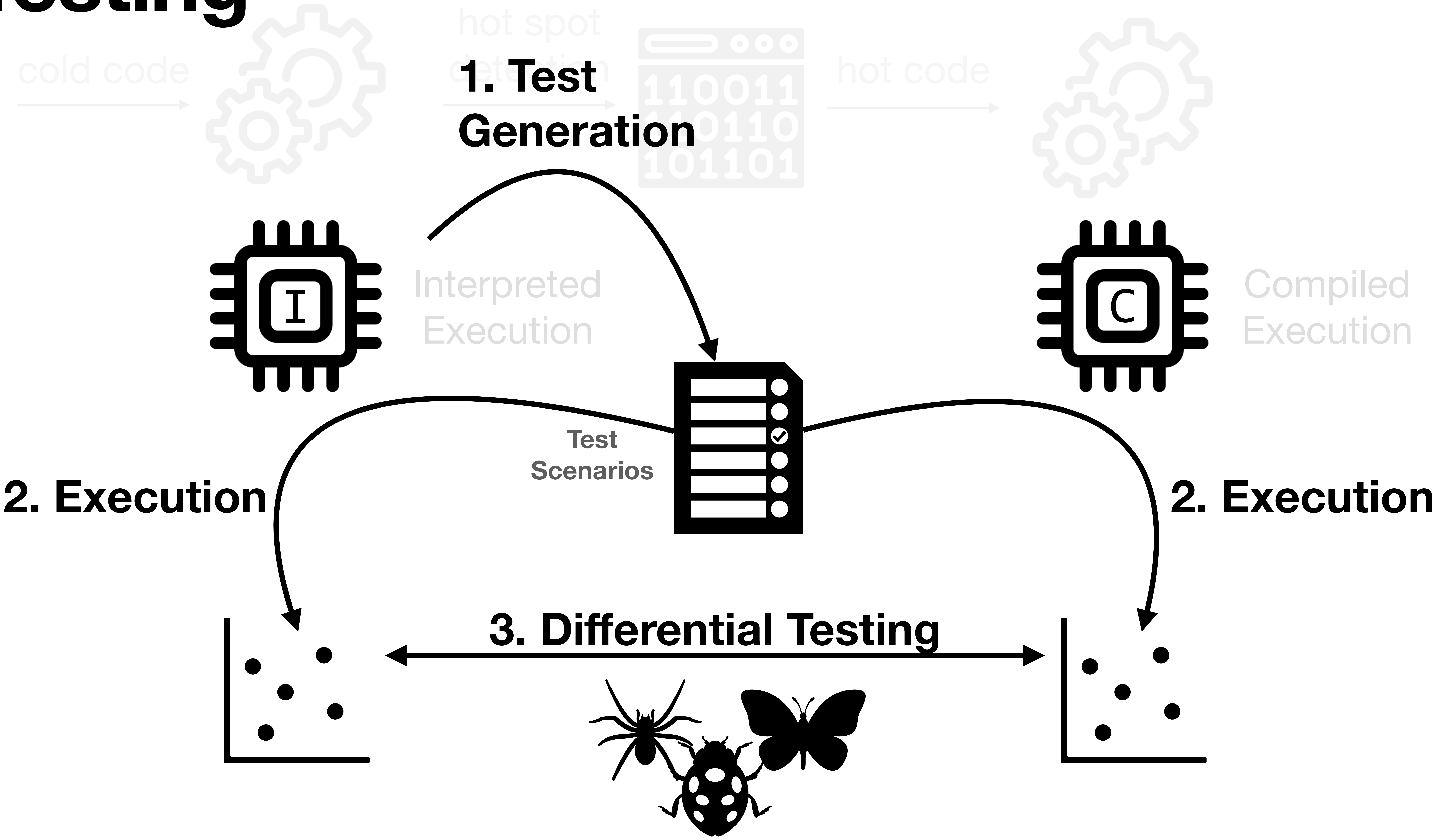


cold code

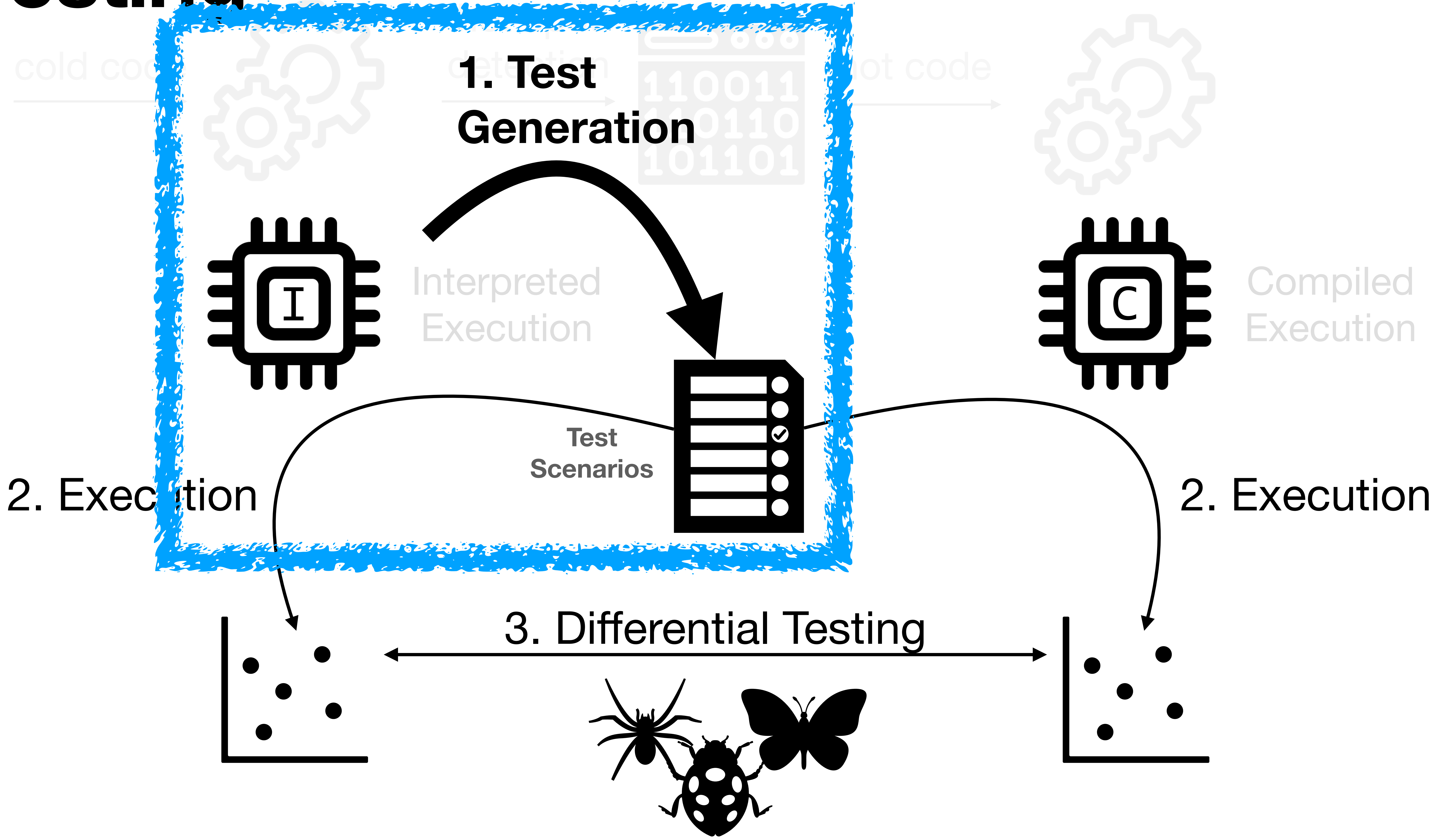
[redacted]

- ✗ Slow
- ✗ Coarse Grained
- ✗ Non Determinism
- ✗ Require Multiple Reference Implementations

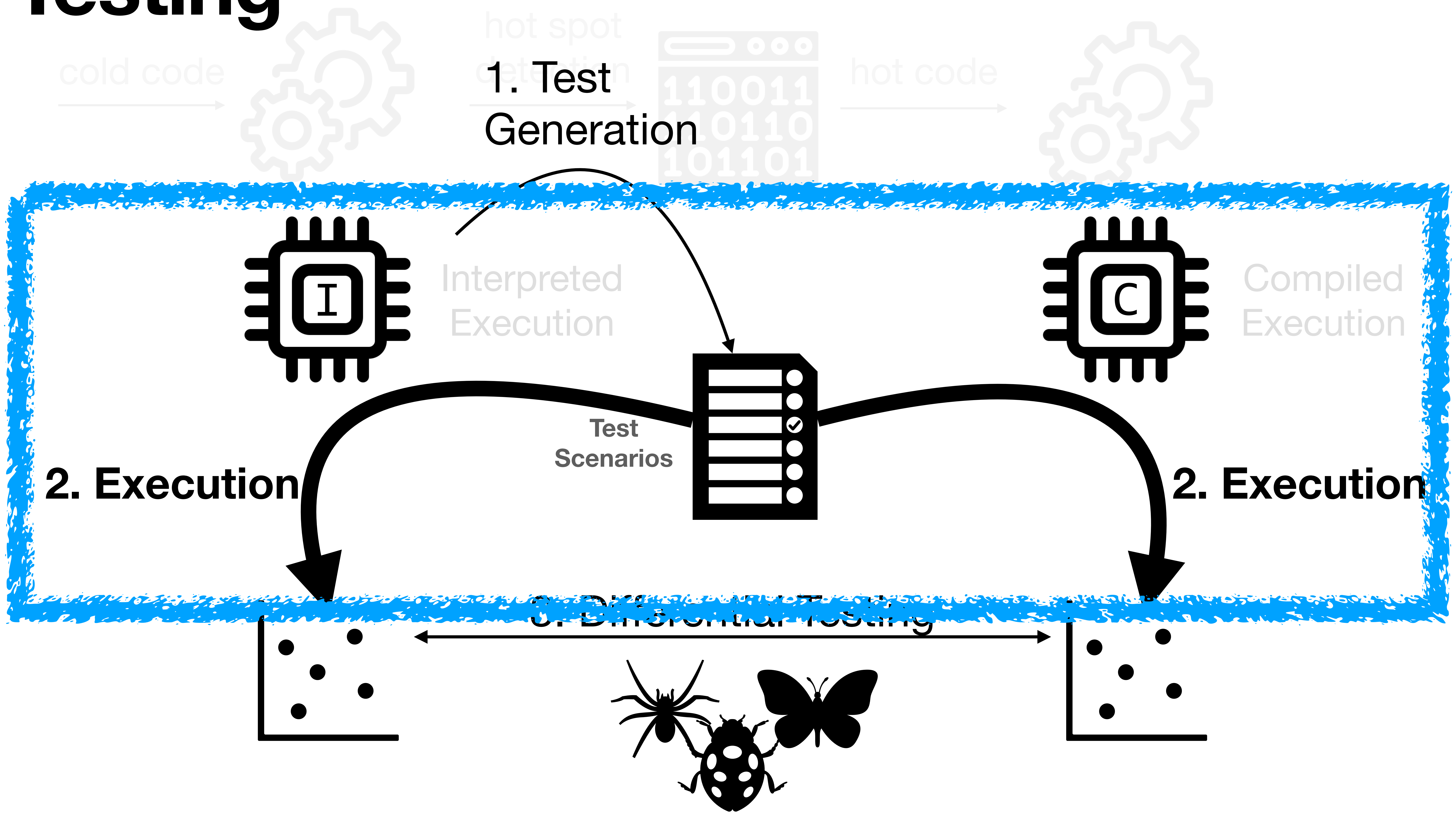
Interpreter-Guided Automatic JIT Compiler Unit Testing



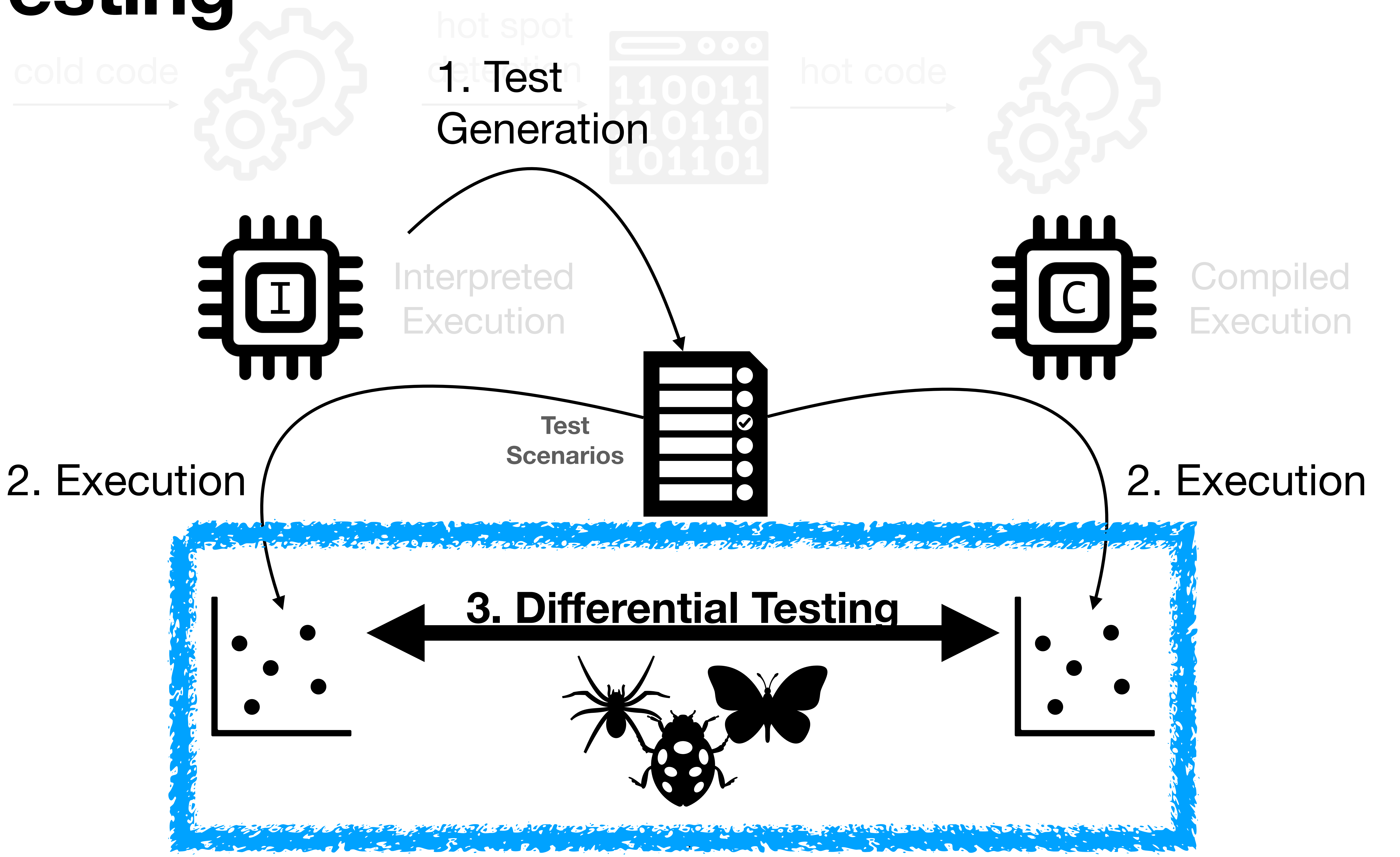
Interpreter-Guided Automatic JIT Compiler Unit Testing



Interpreter-Guided Automatic JIT Compiler Unit Testing



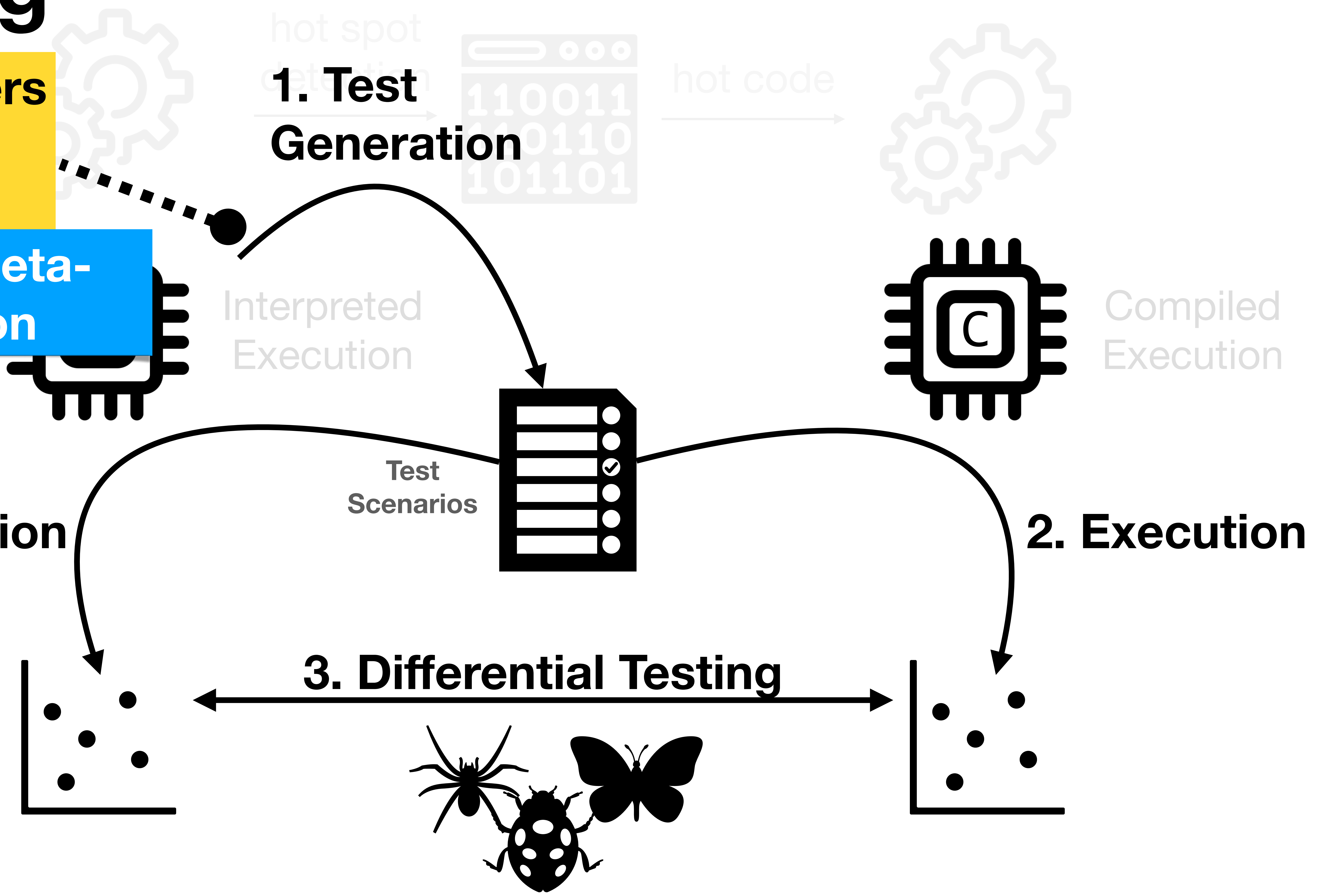
Interpreter-Guided Automatic JIT Compiler Unit Testing



Interpreter-Guided Automatic JIT Compiler Unit Testing

Insight 1: Interpreters are Executable Semantics

=> Concolic Meta-Interpretation



Interpreter are Executable Semantics

Pharo VM Example

```
Interpreter >> bytecodePrimAdd
  | rcvr arg result |
  rcvr := self internalStackValue: 1.
  arg := self internalStackValue: 0.
  (objectMemory areIntegers: rcvr and: arg) ifTrue: [
    result := (objectMemory integerValueOf: rcvr) + (
      objectMemory integerValueOf: arg).
    "Check for overflow"
    (objectMemory isIntegerValue: result) ifTrue: [
      self
        internalPop: 2
        thenPush: (objectMemory integerObjectOf: result).
      ^ self fetchNextBytecode "success"]].
  "Slow path, message send"
  self normalSend
```

If both operands are integers

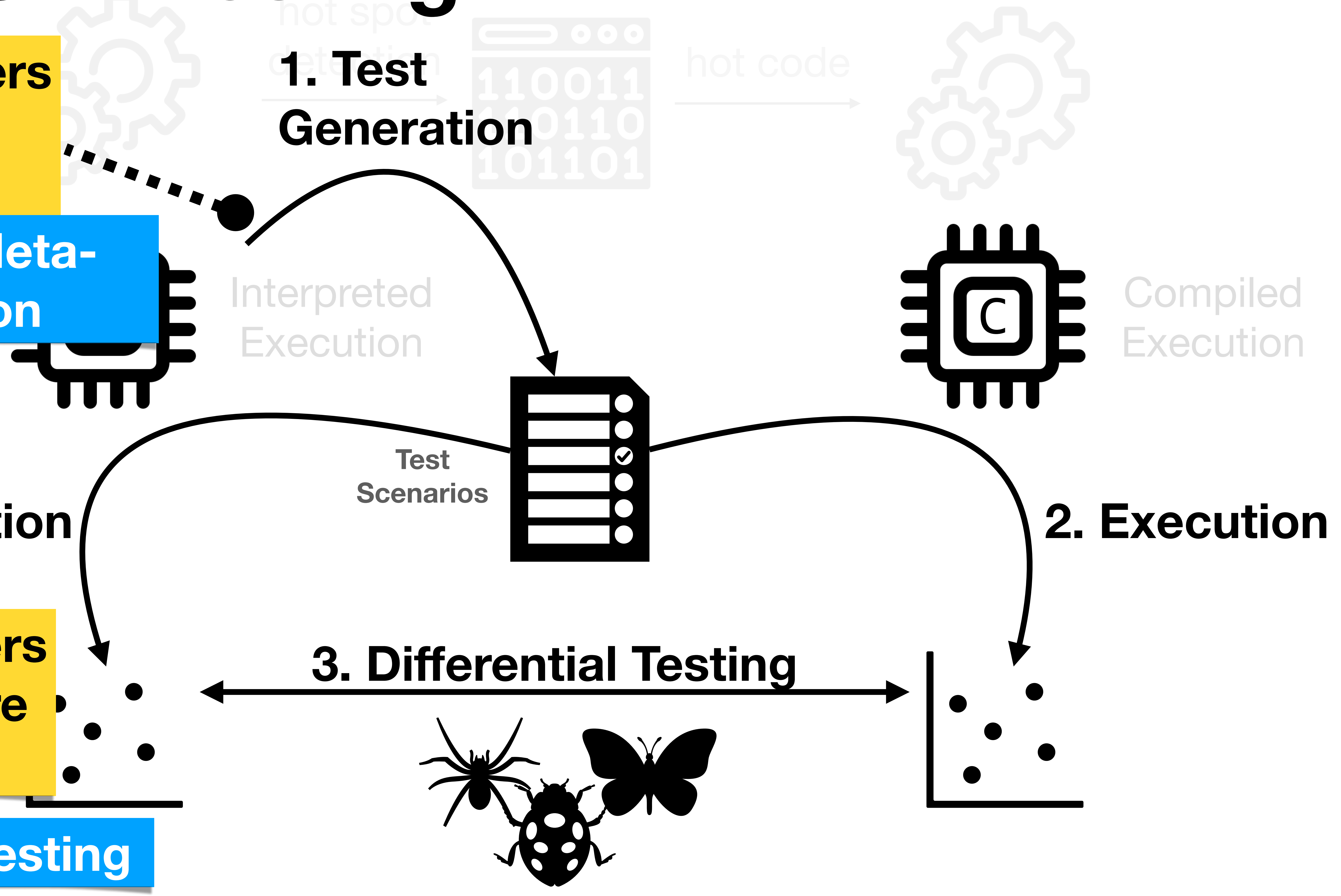
If their sum does not overflow

Else, slow path => message send

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

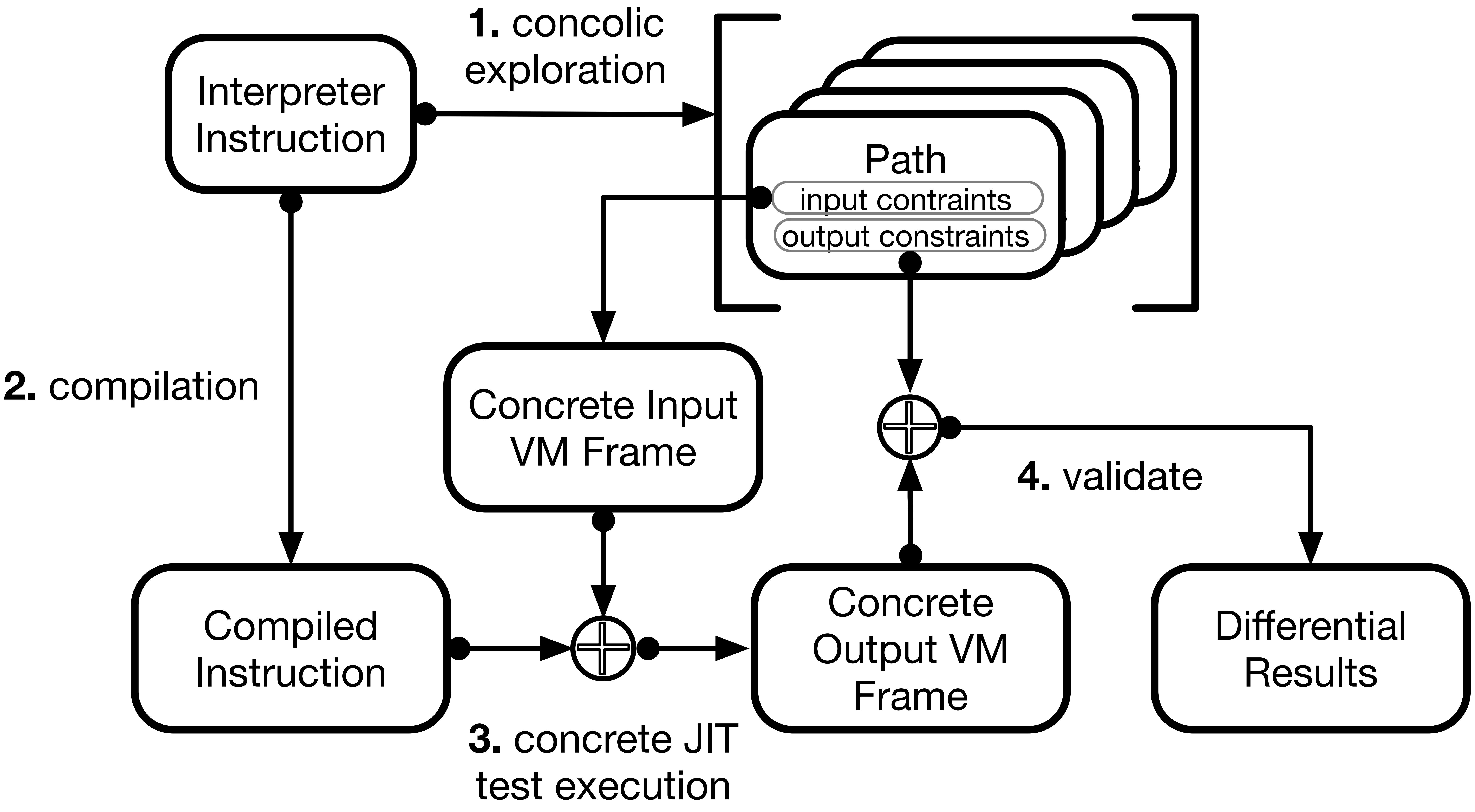
Interpreter VS Compiled Code

Pharo VM Example

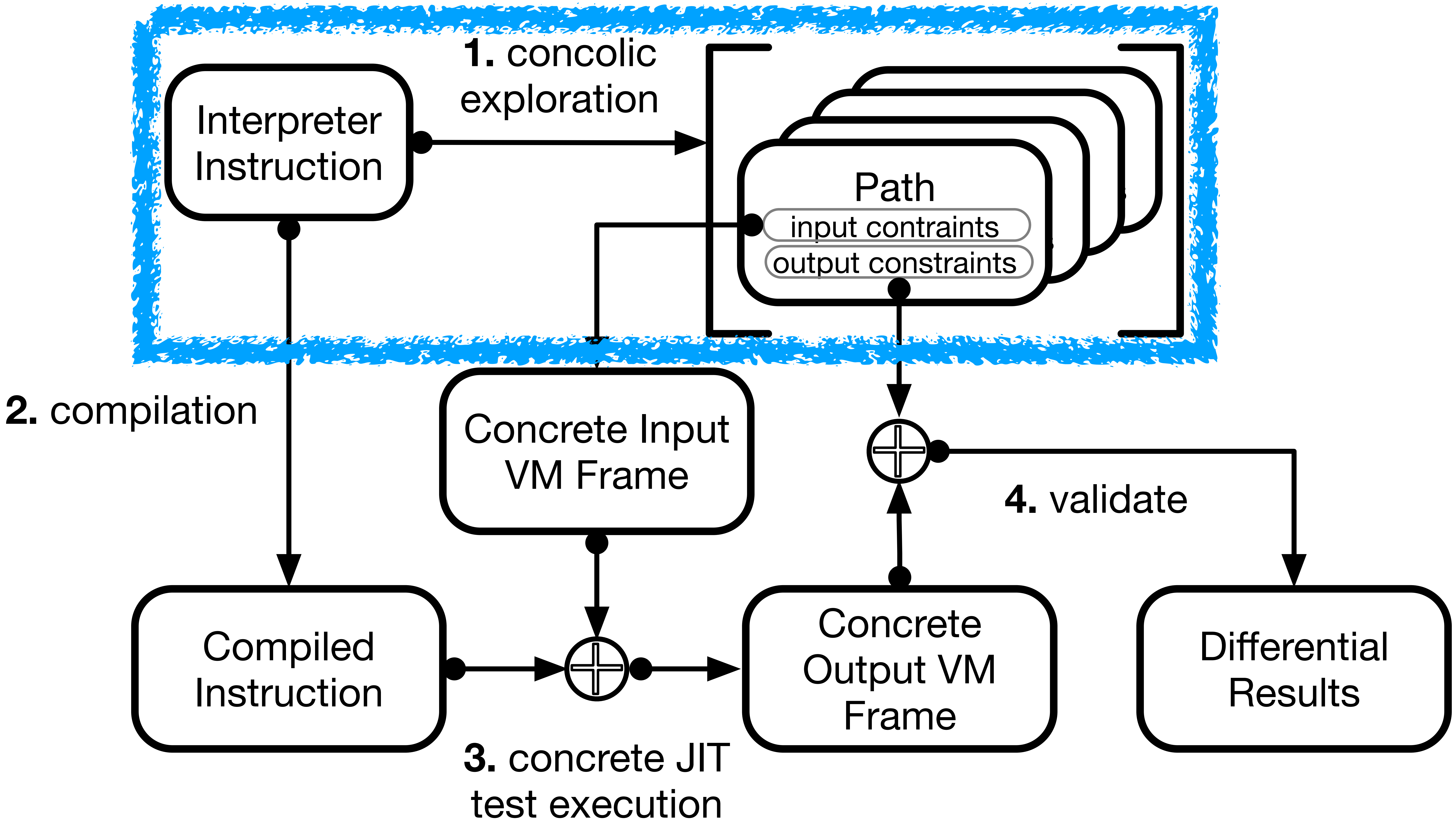
```
Interpreter >> bytecodePrimAdd
  | rcvr arg result |
  rcvr := self internalStackValue: 1.
  arg := self internalStackValue: 0.
  (objectMemory areIntegers: rcvr and: arg) ifTrue: [
    result := (objectMemory integerValueOf: rcvr) + (
      objectMemory integerValueOf: arg).
    "Check for overflow"
    (objectMemory isIntegerValue: result) ifTrue: [
      self
        internalPop: 2
        thenPush: (objectMemory integerObjectOf: result).
      ^ self fetchNextBytecode "success"]].
  "Slow path, message send"
  self normalSend
```

```
    ... # previous bytecode IR
    checkSmallInteger t0
    jumpzero notsmi
    checkSmallInteger t1
    jumpzero notsmi
    t2 := t0 + t1
    jumpIfNotOverflow continue
notsmi: #slow case first send
    t2 := send #+ t0 t1
continue:
    ... # following bytecode IR
```

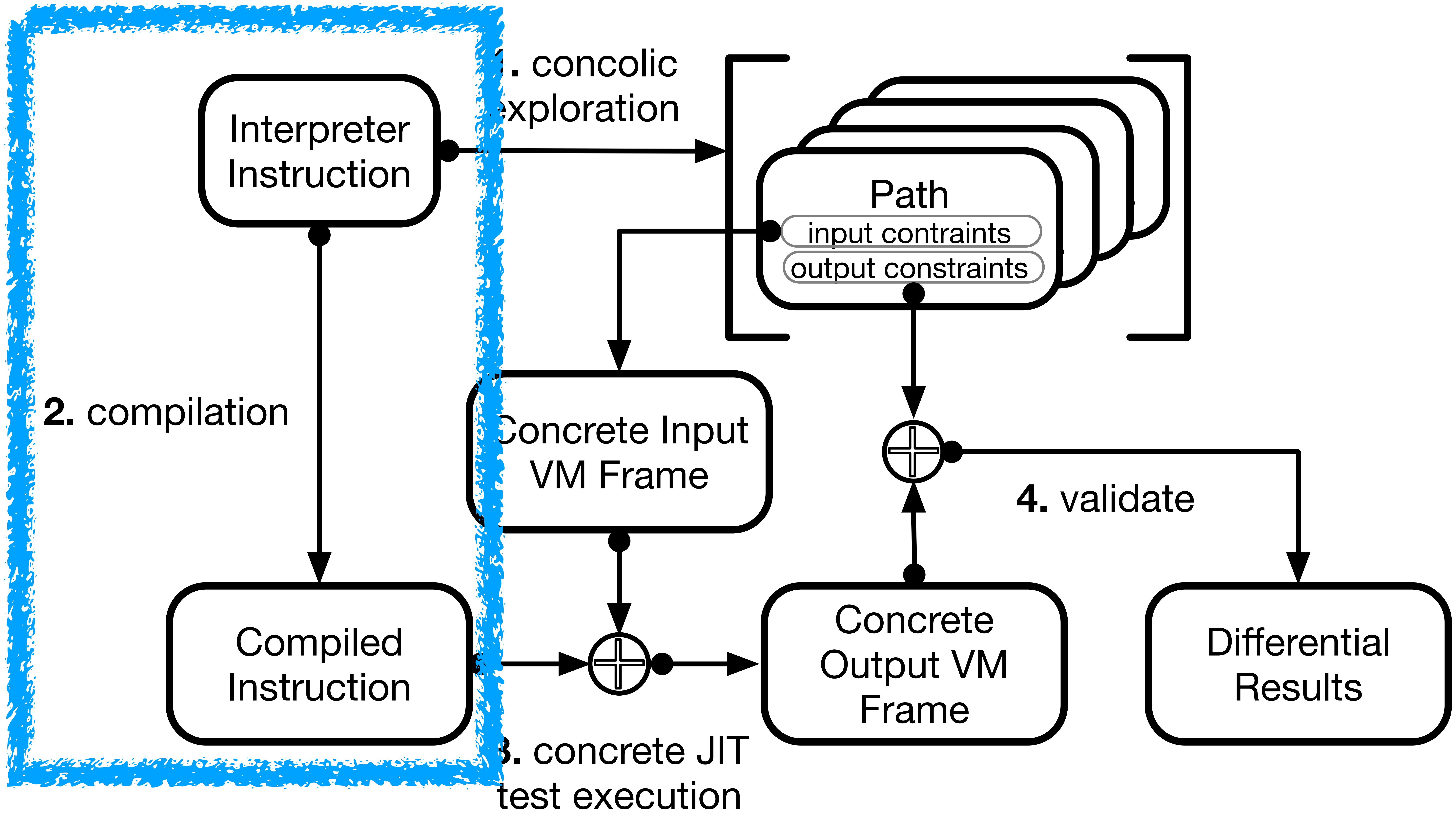
Implementation View



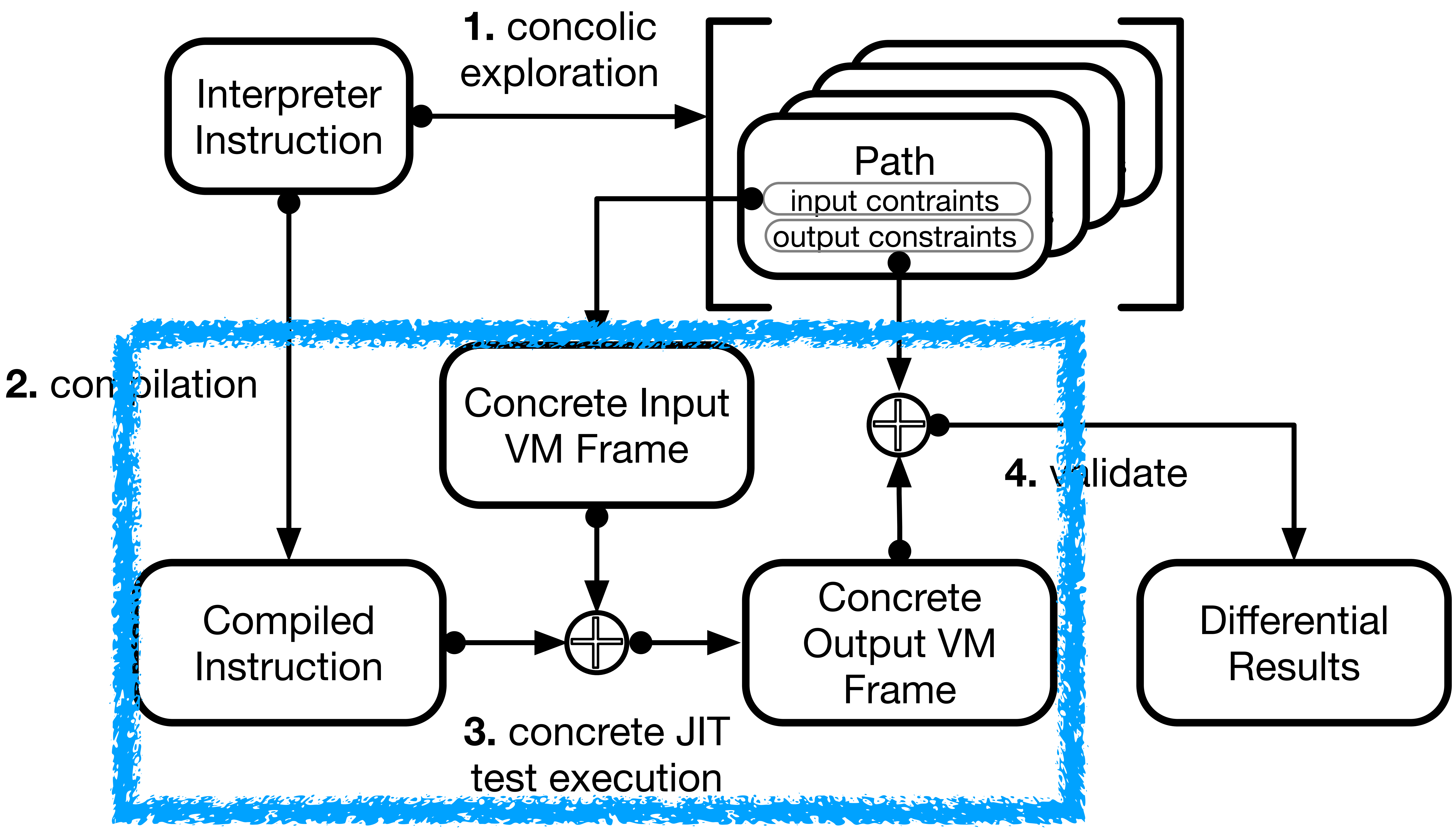
Implementation View



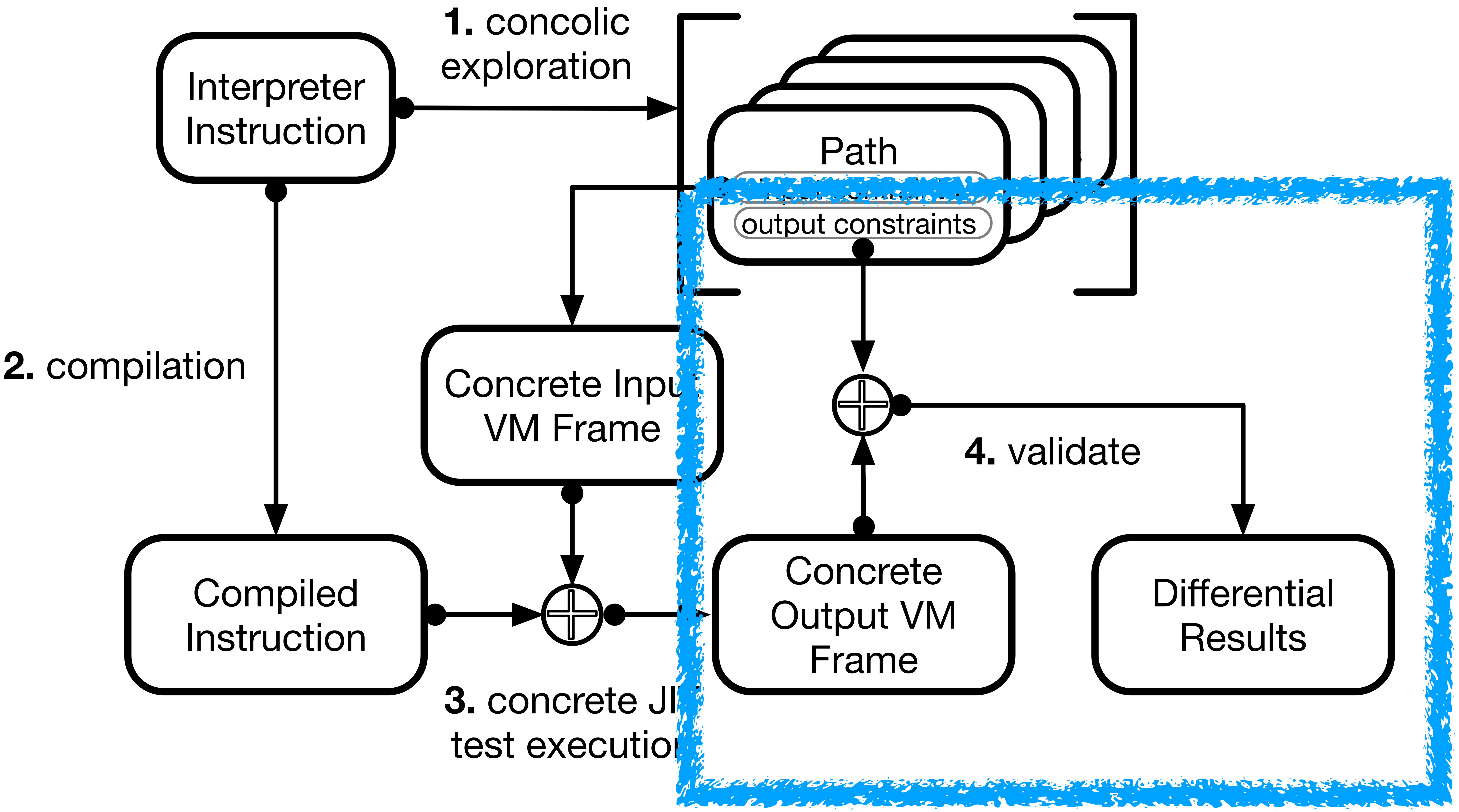
Implementation View



Implementation View

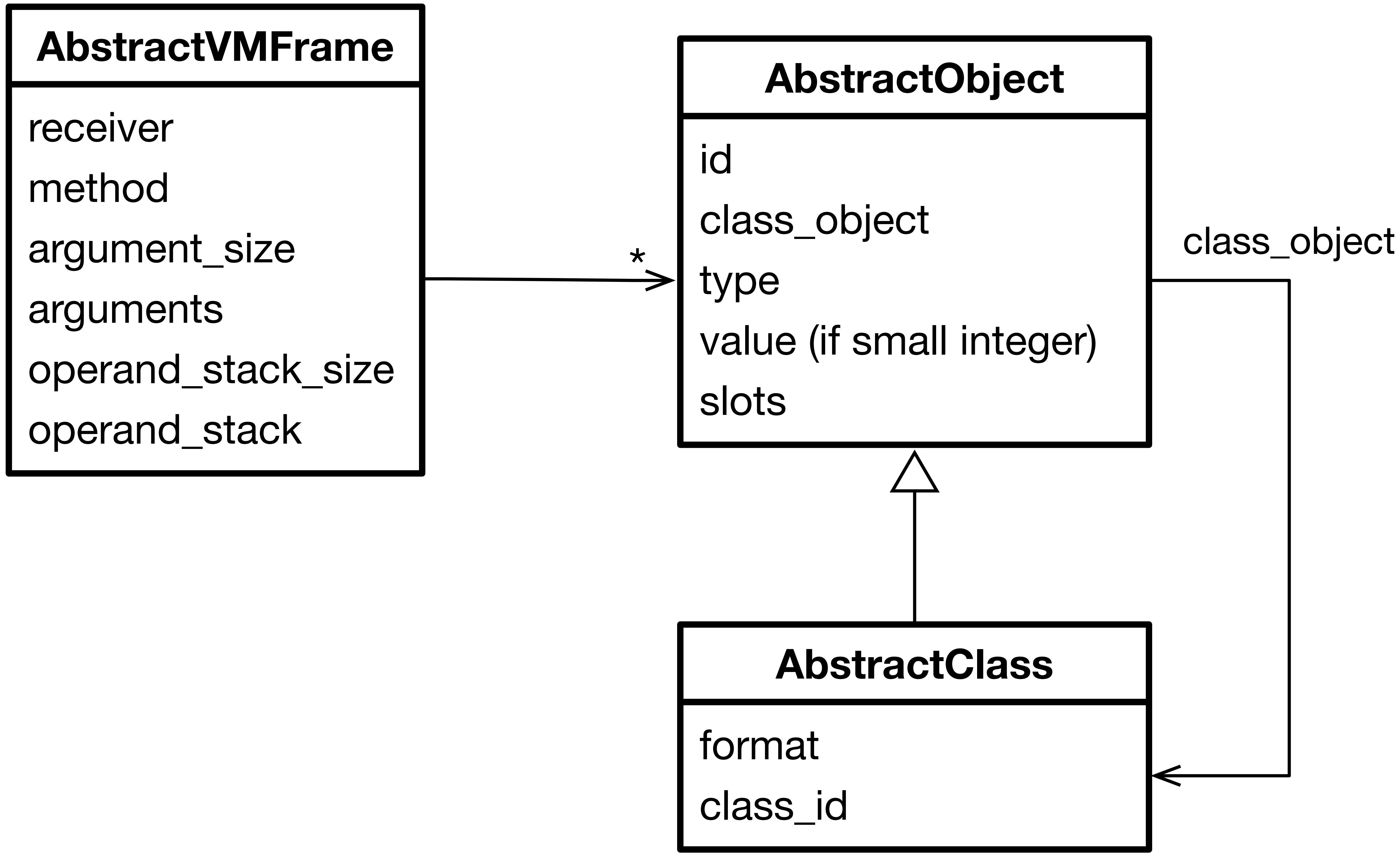


Implementation View



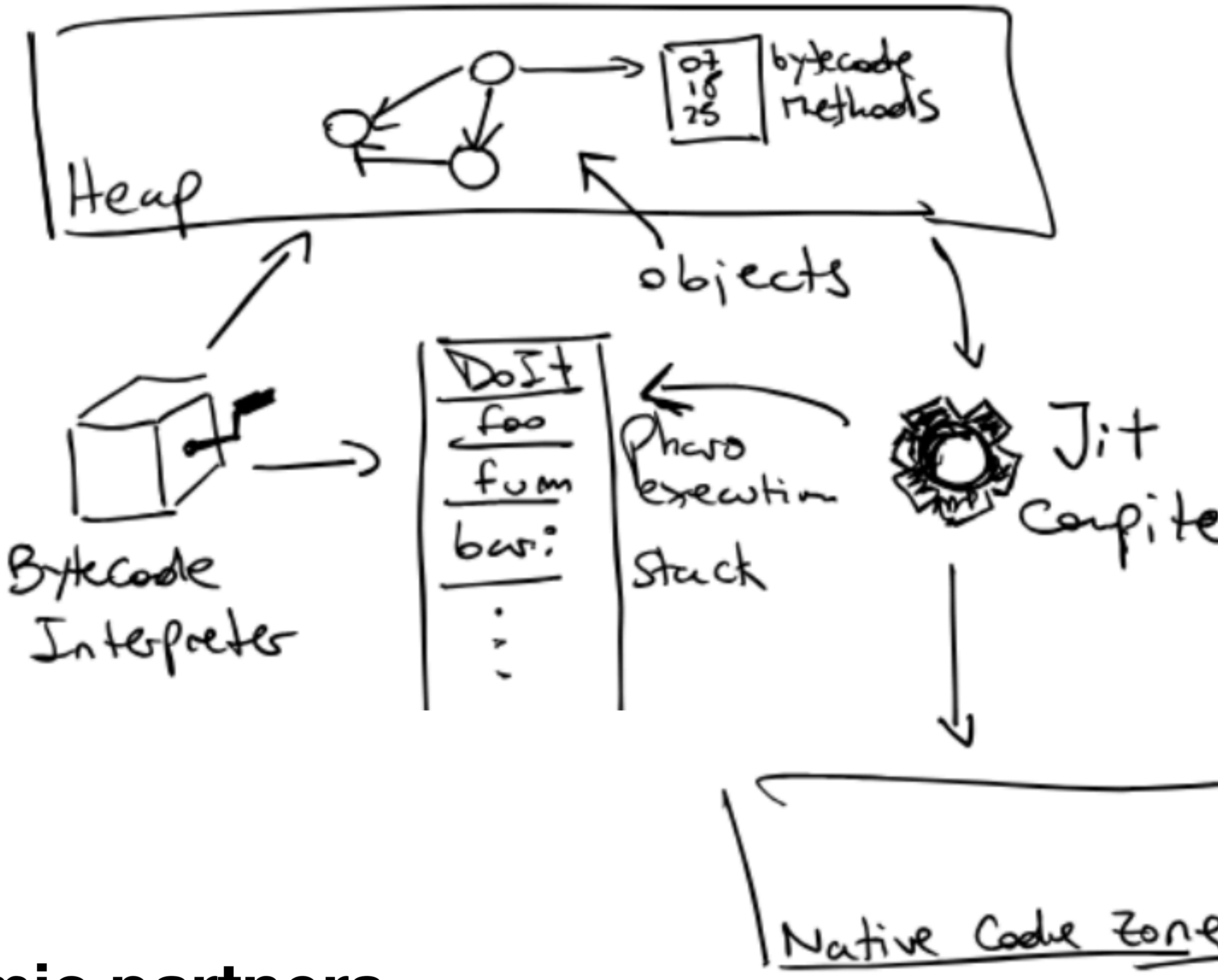
Concolic Meta-Interpretation Model

- Models VM behaviour during concolic execution
  - Frame
  - Objects + types
  - Classes
- Then flattened into SAT solver equations



Experimental Context: The Pharo VM

- Interpreted-compiled mixed execution
- Some numbers:
  - 255 stack based bytecodes
  - ~340 primitives/native methods
  - 146 different IR instructions
  - x86, x86-64, ARMv7, ARMv8, RISC-V
- Industrial consortium:
  - **28 International companies, 26 academic partners**



Previous Manual Testing Effort

- No useful unit tests by ~06/2020
- Large manual testing effort during 2020 while porting to ARM64bits
  - Extended VM simulation with a (TDD compatible) unit testing infrastructure
  - **450+** written tests on the interpreter and the garbage collector*
  - **580+** written tests on the JIT compiler*
  - Parametrisable for 32 and 64bits, **ARM32, ARM64, x86, x86-64**

* Numbers by 05/2021

Cross-ISA Testing of the Pharo VM. Lessons learned while porting to ARMv8 64bits. Polito et al. **MPLR'21**

Evaluation

- 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

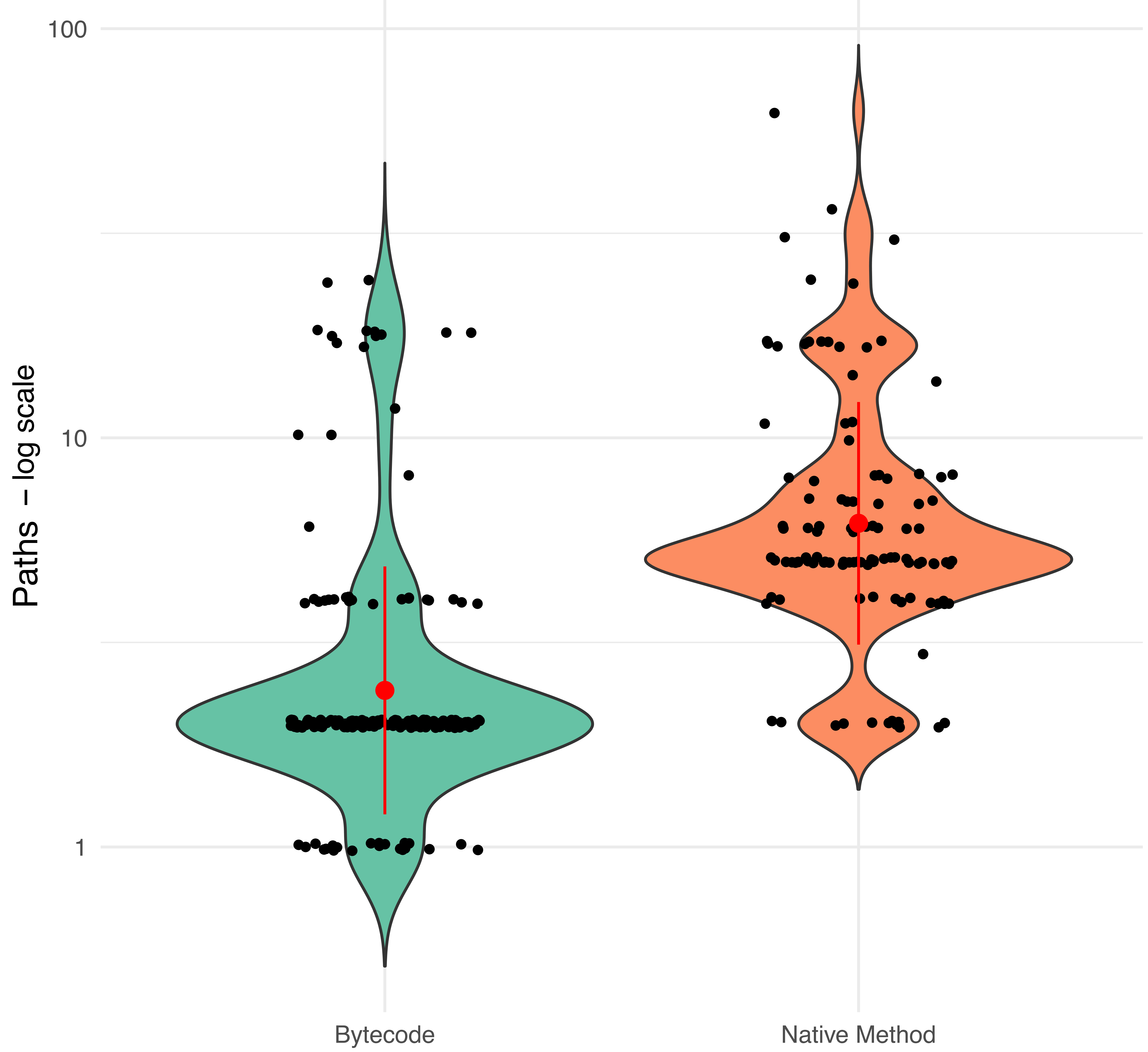
Characterising Concolic Execution

Paths per instruction

- Native methods present in average more paths than bytecode instructions

  => longer time to explore

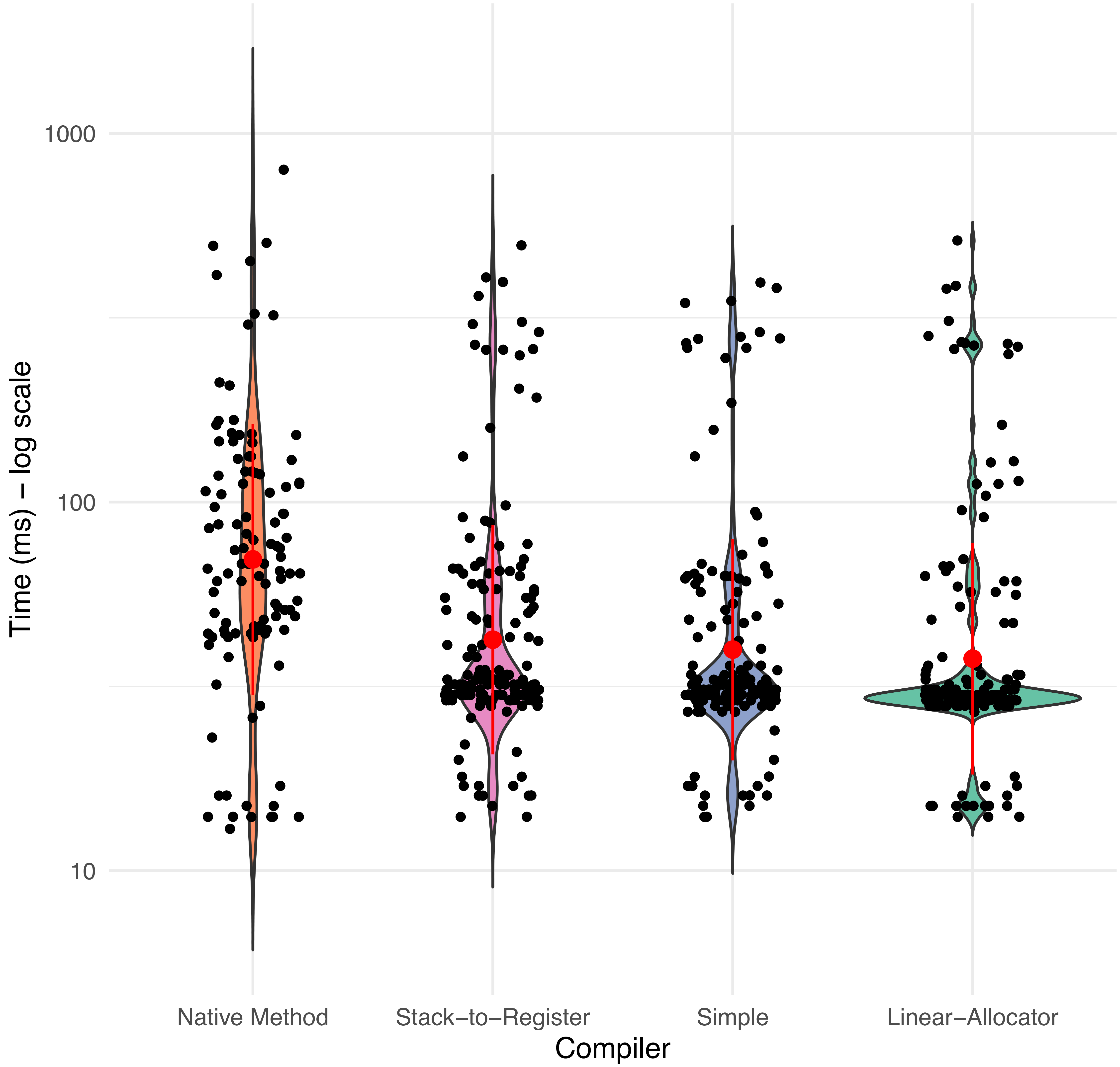
  => potentially more bugs



Paths per Type of Instruction

Practical and Cheap

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



More in the article!

- Discovered Bugs
- Concolic Model
- Testing Infrastructure

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 Lill
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 equivalent. Tests should be duplicated for each execution engine, exercising the same execution paths on each of them.

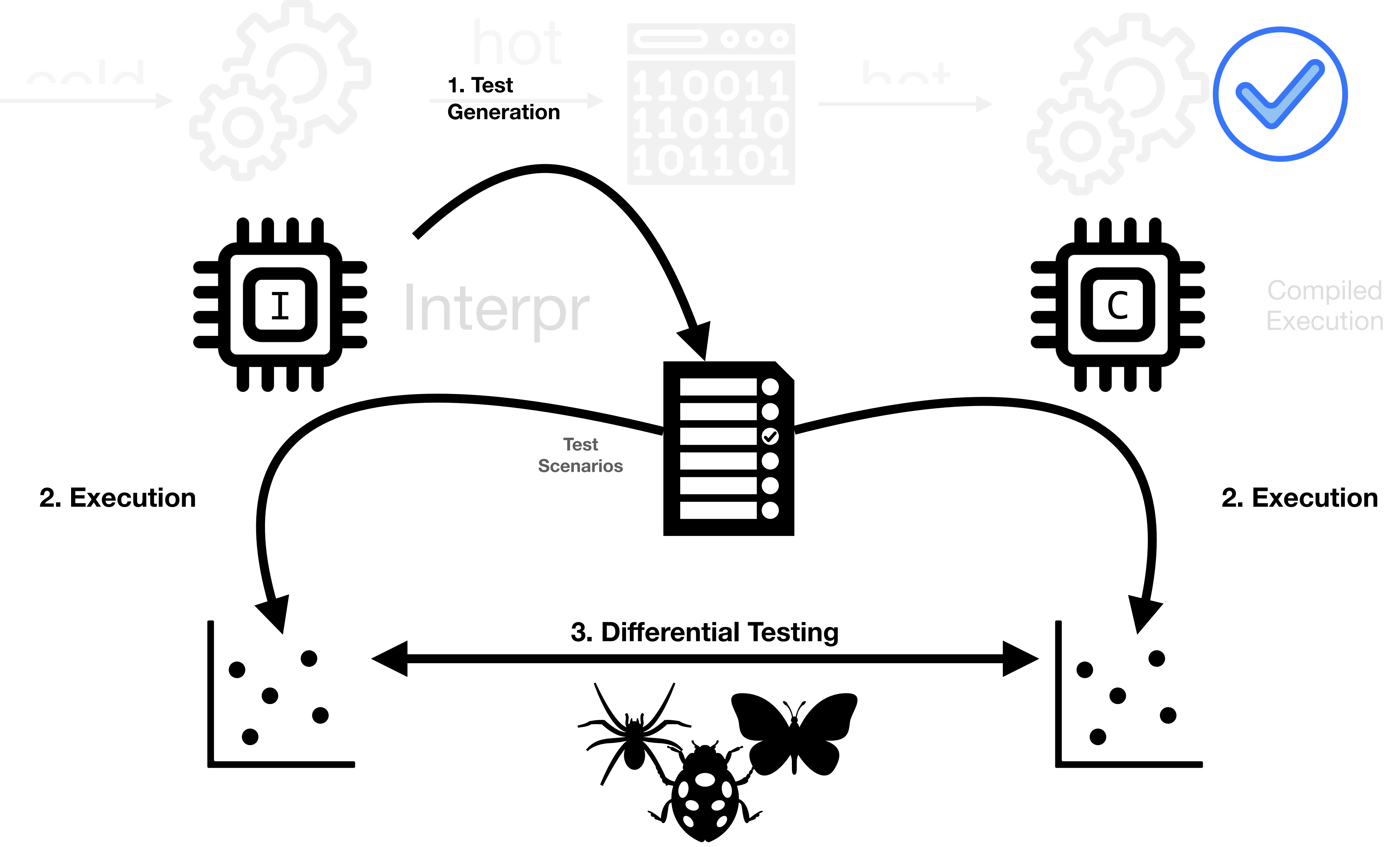
San Diego, CA, USA. ACM, New York, NY, USA, 12 pages. h
//doi.org/10.1145/3519939.3523457

1 Introduction

Modern Virtual Machines support code generation fo
compilation and dynamic code patching for techniques
as inline caching. They are often structured around a b
code interpreter, a baseline JIT compiler, and a specul

Conclusion

- 478 differences found, 91 causes, 6 categories
- Practical:
  - 4928 tests generated in ~8 minutes
  - 4928 tests run in ~40 seconds



Guille Polito - Pablo Tesone - Stéphane Ducasse
guillermo.polito@univ-lille.fr
@guillep

Extras

Concolic Testing through Meta-interpretation

- Idea: Guide test generation by looking at the implementation

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

Different cases
if x > 100 or <= 100!!

Different cases
if x = 1023 or != 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?

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

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)

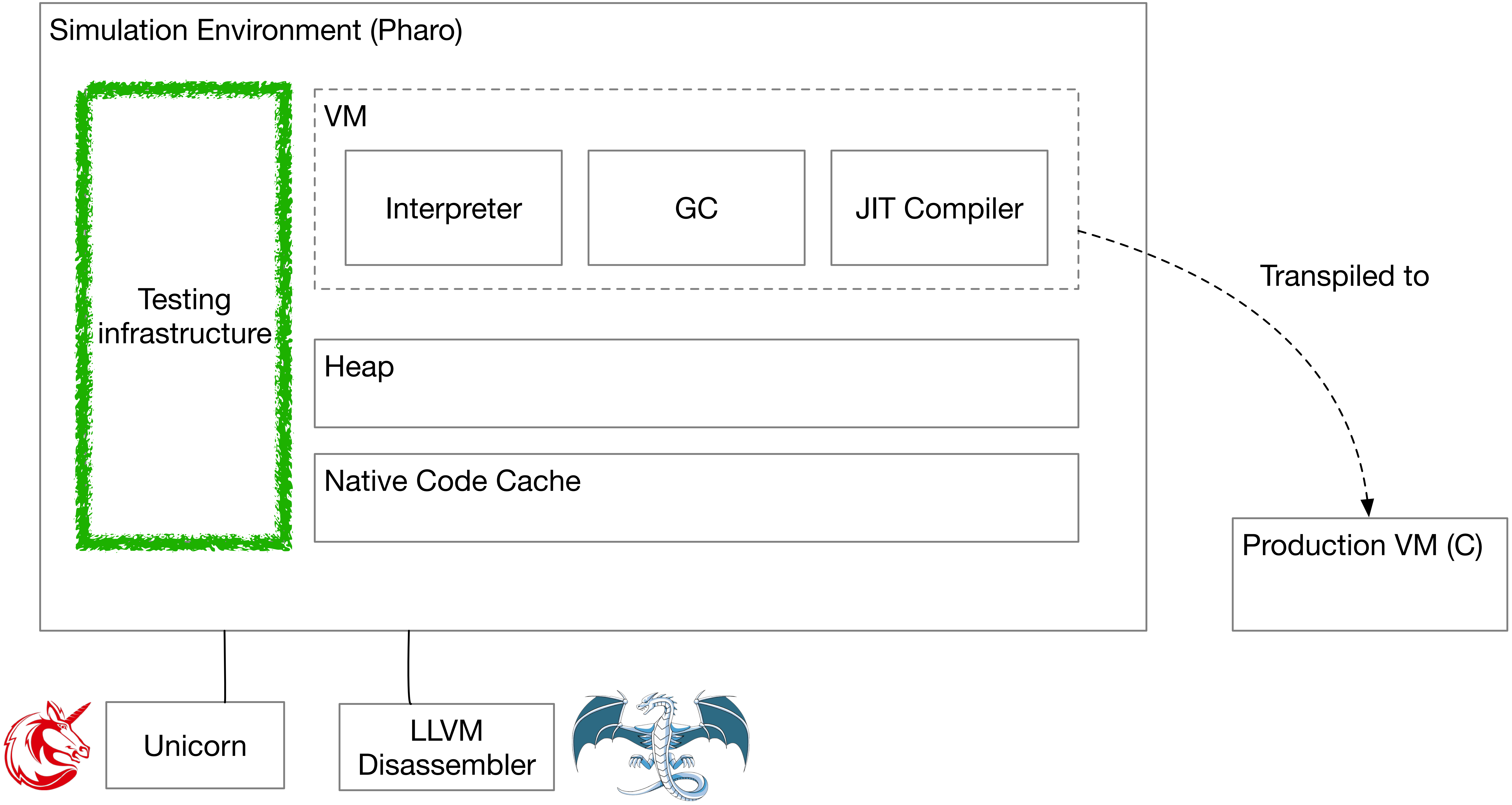
```
Interpreter >> bytecodePrimAdd
  | rcvr arg result |
  rcvr := self internalStackValue: 1.
  arg := self internalStackValue: 0.
  (objectMemory areIntegers: rcvr and: arg) ifTrue: [
    result := (objectMemory integerValueOf: rcvr) + (
      objectMemory integerValueOf: arg).
    "Check for overflow"
    (objectMemory isIntegerValue: result) ifTrue: [
      self
        internalPop: 2
        thenPush: (objectMemory integerObjectOf: result).
      ^ self fetchNextBytecode "success"]].
  "Slow path, message send"
  self normalSend
```

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

```
  ... # previous bytecode IR
        checkSmallInteger t0
        jumpzero notsmi
        checkSmallInteger t1
        jumpzero notsmi
        t2 := t0 + t1
        jumpIfNotOverflow continue
notsmi: #slow case first send
        t2 := send #+ t0 t1
continue:
  ... # following bytecode IR
```

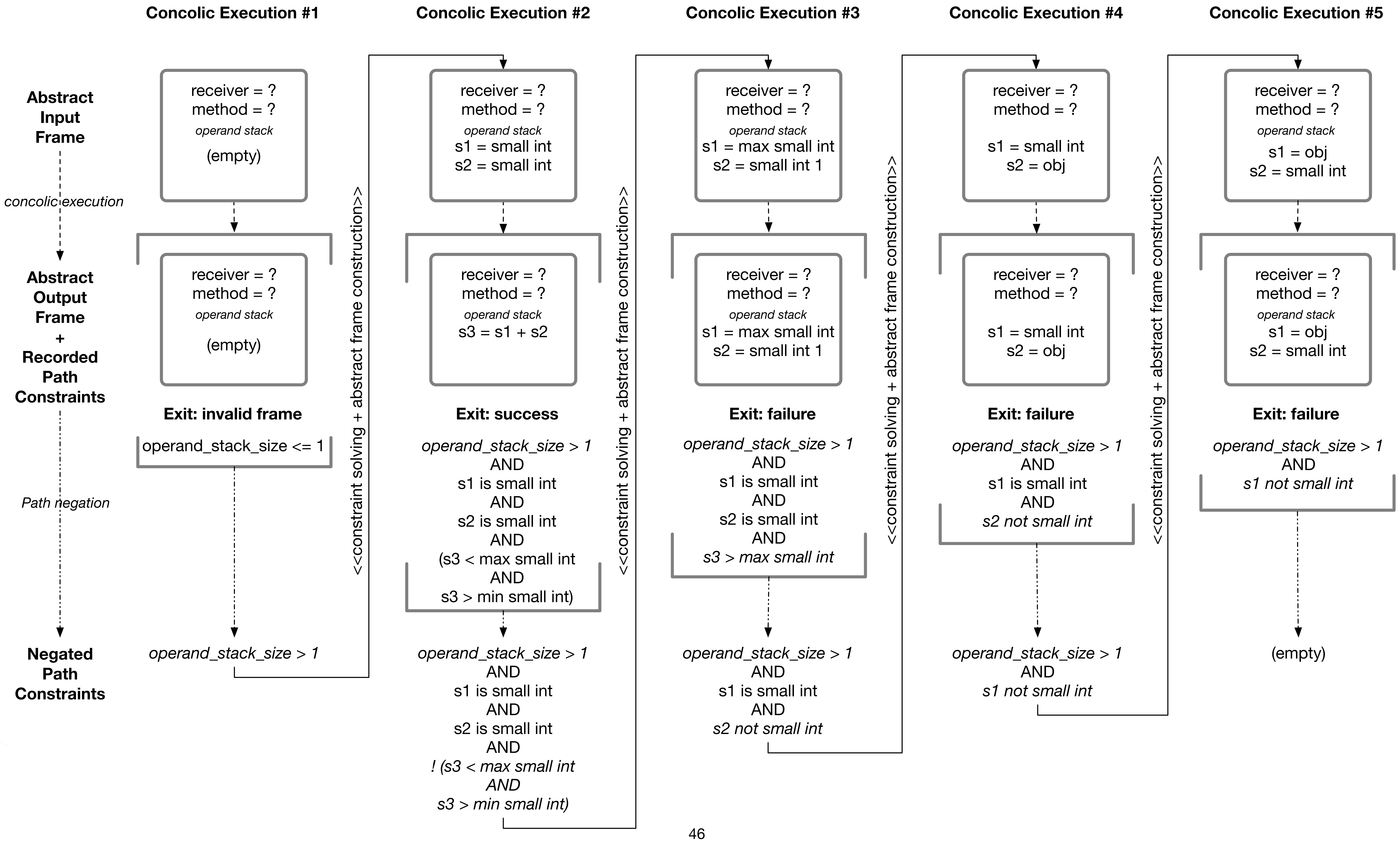
Listing 2. Illustration of the Intermediate Representation instructions created when compiling the byte-code instruction in Listing 1.

Simulation + Testing Environment



Unit Testing Infrastructure Comparison

	Real Hardware Execution	Full-System Simulation	Unit-Testing
Feedback-cycle speed	Very low	Low	High
Availability	Low	High	High
Reproducibility	Low	Low	High
Precision	High	Low	Low
Debuggability	Low	High	High


Concolic Execution #1
Concolic Execution #2
Concolic Execution #3
Concolic Execution #4
Concolic Execution #5
Abstract Input Frame
concolic execution
Abstract Output Frame + Recorded Path Constraints
Path negation
Negated Path Constraints
receiver = ? method = ? operand stack (empty)
receiver = ? method = ? operand stack (empty)
Exit: invalid frame
operand_stack_size <= 1
operand_stack_size > 1
<<constraint solving + abstract frame construction>>
receiver = ? method = ? operand stack s1 = small int s2 = small int
receiver = ? method = ? operand stack s3 = s1 + s2
Exit: success
operand_stack_size > 1 AND s1 is small int AND s2 is small int AND (s3 < max small int AND s3 > min small int)
operand_stack_size > 1 AND s1 is small int AND s2 is small int AND ! (s3 < max small int AND s3 > min small int)
<<constraint solving + abstract frame construction>>
receiver = ? method = ? operand stack s1 = max small int s2 = small int 1
receiver = ? method = ? operand stack s1 = max small int s2 = small int 1
Exit: failure
operand_stack_size > 1 AND s1 is small int AND s2 is small int AND s3 > max small int
operand_stack_size > 1 AND s1 is small int AND s2 not small int
<<constraint solving + abstract frame construction>>
receiver = ? method = ? s1 = small int s2 = obj
receiver = ? method = ? s1 = small int s2 = obj
Exit: failure
operand_stack_size > 1 AND s1 is small int AND s2 not small int
operand_stack_size > 1 AND s1 not small int
<<constraint solving + abstract frame construction>>
receiver = ? method = ? operand stack s1 = obj s2 = small int
receiver = ? method = ? operand stack s1 = obj s2 = small int
Exit: failure
operand_stack_size > 1 AND s1 not small int
(empty)