
ORACLE®

Dynamic Symbolic Execution for Object-Oriented Libraries

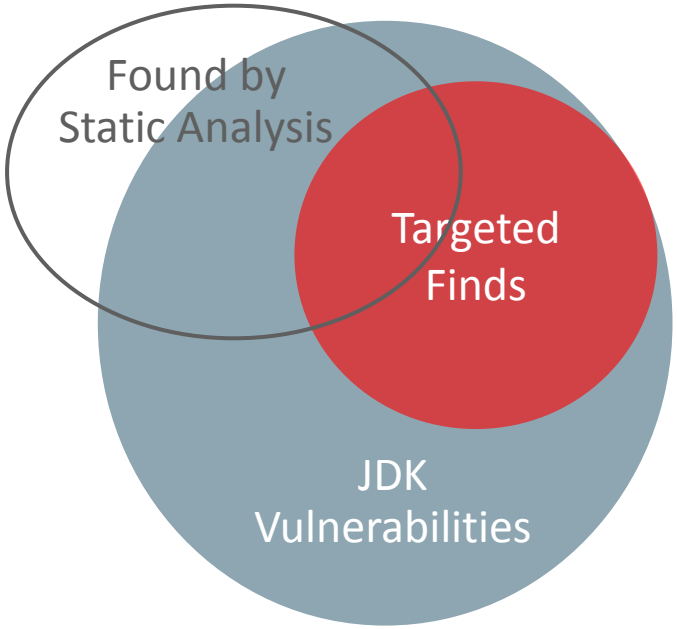
Lian Li and Andrew Santosa
Oracle Labs Australia
November 2014

The following is intended to provide some insight into a line of research in Oracle Labs. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. Oracle reserves the right to alter its development plans and practices at any time, and the development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.

Why Fuzzing?

Hybrid symbolic-analysis exercising vulnerable execution traces

- Address static analysis limitations
  - False positives due to over approximation
  - Complex language aspects: Exception, aliasing
  - Dynamic features of Java: class loading, reflection
- Test generation based on symbolic execution
  - Exercise vulnerable execution traces



Dynamic Symbolic Execution

- Symbolic Execution
  - Program inputs represented as symbols
  - Instructions interpreted as constraints on symbols
  - Solve constraints to generate tests exercising program path

- Dynamic symbolic execution (Con-colic execution)
  - Perform symbolic execution at runtime, together with concrete execution
  - Program path induced by concrete execution
  - Mutate path condition to explore different paths

Challenges in JDK Symbolic Execution

- The complexity of JDK
  - Native library
  - Reflection
  - Exception handling
- Symbolic execution of method invocations
  - Mutation of invocation targets
  - Critical for testing object-oriented libraries and for finding JDK vulnerabilities

Symbolic Execution Example

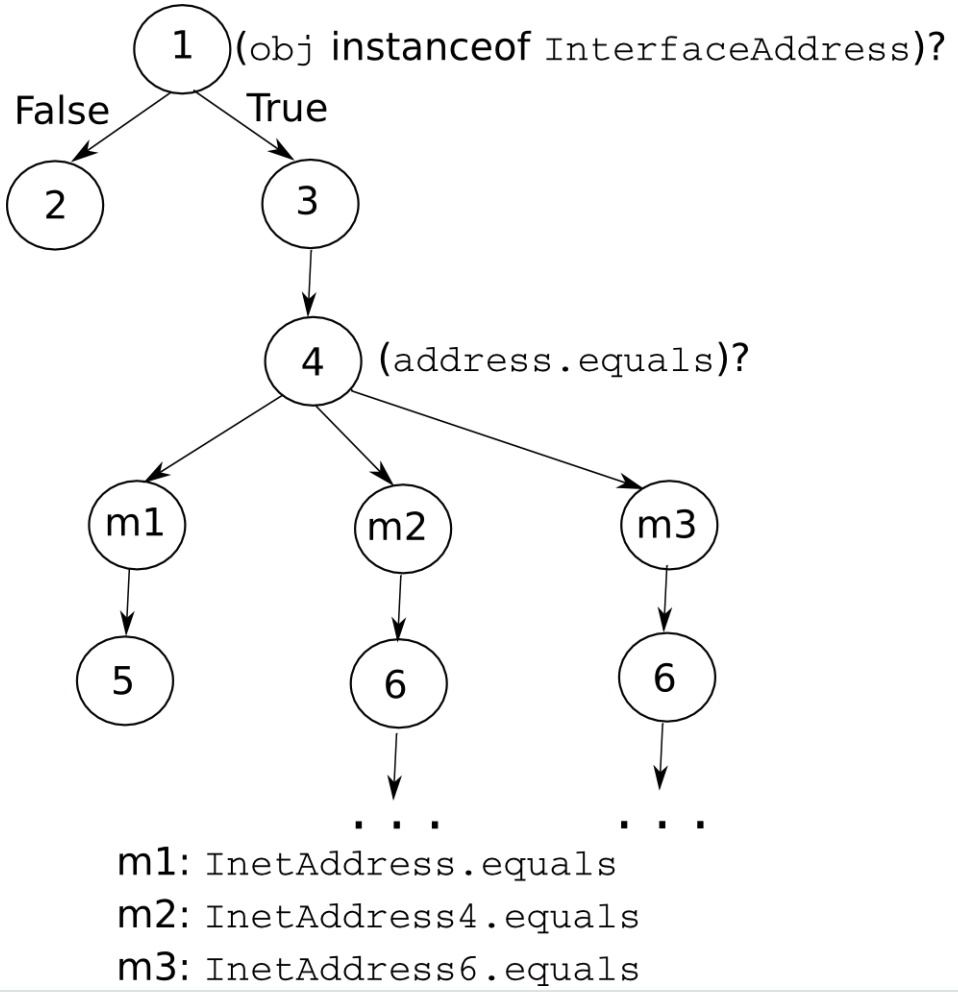
```
public class InterfaceAddress {
    private InetAddress address;
    private InetAddress broadcast;
    . . .
    public boolean equals (Object obj) {
1       if (!(obj instanceof InterfaceAddress))
2          return false;
3       InterfaceAddress cmp = (Interface) obj;
4       if (!address.equals(cmp.address))
5           return false;
6        if (!broadcast.equals(cmp.broadcast))
7           return false;
        . . .
   }
};
```

Null Pointer Exception

Bug 6628576 in
http://bugs.java.com

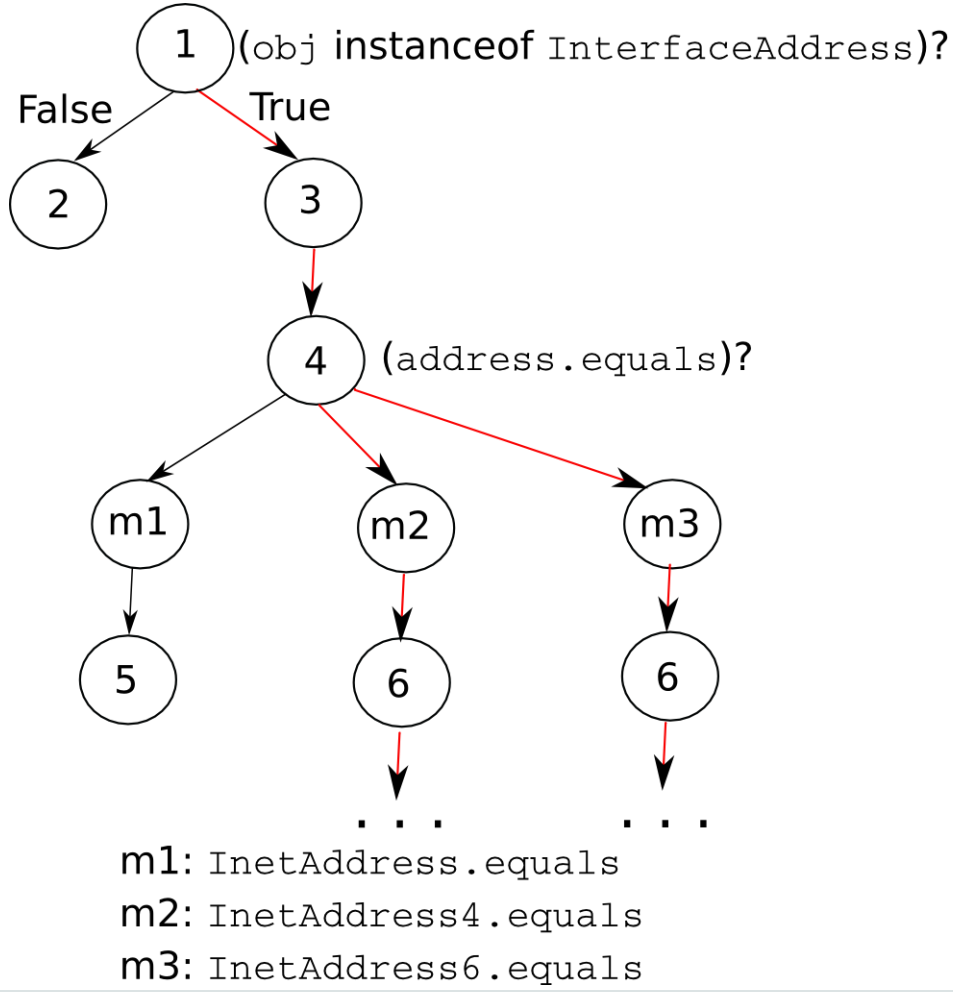
Symbolic Execution Example

```
public class InterfaceAddress {
    private InetAddress address;
    private InetAddress broadcast;
    . . .
    public boolean equals (Object obj) {
1       if (!(obj instanceof InterfaceAddress))
2          return false;
3       InterfaceAddress cmp = (Interface) obj;
4       if (!address.equals(cmp.address))
5           return false;
6        if (!broadcast.equals(cmp.broadcast))
7           return false;
        . . .
   }
};
```



Symbolic Execution Example

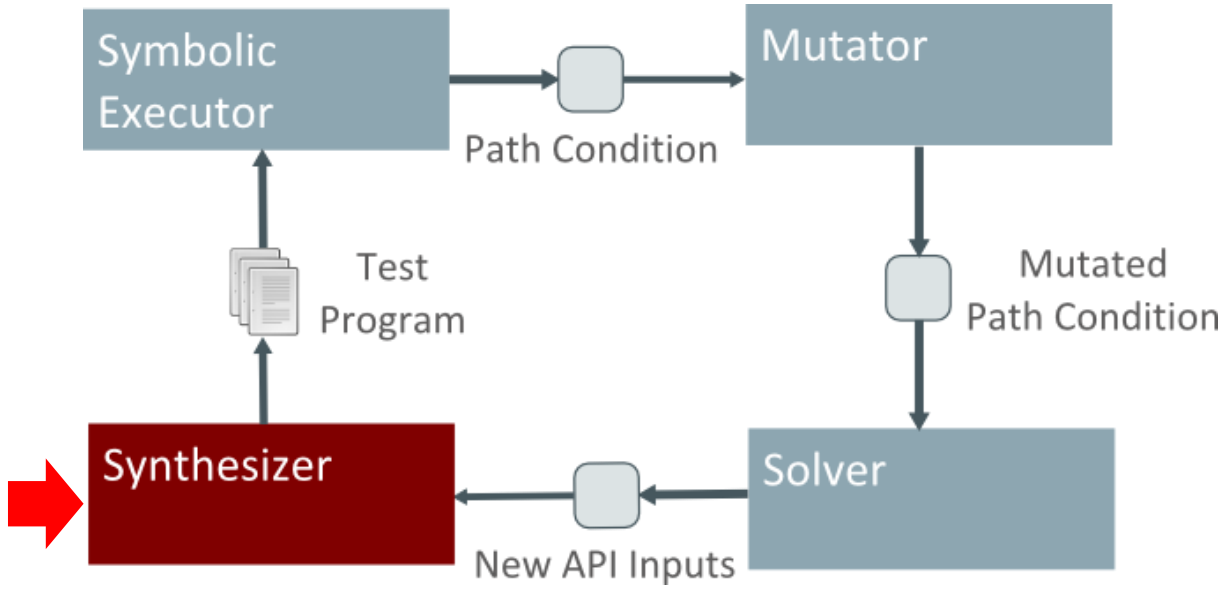
- Mutating types to explore invocation targets
  - `obj` of type `InterfaceAddress`
  - `address` of type `InetAddress4` or `InetAddress6`



Infrastructure for JDK Symbolic Execution

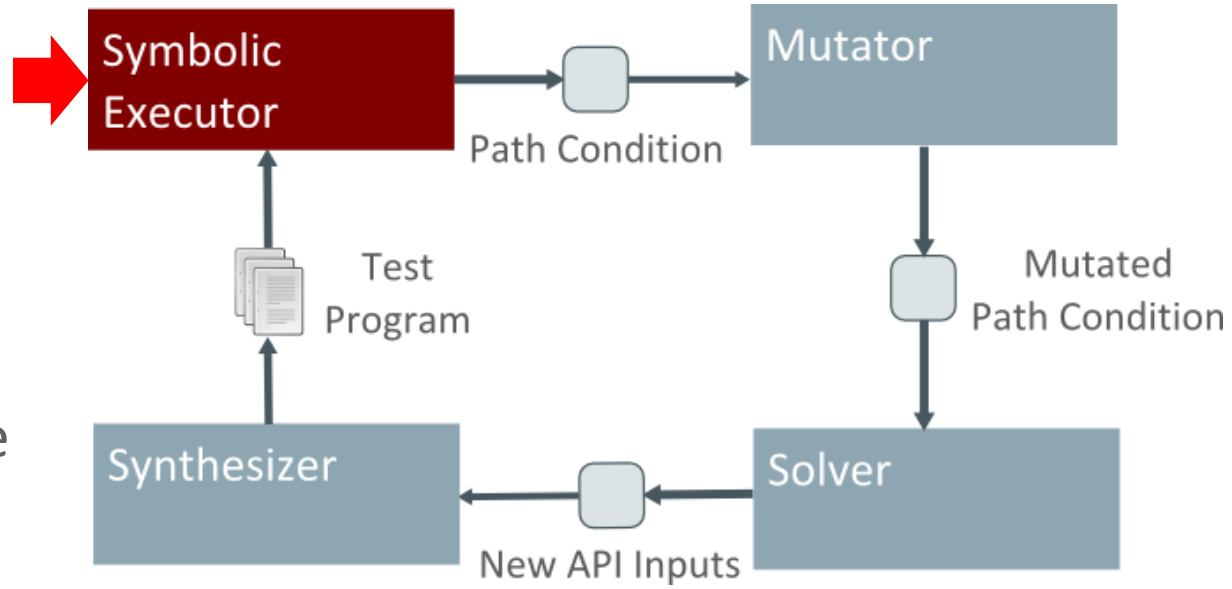
Starting point: Synthesizer

- Synthesizes test programs to invoke JDK API



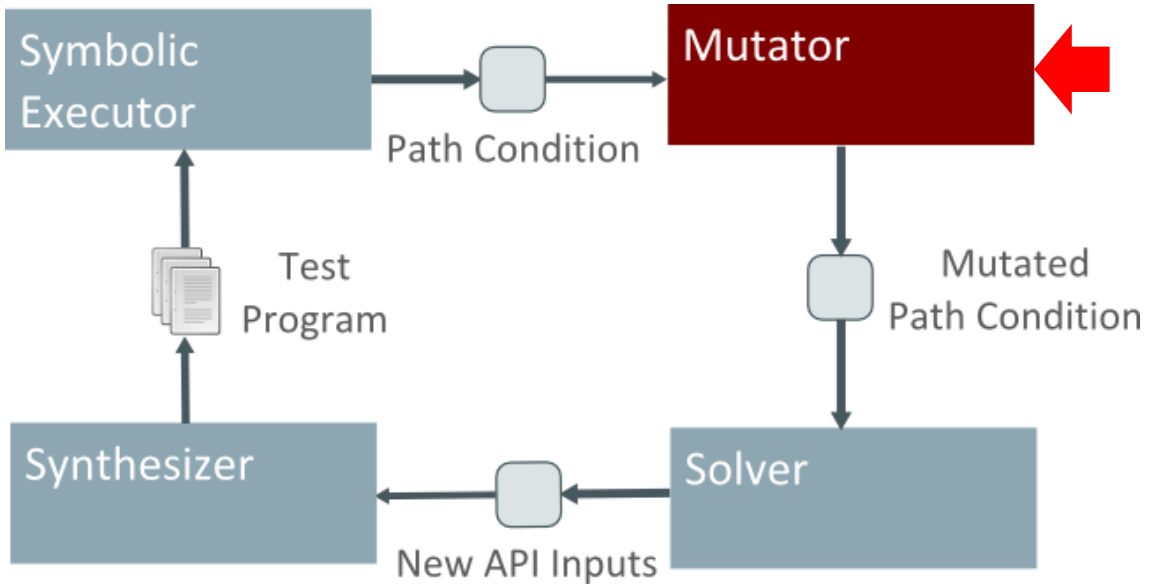
The Symbolic Executor

- Synthesizes test programs to invoke JDK API
- Symbolic execution of synthesized programs
  - Symbolic types for objects referenced by parameters of reference types
  - Generate path condition to represent the execution trace



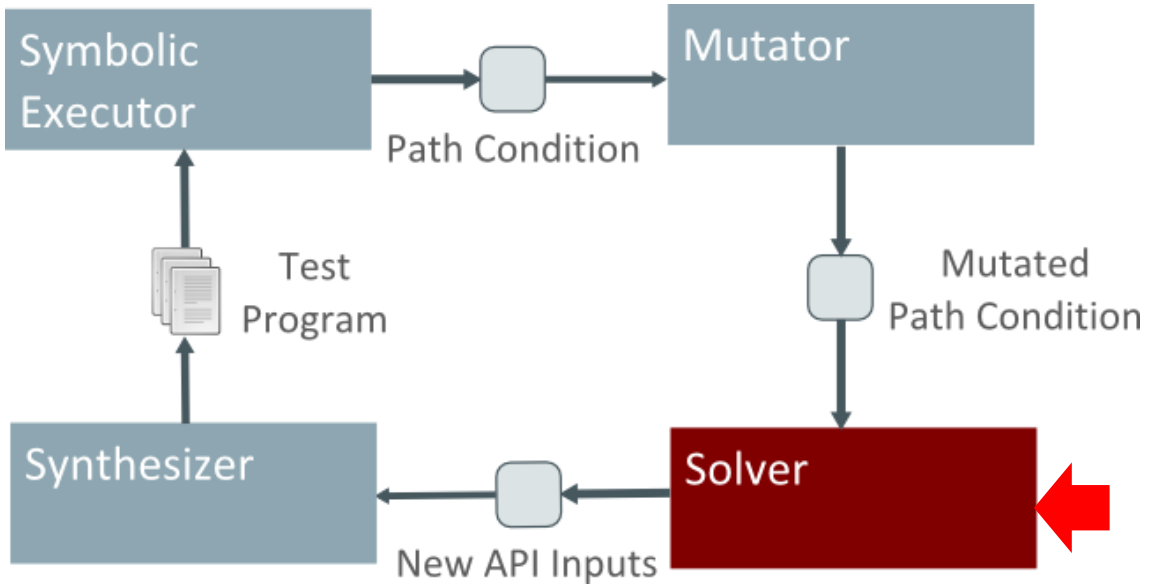
The Mutator

- Synthesizes test programs to invoke JDK API
- Symbolic execution of synthesized programs
- Mutates path conditions to explore new execution paths



The Solver

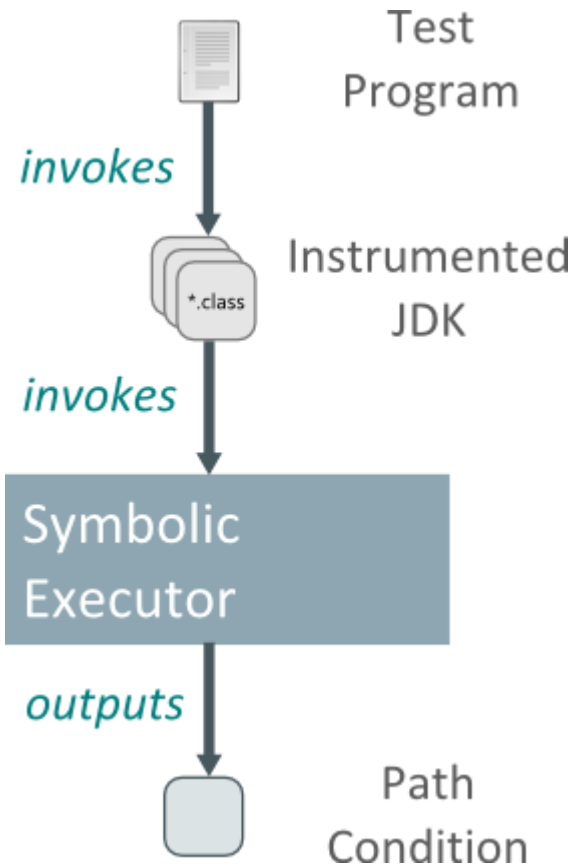
- Synthesizes test programs to invoke JDK API
- Symbolic execution of synthesized programs
- Mutates path conditions to explore new execution paths
- Solves mutated path conditions to generate new tests



Core Component: The Symbolic Executor

Dynamic Symbolic Execution of JDK via Instrumentation

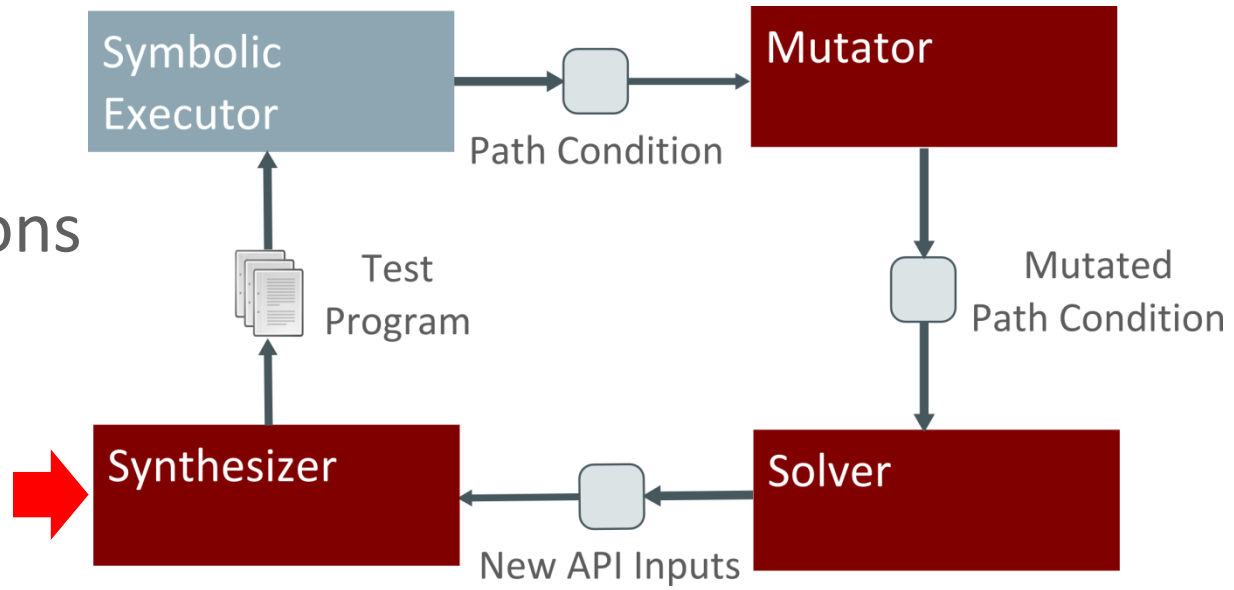
- Introduces symbolic values and types for inputs
- Generates path conditions with ***symbolic types***
- Explores virtual invocation symbolically
- Supports reflection partially
  - Symbolic methods and symbolic fields
- Works with JNI calls



Current Status: Other Components

All Components Functioning

- Synthesizer generates Java byte-code
  - Creates objects using public constructors
- Mutator generates new path conditions
  - Explores different invocation targets
- Solver handles type constraints
  - Generates concrete input types and values



Preliminary Results for Fuzzing OpenJDK-7u7b31

Automatically generate tests using the current infrastructure

Package	#Classes (#Tested/#Total)	#Methods (#Tested/#Total)	Block Coverage (%) For Tested Method	Time
java.lang	91 / 267	978 / 3775	70.1	464m 46s
java.text	28 / 63	273 / 664	58.5	117m 38s
java.nio	47 / 173	147 / 1715	62.0	4m 5s
java.beans	28 / 128	191 / 629	58.8	34m 32s
java.sql	12 / 16	33 / 81	41.5	3m 25s
Total	**206 / 647**	**1622 / 6864**	**Avg: 57.35**	**623m 25s**

Summary

- Infrastructure designed for JDK
  - Symbolically representing types and values
  - Functioning on OpenJDK
- Future work for JDK vulnerability detection
  - Full reflection library support
  - Better synthesis algorithm using static analysis
  - More efficient constraint solver
  - Test generation to expose access control violations in JDK

Hardware and Software
Engineered to Work Together