The JVM is not observable enough (and what to do about it)

Stephen Kell

stephen.kell@usi.ch

"University of Lugano"

joint work with: Danilo Ansaloni, Walter Binder, Lukáš Marek

0xcafebabe

This is a talk about Java bytecode instrumentation

```
00000000  ca fe ba be 00 00 00 32  00 68 0a 00 1b 00 3b 09  |.......2.h....;.|
00000010  00 1a 00 3c 07 00 3d 07  00 3e 0a 00 3f 00 40 0a  |...<..=..>..?.@.|
00000020  00 04 00 41 0a 00 03 00  41 09 00 1a 00 42 07 00  |...A....A....B..|
00000030  43 07 00 44 0a 00 3f 00  45 0a 00 0a 00 46 0a 00  |C..D..?.E....F..|
00000040  09 00 46 09 00 1a 00 47  0a 00 03 00 48 0a 00 03  |..F....G....H...|
00000050  00 49 07 00 4a 0a 00 11  00 4b 0a 00 03 00 4c 0a  |.I..J....K....L.|
00000060  00 3f 00 4c 0a 00 11 00  4d 0a 00 09 00 4e 0a 00  |.?.L....M....N..|
00000070  11 00 4f 0a 00 09 00 50  0a 00 09 00 51 07 00 52  |..O....P....Q..R|
00000080  07 00 53 01 00 06 73 6f  63 6b 65 74 01 00 11 4c  |..S...socket...L|
00000090  6a 61 76 61 2f 6e 65 74  2f 53 6f 63 6b 65 74 3b  |java/net/Socket;|
000000a0  01 00 02 69 73 01 00 19  4c 6a 61 76 61 2f 69 6f  |...is...Ljava/io|
000000b0  2f 44 61 74 61 49 6e 70  75 74 53 74 72 65 61 6d  |/DataInputStream|
000000c0  3b 01 00 02 6f 73 01 00  1a 4c 6a 61 76 61 2f 69  |;...os...Ljava/i|
000000d0  6f 2f 44 61 74 61 4f 75  74 70 75 74 53 74 72 65  |o/DataOutputStre|
000000e0  61 6d 3b 01 00 06 3c 69  6e 69 74 3e 01 00 14 28  |am;...<init>...(|
000000f0  4c 6a 61 76 61 2f 6e 65  74 2f 53 6f 63 6b 65 74  |Ljava/net/Socket|
00000100  3b 29 56 01 00 04 43 6f  64 65 01 00 0f 4c 69 6e  |;)V...Code...Lin|
```

- the Java platform’s *de facto* standard mechanism
- … for *observing* programs in execution
- (non-interactively, usually)

What

- profilers (JP2, . . .)
- data race detectors (FastTrack, . . .)
- white-box / active testing (jCUTE, . . .)
- security monitors (TaintDroid, . . .)
- memory / GC analyses (ElephantTracks, . . .)
- . . .

How

Rewrite the bytecode, adding analysis “snippets”

- on e.g. method entries, object allocations, locking, . . .

Can use libraries that help to munge bytecode

- ASM, BCEL, Javassist, Soot, . . .

Or, some systems abstract the problem a bit more

- Chord, DiSL, BTrace, RoadRunner, . . .

An “innocuous” example (using DiSL)

```
public class TargetClass {
    public static void main(String[] args) {
        System.err.println("MAIN");
} }
```

```
public class DiSLClass {
    @Before(marker = BodyMarker.class, scope = "java.lang.Object.*")
    public static void onMethodExit(MethodStaticContext msc) {
        System.err.print(".");
} }
```

A choice quotation

(from http://docs.oracle.com/javase/6/docs/technotes/guides/jvmti/)

> 'Typically, these alterations are to add "events" to the code of a method —for example, to add, at the beginning of a method, a call to `MyProfiler.methodEntered()`. Since the changes are purely additive, they do not modify application state or behavior.'

Purely additive?

Wishful thinking

Some questions:

- what problems occur writing tools this way?
- can we avoid them?
- what would be a better observation mechanism?

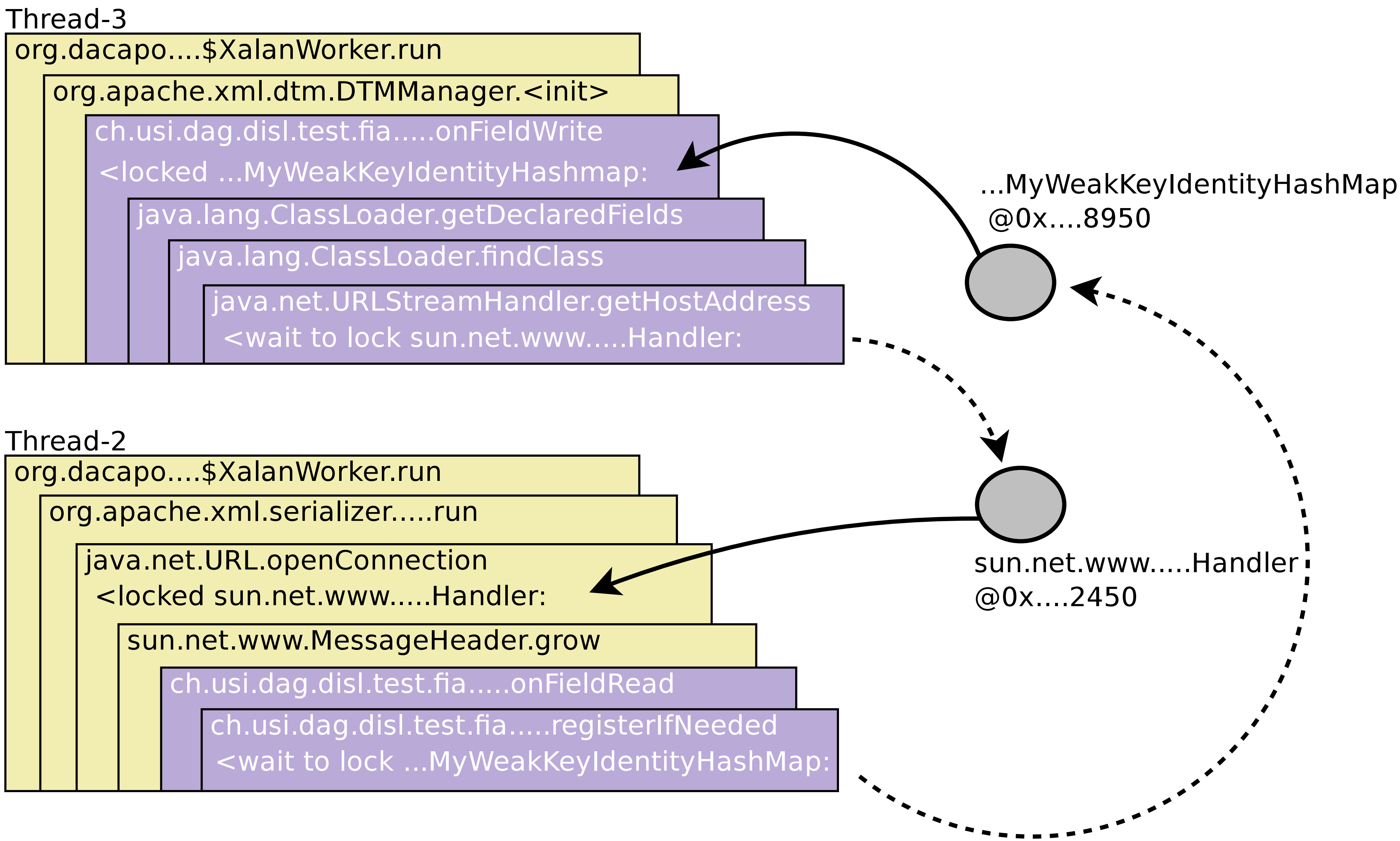
Answers: several; not really; let's talk about it. . .

A summary of the difficulties

In the paper:

- **deadlock between instrumentation and program**
- **state corruption by non-reentrant code**
- **method calls: unsafe but unavoidable**
- “my instrumentation crashes the VM”
- instrumented bytecode that doesn’t verify
- coverage underapproximation (initializers, startup)
- coverage overapproximation (shared threads)

Deadlock



Thread-3
org.dacapo....$XalanWorker.run
org.apache.xml.dtm.DTMManager.<init>
ch.usi.dag.disl.test.fia.....onFieldWrite
<locked ...MyWeakKeyIdentityHashmap:
java.lang.ClassLoader.getDeclaredFields
java.lang.ClassLoader.findClass
java.net.URLStreamHandler.getHostAddress
<wait to lock sun.net.www.....Handler:
...MyWeakKeyIdentityHashMap
@0x....8950
Thread-2
org.dacapo....$XalanWorker.run
org.apache.xml.serializer.....run
java.net.URL.openConnection
<locked sun.net.www.....Handler:
sun.net.www.MessageHeader.grow
ch.usi.dag.disl.test.fia.....onFieldRead
ch.usi.dag.disl.test.fia.....registerIfNeeded
<wait to lock ...MyWeakKeyIdentityHashMap:
sun.net.www.....Handler
@0x....2450

Attempted escape (1): share no mutable state!

Q. Can't we just *never share mutable state*? ($\rightarrow$ no locking)

A. Good idea. But

- this implies calling no methods
- … not even static ones
- does your analysis do I/O? (hint: yes)

Reentrancy example

```
public class TargetClass {
    public static void main(String[] args) {
        System.err.println("MAIN");
} }
```

```
public class DiSLClass {
    @Before(marker = BodyMarker.class, scope = "java.lang.Object.*")
    public static void onMethodExit(MethodStaticContext msc) {
        System.err.print(".");
} }
```

Any guesses about the output?

The output

```
 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
 MAIN.MAIN
.
. . . .
```

Non-reentrant code now called reentrantly

```
package java.io;
class PrintStream {
// ...
void println () {
```

Non-reentrant code now called reentrantly

```
package java.io;
class PrintStream {
//  ...
void println ()  {
   //  ...
   try  {
      this.state  = PENDING;
```

Non-reentrant code now called reentrantly

```
package java.io;
class PrintStream {
// ...
void println () {
  // ...
  try {
    this.state = PENDING;
    while (pos != len) pos = copySome(in, out, pos, len);
```

Non-reentrant code now called reentrantly

```
package java.io;
class PrintStream {
// ...
void println () {
  // ...
  try {
    this.state = PENDING;
    while (pos != len)  pos = copySome(in, out, pos, len);
  } finally {
    assert this.state == PENDING; // FAILS following reentrant call!
    this.state = CLEAR;
  }
} }
```

Attempted escape (2): use native code?

Q. Maybe just do your analysis in native code?

A. Okay, but

- (I thought you liked Java?)
- any library method might be implemented natively…
- *and* might call back into [instrumented] Java
- so sharing can still happen, unbeknownst to analysis

Less likely perhaps, but how to be *safe*?

A known approach we *could* borrow…

Valgrind, Pin, DynamoRIO et al:

- share neither state nor code with the observed program
- $\rightarrow$ private libraries (duplicate libc, etc.)
- $\rightarrow$ avoid signal handling, wait(), shared fds, …

We can do the same, at least from native code…

- maybe from Java too?
- … if can replicate down to Object, ClassLoader etc.

Problem: lost expressiveness!

Expressiveness lost

If we're avoiding shared state, we can't call any Java APIs:

- no reflection
- can't call getters ($\rightarrow$ field access instead)
- can't observe even basic semantics (e.g. equals())
- $\rightarrow$ can't aggregate data using equality
- can't synchronise

One consequence: can't analyse *user-defined abstractions*

- including library-defined abstractions!

Aiming at something better

Wanted: keep the abstraction, but *add* isolation

- bytecode instrumentation (BCI) is an abstraction
- so far, we have made it “safe” by throwing it away

What's the design space of observation?

- isolation: in-process (soft) versus out-of-process (hard)
- abstraction: VM-level (fixed) versus user-level (flexible)
- synchrony…

We have a weird asymmetric isolation requirement.

- observed is *not* influenced by observer
- observer *is* influenced by observed!

Isolated bytecode abstractions

Existing systems we can take inspiration from

- debugger expression eval (VM-style)
- debugger expression eval (native-style)
- Unix `fork()`
- shared memory (is asymmetric. . .)
- isolates, SIPs (MVM, Singularity)
- async assertions (Aftandilian &al, OOPSLA '11)
- JIT purity analysis

Can we share the work with expression eval in debuggers?

Conclusions

Currently, bytecode instrumentors risk

- deadlock, reentrancy-derived corruption, …
- more in the paper!

We can only do things safely by

- trapping to a sharing-free environment ASAP
- avoid interacting with user-defined abstractions

This limits our expressiveness. Real solution:

- an asymmetric “isolated bytecode” abstraction
- might unify/replace a subset of JDWP too! (ask me)

Thanks for listening. Questions?