Understanding Java:
from languages towards systems

Martín Abadi

Systems Research Center

Compaq

What is in the “Java paradigm”?

The Java language?

The Java bytecode language (JVML)?

All of the above, plus libraries?

The Java Virtual Machine (JVM)?

The systems in which these are embedded?

Jini?

All related languages, libraries, and systems?

We have done well on individual languages (“physics”), less well on systems (“chemistry”).

Success stories: the Java language

Decades of formal work on type theory, logics, and semantics have not been completely in vain.

The results of that work apply to Java.

- Several standard concepts and techniques can be adapted to Java with moderate effort:
  - type soundness,
  - program-verification methods,

    ⋮
- Precise definitions, statements, and proofs (even mechanical proofs) are possible.
- Some trouble spots (e.g., dynamic linking) can be explained fairly formally.

This should not be completely surprising.

Unfinished stories

A coherent, formal study of the complete Java language is still work in progress.

Some wishes:

- A precise, useful Java memory model.
  (Cf. Reus et al., Coscia & Reggio.)
  - Threads and locks are part of the language.
  - A good model should help in verification and in optimization (e.g., code motion).
- Practical program verification.
  (Cf. ESC, Jacobs et al..)

Conversely, Java is helping to focus formal work.

Success stories: JVML

In parallel, JVML has been treated rigorously.

We have obtained:

– definitions and theorems,

– a handle on trouble spots (e.g.., interaction of initialization and subroutines).

A coherent, formal study of JVML is still work in progress.

But (roughly) JVML is no harder than Java.

Unfinished stories

Formal work has not yet led to more uniform or efficient JVMs.

It is not entirely clear what all of this has to do with security.

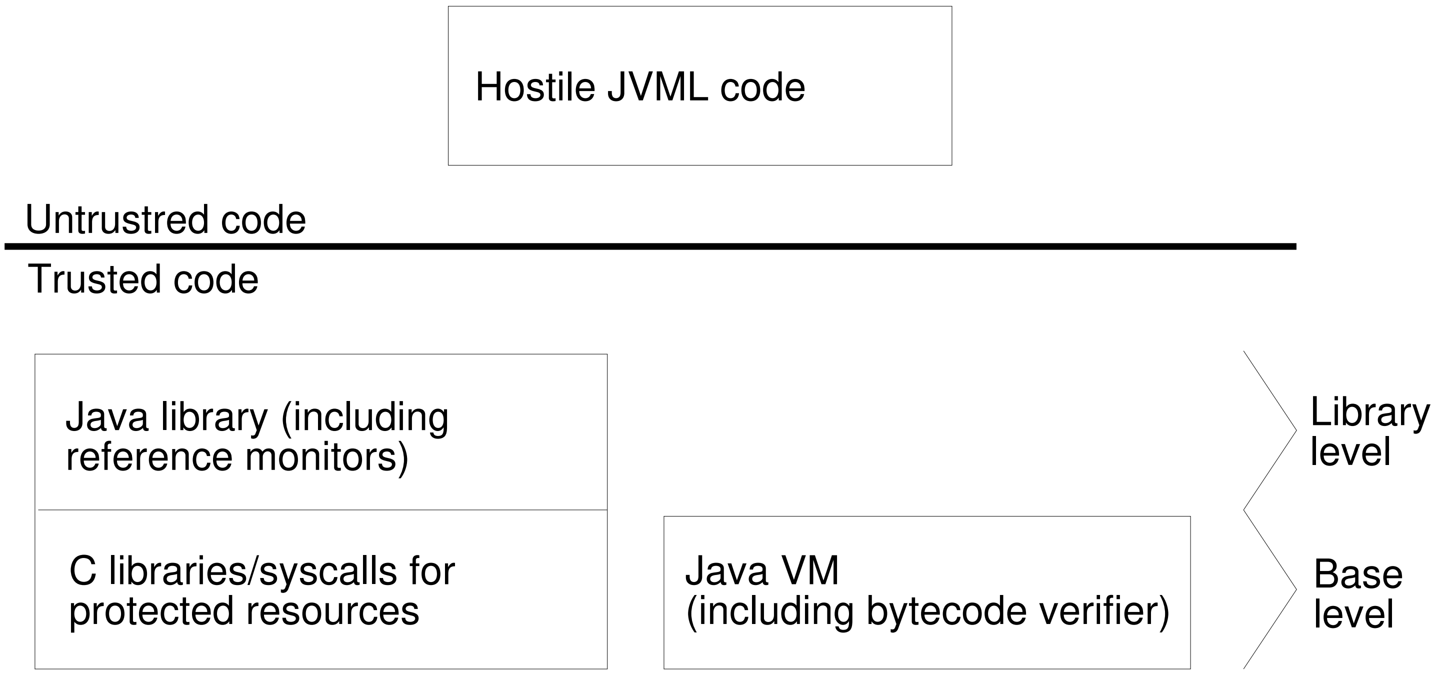
Compiling Java to JVML

An implementation of Java should be

- functionally correct (cf. Börger and Schulte),
- even in interaction with (malicious?) systems that are not the result of compiling Java.

JVML in context

Common Java compilers produce JVML code.



JVMs execute JVML code within web browsers and other applications.

The specification of Java discusses JVML (!).

> Verification ensures that the binary representation of a class or interface is structurally correct. For example, it checks that every instruction has a valid operation code; that every branch instruction branches to the start of some other instruction, rather than into the middle of an instruction; that every method is provided with a structurally correct signature; and that every instruction obeys the type discipline of the Java language.

When does JVML code "obey the type discipline of the Java language"?

> "The question is," said Alice, "whether you *can* make words mean so different things."

Full abstraction

The idea of "full abstraction" is useful for understanding the security of implementations
– in the case of Java,
– for communication abstractions (e.g., RMI),
– perhaps in general.

Loosely:

- Two programs are equivalent in a language if they have the same observable behavior in all contexts of the language (Morris).
- A translation is fully abstract if it respects equivalence (Plotkin, Mitchell, Riecke, . . .).

A class in Java

```
class C {
    private int x;
    public void set_x(int v) {
      this.x = v;
    };
}
```

Conjecture:

In Java, any two instances of C are equivalent.

⇒ A Java programmer may expect the value of x to remain secret.

More substantial and realistic examples can use privacy in similar ways.

The same class in JVML

```
class C {
    private int x;
    public void set_x(int) {
      .framelimits locals = 2, stack = 2;
      aload_0;        // load this
      iload_1;        // load v
      putfield x;     // set x
    };
}
```

The privacy of `x` is preserved (!).

Accessibility in JVML

Privacy at the JVML level can be important for protection **against** mobile objects.

- Java libraries interact with mobile code at the JVML level.
- Inter-applet communication may remain at the JVML level.

Privacy at the JVML is less meaningful for protection **of** mobile objects.

- Mobile objects on untrusted machines are subject to lower-level attacks.

Security and bytecode verification

For security, one cannot ignore

- type errors,
- operand stack overflow or underflow,
- access control violations,

⋮

Otherwise, for example, the qualifier **private** is worth little.

The bytecode verifier helps prevent errors by checking untrusted JVML code before execution.

But bytecode verification is not clearly the same as security.

The gap between Java and JVML

There are discrepancies between Java and JVML.

- Exceptions:

  E.g., in Java, the handler for an exception is never inside the code that the handler protects.

  It may be in JVML.

- Synchronization:

  In Java, method calls and locking are nicely nested (so locks can be placed on the stack).

  JVML allows fairly arbitrary patterns of **monitorenter** and **monitorexit** operations.

- Accessibility:

  Although qualifiers like **private** appear in both Java and JVML, they may not provide the same guarantees.

An inner class in Java

```
class D {
    class E {
        private int y = x;
    };
    private int x;
    public void set_x(int v) {
      this.x = v;
    };
}
```

`E` is an inner class with access to `D`'s `x`.

Translation of an inner class into JVML

```
class D {
    private int x;
    public void set_x(int v) {
      this.x = v;
    };
    static int get_x(D d) {
      return d.x;
    };
}

class E {
    ... get_x ...
}
```

JVML does not have inner classes.

⇒ An accessor method `get_x` is added to `D`.

Security and full abstraction

Since verification restricts JVML contexts,
it "pushes" towards security and full abstraction.

But:

Any class in the same package as `D` can access `x`.

$\Rightarrow$ Despite bytecode verification, the translation from Java to JVML does not preserve all "expected" security properties.

A JVML context that runs `get_x` distinguishes instances of `D` with different values for `x`.

$\Rightarrow$ Despite bytecode verification, full abstraction is lost.

Security and full abstraction (cont.)

What is a secure implementation of Java?

- Full abstraction could be nice.
- It may be too much to ask for.
- Weaker guarantees may suffice.

Alternatives to full abstraction

One may ignore the security of translations

- when low-level code is signed by a trusted party,
- if one analyzes low-level code independently of its source.

These alternatives are not always satisfactory.

In other cases, translations should preserve at least some security properties.

For example, we may use information-flow type systems and prove type preservation.

- Classified data is mapped to classified data.
- Trusted data is mapped to trusted data.

This remains to be done!

Conclusions

The “formal underpinnings of the Java paradigm” could fruitfully explain more than Java or JVML in isolation.

They could address Java systems:
– translators between languages,
– libraries,
– distribution.

Java can help guide formal work:
– with a notation and an architecture,
– with precise instances of general notions,
– with a set of problems.