Java bytecode verification: an overview

Xavier Leroy

INRIA Rocquencourt & Trusted Logic

Overview

Java bytecode verification: a crucial security component in software architectures allowing downloading of untrusted code.

This talk: a survey of known techniques and problems.

Thesis: an excellent application area for formal techniques.

Part 1

The role of bytecode verification

Web applets: the viruses of the '90s?

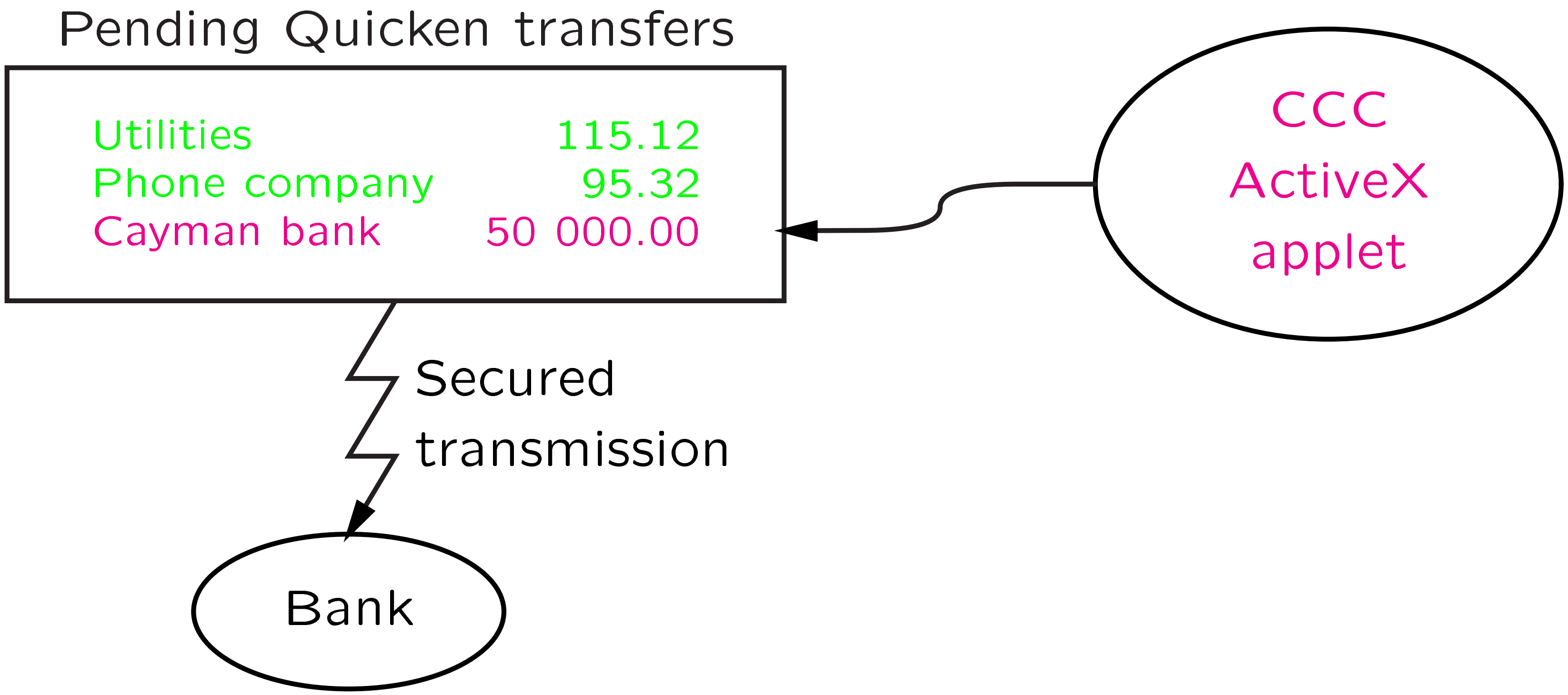
Web applets = automatic execution of untrusted code on a sensitive computer.

(Also: mobile code; servlets; active networks; user-provided kernel extensions; Java smart cards and terminals; etc.)

Obvious security risks:

- Integrity: destroy or modify important data.
- Confidentiality: leak sensitive information.
- Trojan attacks, virus-like behaviors.

An example of a malicious applet



One file access can be very costly . . .

The Java sandbox model

Do not execute applets directly on the hardware, but via a software isolation layer (the *sandbox*).

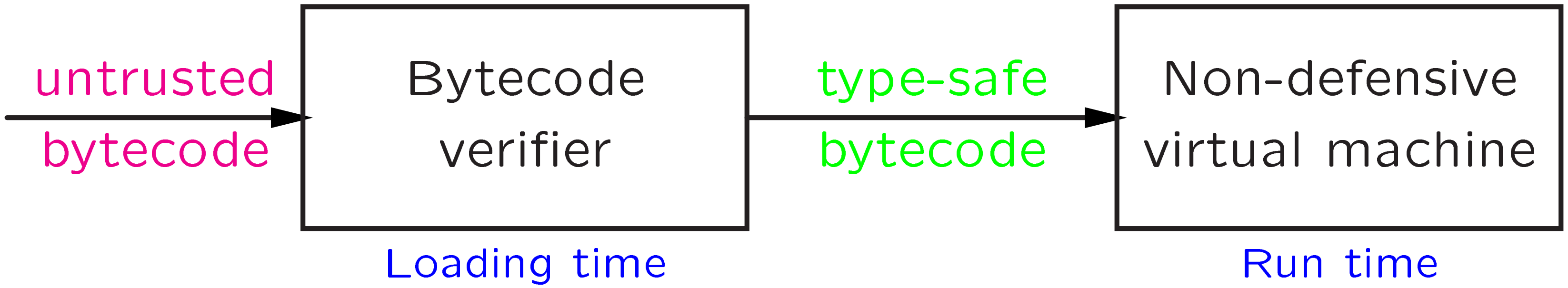
1. Secured APIs performing access control
   - File accesses: prohibited, or just to temporary files.
   - Network accesses: prohibited, or just to originating site.
2. Virtual machine ensuring type safety
   - Data integrity: no pointer forging; bounds checks on arrays.
   - Code integrity: no data $\rightarrow$ code conversions; no jumping into the middle of an API function.
   - Enforce visibility modifiers: no access to `private` API methods or data.

Type safety: defensive VM versus bytecode verification

Type safety in the VM can be achieved in two ways:

1. Defensive virtual machine:
   Performs run-time type checks, along with bytecode execution.
   Slows down execution considerably.

2. Bytecode verification at loading time:
   A separate static code analysis establishes type safety.
   Faster execution by a non-defensive VM.



Properties established by bytecode verification

Well-formedness of the code.

E.g. no branch to the middle of an instruction encoding.

Instructions receive arguments of the expected types.

E.g. `getfield C.f` receives a reference to an object of class `C` or a subclass.

The stack does not overflow or underflow.

Within one method; dynamic check at method invocation.

Local variables (registers) are initialized before being used.

E.g. cannot use random data from uninitialized register.

Objects (class instances) are initialized before being used.

I.e. `new C`, then call to a constructor of `C`, then use instance.

Visibility modifiers are respected.

E.g. no access to a `private` member outside of class.

Technically: checked during linking, not verification.

Part 2

Basics of bytecode verification

(or: Dataflow Analysis 101)

Verifying straight-line code

“Execute” the code with a type-level abstract interpretation of a defensive virtual machine.

- Manipulates a stack of types and a register set holding types.
- For each instruction, check types of arguments and compute types of results.

Example:

```
class C {
    int x;
    void move(int delta) {
        int oldx = x;
        x += delta;
        D.draw(oldx, x);
    }
}
```

```
                              r0: C, r1: int, r2: ⊤        [ ]
ALOAD 0
                              r0: C, r1: int, r2: ⊤        [ C ]
GETFIELD C.x : int
                              r0: C, r1: int, r2: ⊤        [ int ]
DUP
                              r0: C, r1: int, r2: ⊤        [ int ; int ]
ISTORE 2
                              r0: C, r1: int, r2: int     [ int ]
ILOAD 1
                              r0: C, r1: int, r2: int     [ int ; int ]
IADD
                              r0: C, r1: int, r2: int     [ int ]
ALOAD 0
                              r0: C, r1: int, r2: int     [ int ; C ]
SETFIELD C.x : int
                              r0: C, r1: int, r2: int     [ ]
ILOAD 2
                              r0: C, r1: int, r2: int     [ int ]
ALOAD 0
                              r0: C, r1: int, r2: int     [ int ; C ]
GETFIELD C.x : int
                              r0: C, r1: int, r2: int     [ int ; int ]
INVOKESTATIC D.draw : void(int,int)
                              r0: C, r1: int, r2: int     [ ]
RETURN
```

Handling forks and join in the control flow

Branches are handled as customary in data flow analysis:

- Fork points: propagate types to all successors.
- Join points: take least upper bound of types from all predecessors.
- Iterative analysis: repeat until types are stable.

r0: int, r1: C, r2: D

r3: ⊤, r4: ⊤ [ ]

ILOAD 0

r3: ⊤, r4: ⊤ [int]

IFEQ

r3: ⊤, r4: ⊤ [ ]

ICONST 42

r3: ⊤, r4: ⊤ [int]

ISTORE 4

r3: ⊤, r4: int [ ]

ALOAD 1

r3: ⊤, r4: int [C]

r3: ⊤, r4: ⊤ [ ]

ALOAD 2

r3: ⊤, r4: ⊤ [D]

r3: ⊤, r4: ⊤ [Object]

ASTORE 3

r3: Object, r4: ⊤ [ ]

ILOAD 4

ERROR!

More formally . . .

Model the type-level VM as a transition relation:

$$instr : (\tau_{regs},\ \tau_{stack}) \rightarrow (\tau'_{regs},\ \tau'_{stack})$$

e.g. $\mathtt{iadd} : (r,\ \mathtt{int.int.}s) \rightarrow (r,\ \mathtt{int.}s)$

Set up dataflow equations:

$$i : in(i) \rightarrow out(i)$$

$$in(i) = lub\{out(j) \mid j \text{ predecessor of } i\}$$

$$in(i_{start}) = ((P_0, \ldots, P_{n-1}, \top, \ldots, \top), \varepsilon)$$

Solve them using standard fixpoint iteration.

Several proofs of correctness, on paper and on machine (e.g. Nipkow & Pusch).

Part 3

The devil

is

in the details

Bytecode verification goes beyond classic dataflow analysis

1. Interfaces:
   - The subtype relation is not a semi-lattice.
   - Use run-time type checks for interfaces (Sun).
   - Or do Dedekind completion (Knoblock & Rehof).

2. Object initialization protocol:
   - Requires a bit of must-alias analysis during verification.
   - See Bertot's talk.

3. Subroutines.
   - Next slides.

Subroutines in the JVM

A code-sharing device, used to avoid code duplication when compiling `try ... finally`.

```
try {
    ...
    if (cond) { return e; }
    ...
} finally {
    finalization code
}
```

Without subroutines

```
    ...
    iload cond
    ifne Early_return
    ...
    finalization code
    ...

Early_return:
    compute e
    istore 2
    finalization code
    iload 2
    ireturn

Exception_handler:
    astore 2
    finalization code
    aload 2
    athrow
```

With subroutines

```
    ...
    iload cond
    ifne Early_return
    ...
    jsr Subroutine
    ...

Early_return:
    compute e
    istore 2
    jsr Subroutine
    iload 2
    ireturn

Exception_handler:
    astore 2
    jsr Subroutine
    aload 2
    athrow

Subroutine:
    astore 3
    finalization code
    ret 3
```

The problem with subroutines

Treating `jsr` and `ret` like branches causes excessive type merging.

```
istore 2
        r2: int
jsr Subroutine
         r2: ⊤
iload 2
         ERROR!
ireturn

...

astore 2
  r2: Throwable
jsr Subroutine
         r2: ⊤
aload 2
         ERROR!
athrow
```

```
Subroutine:
  r2: ⊤
    astore 3

    finalization code

    ret 3
  r2: ⊤
```

Solution 1: polymorphism (Sun)

Registers that are not used in a subroutine keep their types across a `jsr` to this subroutine.

```
istore 2
            r2: int, r3: ⊤
jsr Subroutine
            r2: int, r3: retaddr
iload 2
ireturn
...
astore 2
            r2: Throwable, r3: int
jsr Subroutine
            r2: Throwable, r3: retaddr
aload 2
athrow
```

```
Subroutine:
            r2: ?, r3: ⊤
    astore 3
    finalization code
        (not using r2)
    ret 3
            r2: ?, r3: retaddr
```

Problems with this solution

Reminiscent of parametric polymorphism in type systems:

$$\mathtt{Subroutine} : \forall\alpha.\ \{\mathtt{r2} : \alpha;\ \mathtt{r3} : \top\} \to \{\mathtt{r2} : \alpha;\ \mathtt{r3} : \mathtt{retaddr}\}$$

One issue remains:
– match `jsr` instructions with `ret` instructions;
– determine registers used by each subroutine.

Since subroutines are not syntactically delimited in the bytecode, this requires a complementary "subroutine structure" analysis.

Some formalizations of subroutine structure analysis:
– Abadi & Stata: incomplete (does not extend to exceptions).
– Qian: close to Sun's reference implementation.

Soundness of this approach still not proved for the full JVM.

Solution 2: polyvariance

Analyse subroutine bodies once per call site.

Instead of having one (stack type, register type) per PC, have one per PC and per contour.

The contour approximates the subroutine call sequence:

ε	outside of any subroutine
L	in a subroutine called from PC L
$L_1.L_2$	in a subroutine called from L_2, itself in a subroutine called from L_1

```
   istore 2
           ε: r2: int, r3: ⊤
A: jsr Subroutine
           ε: r2: int, r3: retaddr
   iload 2

   ireturn


   ...

   astore 2
           ε: r2: Throwable, r3: int
B: jsr Subroutine
           ε: r2: Throwable, r3: retaddr
   aload 2

   athrow
```

```
Subroutine:
     A: r2: int, r3: ⊤
     B: r2: Throwable, r3: int
   astore 3

   finalization code

   ret 3
     A: r2: int, r3: retaddr
     B: r2: Throwable,
            r3: retaddr
```

Solution 3: enumerative model-checking

(Posegga & Vogt; Brisset)

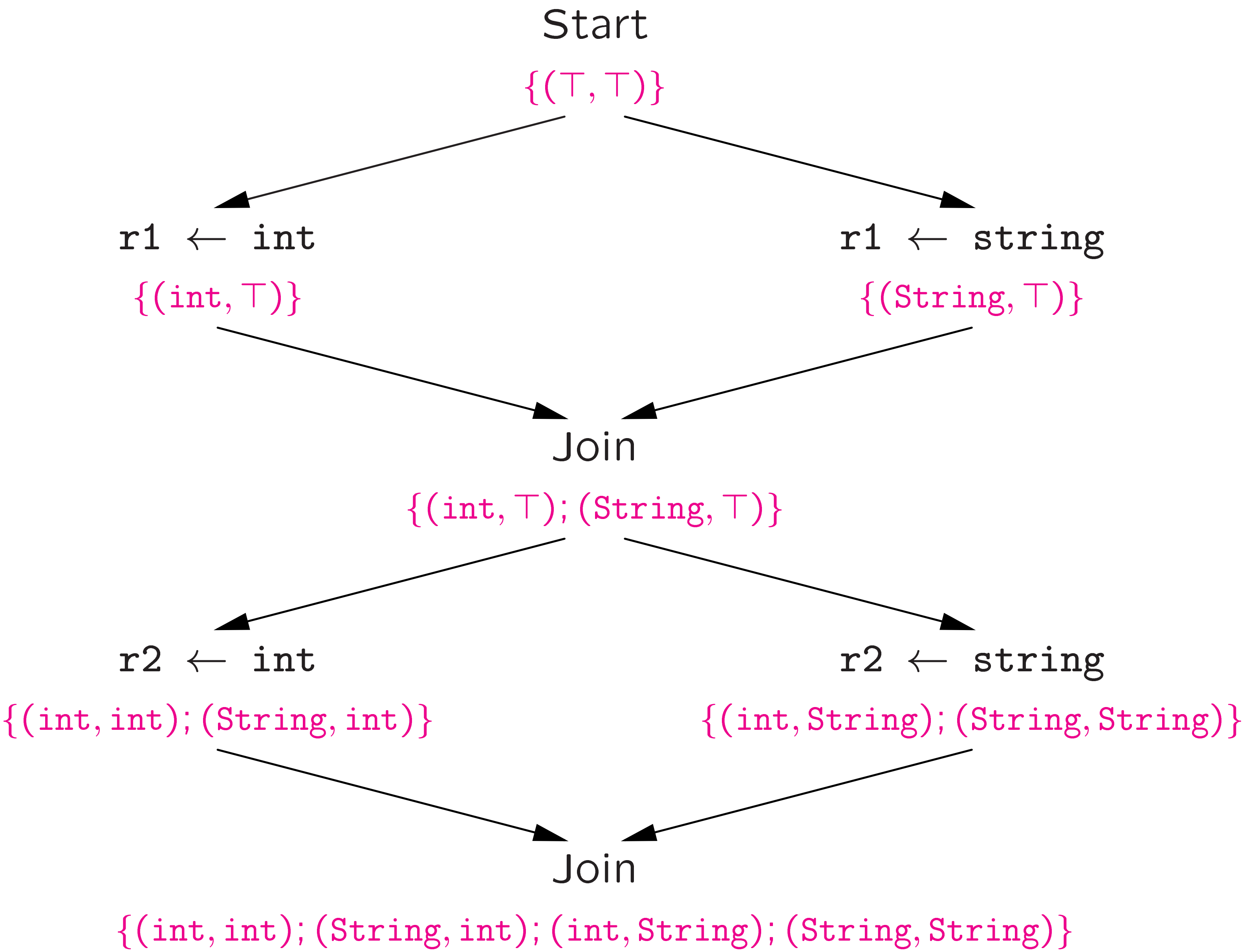
Model the type-level VM as a transition relation:

$$(PC_{instr},\ \tau_{regs},\ \tau_{stack}) \rightarrow (PC_{succ},\ \tau'_{regs},\ \tau'_{stack})$$

and explore all reachable states.

No type merging at all – the ultimate polyvariant analysis!

Computationally expensive, but useful to prove the correctness of more approximate analyses.


Start
{(⊤, ⊤)}
r1 ← int
{(int, ⊤)}
r1 ← string
{(String, ⊤)}
Join
{(int, ⊤); (String, ⊤)}
r2 ← int
{(int, int); (String, int)}
r2 ← string
{(int, String); (String, String)}
Join
{(int, int); (String, int); (int, String); (String, String)}

Part 4

Bytecode verification

on small embedded computers

Example: Java-enabled smart cards

Java Card smart cards support downloading of Java applets, and have strong security requirements.

But: Java Cards offer very limited computing resources:
– approx. 2 kilo-bytes of RAM
– approx. 16 kilo-bytes of EEPROM
– slow 8-bit processor (approx. 1/1000 of a PC).

The challenge of on-card verification

Bytecode verification on Java Card (and similar embedded devices) is challenging:

- Time: complex process, e.g. fixpoint iteration.

- Space: the memory requirements of Sun's algorithm are

$$3 \times (M_{stack} + M_{regs}) \times N_{branch}$$

 (to store the inferred types at each branch target point).

 E.g. $M_{stack} = 5$, $M_{regs} = 15$, $N_{branch} = 50 \Rightarrow$ 3450 bytes.

Solution 1: lightweight bytecode verification

(Rose & Rose; an application of Proof Carrying Code.)

Transmit the stack and register types at branch target points along with the code (*certificates*).

The verifier checks this information rather than inferring it.

Benefits:
– Fixpoint iteration is avoided; one pass suffices.
– Certificates are read-only and can be stored in EEPROM.

Limitations:
– Not known how to handle subroutines.
– Certificates are large (20–100% of code size).

Solution 2: restricted verification + code transformation

(Leroy, Trusted Logic.)

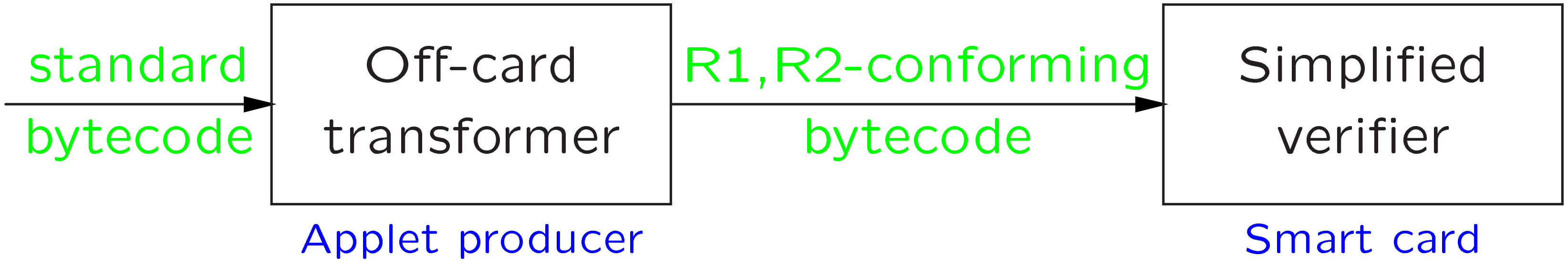
The on-card verifier puts two additional requirements on verifiable code:

– R1: The stack is empty at all branches.

– R2: Each register has the same type throughout a method.

Requires only one global stack type and one global set of register types ⇒ low memory $3 \times (M_{stack} + M_{regs})$.

An off-card code transformer rewrites any legal bytecode into equivalent bytecode meeting these requirements.



Code size increase due to transformation: less than 2 %.

Part 5

Conclusions and perspectives

Conclusions

Bytecode verification is an excellent playground for formal techniques:

- High stakes (security).
- Significant demands from industry
  (e.g. in the smart card world).
- Relatively small, well-delimited problem.
- Complex algorithms.

What can formal techniques provide?

- Formal specifications:
  many partial specs, but the only complete reference is Sun's implementation.

- Formal verification of bytecode verifiers:
  mostly proof checking so far; model checking?

- Design of new algorithms:
  combinations of abstract interpretation and enumerative model checking.

- Generation of test cases:
  currently, hand-written test suites + random mutation testing.

Perspectives

Verifying actual machine code rather than VM code:
– Typed Assembly Language (Morrisett et al).

Verifying more than just type safety:
– resource bounds (memory, time, reactiveness);
– access control, information flow.

Formal verification of Java APIs:
– APIs is where all the high-level security properties are enforced;
– verified applets can (and do) still exploit holes in APIs.