Exploiting type systems and static analyses for smart card security

Xavier Leroy

INRIA Rocquencourt & Trusted Logic

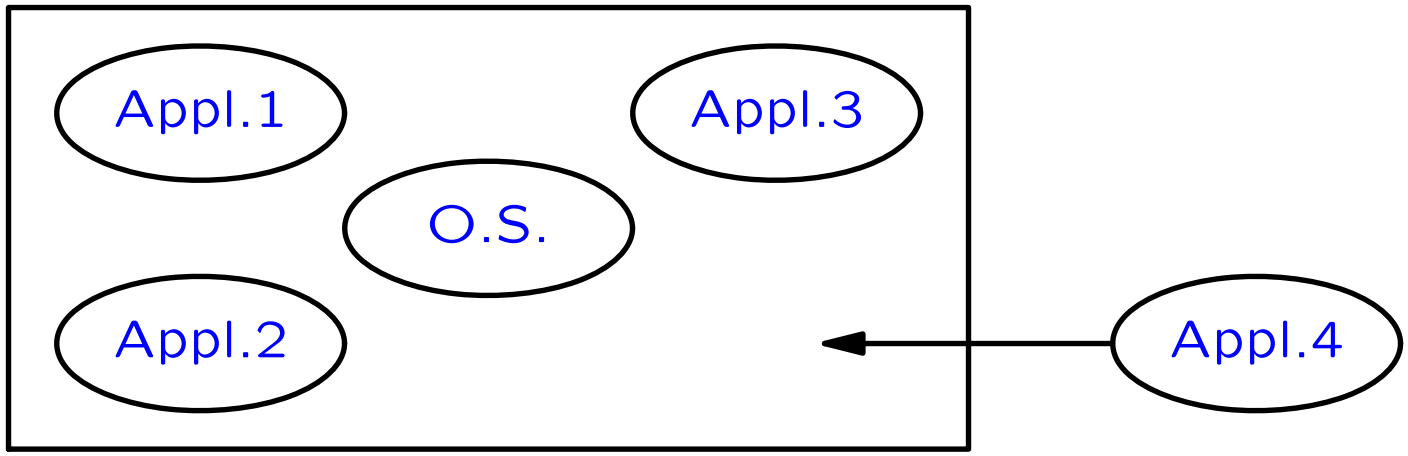
Outline

1. What makes strongly typed programs more secure?
2. Enforcing strong typing (bytecode verification).
3. Static analyses beyond strong typing.

Part 1

The role of strong typing

Context



A secure system running multiple applications, possibly untrusted:

- Workstation: OS kernel + users' applications.
- Web browser: browser + Web applets + Web scripts.
- Java Card: OS + applications + cardlets.

Must ensure **isolation** between the applications (integrity, confidentiality), as well as **controlled communications** between them.

Enforcing isolation between applications

Hardware protection mechanisms: (classic OS model)

- Hardware access control on memory pages and devices.
- Forces dangerous operations to go through the OS.
- Not always available; sometimes too coarse.

Software-only isolation: (the Java model)

- Execute applications via a software isolation layer (*sandbox*): virtual machine + native APIs performing access control.
- How to ensure that the isolation layer is not bypassed?

Strong typing to the rescue

Strong typing in programming languages imposes a discipline on the use of data and code:

- a reference cannot be forged by casting from an integer;
- one cannot jump in the middle of the code of a function;
- etc.

This discipline is good for program reliability in general.
(Avoid "undefined behaviors", e.g. crashes.)

But it can also be exploited to ensure that a software isolation layer cannot be bypassed.
(Avoid "undefined behaviors", e.g. attacks.)

Strong typing and virtual machine integrity

Consider a malicious cardlet such as the "memory dump" cardlet:

```
class MaliciousCardlet {
  public void process(APDU a) {
    for (short i = -0x8000; i <= 0x7FFC; i += 2) {
      byte [] b = (byte []) i;
      send_byte((byte) (b.length >> 8));
      send_byte((byte) (b.length & 0xFF));
    }
  }
}
```

A typing violation (the cast in red) translates to a security attack.

Strong typing and virtual machine integrity

There are many other ways by which type-unsafe code can cause the virtual machine to malfunction and break isolation:

- **Pointer forging**:
  via casts of well-chosen integers `(byte []) 0x1000`
  or via pointer arithmetic `(byte [])((int)a + 2)`.
  Infix pointers obtained by pointer arithmetic can falsify the firewall determination of the owner of an object.

- **Illegal cast**:
  casting from `class C { int x; }` to `class D { byte[] a; }`
  causes pointer `a` to be forged from integer `x`.

Strong typing and virtual machine integrity

- **Out-of-bounds access**:
  if `a.length == 10`, referencing `a[20]` accesses another object.
  Buffer overflows in general.

- **Stack smashing**:
  read or modify the caller's local variables;
  overwrite return address.

- **Context switch prevention**:
  replace `obj.meth(arg)` (virtual method call, context switch)
  by `meth(obj, arg)` (static method call, no context switch).

- **Explicit deallocation**:
  free an object, keep its reference around, wait until memory manager reallocates the space.

- . . . and more.

Strong typing and basic resource confinement

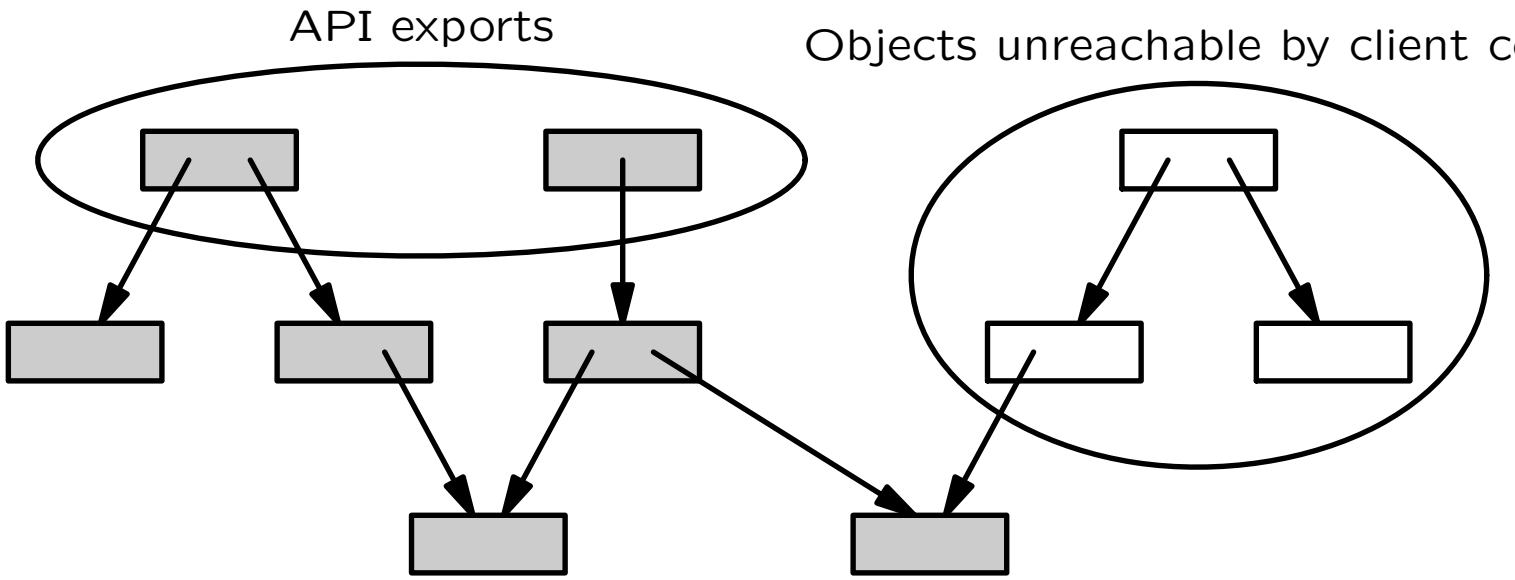
In addition to VM integrity, strong typing implies basic confinement properties between applications:

- Memory reachability guarantees.
- "Procedural encapsulation"
- "Type abstraction"

The latter two rely on enforcement of Java's **visibility modifiers** as part of strong typing.

(Even though violation of visibility does not endanger VM integrity.)

Reachability-based guarantees



The only objects that the client code can access are those that are exported by the API, those reachable from the latter by following pointers, and those allocated by the client code itself.

"Procedural encapsulation"

```
public class Crypto {
    static private SecretKey key;
    static public void encrypt(...) { ... key ... }
    static public void decrypt(...) { ... key ... }
}
```

Although formally reachable from an instance of `Crypto`, the field `key` can only be accessed by the methods of the `Crypto` class, but not by client code.

The `encrypt` and `decrypt` methods encapsulate the resource `key`. They can additionally perform access control.

“Type abstraction”

```
public class Capability {
    private Capability(...) { ... }
    private boolean check(...) { ... }
}
```

Other applications cannot construct instances of class `Capability`.

All they can do is pass capabilities provided by the current application back to that application, unchanged.

Leveraging basic confinement

Secure APIs can be written in a way that leverages these basic confinement properties implied by strong typing.

$\rightarrow$ "Design patterns" for APIs?

The Java APIs rely strongly on these properties. In Java Card, they are partially redundant with the firewall checks.

Part 2

Enforcing strong typing

Enforcing strong typing

Strong typing guarantees can be achieved in two ways:

- **Dynamic enforcement** during execution
  $\longrightarrow$ defensive virtual machine.

- **Static enforcement** during applet loading
  $\longrightarrow$ bytecode verification.

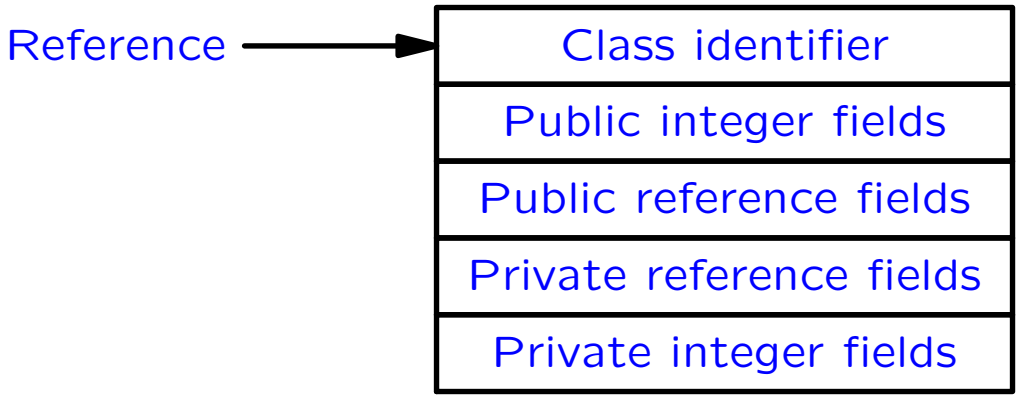
Dynamic enforcement

A defensive virtual machine:

- Attaches type information to values manipulated by the VM.
- Checks type constraints on instruction operands before executing the instruction.
- Also checked dynamically: stack overflows and underflows, array bound checks, ...

Simple and robust, but additional costs in execution speed and memory space (for storing type information).

Efficient representation of type information



In Java: all objects carry their type (for checked downcasts).

In Java Card: instance fields are partitioned into reference / integers and public / package-private.

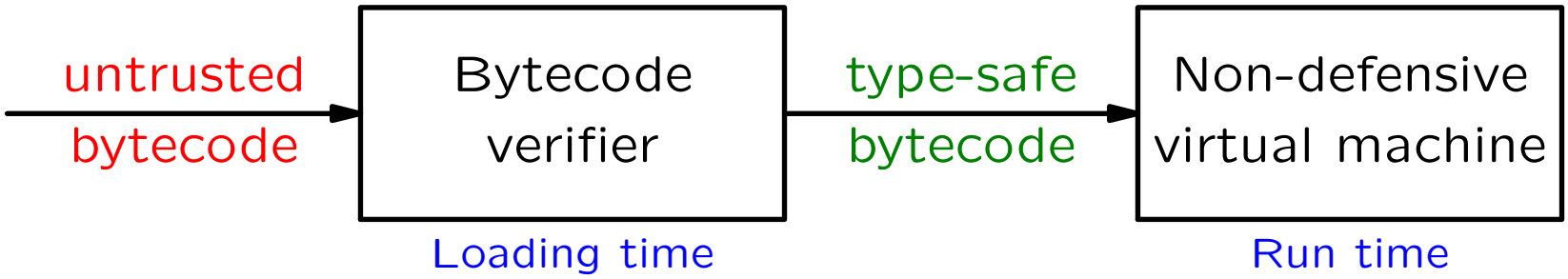
$\rightarrow$ The only additional run-time type information needed is on local variables and stack slots.

$\rightarrow$ That information is just "reference" or "integer" or "return address".

Static enforcement: bytecode verification

Bytecode verification is a static analysis performed on compiled bytecode that establishes that it is type-safe.

Allows faster execution by a non-defensive VM.



Properties statically established by bytecode verification

Well-formedness of the code.

E.g. no branch to the middle of another method.

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 expression 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.

Caveat: other checks remain to be done at run-time (array bounds checks, firewall access rules). The purpose of bytecode verification is to move some, not all, checks from run-time to load-time.

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

```
                          r0: C, r1: short, r2: ⊤      [ ]
ALOAD 0
                          r0: C, r1: short, r2: ⊤      [ C ]
DUP
                          r0: C, r1: short, r2: ⊤      [ C; C ]
GETFIELD C.x : short
                          r0: C, r1: short, r2: ⊤      [ C; short ]
DUP
                          r0: C, r1: short, r2: ⊤      [ C; short ; short ]
SSTORE 2
                          r0: C, r1: short, r2: short   [ C; short ]
SLOAD 1
                          r0: C, r1: short, r2: short   [ C; short ; short ]
SADD
                          r0: C, r1: short, r2: short   [ C; short ]
SETFIELD C.x : short
                          r0: C, r1: short, r2: short   [ ]
SLOAD 2
                          r0: C, r1: short, r2: short   [ short ]
ALOAD 0
                          r0: C, r1: short, r2: short   [ short ; C ]
GETFIELD C.x : short
                          r0: C, r1: short, r2: short   [ short ; short ]
INVOKESTATIC D.draw : void(short,short)
                          r0: C, r1: short, r2: short   [ ]
RETURN
```

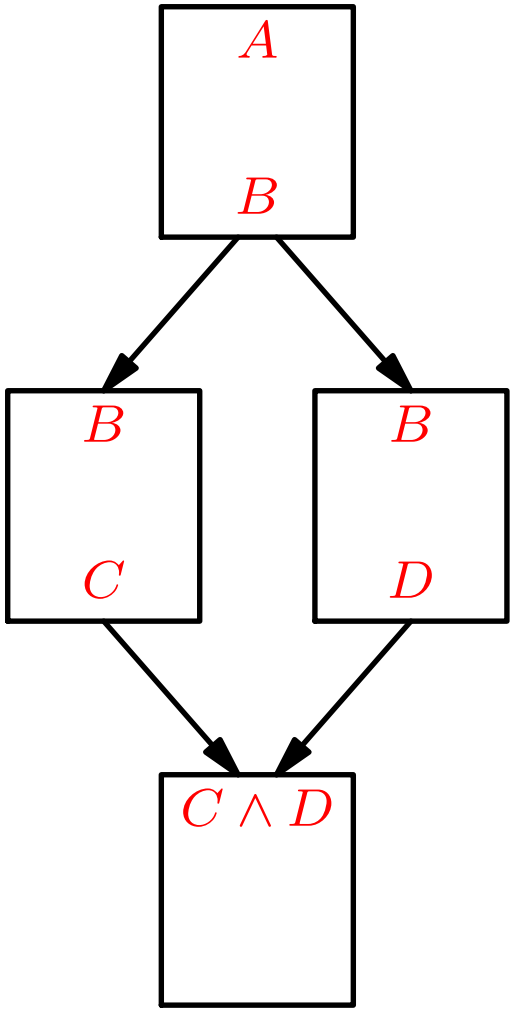
Handling forks and join in the control flow

Branches are handled as usual 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.



More formally . . .

Model the type-level VM as a transition relation:

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

e.g. $\texttt{sadd} : (r,\ \texttt{short.short.}s) \rightarrow (r,\ \texttt{short.}s)$

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.

The devil is in the details

Several aspects of bytecode verification go beyond classic dataflow analysis:

- Interfaces:
  The subtype relation is not a semi-lattice.

- Object initialization protocol:
  Requires some must-alias analysis during verification.

- Subroutines:
  A code sharing device, requires polymorphic or polyvariant analysis (several types per program point).

(See the book *Java and the Java virtual machine* and my survey in *Journal of Automated Reasoning* 2003.)

On-card bytecode verification

The bytecode verifier is part of the Trusted Computing Base
$\rightarrow$ better to perform it on-card.

However, bytecode verification on a smart card is challenging:

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

Space: memory requirements of the standard 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.

This is too large to fit in card RAM. Fits barely in NVM.
(See Deville & Grimaud, WIESS 2002).

Solution 1: lightweight bytecode verification

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

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

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:
– Certificates are large (50% of code size).

Solution 2: restricted verification + code transformation

(Leroy, Trusted Logic.)

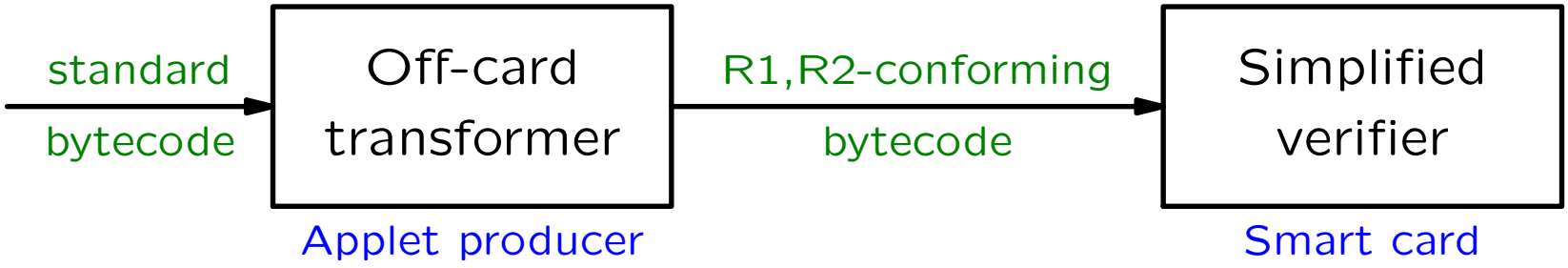
The on-card verifier puts two additional requirements on verifiable code:

– R1: The expression stack is empty at all branches.

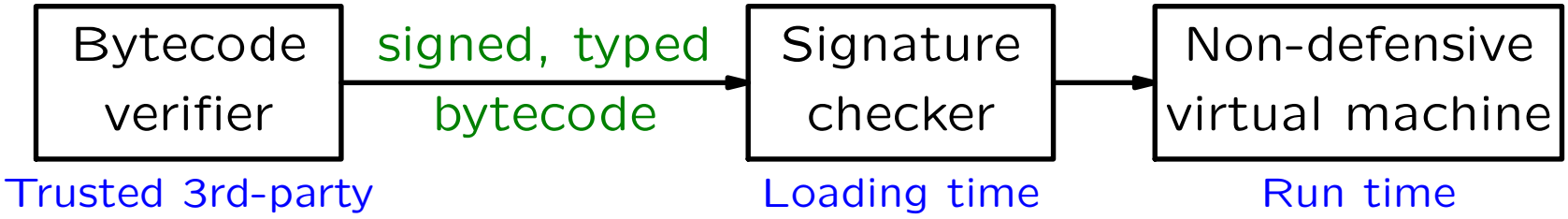
– R2: Each register has the same type throughout a method.

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

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



Solution 3: off-card verification



Bytecode verification can also be performed off-card and attested by a digital signature.

The off-card verifier can run on an Hardware Security Module.

Issues of trust and key management.

Part 3

Beyond strong typing

Enforcing functional properties beyond type safety

Strong typing + secured APIs enforce a number of desirable, basic security and reliability properties.

Can we find similar combinations of static analyses and run-time checks that enforce finer properties?

- Security properties:
  (semi-)static enforcement of firewall rules;
  richer access control policies (security automata);
  information flow and non-interference properties.

- Reliability properties:
  absence of unhandled run-time errors (security or otherwise);
  proper utilization of API functions (e.g. transactions).

- Controlling resource utilization:
  memory consumption, reactiveness, . . .

Why use static analysis?

Often, dynamic enforcement of the properties is sufficient from a security standpoint, and much easier to implement than static enforcement.

Static enforcement via program analysis is nevertheless desirable in some cases:

- For security properties that are too difficult to check at run-time.

  Example: information flow.

- For static debugging and validation: tell the programmer or the reviewer about possible coding errors.

  The concern here is both reliability and security.

Example 1: type systems for information flow analysis

The confidentiality properties enforced by access control are weak:

CEO: `access(secret), access(trashcan)`

Janitor: `access(trashcan)`

To prevent such breaches of confidentiality, attach information levels to every variable (or more generally to every piece of input and output data).

Information levels

Classic model (Bell & La Padula, Denning):

level = `H` (secret) or `L` (public), with `H` $>$ `L`.

Allow information to flow from x to y only if $\texttt{level}(x) \leq \texttt{level}(y)$.

Non-interference property: varying the values of secret inputs does not cause the values of public output to change.

JFlow model (A. Myers):

level = set of (owner, set of allowed readers).

Allow read only if reader is allowed by all owners.
Permit explicit declassification by owners.

Why dynamic enforcement of confidentiality is difficult

Idea: replace each data by pairs (data, level). Check levels dynamically at every operation.

Works fine for catching direct information flows:

```
y = x;      ---->      if (x.level <= y.level)
                          y.value = x.value;
                       else
                          throw (new SecurityException());
```

Difficult to extend to indirect information flows:

```
if (x) {
    y = true;
} else {
    y = false;
}
```

Revert to static analysis: approximate conservatively the levels attached to data, and check them statically.

Type systems for information flow

(Volpano & Smith; Heintze & Riecke; Myers; Pottier & Simonet; . . .)

Take an existing type system and annotate every type constructor with an information level, e.g.

$$\texttt{int}^H \qquad \texttt{array}^L(\texttt{int}^H)$$

Typing rules for expressions propagate levels in the natural way:

$$\frac{\Gamma \vdash e_1 : \texttt{int}^\alpha \qquad \Gamma \vdash e_2 : \texttt{int}^\beta \qquad \delta = \max(\alpha, \beta)}{\Gamma \vdash e_1 + e_2 : \texttt{int}^\delta}$$

Catching direct and indirect flows

To capture indirect flows, the typing judgement for statements $\Gamma \vdash S \texttt{ secure at } \delta$ is parameterized by an information level δ for the context.

$$\frac{\Gamma \vdash x : \tau^{\alpha} \qquad \Gamma \vdash e : \tau^{\beta} \qquad \beta \leq \alpha \qquad \delta \leq \alpha}{\Gamma \vdash (x = e) \texttt{ secure at } \delta}$$

$$\frac{\Gamma \vdash S_1 \texttt{ secure at } \delta \qquad \Gamma \vdash S_2 \texttt{ secure at } \delta}{\Gamma \vdash (S_1; S_2) \texttt{ secure at } \delta}$$

$$\frac{\Gamma \vdash e : \texttt{boolean}^{\alpha} \qquad \Gamma \vdash S \texttt{ secure at } \max(\alpha, \delta)}{\Gamma \vdash (\texttt{if } (e)\ S) \texttt{ secure at } \delta}$$

(Green: direct flow avoidance; Red: indirect flow avoidance.)

Example 2: model-checking of control-flow properties

(Jensen et al; Chugunov, Fredlund and Gurov, CARDIS'02)

An instance of D. Schmidt's slogan:

> "Data flow analysis is model checking of abstract interpretations"

Concretely:

- Abstract the flow of control as a method call graph.
- Express control properties of interest in temporal logic.

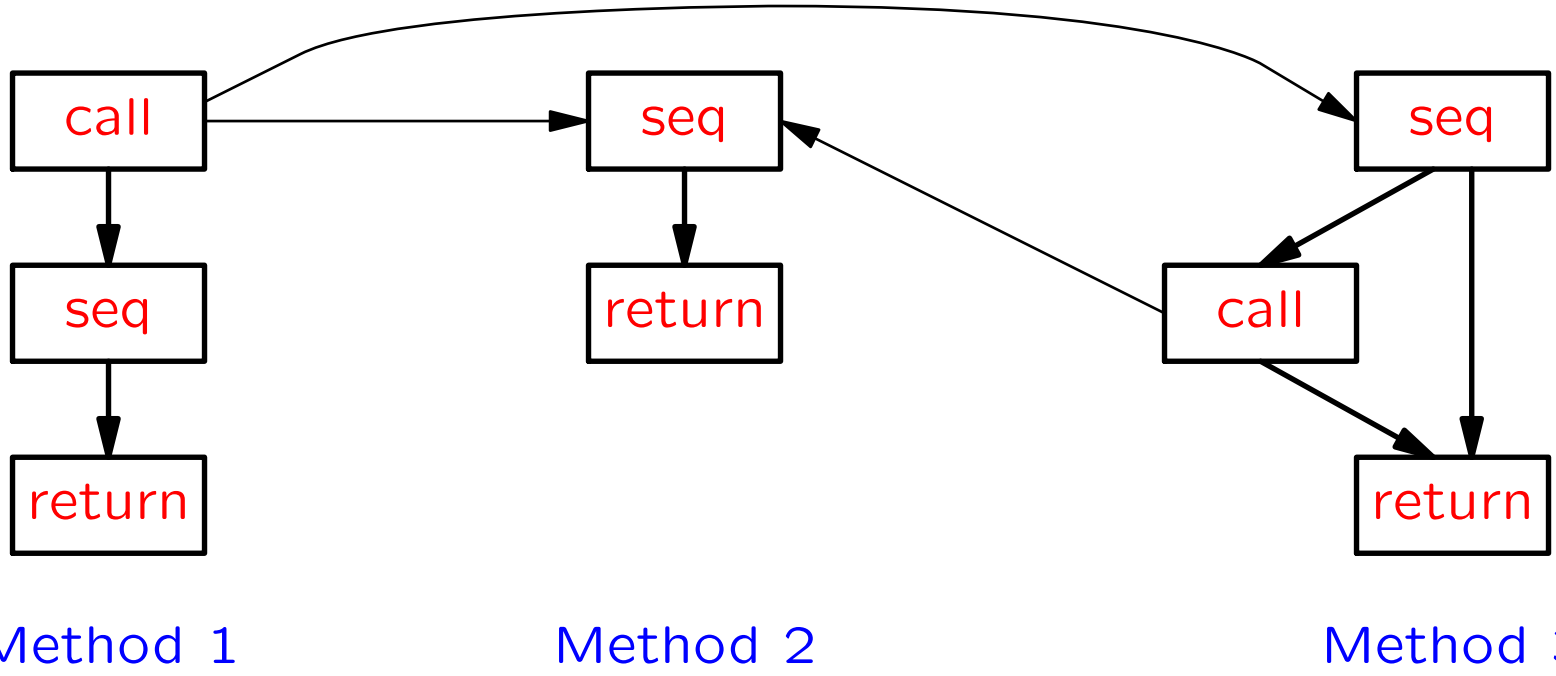
  Example: "method m_2 is always called after having called method m_1"
- Check these properties on the abstraction using model checking.

Method call graphs

Nodes: program points + annotation `seq`, `call`, `return`

Edges: intra-method branches or cross-method calls.

(Due to virtual dispatch, can have several target methods for a `call` node.)



(Semantics given in terms of pushdown systems.)

Expressing control properties

Atomic propositions: "we are at program point pc"

Connectives: those of temporal logic $\vee$, $\wedge$, $\neg$, $\Rightarrow$, `next`, `until`, `eventually`, `never`, ...

Examples of derived specification patterns:

> "we are in method m", "in package p", "in API"
>
> "m_2 can only be called after a call to m_1"
>
> "m_2 can only be called from m_1"
>
> "a call to m_1 never triggers a call to m_2"
>
> "if m_1 is called, m_2 will eventually be called later"
>
> "method m cannot be directly called from package p"

These formulae can be checked against a method call graph using standard model checkers.

Examples of properties that can be checked automatically

Basic coding conventions:

- No recursion
  "method m never triggers method m"

- No allocation during `process`
  "method `process` never triggers a call to a constructor"

- Absence of nested transactions
  "between two calls to `beginTransaction` there is always a call to `endTransaction` or to `abortTransaction`"

- Transactions are matched
  "a call to `beginTransaction` is eventually followed by a call to `endTransaction` or to `abortTransaction`"

More properties

Avoidance of unportable or dangerous constructs:

- No allocation within a transaction.

Conformance with high-level application policies:

- `grantLoyaltyPoints` is only called from `debitPurse`.
- `cashLoyaltyPoints` never triggers `grantLoyaltyPoints`.

Conclusions

The "success story" of strong typing

Strong typing as embodied by the Java Card virtual machine has been relatively successful security-wise:

- VM integrity without hardware protection.
- Basic confinement properties. APIs can take advantage of them.
- On-card enforcement is relatively easy and inexpensive.

Can we repeat this success for other properties?

Other static analyses could be applied to increase security and reliability of Java Card code.

Some practical issues to keep in mind:

- No consensus on the properties of interest.
- Complex static analyses, often specialized.
- Too expensive to be performed on-card or in a HSM.
- Not obvious how to apply them to bytecode.
- Not obvious how to generate checkable certificates.