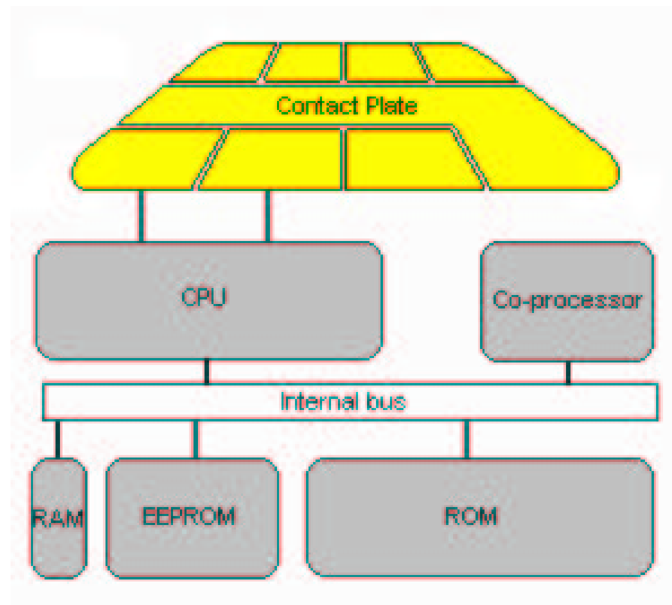
Smart card security

from a programming language and static analysis perspective

Xavier Leroy

INRIA Rocquencourt & Trusted Logic

Smart cards



- A small embedded computer.
  Shaped as a credit card or a SIM card.
  Low processing power (8-bit CPU, 5 MHz clock).
  Small memory (4 Kb RAM, 16 Kb EEPROM, 64 Kb ROM).
- Secure (tamper-resistant).
- Inexpensive (average 3 Euros).

Using smart cards as security tokens

- Authentication of the card holder:

  To have (the card) and *to know* (a PIN code).

  Credit cards; SIM cards for GSM; pay TV; electronic locks.

- Storing sensitive information:

  Credit cards: number, expiration date, transaction log, . . .

  Phone book in SIM cards.

  Medical data.

Smart card software

Traditional smart cards:

- One card = one function.
- Software is in ROM, cannot be updated.
- Propietary software, developed by card manufacturer.
- Written in C or in assembly.

Modern smart cards: (Java Card, MultOS)

- Multiple applications, with controlled sharing of information.
- Post-issuance downloading of applications (*cardlets*).
- Software written in a subset of Java against a standard API.
- Often written by others than the card manufacturer.

Risks of applet-based architectures

Post-issuance downloading of cardlets brings a lot of flexibility, but raises significant security issues.

A malicious cardlet can do a lot of harm:

- Destroy or modify important data (integrity).
- Leak sensitive information outside (confidentiality).
- Cause other card applications to malfunction (availability).

(Same problems with Web applets and mobile code in general.)

Digital signatures on cardlets is not a complete solution.
(The signature guarantees the origin of the cardlet, not that it is innocuous.)

An example of a malicious cardlet

The "memory dump" cardlet.

```
class MaliciousCardlet {
  public void process(APDU a) {
    for (short i = 0; i <= 0xFFF8; i += 2) {
      byte [] b = (byte []) i;
      send_byte((byte) (b.length >> 8));
      send_byte((byte) (b.length & 0xFF));
    }
  }
}
```

Sends the whole contents of the card memory to the terminal.

The Java “sandbox” model for software isolation

Untrusted code is not executed directly on the hardware, but via a software isolation layer (the sandbox):

1. Secured runtime execution environment (applet API). Provides controlled access to system resources (files, communication devices, etc).

2. Untrusted code is executed by a defensive virtual machine that ensures 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.

Part 1

Securing the execution environment

Access control in full Java: the security manager

Each method has an associated set of privileges (capabilities).
Depends on the classloader used to load the code: system code has full privileges, Web applets have low privileges.

Stack inspection: when a sensitive operation is attempted, take the intersection of the privileges of all methods on the call stack.
(System code (API methods) called by an applet thus have applet privileges only.)

Privilege amplification: a method can explicitly request to run with its full privileges, even if called from less privileged code.
(Example: drawing text on screen may require reading font files.)

Access control in Java Card: the firewall

A simplified security model, without stack inspection, but with stronger isolation guarantees.

Each object is owned by the principal (cardlet or system) that created it.

Firewall rules in the virtual machine prevent a cardlet from accessing an object that it does not own...

...except for explicitly-designated shareable objects, where interface method invocation is allowed.

Interface and virtual method invocation causes a context switch: the method code is executed with the privileges of the owner of the object.

Static method invocation preserves the context: the code is executed with the privileges of the caller.

Using these access control mechanisms

Both Java's security manager and Java Card's firewall are low-level mechanisms to implement high-level security policies in runtime environments and in applets, such as:

> "Applets cannot open files and cannot make network connections to any host other than their originating site."

> "The field `balance` of class `ElectronicPurse` cannot be affected by the execution of any other cardlet."

There is a semantic gap between the policies and the mechanisms. For instance, stack inspection doesn't always do what the API programmer had in mind.

Example of an API security hole

Suddenly, you Netscape 4 browser turns into a Web server...

In the API:

```
class SocketServer {
  protected final void implAccept(Socket socket) {
    try {
      // accept network connection and bind it to socket
      securityManager.checkAccept(socket.getHostAddress(), socket.getPort());
    } catch (SecurityException e) {
      socket.close();
    }
  }
```

Malicious applet subclasses `Socket` so that the connection stays open even after a security exception:

```
class EvilSocket extends Socket {
  public void close() { // do nothing }
}
```

Semantics and static analysis to the rescue

Formal semantics and equational theory for stack inspection (Gordon and Fournet),

Static analyses of the Java "stack inspection" security policy:
By abstract interpretation of call stacks + model checking (Jensen et al);
By constraint-based type-checking (Pottier, Skalka, Smith).

Static analysis of the JavaCard "firewall" security policy:
By abstract interpretation and model checking (Chugunov et al).

Part 2

Ensuring type safety

The need for type safety

There are many ways in which type-unsafe code can circumvent the access control mechanisms implemented in the API and the firewall:

- 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`.

The need for type safety

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

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

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

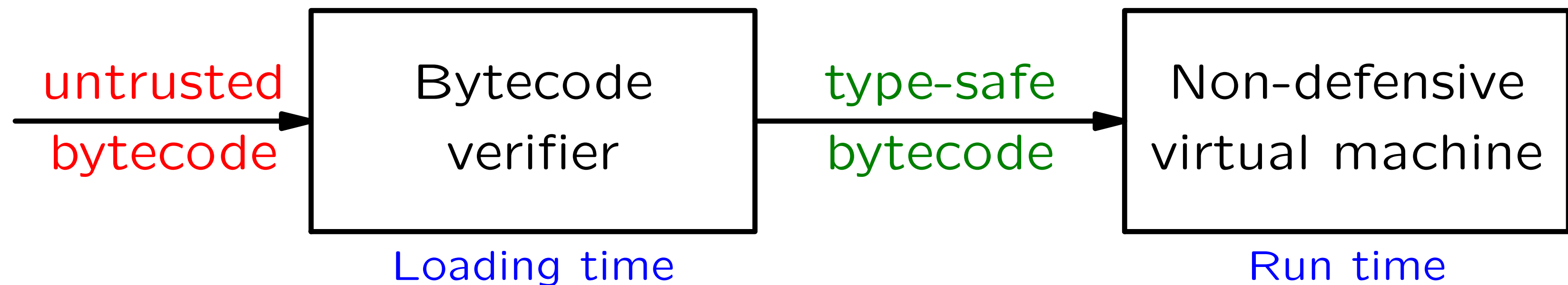
- and much more.

Type safety: defensive VM versus bytecode verification

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

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

2. Bytecode verification at loading time:
   A separate static code analysis establishes type safety.
   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:

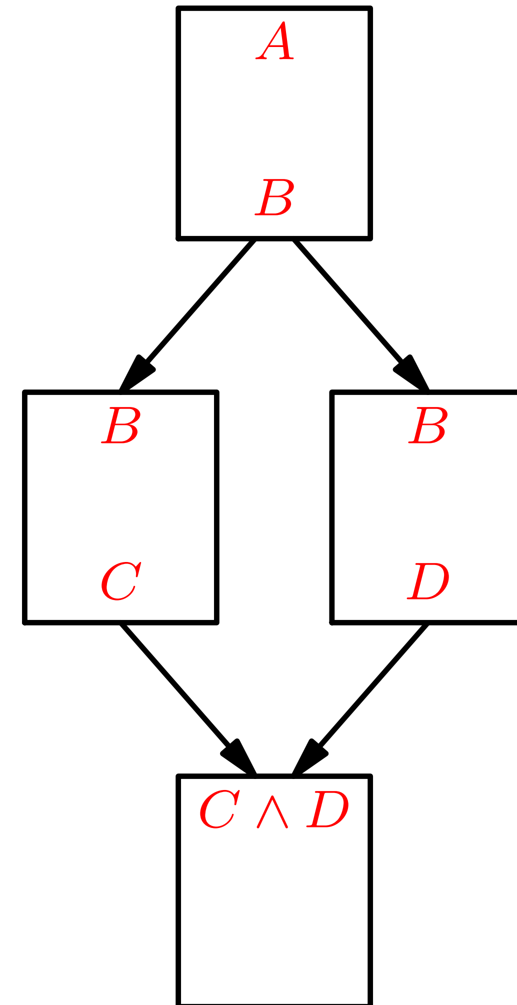
```
class C {
    int x;
    void move(int delta) {
        int oldx = x;
        x += delta;
        D.draw(oldx, x);
    }
}
```

```
                                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 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. $\texttt{iadd} : (r,\ \texttt{int.int.}s) \rightarrow (r,\ \texttt{int.}s)$

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.

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 a bit of must-alias analysis during verification.
- Subroutines:
  A code sharing device, requires polymorphic / polyvariant analysis (several types per program point).

In addition, bytecode verification is not 100% specified.
Informal description; one reference implementation; many after-the-fact formalizations that don't fully agree.

Bytecode verification on small devices

Bytecode verification on a smart card is challenging:

- Time: complex process, e.g. fixpoint iteration.
- Space: the 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 RAM.

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 (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.)

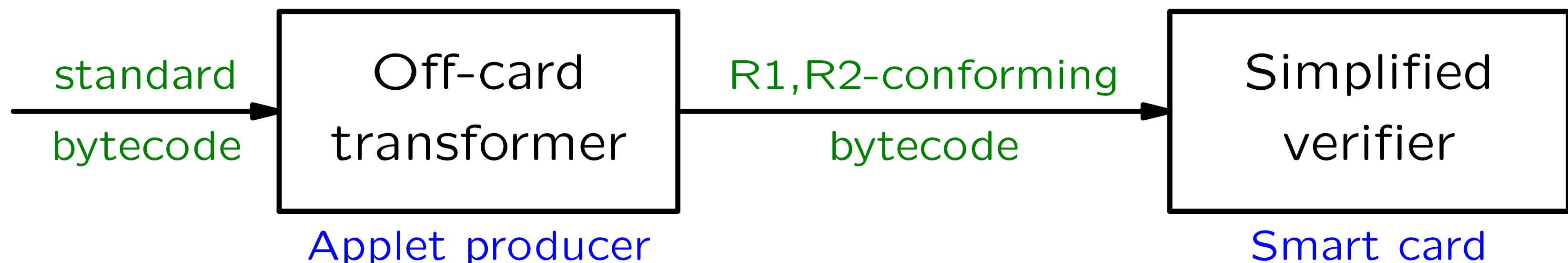
The on-card verifier puts two additional requirements on verifiable code:

– 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 ⇒ low memory $3 \times (M_{stack} + M_{regs})$.

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



Formal methods applied to Java bytecode verification

Specifications and (machine) proofs of type soundness.
(Nipkow; many others.)

$$\text{bytecode verifier} + \text{non-defensive VM} \geq \text{defensive VM}$$

Systematic derivation of bytecode verifiers and non-defensive VM from a defensive VM.
(Barthes et al.)

Beyond bytecode verification

Typed Assembly Language:

Static type-checking of x86 assembly code, including advanced idioms (Morrisett et al).

Typing legacy C code:

static debugging of buffer overflows, illegal casts, memory management errors.

Typing with dependent types:

Static checking of array bounds, and more (Xi, Shao, Crary, et al).

Proof-carrying code:

Replace type-checking by proof-checking. The code is accompanied by a certificate that is a proof of correctness w.r.t. an arbitrary safety policy (Lee and Necula).

Part 3

Physical attacks on smart cards

Physical attacks on smart cards

Unlike other secure computers, smart cards are physically in the hands of the attacker.

- Observation: observe power consumption and electromagnetic emissions as a function of time.
- Invasion: expose the chip and implant micro-electrodes on data paths.
- Temporary perturbations: glitches on power supply or external clock; flash it with high-energy radiations.
- Permanent modifications: destroy connections and transistors; grow back fuses.

Effect of these attacks: sometimes, read directly secret information; more often: cause the program to mal-function and reveal a secret or grant a permission.

Examples of hardware attacks

```
if (permission_check) { do_privileged_action(); }
```

Invalidate the test on `permission_check`, or modify its boolean value.

```
for (p = buffer;  p != buffer + length;  p++) {
  output_on_serial_port(*p);
}
```

Invalidate the stop condition, or modify the current value of `p` → dump the whole memory on the serial port.

```
r = 1
for each bit b in secret RSA key d {
  r = r * r mod pq;
  if (b is set) r = r * m mod pq;
}
```

By correlating the power consumption with the message `m`, it is possible to reconstruct the bits of the key `d`.

Counter-measures

Hardware countermeasures against these attacks exist: protection layers, obfuscation of the chip layout, encryption of the memory bus, hardware memory access control, . . .

Amazingly, software can also be hardened (to some extent) against hardware attacks.

- Destructive attacks: precise, but not reversible
  → periodic self-tests; redundant storing of data.

- Perturbation attacks: temporary, but imprecise
  → redundancy within data (checksums) and between data and control.

- Observation attacks:
  → randomized execution.

Examples of software counter-measures

Redundancy between control and data:

```
trace = 0;
if (! condition1) goto error;
trace |= 1;
if (! condition2) goto error;
trace |= 2;
sign_transaction_certificate(cert, key + trace - 3);
```

Doubly-counted loops:

```
for (p = buffer, i = 0, j = length;  p != buffer + length;  p++) {
  if (i >= length || j <= 0 || i + j != length) halt();
  output_on_serial_port(*p);
}
```

Examples of software counter-measures

Randomized control:

```
if (random_bit()) {
    do_something();
    do_something_else();
} else {
    do_something_else();
    do_something();
}
```

Randomized data (RSA blinding):

```
blinding = random number relatively prime to d;
m = m * blinding;
r = 1
for each bit b in secret RSA key d {
  r = r * r mod pq;
  if (b is set) r = r * m mod pq;
}
r = r * blinding^-e mod pq;
```

Reasoning about hardware attacks and counter-measures

Timing and power analysis attacks have been studied extensively in the context of traditional security and of cryptography.

A lot of semantic and programming language work remains to be done:

- Develop probabilistic semantics that reflect the characteristics of hardware attacks

  (precise + irreversible or imprecise + temporary).

- Use these semantics to reason about software counter-measures.

  (Is it the case that the hardened schemes outlined above increase the probability of failing cleanly in the presence of an attack?)

- Systematize software hardening schemes and implement them as (semi-)automatic program transformations.

  (A form of aspect-oriented programming?)

Part 4

Conclusions

Building a secure computer application

Generalizing from the smart card examples, we see a three-part process:

1. Design appropriate security policy.
   Requires security experts and domain experts.
   Not computer-specific.

2. Implement it as a correct program.
   "Business as usual" for us software people:
   specification, programming, testing, verification, . . .

3. Protect against every way in which the security mechanisms could be circumvented.
   Attackers do think out of the box.

"Programming Satan's computer" (Anderson & Needham).

What I learned

Security is a "holistic" property that cannot be completely reduced to independent sub-problems.

Still, software techniques are relevant to computer security:

- Programming languages and static analysis (this talk);
- Applied π-calculus for cryptographic protocols;
- Formal methods in general.

Hope: semantics can help gain a better understanding of the security benefits and risks associated with various software practices.