Language-based security for mobile code with applications to smart cards

Xavier Leroy

INRIA Rocquencourt & (formerly) Trusted Logic S.A.

This lecture

An exploration of several topics at the border between security and programming languages.

Or: security from the perspective of a programming languages and semantics person.

Builds on / generalizes from two historical examples:

- Web applets;
- Java smart cards.

Outline

Language-based security for mobile code:

1. The sandbox model

   (how to execute untrusted code securely)

2. Access control in API

   (Java stack inspection)

3. Security implications of type safety

   (or, ensuring that API access control is not bypassed)

4. Bytecode verification

   (ensuring type safety of untrusted compiled code)

5. Information flow analysis

   (another security-relevant static analysis)

Outline

Smart cards:

1. Overview of smart cards

   (what they are, what they are used for, how they are programmed)

2. Adapting the Java security model to Java Card

   (or, Java in the very small)

3. Hardware attacks and countermeasures

   (tamper-resistance)

The sandbox model
for execution of untrusted code

Automatic execution of untrusted code

Idea: enhance user experience by automatically downloading and executing code embedded in various documents.

First published instance: N. Borenstein, *E-mail with a mind of its own*, 1994. (MIME mail containing TCL scripts.)

This idea became famous circa 1995 with Java Web applets (Java programs embedded in Web pages).

Applets today

Java Web applets have nearly disappeared, but automatic execution of untrusted code has become ubiquitous in desktop computing:

- In Web pages:
  JavaScript, Macromedia Flash, ActiveX components, . . .

- In other documents:
  Word and Excel macros, Postscript documents, . . .

Applet-style downloading of applications is also becoming popular in certain embedded systems:

- Mobile phones (Java “midlets” for MIDP-enabled phones).

- Smart cards (Java Card).

Security risks associated with applets

A malicious applet running on your PC, mobile phone or smart card can do a lot of harm:

- Destroy files and applications

  e.g. reformat hard drive

- Divulge confidential information

  financial info, address book, Web navigation history, identity theft

- Perform actions on your behalf

  financial transactions, spam relaying, DDOS attacks

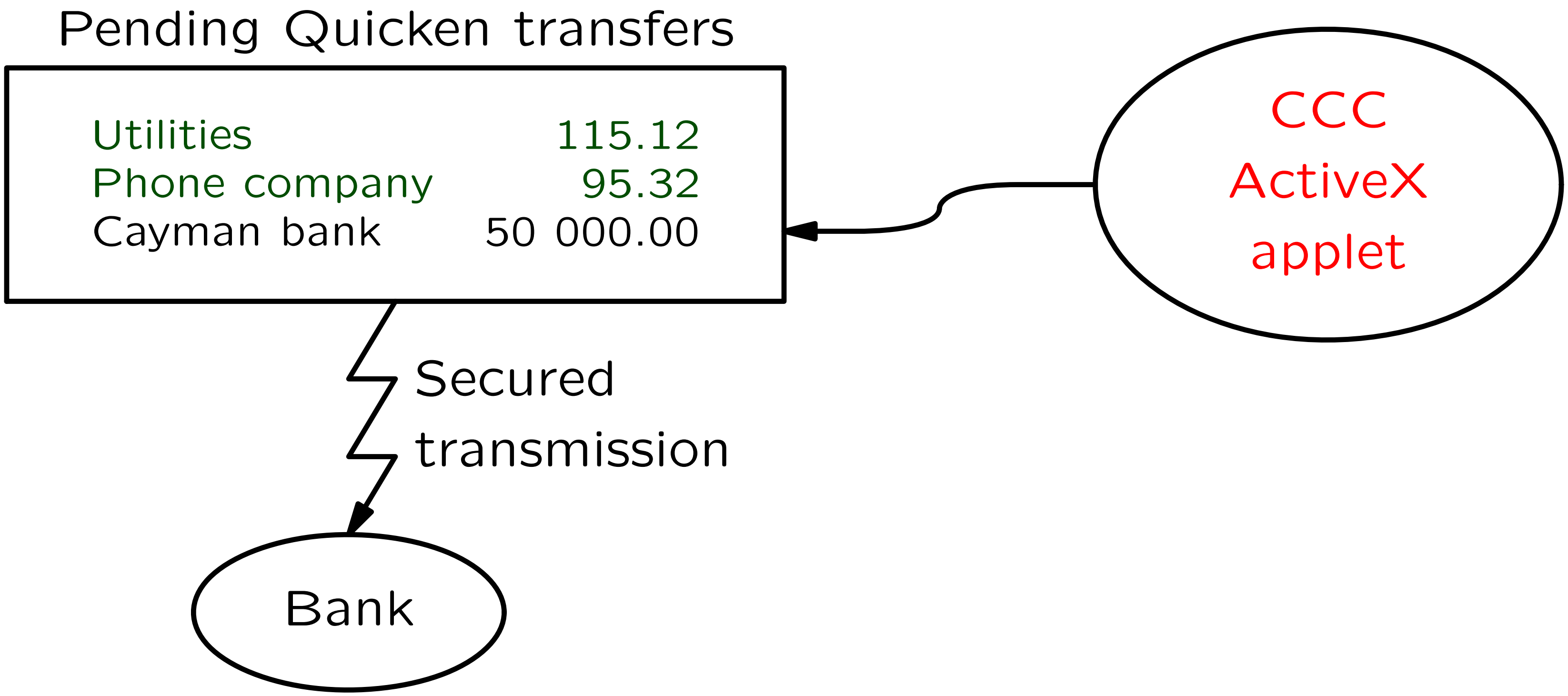
- Render other applications unsecure

  Web page spoofing, installation of spyware

The cracker's dream (remote code execution) made true?

An example of a malicious applet

(Demo attack on ActiveX, Chaos Computer Club, 1997.)



Pending Quicken transfers

Utilities	115.12
Phone company	95.32
Cayman bank	50 000.00

CCC ActiveX applet

Secured transmission

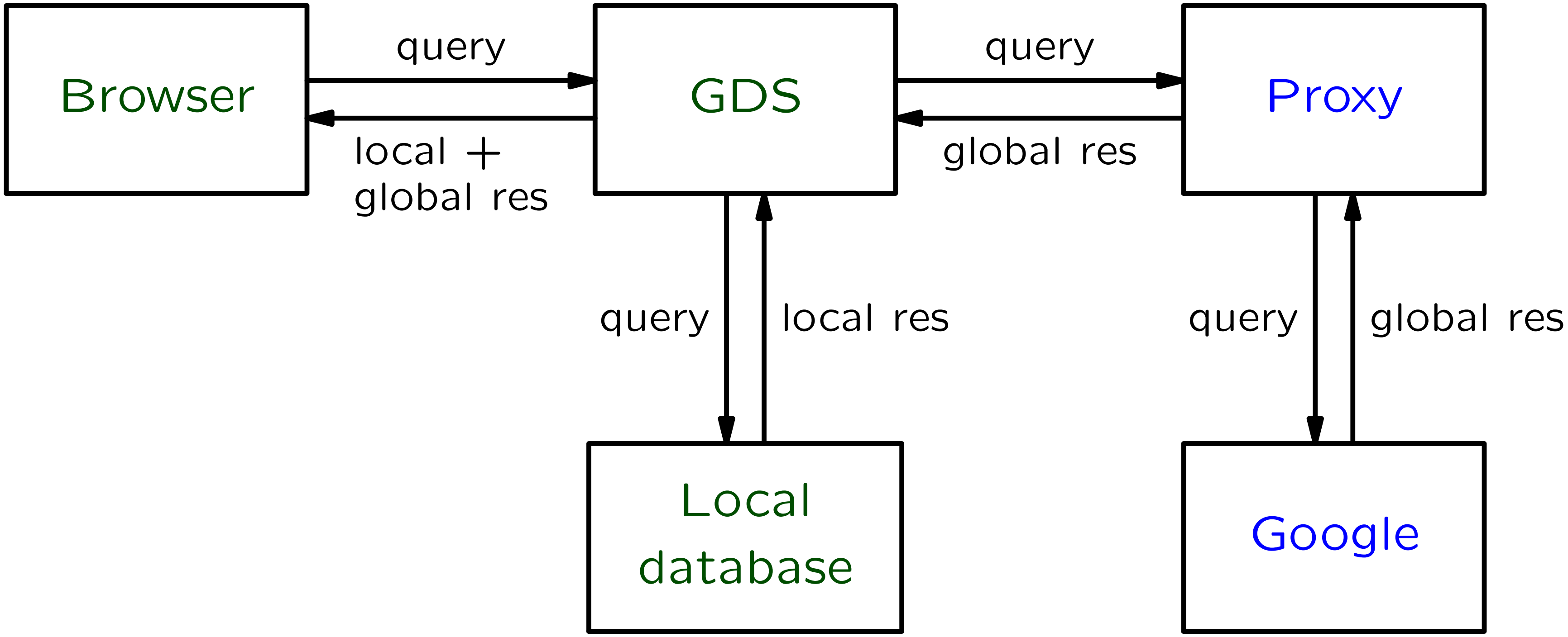
Bank

One file access can be very costly . . .

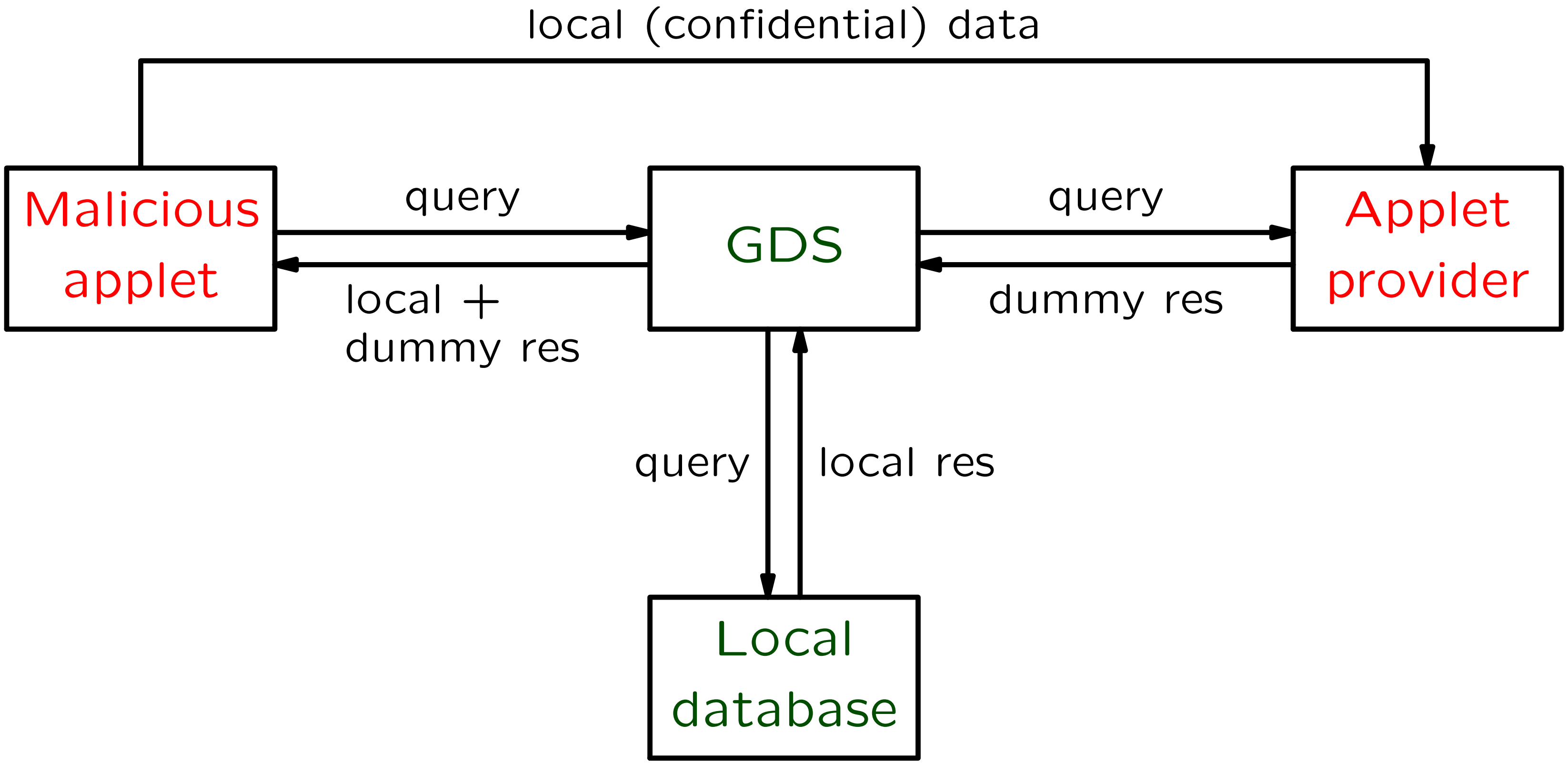
An attack on Google Desktop Search

(Nielson, Fogarty & Wallach, 11/2004, fixed 12/2004).

GDS = local indexing of personal files, integrate results of local searches into results of Google searches.

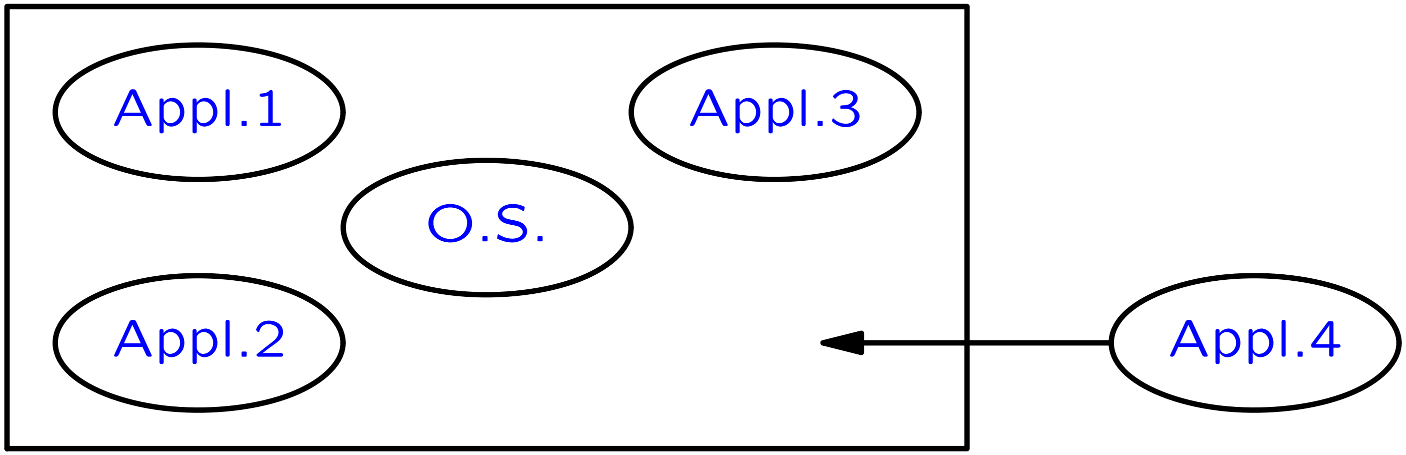


GDS + malicious applet = information leak



Note: the default Java security model for applets allow them to make network connections to their originating host.

The general context



A secure system running multiple applications, possibly untrusted:

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

Between applications, must ensure isolation (integrity, confidentiality) and controlled communications.

Enforcing isolation between applications

The classic solution, as found in multiuser operating systems, is to exploit hardware protection mechanisms:

- Hardware access control on memory pages and devices.
- Forces dangerous operations to go through the OS.

This model has known limitations:

- Not always available:

  small embedded systems lack hardware memory protections and user-mode/kernel-mode distinction.

- Often too coarse:

  e.g. hard to create a new Unix user for every Web applet.

- Sometimes too slow:

  e.g. run dubious code in the kernel to avoid the cost of context switches.

Software-only isolation: the sandbox model

Java applets have revived another model for software isolation:

Do not run untrusted applications on the bare hardware.

Instead, execute them via a software isolation layer (sandbox), consisting of:

- a virtual machine that enforces type- and memory-safety;
- native APIs that perform access control.

Rely on safety properties of the programming language and its execution environment to ensure that this isolation layer is not bypassed.

Access control in API

The Java "SecurityManager" model

(Similar model used in C#)

Each method has an associated set of permissions (capabilities):

- File permissions: read, write, execute, delete (per file).
- Socket permissions: connect, accept connections (per host).
- Runtime permissions: exit VM, load native code, define own class loader, define own security manager, . . .
- GUI permissions: access clipboard, read pixels on screen, ...
- . . . and much more.

Associating permissions to methods

In Java 1: the permissions depends on the classloader used to load the code.

- Code loaded from local files: full permissions.
- Applet loaded from the network:
  no file accesses;
  network connections with originating host only;
  no loading of native code;
  etc.

In Java 2: cryptographically signed applets can receive additional permissions. This is governed by locally-managed resource files.

Checking permissions

API methods call `SecurityManager.checkPermission` and its variants (`checkRead`, `checkAccept`, etc) before attempting a sensitive operation.

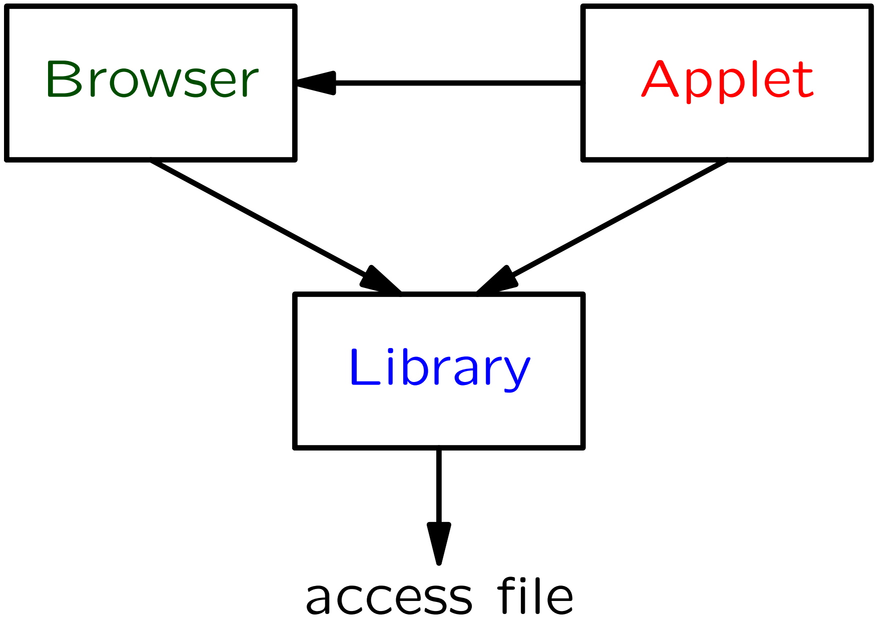
If the current method and all its calling methods possess the permission, `checkPermission` returns normally (access granted).

If one of the callers does not have the permission, `checkPermission` raises a `SecurityException` exception (access denied).

Call stack inspection

The permission check on the current method and all its callers is called stack inspection.

It allows libraries to behave differently depending on whether they execute on behalf of trusted code (the browser) or untrusted code (an applet).



Browser → Library:
access granted.

Applet → Library:
access denied.

Applet → Browser → Library:
access denied.

Privilege amplification

Sometimes, a library may need to perform a sensitive operation regardless of the caller.

Example: drawing text in a GUI can require reading fonts from local files – even if called from an applet.

The font loading code can call `doPrivileged` to request execution with its own privileges, even if called from less privileged code.

```
void loadFontFile(String name) {
    // safety checks
    AccessController.doPrivileged(new PrivilegedAction() {
        public Object run() {
            // open and read font file
            return null;
        }
    });
}
```

The permission checking algorithm

```
Repeat
    Check if current method has requested permission.
        If not, throw SecurityException.
    Check if current method has amplified privileges.
        If so, grant permission.
    Consider calling method (move up call stack).
Until call stack is empty.

Check if thread inherited the requested permission.
    If not, throw SecurityException.
    If so, grant permission.
```

The “confused deputy” problem

An untrusted principal coaxes a trusted principal into performing a sensitive operation on its behalf.

Stack inspection protects the callee from the caller: it addresses the case where the untrusted principal calls the trusted principal.

It offers no such protection of the caller from the callee, as can happen in callback- or event-based architectures.

Examples of confused deputies

(Abadi and Fournet, 2003)

```
class Trusted {
  // full privileges
  static void main() {
    String s = BadPlugin.tempfile();
    File.delete(s);
  }
}

class BadPlugin {
  // low privileges
  static String tempfile() {
    return "/etc/passwd";
  }
}
```

```
abstract class Trusted {
  // full privileges
  String tempfile = "/tmp/tempfile";
  abstract void proceed();
  void main() {
    try {
      proceed();
    } catch (Exception e) {
      File.delete(tempfile); throw(e);
    }
  }
}

class BadPlugin extends Trusted {
  // low privileges
  void proceed() {
    tempfile = "/etc/passwd";
    throw new Exception();
  }
}
```

Access control based on execution history

(Abadi and Fournet, 2003)

As an alternative to stack inspection, we can also accumulate in a per-thread variable the intersection of the permissions of all methods invoked since start of thread, and base permission checks on this intersection.

→ Protects caller from callee as well as callee from caller.

From security mechanisms to security policies

Stack inspection or execution histories do not guarantee security by themselves: they are (powerful) mechanisms that can be exploited to implement the desired security policies.

The difficulty is to ensure that local checks (mechanisms) actually implement the desired global security policy.

Some common pitfalls:

1. Mechanisms are too coarse to express desired policy.
2. Mechanisms are incorrectly used

   (missing `checkPermission` or excessive `doPrivileged` in API).
3. Mechanisms are so extensible that attacker can circumvent them.

Security policies that cannot be expressed precisely

Consider the following policy:

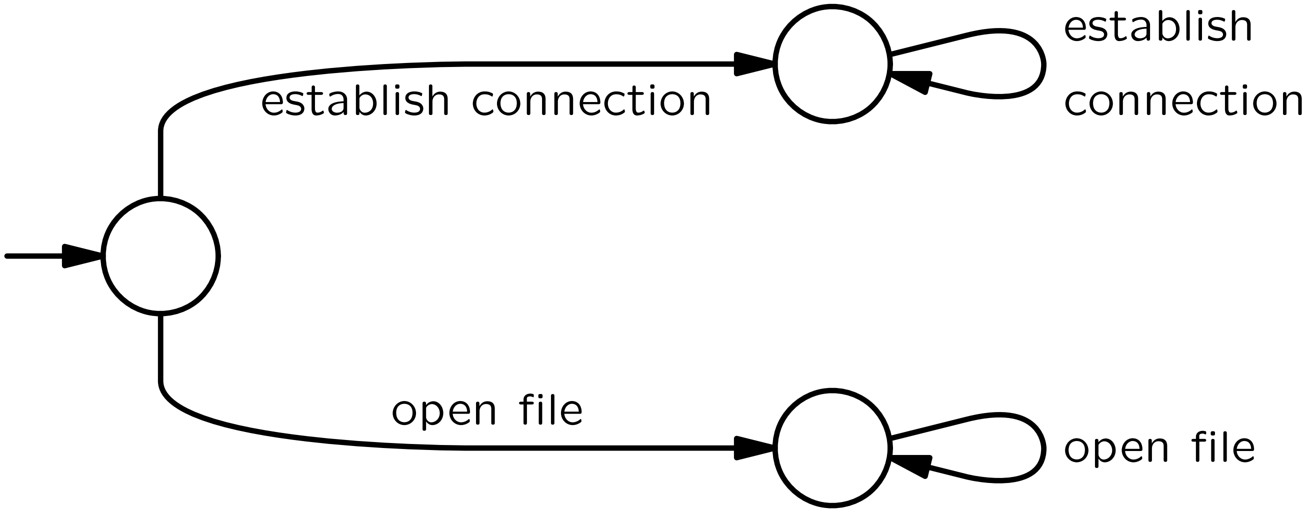
> An applet can read local files or establish network connections, but not both.

(Useful e.g. to view local images with an applet.)

This policy cannot be enforced by stack inspection.

Security automata (F. Scheider)

A generalization of access control where decisions depend on previous actions, as summarized in the current state of an automaton.



Permission is granted iff automaton can make a transition from current state on requested action.

Extensibility versus security

There is a tension between

- Hard-wired, established security mechanisms that cannot be circumvented.
- The desire to allow programmers to customize security mechanisms and other security-sensitive API services.

Object-oriented languages have a tradition of extensibility (via subclassing and method overriding) that can raise security challenges.

Example of an attack by subclassing

(D. Brumleve, 1998)

A Java applet that turns a Netscape 4 browser into a Web server serving all local files for the world to see.

Violates the security policy for Java applets in two ways:

- Reads local files.

  Via a bug in the way Netscape handled `file:///` URLs.

- Accepts network connections from any host.

  Via clever use of subclassing.

Analysis of the second security hole

In the `SocketServer` class of the Netscape Java API:

```
protected final void implAccept(Socket socket) {
    try {
        ...
        impl.accept(socket.impl);   // OS-level accept connection
        sm = System.getSecurityManager();
        if (sm != null) {
            // Check that connecting host is authorized
            sm.checkAccept(...,
              socket.getHostAddress(),
              socket.getPort());
        }
        return;
    } catch (SecurityException e) {
       // close socket connected to non-authorized host
        socket.close();
        ...
    }
}
```

The attacker can subclass the Socket class . . .

. . . providing his own `close` method:

```
public class EvilSocket extends Socket {

    public void close() {
        // do nothing
    }

}
```

. . . and recover from failed permission check

```
class EvilServer extends java.net.ServerSocket {

    Socket accept() {
        Socket s = new EvilSocket();
        try {
            implAccept(s);
        } catch (SecurityException e) {
            // do nothing
        }
        return s;
    }

}
```

Access control in the API is completely bypassed.

Static analysis of API access control checks

(Thomas Jensen et al, 2001)

Detect potential security holes in APIs using a combination of model checking and abstract interpretation:

- Represent all possible execution paths
  as a finite method call graph.

- Express control properties of interest in temporal logic.

  "if method m_2 is called, method m_1 must have been called before"

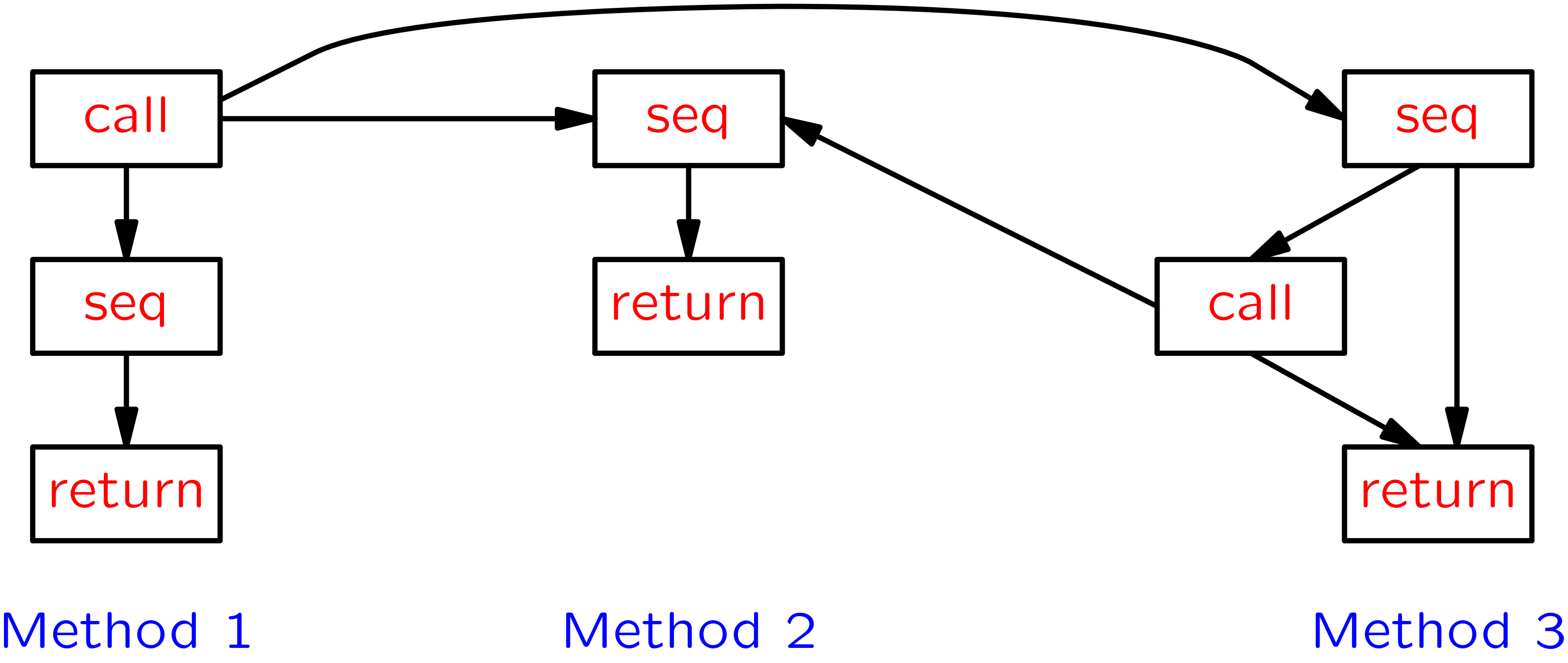
- Check these properties on the abstraction
  using model checking.

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.)

Checking properties of the call graph

Using temporal logic, we can express properties such as

> "a method from applet A cannot call a method from applet B"
>
> "`checkAccept` is always called and succeeds before we call `OS.accept`"
>
> "if `checkAccept` fails, we will eventually call `OS.close`"

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

Summary

Access control at the level of APIs is attractive because it is inexpensive, flexible, and has easy access to a lot of context (call stack, execution history) on which to base its decisions.

It is also quite error-prone and hard to combine with OO-style software extensibility.

Static analyses and formal methods in general can help hardening API access control.

The role of strong typing

Circumventing API access control

Example: we want to read a file which the file I/O API doesn't allow us to read.

- Jump in the middle of the API `readfile` function, skipping over the access control checks.
- Or, generate machine code for a kernel syscall in a byte array, then jump to it.
- Or, peek and poke bytes in the IDE hard drive controller.
- Or, modify our stack frames so that we look like privileged code.
- . . . and much more.

(Keep in mind: attackers do think outside of the box. . .)

The “memory dump” cardlet

On small embedded systems such as smart cards, a malicious cardlet could just send the whole contents of memory over the serial link → all card secrets (keys, PINs, etc) are in the open.

```
class MaliciousCardlet {
  public void process(APDU a) {
    for (short i = -0x8000; i <= 0x7FF8; i += 2) {
      byte [] b = (byte []) i;
      send_byte((byte) (b.length >> 8));
      send_byte((byte) (b.length & 0xFF));
    }
  }
}
```

(Assumes the length of an array is stored as the first 2 bytes in the array.)

Strong typing to the rescue

Strong typing is a feature of programming languages that enforces basic correctness properties on how the program manipulates data:

- An integer is not an object reference (pointer).
- A byte array is not a function.
- An array or string cannot be accessed beyond its bounds.
- Variables are initialized before being read.
- . . . and more.

Can be enforced dynamically (ill-typed program halts) and/or statically (ill-typed program is rejected at compile-time).

Strong typing for software reliability and security

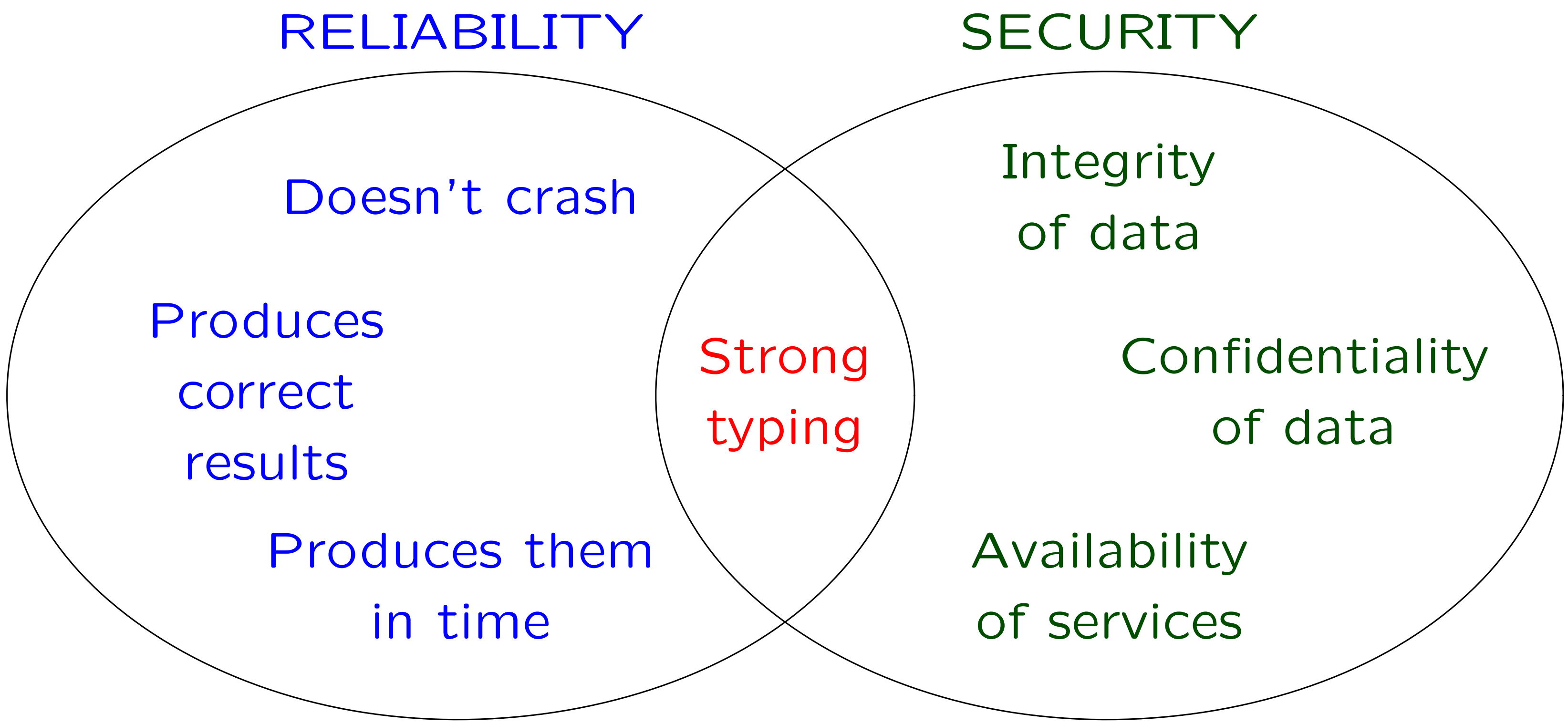
Strong typing has proved successful to increase program reliability (catches all sorts of programming bugs).

→ Strong typing avoids “undefined behaviors”, i.e. crashes.

As we shall now see, strong typing is also very effective at increasing program security.

→ Strong typing avoids “undefined behaviors”, i.e. attacks.

The reliability-security spectrum



RELIABILITY

SECURITY

Doesn't crash

Produces correct results

Produces them in time

Strong typing

Integrity of data

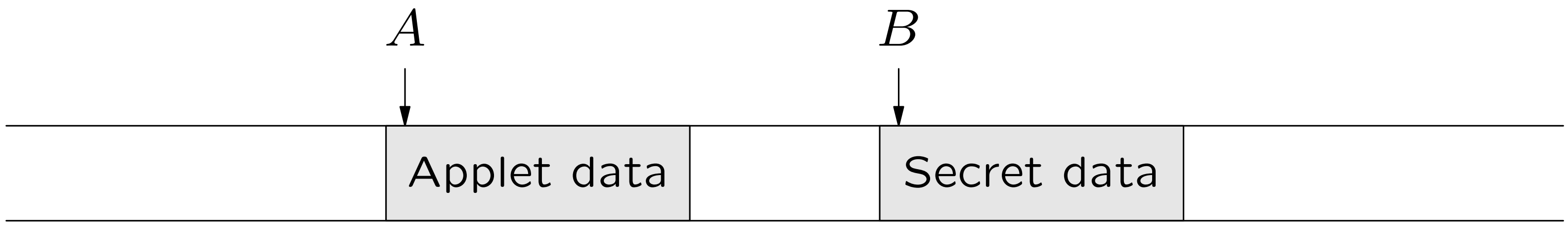
Confidentiality of data

Availability of services

Works well in normal usage conditions

Holds well against attacks

Insecurities caused by lack of strong typing



Many ways in which an ill-typed code can get access to B:

- Pointer forging:

  E.g. via pointer arithmetic `(byte [])((int)`A` + 200)`.

- Out-of-bounds access:

  Use $A[n]$ where n is large enough to hit B.

- Explicit deallocation:

  Free the block A, keep the reference A around, wait until memory manager moves B into A's spot.

More insecurities caused by lack of strong typing

- Illegal casts between unrelated classes:

  Casting from `class C { int x; }` to `class D { byte[] a; }` causes pointer `a` to be forged from integer `x`.

  Casting from `class C { private int x; }` to `class D { public int x; }` gives access to private data.

- V-tables hacking (method dispatch tables):

  Writing well-chosen integers or references in vtables allows arbitrary code to be executed.

- Stack smashing:

  Overflow stack-allocated buffer, replacing return address to caller by a pointer to malicious code.

  Change return addresses to fool stack inspection.

Security offered by strong typing?

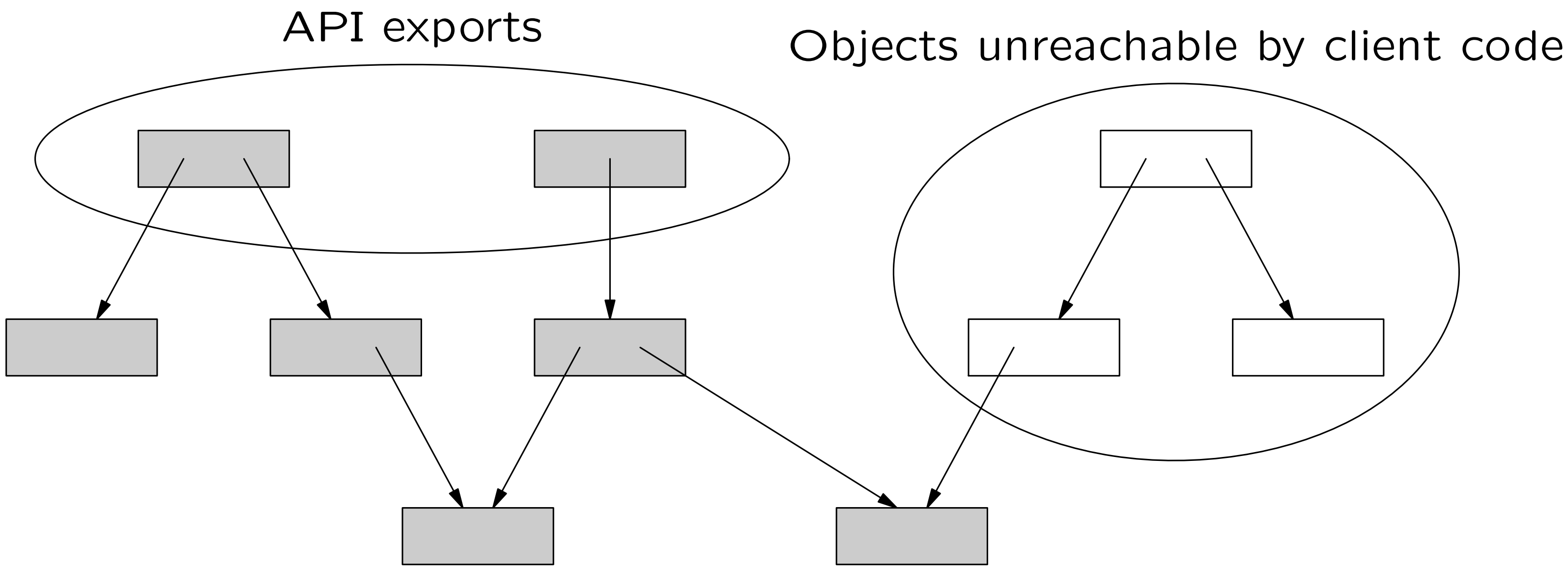
It is much harder to characterize formally the security guarantees implied by strong typing, and understand how to take advantage of these guarantees while writing a secure API.

(The standard characterization of type safety, "well-typed programs do not go wrong", is useless: malicious programs do not go wrong either.)

Three broad families of type-based guarantees:

1. Reachability (GC safety)
2. Procedural encapsulation (cannot access some fields)
3. Type abstraction (cannot construct some objects)

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 (private fields)

```
public class Uid {
    static private int id;
    static public void get() {
        return id;
    }
    static public void set(int new_id) {
        if (id == 0)
            id = new_id;
        else
            throw new SecurityException();
    }
}
```

Although formally reachable from `Uid`, the instance field `Uid.id` can only be accessed by the methods of the `Uid` class.

If `Uid.id` $\neq 0$ initially, the client code cannot modify `Uid.id`.

The `Uid.get` and `Uid.set` methods encapsulate the resource `Uid.id` and perform access control.

Type abstraction (private constructors)

```
public class Capability {
    private int capabilities;

    packageprivate Capability(int c)
        { capabilities = c; }

    public boolean test(int capnum)
        { return capabilities & (1 << n) != 0; }

    public Capability drop(int capnum)
        { return new Capability (capabilities & ˜(1 << n)); }
}
```

Client code (outside the defining package) cannot construct directly objects of type `Capability` and must go through the public method `drop` to do so. Thus, it cannot acquire capabilities that it did not initially have.

Summary

Strong typing is an essential ingredient of the sandbox model: it ensures that API-level access control cannot be circumvented.

Additionally, strong tying can be leveraged to build secure APIs: procedural encapsulation, type abstraction, etc, are some “design patterns” for secure APIs.

One question remains:
how to make sure that untrusted client code is strongly typed?

Bytecode verification

Enforcement of type safety

Traditionally, type-safe execution of a program can be achieved in two ways:

1. Dynamic typing: check types during execution; tag data with their types.

   Example: when evaluating `x + y`, check that `x` and `y` contain integers; compute sum and tag it as integer.

2. Static typing: check types at compile-time, using static analysis; execution proceeds without type tests, on untagged data.

   Example: When compiling `x + y`, check that `x` and `y` have been declared as integers; record that this expression has type `int`.

Type safety for mobile code

For mobile code, the two traditional approaches are applicable if applets are transferred as source code, e.g. JavaScript code, Flash animations, etc.

In the case of Java applets, compiled code (JVM bytecode) is transferred. The Java → JVM compiler performs static type checking, but cannot be trusted:

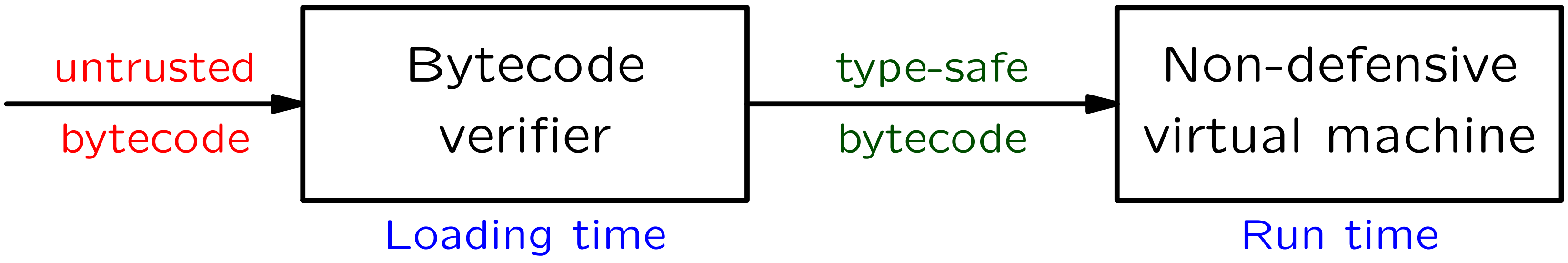
- A malicious applet provider could have written bad bytecode by hand.
- Cryptographic signatures on the bytecode can establish the identity of the applet provider, but not that the applet is innocuous.

Type safety for compiled code

The dynamic typing / static typing alternative applies to compiled code as well:

1. Defensive virtual machine: execute compiled bytecode with a virtual machine that checks types during execution and tags data with their types.

2. Bytecode verification: before the first execution, subject the untrusted bytecode to a static analysis that establishes type safety guarantees. Execute verified code faster using a non-defensive virtual machine.



untrusted bytecode

Bytecode verifier

Loading time

type-safe bytecode

Non-defensive virtual machine

Run time

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 leftover 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 defining class.

A caveat about bytecode verification

These statically-checked properties do not suffice to ensure type safety, let alone security.

Other checks remain to be done at run-time:

- array bound checks;
- null pointer checks;
- checked type casts (e.g. `(String) x`);
- stack inspection.

The purpose of bytecode verification is to move some time-consuming checks from run-time to loading-time, but not all such checks can be done statically.

Verifying straight-line code

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

- Manipulates a stack of types and a register set holding types.

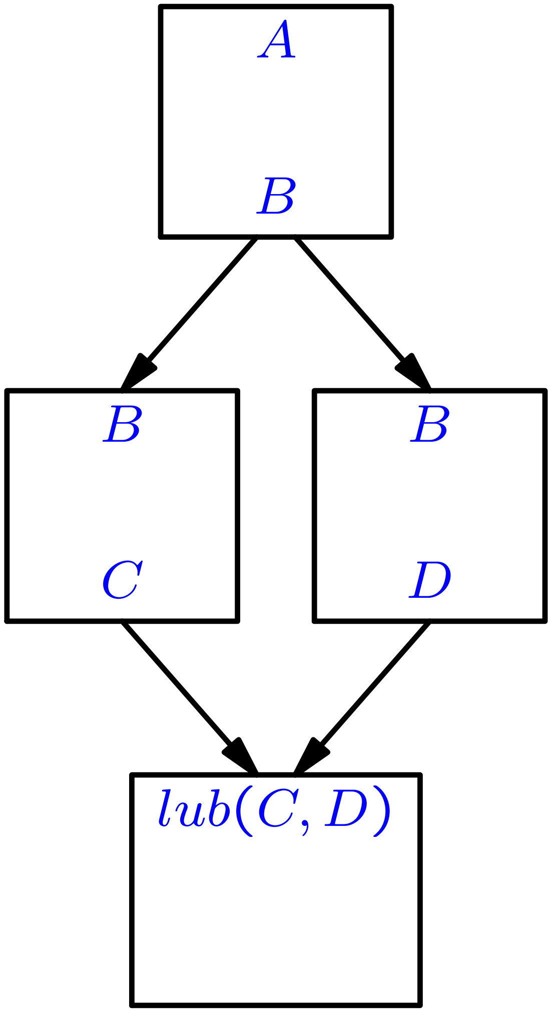
- For each instruction, checks types of arguments and compute types of results.

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 ]
DUP
                           r0: C, r1: int, r2: ⊤      [ C; C ]
GETFIELD C.x : int
                           r0: C, r1: int, r2: ⊤      [ C; int ]
DUP
                           r0: C, r1: int, r2: ⊤      [ C; int ; int ]
ISTORE 2
                           r0: C, r1: int, r2: int     [ C; int ]
ILOAD 1
                           r0: C, r1: int, r2: int     [ C; int ; int ]
IADD
                           r0: C, r1: int, r2: int     [ C; int ]
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.

```
r0: int, r1: C, r2: D                   r3: ⊤, r4: ⊤    [ ]
                              ILOAD 0
                                        r3: ⊤, r4: ⊤    [ int ]
                              IFEQ




r3: ⊤, r4: ⊤   [  ]                             r3: ⊤, r4: ⊤    [  ]
                    ICONST 42       ALOAD 2
r3: ⊤, r4: ⊤   [ int ]
                    ISTORE 4
r3: ⊤, r4: int  [  ]
                    ALOAD 1
r3: ⊤, r4: int  [ C ]                           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:

$$\mathit{instr} : (T_{regs},\ T_{stack}) \rightarrow (T'_{regs},\ T'_{stack})$$

For instance:

$\texttt{iadd} \ : \ (\texttt{int}.\texttt{int}.S,\ R) \rightarrow (\texttt{int}.S,\ R)$

$\texttt{iconst}\ n \ : \ (S,\ R) \rightarrow (\texttt{int}.S,\ R)$ if $|S| < M_{stack}$

$\texttt{aload}\ n \ : \ (S,\ R) \rightarrow (R(n).S,\ R)$
if $0 \leq n < M_{reg}$ and $R(n) <: \texttt{Object}$ and $|S| < M_{stack}$

$\texttt{astore}\ n \ : \ (\tau.S,\ R) \rightarrow (S,\ R\{n \leftarrow \tau\})$
if $0 \leq n < M_{reg}$ and $\tau <: \texttt{Object}$

$\texttt{getfield}\ C.f.\tau \ : \ (\tau'.S,\ R) \rightarrow (\tau.S,\ R)$ if $\tau' <: C$

$\texttt{putfield}\ C.f.\tau \ : \ (\tau_1.\tau_2.S,\ R) \rightarrow (S,\ R)$ if $\tau_1 <: \tau$ and $\tau_2 <: C$

The dataflow equations for bytecode verification

Write $in(i)$ for the stack and register types before executing instruction i, and $out(i)$ for the types after executing i.

Set up dataflow equations between in and out types:

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

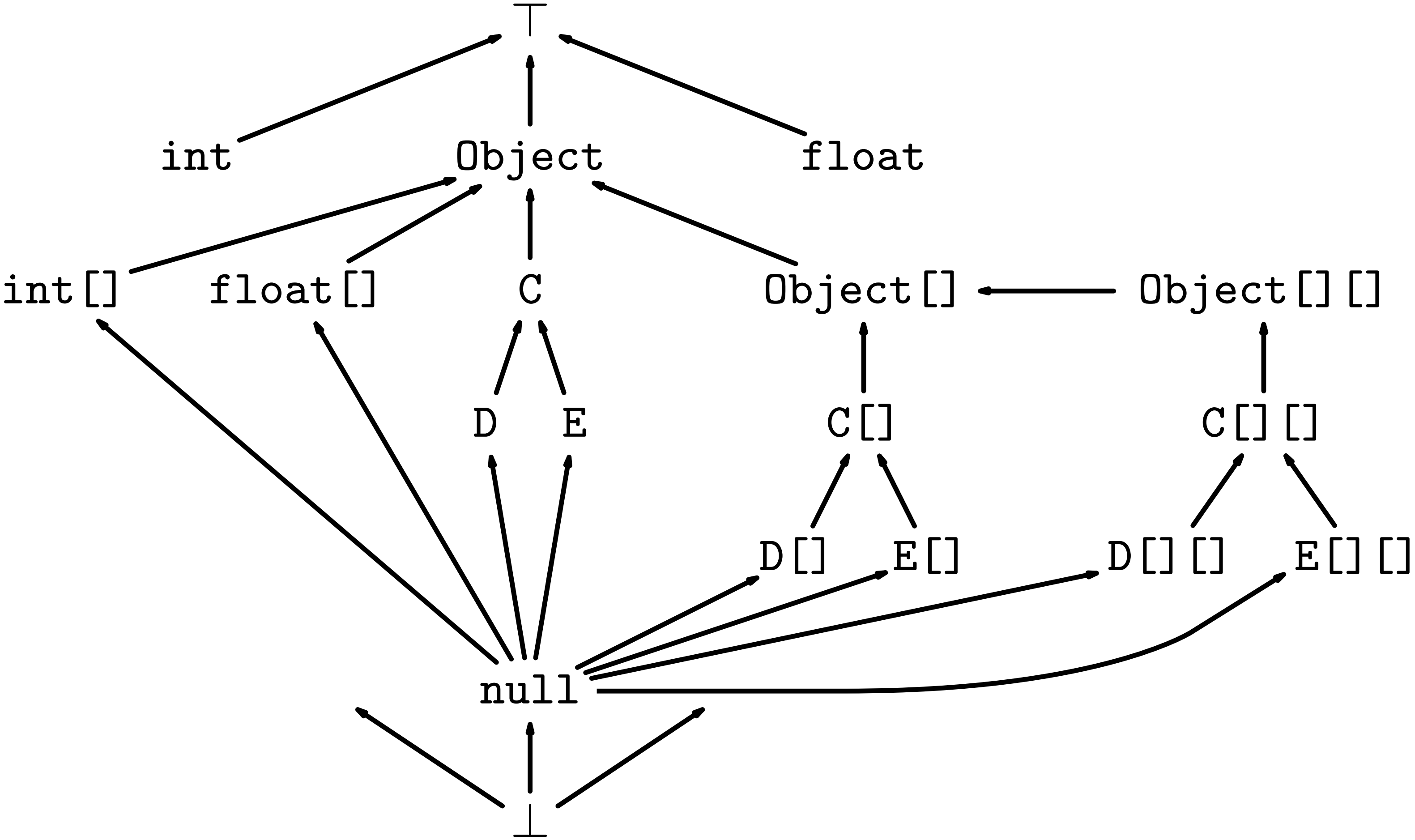
$$in(i) = lub\{out(j) \mid j \text{ predecessor of } i\}$$

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

Here, lub denotes the least upper bound in the lattice of types, i.e. $lub(\tau_1, \tau_2)$ is the smallest type that is super-type of τ_1 and τ_2.

$P_0, \ldots, P_{n-1}$ are the types of the method parameters, as found in the method signature.

The lattice of JVM types (parts of)



Solving dataflow equations

Kildall's worklist algorithm:

Initialization:

$in(i_{start}) = ((P_0, \ldots, P_{n-1}, \top, \ldots, \top), \varepsilon)$
$in(i) = \bot$ for all $i \neq i_{start}$
$out(i) = \bot$ for all i
$W = \{i_{start}\}$ (initial worklist)

Fixpoint iteration:

```
While $W$ is not empty:
    Pick $i$ in $W$
    $W = W \setminus \{i\}$
    Find $O$ such that $i : in(i) \to O$.
    If no such $O$, abort (type error); otherwise, set $out(i) = O$.
    For each successor $j$ of $i$:
        $T = lub(in(s), out(i))$
        If $T \neq in(j)$, set $in(j) = T$ and $W = W \cup \{j\}$
      End If
    End For each
End While.
```

If this algorithm terminates without error, the bytecode is type-safe.

Bytecode verification goes beyond classic dataflow analysis

Several features of JVM bytecode introduce serious complications in bytecode verification and cannot be handled by basic dataflow analysis:

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

2. Object initialization protocol:
   Requires a bit of must-alias analysis during verification.
   (Not discussed here.)

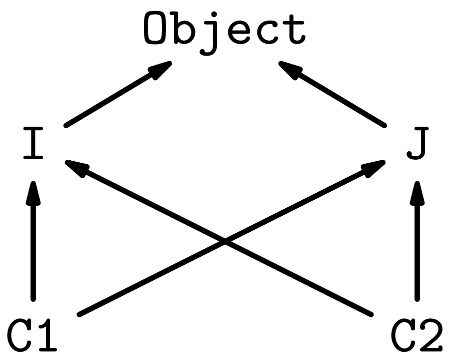
3. Subroutines:
   Require polyvariant analysis
   (several types per program point).

(Note: the MSIL bytecode used by C# avoids complications 2 and 3.)

The problem with interfaces

Consider the following Java classes:

```
interface I  { ... }
interface J  { ... }
class C1 implements I, J   { ... }
class C2 implements I, J   { ... }
```

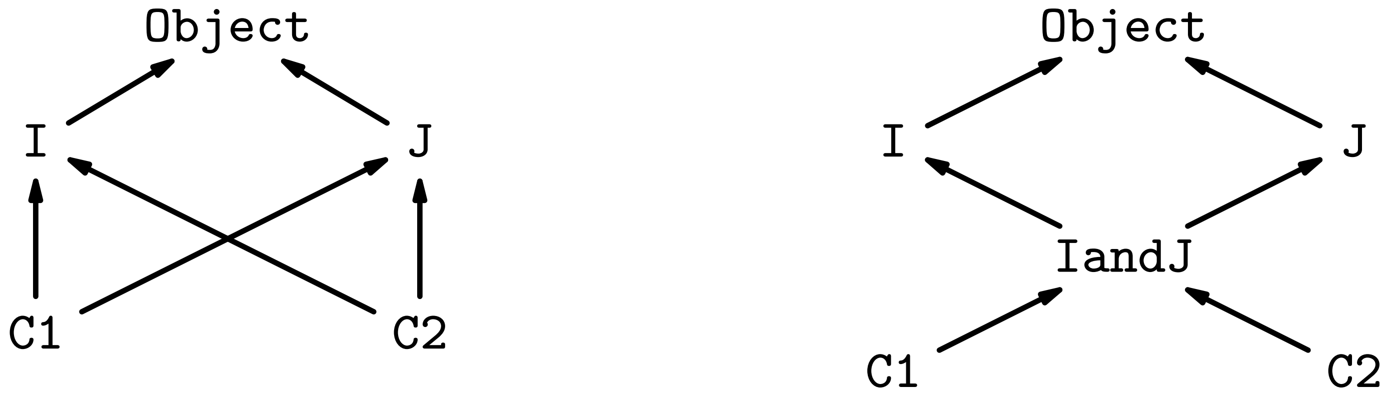


The induced subtyping relation is not a semi-lattice: `C1` and `C2` have two incomparable common super-types `I` and `J`.

Thus, $lub(\texttt{C1}, \texttt{C2})$ is not defined.

Solution 1: lattice completion

Theorem (Dedekind-MacNeille): any partially-ordered set can be injected into a lattice.

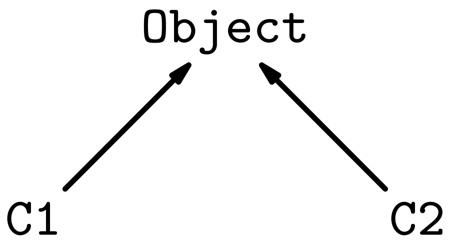


The pseudo-class `IandJ` was added to serve as l.u.b. of `C1` and `C2`.

This completion can be done before bytecode verification, which then doesn't need to manipulate sets of types.

Solution 2: Sun's solution

Sun's bytecode verifier sidesteps the problem by ignoring interfaces: interfaces are not part of the type lattice, and are treated like `Object`.



Thus, $lub(\texttt{C1}, \texttt{C2}) = \texttt{Object}$. L.u.b always exist because of single inheritance between classes: the inheritance relation is a tree.

Drawback: invocation of an interface method `I.m` on an object x does not statically guarantee that $x : \texttt{I}$ (all we know is $x : \texttt{Object}$). Therefore, a run-time check is needed to make sure that the class of x implements `I`.

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 SUB
          r2: ⊤
iload 2
          ERROR!
ireturn                                            SUB:
                                                      r2: ⊤
...                                                     astore 3

astore 2                                                finalization code
          r2: Throwable
jsr SUB                                                 ret 3
          r2: ⊤                                        r2: ⊤
aload 2
          ERROR!
athrow
```

Sun’s solution: polymorphism

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

```
istore 2
            r2: int, r3: ⊤
jsr SUB
            r2: int, r3: retaddr
iload 2

ireturn                                      SUB:
                                                r2: ?, r3: ⊤
...                                               astore 3

astore 2                                          finalization code
            r2: Throwable, r3: int                    (not using r2)
jsr SUB                                           ret 3
            r2: Throwable, r3: retaddr          r2: ?, r3: retaddr
aload 2

athrow
```

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.

All such published analyses are incomplete.

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

A better solution: polyvariant analysis

Allow several (stack type, register type) per program point.

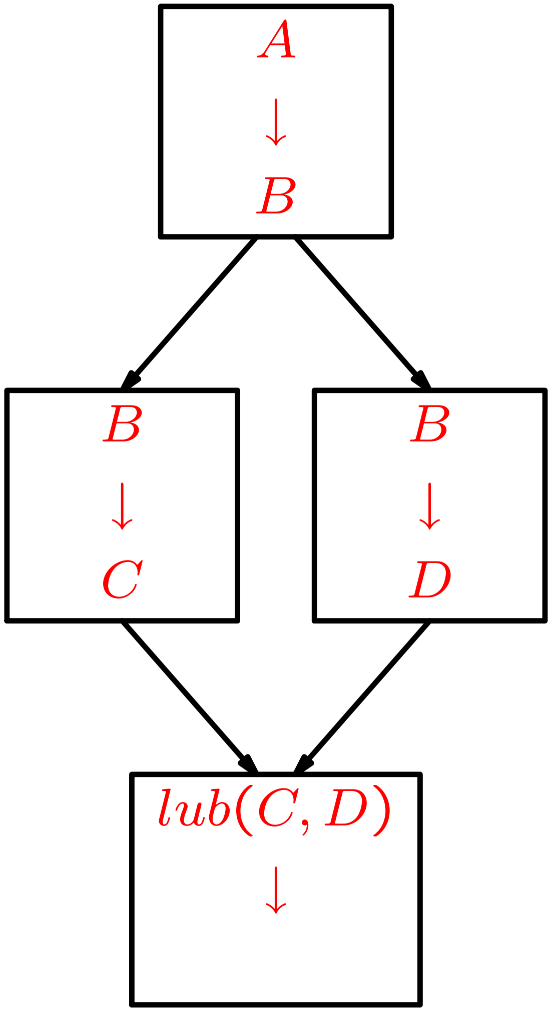
→ No need to take least upper bounds at merge points.

→ Instead, analyze instructions several times if needed.

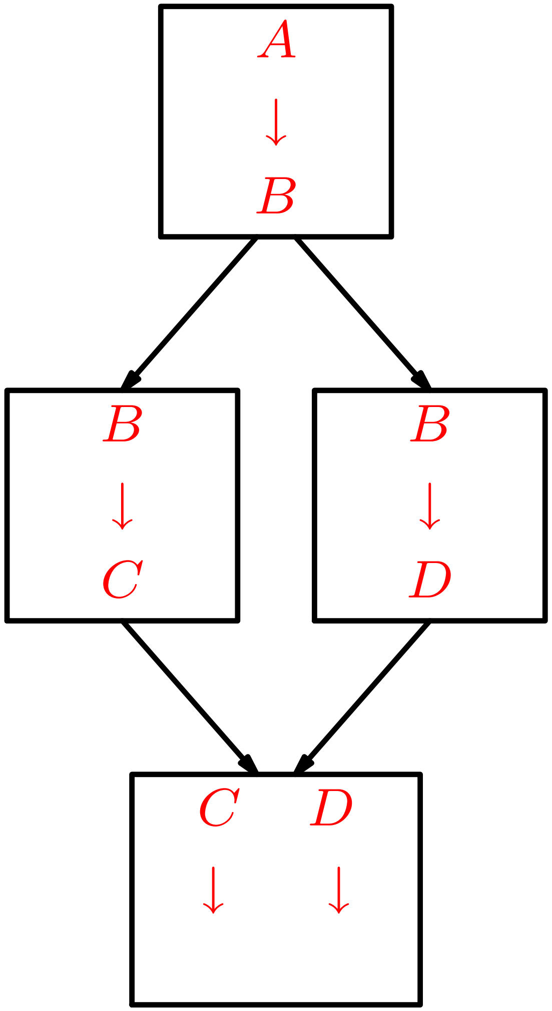
This is called a polyvariant analysis, as opposed to the original dataflow approach which is monovariant.

Handling forks and joins, revisited

Monovariant:



Polyvariant:



The subroutine problem: solved!

```
   istore 2
          r2: int, r3: ⊤
A: jsr SUB
          r2: int, r3: retaddr(A)
   iload 2

   ireturn
                                        SUB:
   ...                                     astore 3
                                               r2: int, r3: retaddr(A)
                                               r2: Throwable, r3: retaddr(B)
   astore 2                                finalization code
          r2: Throwable, r3: ⊤                r2: int, r3: retaddr(A)
B: jsr SUB                                     r2: Throwable, r3: retaddr(B)
          r2: Throwable, r3: retaddr(B)    ret 3
   aload 2

   athrow
```

No merging on the type of `r2` because `SUB` is analyzed twice.

More formally...

Define the state transitions of the type-level VM as follows:

$$\begin{array}{rcl} (pc, R, S) & \Rightarrow & (pc', R', S') \\ & & \quad \text{if } instr(pc) : (R, S) \rightarrow (R', S') \\ & & \quad \text{and } pc' \text{ is a successor of } pc \\ (pc, R, S) & \Rightarrow & \texttt{Error} \\ & & \quad \text{if } instr(pc) : (R, S) \not\rightarrow \end{array}$$

The relation $\Rightarrow$ represents one step of type-level execution of the bytecode.

Explore all states reachable by successive applications of $\Rightarrow$ from the initial state $(start, (P_0, \ldots, P_{n-1}, \top, \ldots, \top), \varepsilon)$.

If the state `Error` cannot be reached, the bytecode is type-safe.

The algorithm for state exploration

Initialization: $\Sigma = W = \{(start, (P_0, \ldots, P_{n-1}, \top, \ldots, \top), \varepsilon)\}$

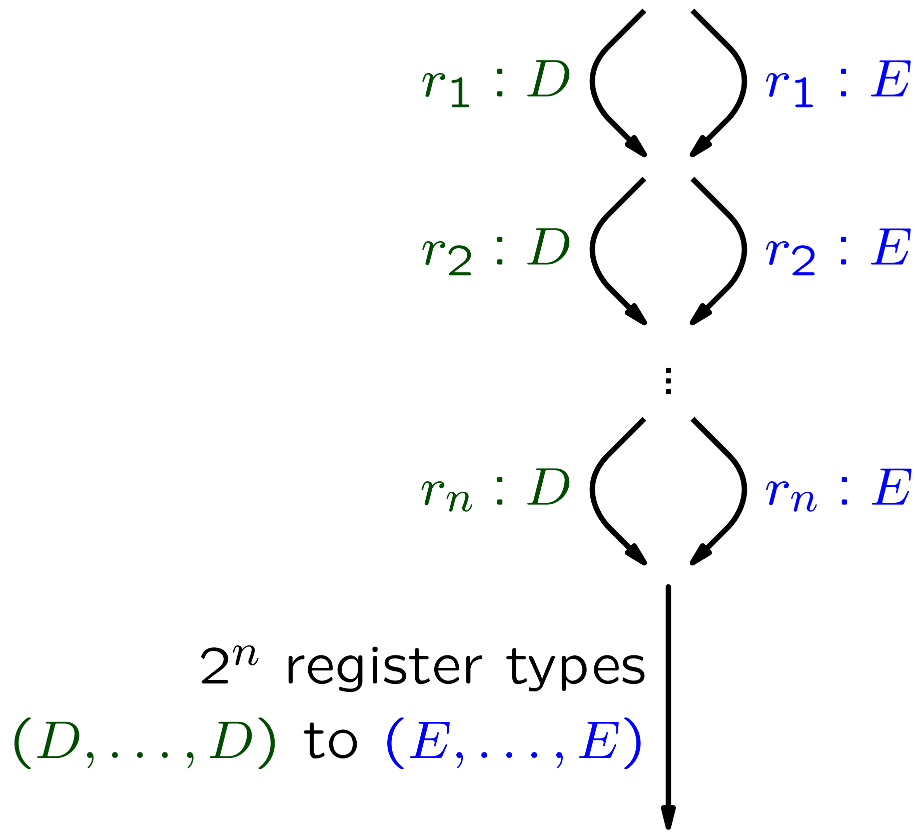
Fixpoint iteration:

```
While $W$ is not empty:
    Pick $s \in W$
    $W = W \setminus \{s\}$
    For each $s'$ such that $s \Rightarrow s'$, do:
        If $s' = \texttt{Error}$, abort (type error).
        If $s' \notin \Sigma$, set $\Sigma = \Sigma \cup \{s'\}$ and $W = W \cup \{s'\}$
    End For each
End While.
```

If this algorithm terminates without error, the bytecode is type-safe.

State explosion

Consider a method with n `if-then-else` in sequence:



We end up analyzing the bottom instructions 2^n times.

Accelerating state exploration by widening

At any step in the state exploration algorithm, we can replace the set Σ of states reachable so far by any other set Σ' such that

$$\forall s \in \Sigma,\ \exists s' \in \Sigma',\ s <: s'$$

This is called a widening step.

Widening reduces the precision of the analysis, but can speed it up if Σ' is chosen smaller than Σ.

Widening is always correct: if `Error` is not reachable after applying widening steps, it is not reachable either in the original, non-widening algorithm.

Monovariant analysis as widened state exploration

The original monovariant analysis is a trivial instance of widened state exploration: we replace all states having the same PC

$$(pc, R_1, S_1), \ldots, (pc, R_n, S_n)$$

by their least upper bound:

$$(pc, lub(R_1 \ldots R_n), lub(S_1 \ldots S_n))$$

This reduces complexity from exponential back to polynomial.

Combining polyvariance and efficient widening

The best known compromise between precision and efficiency for bytecode verification is obtained by the following widening scheme: replace sets of states having the same PC

$$(pc, R_1, S_1), \ldots, (pc, R_n, S_n)$$

by their least upper bound:

$$(pc, lub(R_1 \ldots R_n), lub(S_1 \ldots S_n))$$

if and only if the R_i and the S_i agree on return addresses, i.e. contain the same `retaddr` types in the same places.

The optimal bytecode verification algorithm?

For subroutine-free code, no `retaddr` types appear
→ we recover the efficiency of monovariant analysis.

For code containing subroutines, the criterion prevents merging states that correspond to different invocations of a subroutine (they differ by at least one `retaddr` type)
→ sufficient polyvariance is preserved.

It can be shown that this widening scheme accepts exactly the same programs as the non-widened state exploration scheme
→ no precision is lost.

Summary on bytecode verification

Bytecode verification: the paradigmatic example of a security component based on static analysis of compiled code.

The basic idea (dataflow analysis on a type-level VM) is effective and unproblematic.

Extending it to the full JVM (with subroutines, etc) is problematic:

- Originally, no formal specification of bytecode verification.
- Sun's reference implementation as a de facto spec.
- Dozens of academic papers proposing different specifications and algorithms.

The presentation in terms of state space exploration and widening unifies many of these works, but comes too late.

Bytecode verification and formal methods

Many formalizations and proofs of correctness of bytecode verification, on paper but also on machine using theorem provers (Nipkow & Pusch, Barthes et al, Gimenez & Barras, etc).

EAL7 certification (Gimenez & Barras, Trusted Logic).

Bytecode verification is arguably the security component that has been most formalized and mechanically verified using formal methods.

Further work

Bytecode verification in resource-constrained environments.

Typed Assembly Language (Morrisett et al):
static type-checking of annotated x86 assembly code.

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

Beyond strong typing:
the example of information flow analysis

Enforcing functional properties beyond type safety

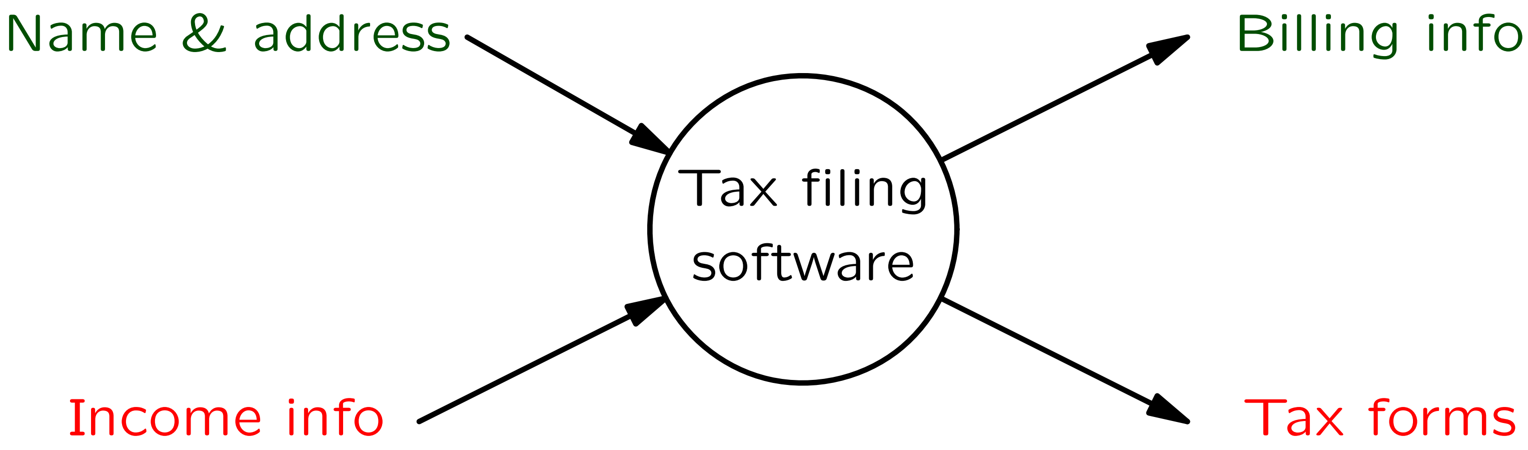
Strong typing + secured APIs enforce a number of desirable, basic security and reliability properties.

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

One example: information flow analysis.

Why control information flow?

Consider a pay-per-use tax filing program:



To preserve confidentiality, the billing information sent to the software provider must not reveal any of the income info and must depend only on identity.

Inadequacy of access control

Access control (restricting who can read confidential information) is inadequate because it does not prevent a privileged principal from divulging the information just read.

In the tax software example, the program must be allowed to read both public info (identity) and confidential info (income).

Security policies based on information flow address this need to track the propagation of confidential information.

Information levels

Attach information levels to every piece of data manipulated by the program.

In the classic model (Bell & La Padula, Denning):

$$\text{level} = \texttt{H} \text{ (secret) or } \texttt{L} \text{ (public), with } \texttt{H} > \texttt{L}.$$

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.

Information levels

Richer models are possible:

- Multi-level security: e.g.

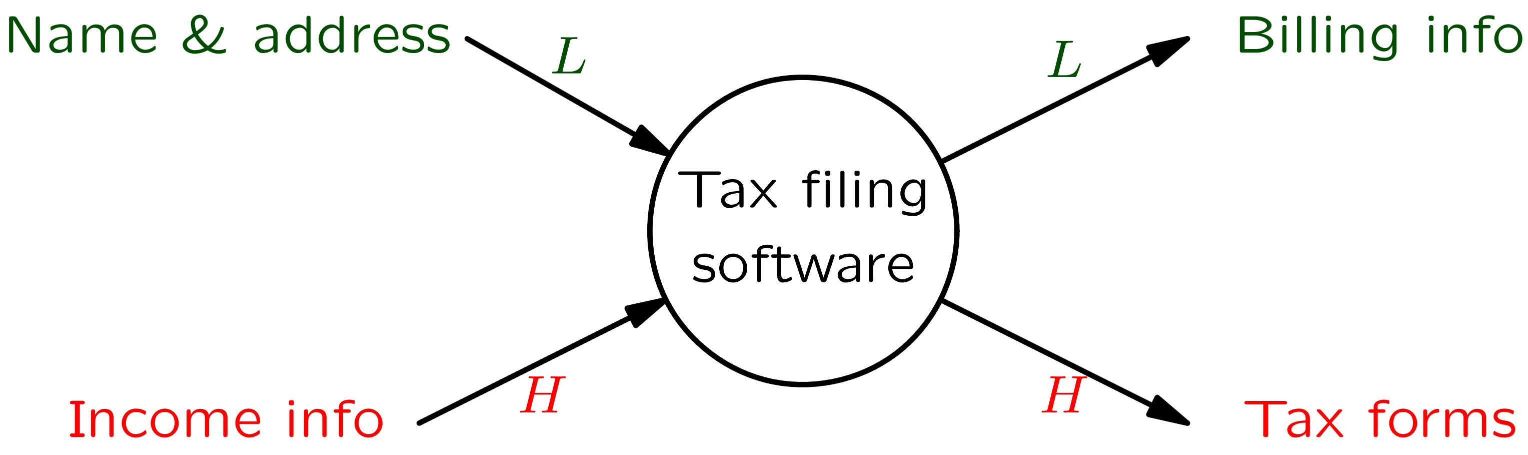
$$\mathsf{public} < \mathsf{restricted} < \mathsf{confidential} < \mathsf{secret}$$

- Sets of principals (those allowed to read the data):

$$\mathsf{everyone} < \{\mathsf{Alice}, \mathsf{Bob}\} < \{\mathsf{Bob}\} < \emptyset$$

More generally: a lattice of information levels (partially-ordered set with min and max operations).

Tax filing revisited



Our goal is to ensure that the software behaves as specified by the L, H labeling above.

(I.e. that L outputs do not change when H inputs change.)

Dynamic enforcement of confidentiality?

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

Works fine for catching direct information flows:

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

The problem with indirect information flows

Difficult to extend to indirect information flows:

```
static boolean y;

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

This code sets `y` to the same value as `x`, yet no direct assignment `y = x` takes place.

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

Can be presented as an enriched type system.

A quick introduction to type systems

A type system is a formal specification of what a type checker does. It comprises:

- A type algebra (set of type expressions):

$$\tau ::= \texttt{int} \mid \texttt{boolean} \mid \texttt{array}(\tau)$$

- One or several typing judgements associating types to program fragments (expressions, statements, . . .):

$\Gamma \vdash e : \tau$	Under hypotheses Γ on the types of variables, the expression e has type τ
$\Gamma \vdash s\ \texttt{ok}$	Under hypotheses Γ on the types of variables, the statement s is well-typed

- Typing rules that define when the judgements hold.

Examples of typing rules

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

(If x is declared with type τ, an occurrence of x in an expression has type τ)

$$\frac{\Gamma \vdash e_1 : \mathtt{int} \qquad \Gamma \vdash e_2 : \mathtt{int}}{\Gamma \vdash (e_1 + e_2) : \mathtt{int}} \qquad \frac{\Gamma \vdash e_1 : \mathtt{array}(\tau) \qquad \Gamma \vdash e_2 : \mathtt{int}}{\Gamma \vdash e_1[e_2] : \tau}$$

Examples of typing rules

$$\frac{\Gamma(x) = \tau \qquad \Gamma \vdash e : \tau}{\Gamma \vdash (x = e)\ \texttt{ok}}$$

(If e and x have the same type τ, the assignment $x = e$ is well typed.)

$$\frac{\Gamma \vdash e_1 : \texttt{array}(\tau) \qquad \Gamma \vdash e_2 : \texttt{int} \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (e_1[e_2] = e_3)\ \texttt{ok}}$$

$$\frac{\Gamma \vdash S_1\ \texttt{ok} \qquad \Gamma \vdash S_2\ \texttt{ok}}{\Gamma \vdash (S_1; S_2)\ \texttt{ok}} \qquad \frac{\Gamma \vdash e : \texttt{boolean} \qquad \Gamma \vdash S\ \texttt{ok}}{\Gamma \vdash (\texttt{if}(e)\ S)\ \texttt{ok}}$$

Type systems for information flow

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

Take an existing type system and annotate type expressions with information levels ℓ:

$$\tau ::= \texttt{int}^{\ell} \mid \texttt{boolean}^{\ell} \mid \texttt{array}^{\ell}(\tau)$$

Examples:

$\texttt{int}^{\mathrm{L}}$	a public integer
$\texttt{boolean}^{\mathrm{L}}$	a confidential boolean
$\texttt{array}^{\mathrm{L}}(\texttt{int}^{H})$	a public array of confidential integers

Tracking information levels in typing rules

Typing rules for expressions propagate levels in the natural way:

$$\frac{\Gamma \vdash e_1 : \mathtt{int}^{\alpha} \qquad \Gamma \vdash e_2 : \mathtt{int}^{\beta} \qquad \alpha \le \delta \qquad \beta \le \delta}{\Gamma \vdash (e_1 + e_2) : \mathtt{int}^{\delta}}$$

In general: the result of an expression is at least as confidential as any of its operands.

Exception: the result of a cryptographic hash $H(e)$ could be given type $\mathtt{string}^{\mathrm{L}}$ even if e is confidential, as the hash reveals (almost) no information on the value of e.

Tracking direct and indirect flows

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

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

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

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

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

Indirect flow avoidance

Consider the statement $\texttt{if}\ (b)\ x = y$.

The constraints enforced by the typing rules are

$$\texttt{level}(y) \leq \texttt{level}(x) \qquad \texttt{level}(b) \leq \texttt{level}(x)$$

Considering all cases:

$\texttt{level}(x)$	$\texttt{level}(y)$	$\texttt{level}(b)$	Accepted?
H	any	any	OK
L	L	L	OK
L	H	any	rejected (direct flow)
L	any	H	rejected (indirect flow)

From a type system to a static analysis tool

1. Develop type inference algorithms.

   Synthesize and check levels from mostly un-annotated source code.

   Examples: JFlow (A. Myers), FlowCaml (V. Simonet).

2. Perform inference and verification on (unstructured) compiled bytecode instead of (structured) source code.

   A largely open question.

Summary on information flow analysis

Information flow has generated considerable academic work, both as a security policy and as a program analysis problem.

This work is rarely used in practice:

- Strict non-interference is often too restrictive.
- Side channels (e.g. timing leaks) are not accounted for.
- Need to reason in terms of information leakage rate (N bits of info leak per second) rather than flow / no flow.

Summary on static analyses for security

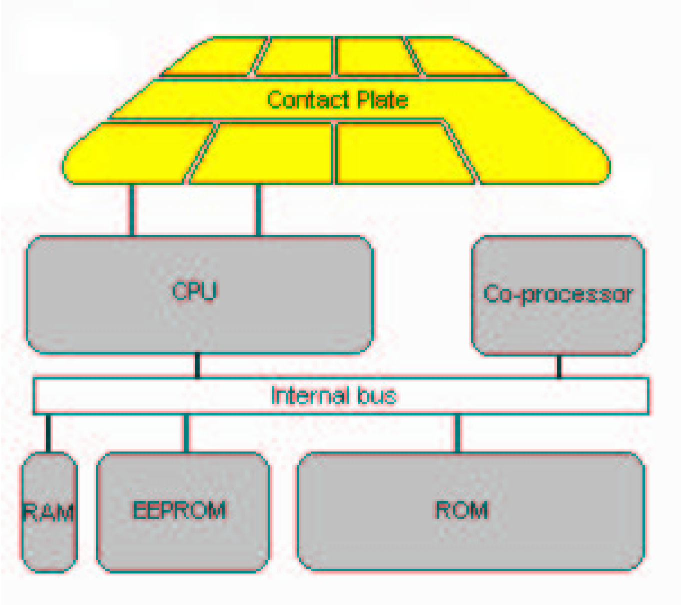
Bytecode verification / strong typing has been relatively successful security-wise: widely deployed, good benefits/cost ratio.

It is unclear whether this “success story” can be repeated for finer static analyses for security:

- No consensus on the properties of interest.
- Complex and expensive static analyses, often specialized.
- Not obvious how to apply them to bytecode.

Smart cards and Java Card

Smart cards



- A small embedded computer.
  Shaped as a credit card or a SIM card.
- Secure (tamper-resistant).
- Inexpensive (average 3 USD).

Smart card hardware

- Processor:
  low processing power (8-bit CPU, 5 MHz clock)
  sometimes complemented with a cryptographic coprocessor.

- Memory:
  1–4 Kbytes of volatile RAM (scratch memory);
  16 Kb of EEPROM or Flash (persistent rewritable memory);
  64 Kb of ROM (persistent, non-writable).

- Communication links:
  serial link via the contact area (also: power & clock);
  or radio link (contactless smart cards).

Some typical uses of smart cards

Credit cards (Visa, Mastercard, Eurocard):
the card checks the user's PIN code and authorizes off-line transactions.

SIM cards for GSM mobile phones:
the card authentifies the user with the mobile phone provider. Also stores a phone book.

Electronic purses (Proton, Moneo):
small transactions without a PIN; refilled from a credit card.

Pay TV:
the card generates a temporary descrambling key based on per-user code sent monthly or quarterly by mail.

Some typical uses of smart cards

Public transportation pass (Navigo):
contactless smart card to zip through ticket booths; refilled monthly or yearly.

Social security data (Vitale):
stores social security and medical insurance info; replaces paper forms.

Electronic locks:
some cars use contactless smart cards instead of ignition keys.

Using smart cards as security tokens

- Authentication of the card holder:

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

  Credit cards; GSM SIM cards; electronic locks.

- Storing sensitive information:

  Credit cards: number, expiration date, transaction log, . . .
  Phone book in SIM cards.
  Social security data.

- Generating certificates or session keys:

  Pay TV descrambling; credit cards (transaction certificate);
  GSM (session key for voice encryption).

Detailed example: the B0' protocol (1st-gen credit cards)

The card provides the terminal with the credit card number, the expiration date, and the authentication value (RSA signature of number & date).

The terminal verifies the authentication value using RSA public key.

User types PIN on terminal, which sends it to the card.

The card checks the PIN code. It locks itself after 3 incorrect attempts.

The card records the transaction in its internal log.

The card generates a cryptographic certificate (DES MAC) of the transaction and sends it to the terminal.

Merchant accepts the payment and transmits it later to its bank.

The importance of the authentication value

Without the authentication value, any attacker could manufacture a so-called yes-card:

- It carries the CC number and dates of your choice.
- It answers OK to the PIN code of your choice (or to any PIN code)
- It generates a bogus transaction certificate
(it cannot be checked during the transaction).

The a.v. signature is the only thing that distinguishes a valid credit card (issued by an authorized bank) from a yes-card.

If RSA secret key leaks or RSA public key is factored, attacker can still manufacture a yes-card carrying a correct a.v.

Attacks on `B0'`

First-generation credit cards used a 320 bit RSA key.

(That size was chosen in the early 1980's. It was the maximum that terminals of that time could check in reasonable time. A 1988 academic publication warns of this weakness: L. Guillou and J.-J. Quisquater, "Techniques à clé publique", *Annales des Télécommunications* 43(9–10).

The key was broken and published in 1999 (the Humpich affair).

Attacks on `B0'`

Newer credit cards use a similar protocol but are signed with a 768 bit RSA key. However, one vulnerability remains. . .

The authentication value for a credit card is readable by anyone, without presenting the PIN.

→ can copy the number, date and a.v. from a valid card and put them on a yes-card.

→ anyone with physical access to a credit card can "clone" it.

Constraints on smart card systems

- High security requirements:
  no human lives in danger, but significant money is at stake.
  Objective: attacks should be more costly than expected gain.

- Tiny computing resources:
  about 1/10,000th that of a PC.
  Limits amount of crypto that can be done.
  Increase slowly because cards must remain inexpensive.

- Possibility of hardware attacks:
  the card is physically in the hands of the attacker.

Smart card software

The traditional software architecture for smart cards (till 2000):

- Single application: one card = one function.
- Software is in ROM, cannot be updated post-issuance.
- Software usually structured as an operating system (memory management, communications, crypto primitives) plus one application.
- Software is propietary, developed by card manufacturer, often very strongly tied to the underlying hardware.
- Written in assembly language or in C.
- A lot of security by obscurity (trade secrets).

Open architectures for smart cards

Circa 2000, the smart card industry switched to much more open software architectures such as Java Card:

- Multiple applications, with controlled sharing of information.
- Post-issuance downloading of applications (*cardlets*).
- Programmed in a subset of Java against a standard API.
- Software comprises:
  an operating system
  Java Card VM, APIs and possibly cardlet loader
  applications written in Java Card.
- Applications are often written by others than the card manufacturer.
- Still a lot of trade secrets, but VM and APIs are public standards.

The Java Card subset of Java

- Lacks the types `long`, `float`, `double`, `char` and associated arithmetic operations.

- Most arithmetic is done on `byte` and `short` (8 and 16-bit integers).
  Support for `int` is optional.

- Objects and arrays are persistent (stored in EEPROM).

- GC is optional → must allocate all needed objects at cardlet installation time.

- No threads; no class loaders.

- Different set of standard classes: communications, crypto and transactions.

The general shape of a Java Card application

```
public class MyApplet extends javacard.framework.Applet {
    MyApplet() {
        // initialize instance variables, allocate space
        super.register();
    }
    public static void install(...) {
        new MyApplet();
    }
    public boolean select() {
        // per-session initialisation
    }
    public void deselect() {
        // per-session cleanup
    }
    public void process(APDU b) {
        // process commands found in b
    }
}
```

Translating the sandbox model to Java Card

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 owned by another cardlet.

	Owner of object		
Principal	System	Applet A	Applet B
System	ok	ok	ok
Applet A	NO (*)	ok	NO
Applet B	NO (*)	NO	ok

(*) Except for specially designated “JCRE entry point” objects.

Access control in Java Card: the firewall

An applet can designate explicitly some of its objects as shareable. Other applets are then allowed to invoke interface methods on this object.
→ a form of inter-applet communication.

Interface and virtual method invocations cause an 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.

Enforcement of type safety

Strong typing is as essential to Java Card security as it is to Java security.

The firewall checks in the Java Card VM make it harder but not impossible to mount attacks based on type violations.

(Infix pointers obtained by pointer arithmetic can falsify the firewall's determination of the owner of an object.)

Enforcement of type safety can be achieved by:

- Off-card bytecode verification + cryptographic signatures.
- Defensive virtual machine.
- On-card bytecode verification.

(Often combine the first 2 for additional security.)

On-card bytecode verification

Bytecode verification on Java Card is challenging because of the low resources of a card:

- Time: complex process involving fixpoint iteration.
- Space: the memory requirements of Sun's algorithm are

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

where

$$\begin{aligned} M_{stack} &= \text{max size of expression stack} \\ M_{regs} &= \text{number of registers used by method} \\ N_{branch} &= \text{number of branch target points in method} \end{aligned}$$

The space above is needed 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 much to fit in RAM on a smart card.

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

Not all verifiable Java Card bytecode produced by a standard Java compiler meets these requirements.

However, all verifiable bytecode can be transformed off-card into equivalent bytecode that meets R1 and R2.

Emptying the stack at branch points

Branches arising from loops or `if...else...`statements are naturally taken on an empty expression stack.

The problem arises with conditionals such as `a + (b ? c : d)`.

```
    compute a
    compute b
    ifeq L1     // stack contains $v_a$
    compute c
    goto L2     // stack contains $v_a$, $v_c$
L1: compute d
L2: iadd
```

Emptying the stack at branch points

Solution: use temporary registers to save and reload stack slots around branches.

```
    compute a
    compute b
    swap
    istore TEMP1   // TEMP1 = $v_a$
    ifeq L1        // empty stack
    compute c
    istore TEMP2   // TEMP2 = $v_c$
    goto L2        // empty stack
L1: compute d
    istore TEMP2   // TEMP2 = $v_d$
L2: iload TEMP1
    iload TEMP2
    iadd
```

Equivalent to transforming `x = a + (b ? c : d)` into

```
    TEMP1 = a;
    if (b) TEMP2 = b; else TEMP2 = c;
    x = TEMP1 + TEMP2;
```

Ensuring unique types for registers

In Java source code, every local variable is used with only one type: the type it was declared with.

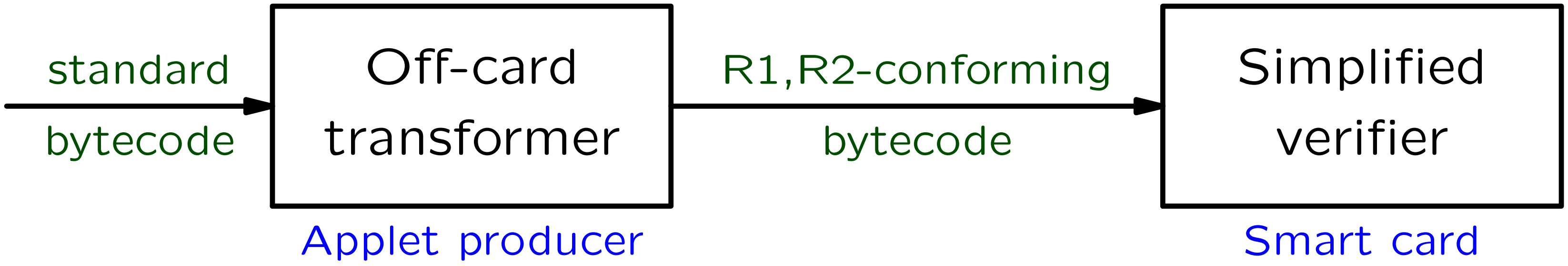
However, the Java compiler can pack variables with different types and non-overlapping scopes into a single register:

```
void foo() {
    {
      int x;          // mapped to register 1
      ...
    }
    {
      Object r;       // mapped to register 1
      ...
    }
}
```

Solution: undo this packing by live range splitting.

E.g. rename register 1 to register 2 in second half of code.

The general architecture



The code transformation is complex but can be performed off-card by the applet producer, since it does not need to be trusted.

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

Summary

The Java “sandbox” model can be transferred to small embedded systems such as smart cards.

On-card bytecode verification is feasible but requires special algorithms and infrastructure.

Hardware attacks and hardware and software countermeasures.

Hardware attacks

The security of a PC or server is void if the attacker has physical access to the machine:

- Pop the hard drive, put it in attacker's machine, read and modify data and programs.
- Key loggers to circumvent cryptography.

Smart cards and other embedded systems are in the hands of the attacker → need to make them tamper-resistant.

Tampering with smart cards

Using a good micro-electronics lab, the attacker can do all of:

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

Observation attacks

See J.R. Rao’s talk.

Temporary perturbations

On EEPROM accesses:

- A lower-than-usual voltage causes EEPROM writes to fail.
  → no increment of the “wrong PIN” count for instance.

- A higher-than-usual voltage can cause EEPROM reads to return all zeroes.
  → crypto becomes done with known keys (all 0).

On the code memory or data bus:

- For a while, the processor executes NOP-like instructions instead of program instructions.

- Then, execution resumes at some further point in the code
  → parts of the program have been skipped over.

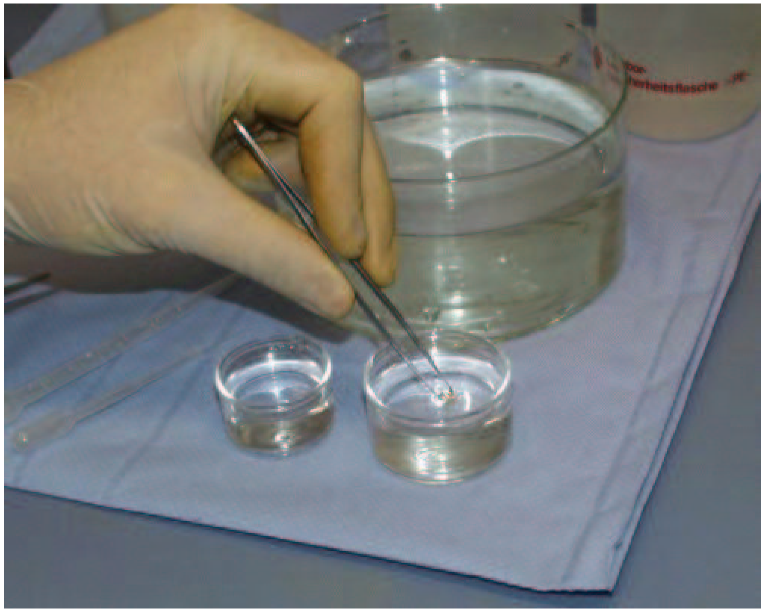
Example of attacks by perturbation

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

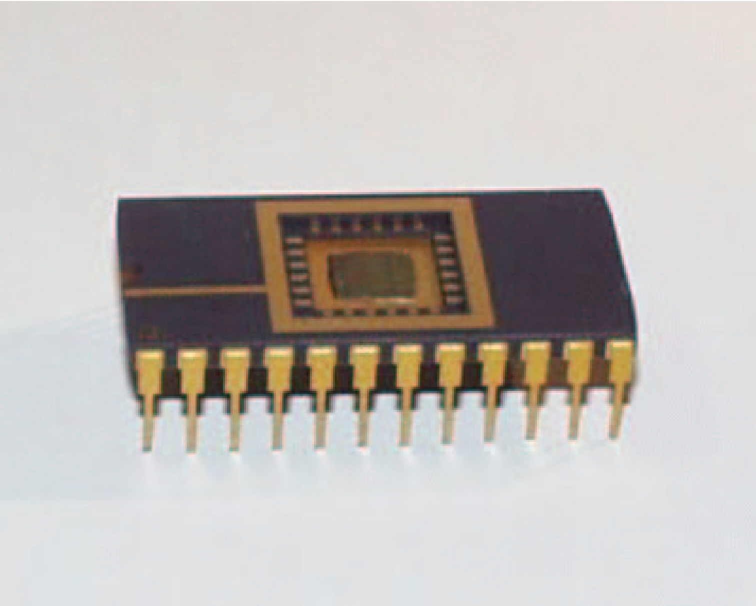
Skip over the stop condition, or modify the current value of `p` so that it jumps over the bound `buffer + length`
→ dumps the whole memory on the serial port.

Invasive attacks: De-packaging the processor

(Photos taken from Oliver Kömmerling, Markus G. Kuhn: Design Principles for Tamper-Resistant Smartcard Processors, USENIX Workshop on Smartcard Technology, 1999, `http://www.cl.cam.ac.uk/Research/Security/tamper/`.)

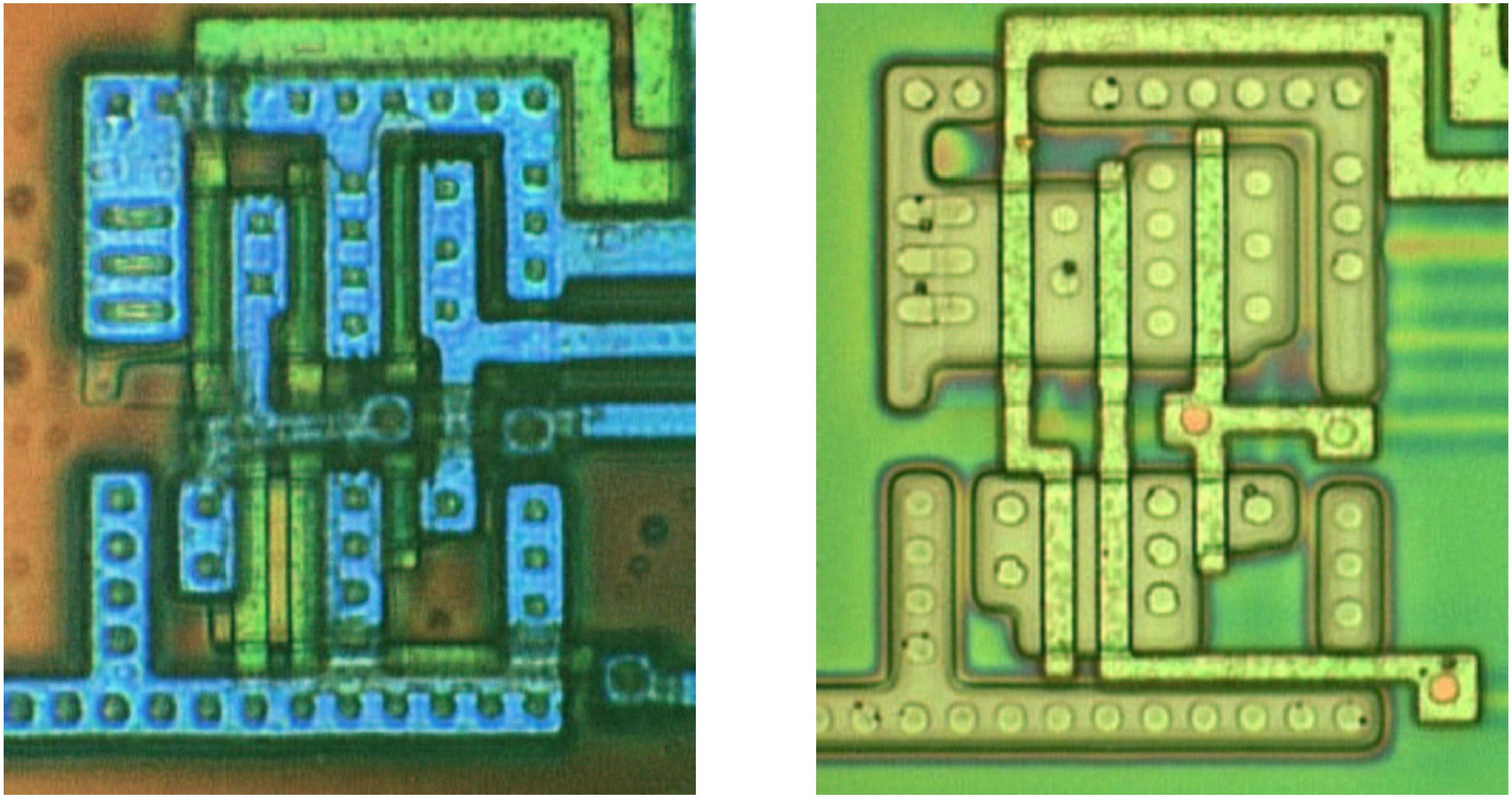


Invasive attacks: Re-bonding the processor



Attacker can now run the device while examining or modifying it.

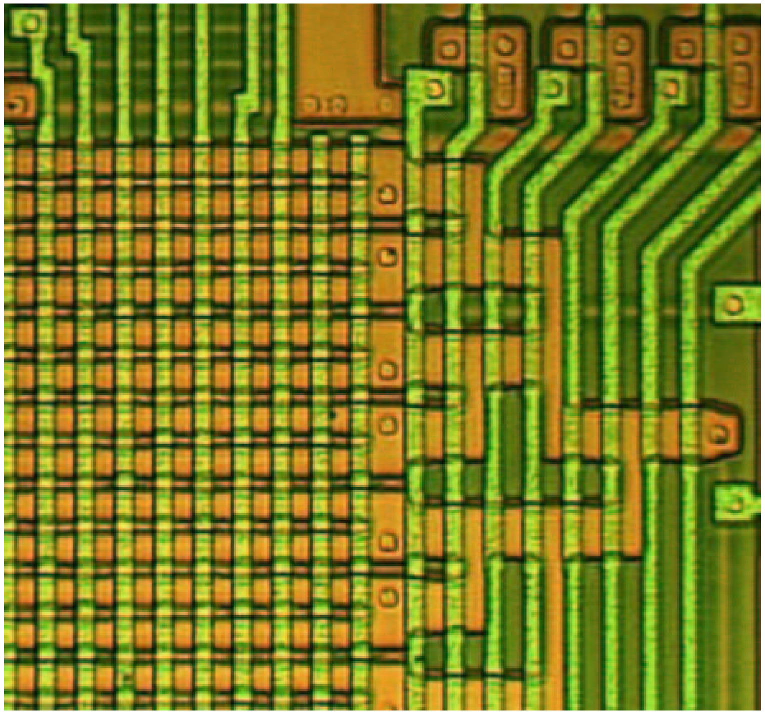
Invasive attacks: Reverse-engineering of circuits

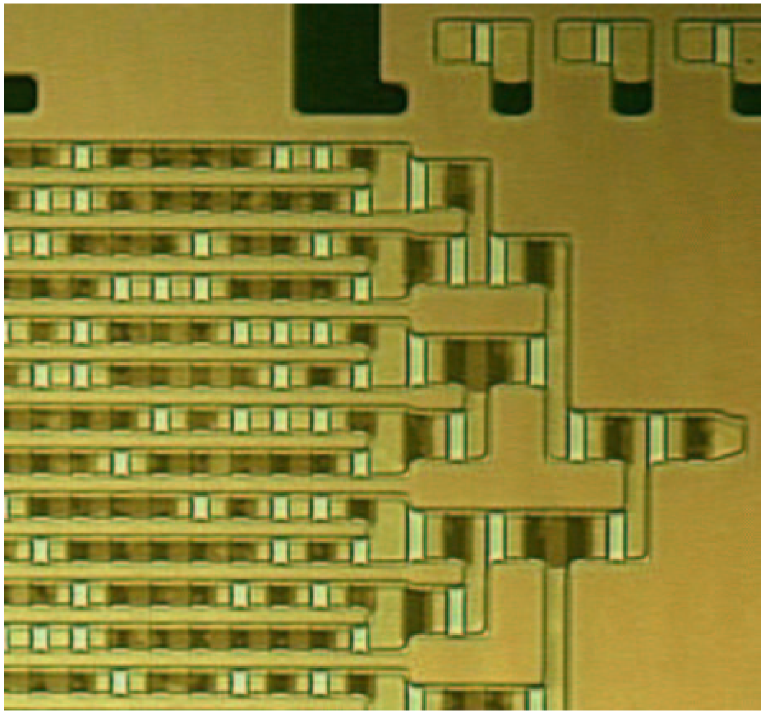


Attacker can locate interesting parts of the circuit, e.g. the program counter, the data bus, etc.

Invasive attacks: Read contents of ROM area

Using crystallographic staining techniques, 0 bits show differently than 1 bits.

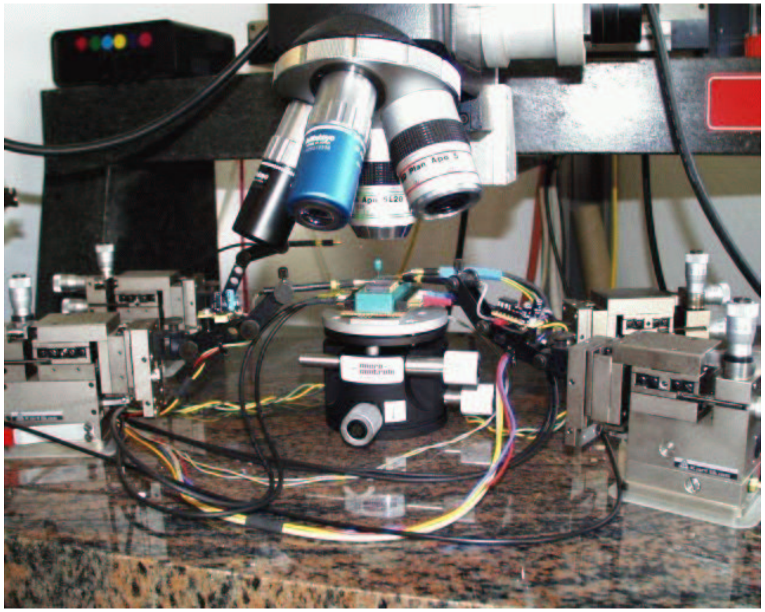




Attacker can read the program code and whatever data is in ROM.

Invasive attacks: Insert microprobes on CPU bus

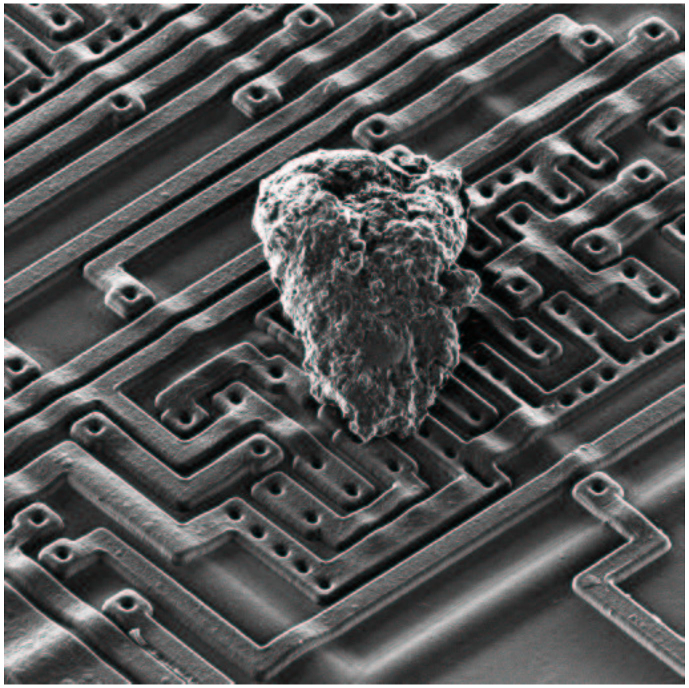
A microprobing station allows “depassivation” (exposure) of data bus lines and attaching microprobes to them.

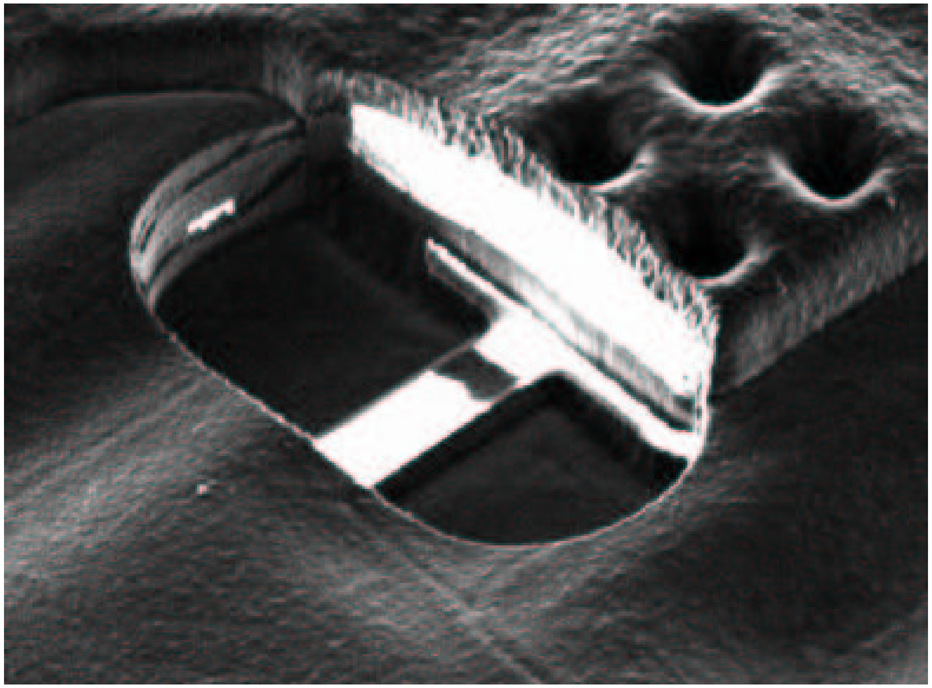


Attacker can sniff all data stored into memory or read from memory.

Invasive attacks: Cut or add connections

Focused ion-beam machines allow removing or depositing conductive material on chip.





Attacker can e.g. grow back fuses that were blown during manufacturing to disable testing equipment or prevent rewriting write-once memory.

Hardware counter-measures

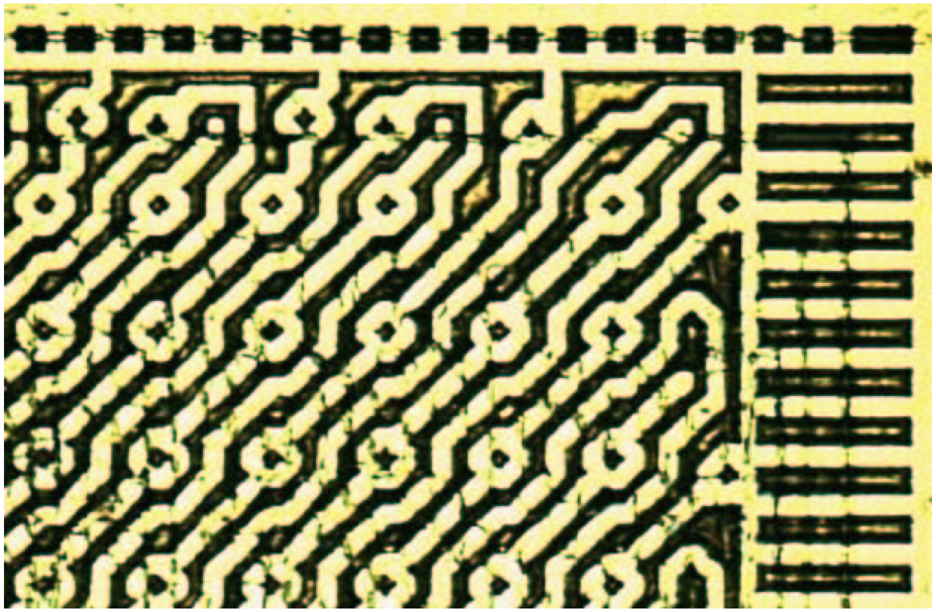
Many hardware techniques to protect against hardware tampering:

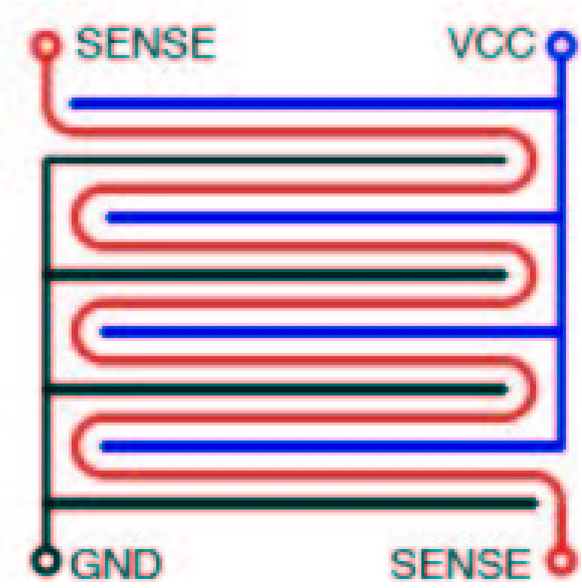
- Detect power glitches and shut down.
- Obfuscation of the chip layout.
- Encryption of the data bus and memory contents.
- Hide circuits under protective layers.
- Memory access controllers:

  Restrict what ranges of PC can access what range of memory addresses.
- Limited-excursion addressing:

  16-bit constant base + 8-bit index register.

Example: protective layers





Trying to remove this layer cuts the power supply or the “sense” line.

Software counter-measures

In general, software cannot protect itself against being executed by a malicious processor (e.g. a debugger or emulator).

Nonetheless, software can also be hardened (to some extent) against hardware attacks, by exploiting characteristics of these attacks:

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

- Perturbation attacks are temporary and reversible, but imprecise
  → redundancy within data and between data and control.

- Observation attacks rely on deterministic program execution.
  → randomized execution.

Randomization

Randomized control:

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

Randomized data (RSA message 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;
```

Redundancy within data

Redundant storage of data: store sensitive data $a_0, \ldots, a_n$ (e.g. a PIN code) as three arrays b, c and d such that

$$b[i] = \text{random integer} \qquad c[i] = b[i] \oplus a_i \qquad d[i] = c[i] \oplus C$$

where C is a constant, e.g. `0x55`.

Use $b[i] \oplus c[i]$ when a_i is needed, after checking that $c[i] \oplus d[i] = C$.

Checksumming of data: store sensitive data along with a checksum or crypto hash, and check the hash before any operation on the data.

Redundancy over control

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);
}
```

Summary

Hardware attacks can undermine even perfect software security.

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

Also a lot of work on hardware countermeasures, but most of it is trade secrets.

Software countermeasures are used in smart card operating systems, not yet in Java Card VMs. They remain poorly understood.

Formal understanding of software hardening

A challenge for semantics and programming language people:

- Develop probabilistic semantics that reflect the characteristics of hardware attacks

  (precise + irreversible or reversible + imprecise).

- 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?)

Conclusions and perspectives

Building a secure computer application

Generalizing from 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).

The “perfect component” fallacy

Security is a “holistic” property that cannot easily be reduced to independent sub-problems.

A perfect bytecode verifier is useless if API contain holes.

A perfect Java runtime environment is useless if hardware is easily attacked.

A perfect cryptographic protocol is useless if the keys are too small or not securely stored.

Security holes are often at the interface between two otherwise secure subsystems.

The language-based approach to security

Not a complete solution, but a set of tools and techniques that usefully complement more traditional approaches.

Pros and cons:

– The Trusted Computing Base is large.

  (VM + JIT + loader + verifier + APIs + . . .)

+ High portability, hardware and OS independence.

+ Standardized architectures and concrete issues that can be studied in-depth even by academics.

+ Formal methods apply well. Can reuse much experience gained with verification of safety properties.

The future of smart cards

Smart cards: the only example so far of mass deployment of highly secure computer systems.

Socially well accepted.

Java Card: both a blessing (opens up the area) and a curse (creates new security problems).

Smart cards remain confined to a few application areas.

Challenge: integrate them better in (distributed) computing infrastructures.