Formal methods for software security

Gilles Barthe

INRIA, Sophia-Antipolis, France

Programme

- Day 1: Formal methods for security
- Day 2: Progam verification
- Day 3: Information flow

Today

- Motivations
- Bytecode verification
- Proof Carrying Code
- The Mobius project
- State-of-the-art

Fact sheet

- Integrated Project, started Sept 2005, 4 years duration.
- Part of the FET pro-active initiative Global Computing II
- 16 members

INRIA	*LMU Münich*
RU Nijmegen	*ETH Zürich*
U. Edinburgh	*Chalmers U.*
Tallinn U.	*Imperial College*
UC. Dublin	*U. Warsaw*
UP. Madrid	*TLS*
SAP Research	*France Telecom*
Trusted Logic	*RWTH Aachen*
	(TU Darmstadt)

- Scientific Advisory Board:

  Martin Abadi *Amy Felty* *Rustan Leino*

Objective

Establish a security architecture appropriate for global computers:

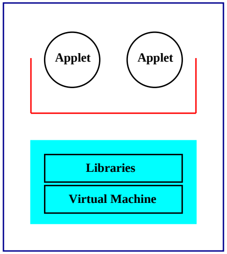
1. adopt a computational model that captures faithfully fundamental aspects of global computers
2. identify the trust and security requirements of such a model
3. develop on top of the computational model a security framework that enforces these requirements
4. provide the enabling technologies necessary for implementing the framework
5. validate the architecture

Computational model

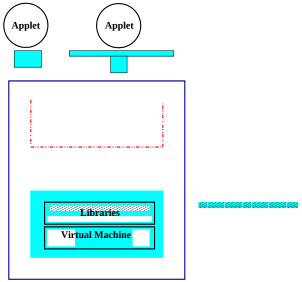
Very large distributed networks of JVM-enabled devices:

- *uniformity vs heterogeneity:* aimed at providing a global and uniform access to services, yet subject to resource constraints
- *flexibility vs security:* open architectures, yet subject to security constraints
- *mobile components:* code and computational infrastructures

General view of computational model



Current scenario



Mobius scenario

Security framework

Need of a framework:

- providing precise, efficient, resource-aware enforcement mechanisms
- for a wide range of expressive policies, including
  - Information flow and resource control
  - Framework-level and application-level
  - Safety and functional correctness
- for a widely deployed infrastructure
- with the highest guarantees

Security framework

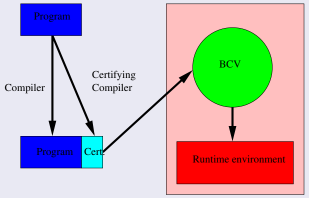
To deliver a framework with appropriate characteristics, we adopt ideas from *Proof Carrying Code* (PCC), proposed by Necula and Lee around 1996.
Downloaded components come equipped with certificates, where certificates:

- are condensed and formalized mathematical proofs/hints
- are self-evident and unforgeable,
- can be checked efficiently.

PCC certificate is different from standard crypto-based certificate.

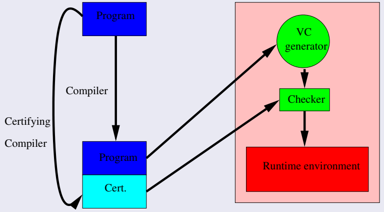
Main flavors of PCC

Type-based PCC



- Widely deployed in CLDC (and soon in J2SE)
- On-device checking is possible

Logic-based PCC



- Original scenario
- Applications to type safety and memory safety

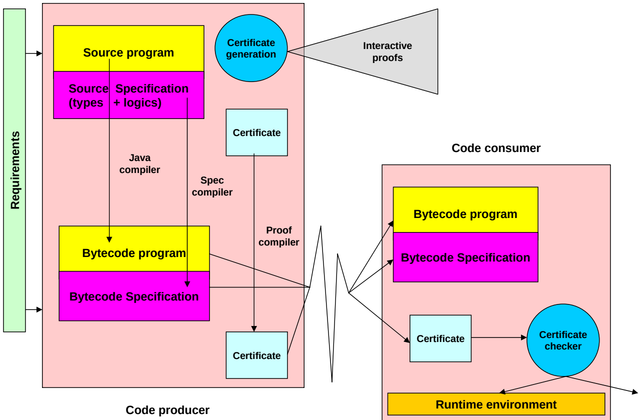
Enabling technologies

Enabling technologies should provide enough precision and automation to guarantee applicability and scalability.

- Type systems/static analyses:
  - efficient, automatic, but specialized and imprecise
  - used for information flow, resource usage, aliasing
- Program logics:
  - general, precise, but interactive
  - used for high-level security policies, non-functional properties, and functional correctness

The goal is to provide integration and support for type systems and logics within the Mobius tool.

Mobius vision



Specializing the architecture

In order to target different application domains, we can select different layers in the framework

- Enhanced bytecode verification for efficient and automatic verification of generic security properties
- Logical verification of basic security rules:
    - annotation assistants
    - proof inference
- Logical verification of complex security and functionality properties:
    - component validation
    - proof construction
    - proof checking

Scientific achievements to date

- *Requirements and scenarios*.
  - Information flow and resource control requirements
  - Framework specific and application specific requirements
  - Proof Carrying Code scenarios
- *Type-based verification*.
  - Information flow
  - Resource control
  - Aliasing
- *Logic-based verification*.
  - Bytecode logic, modeling language and verification condition generator
  - Prototype implementation of the verification environment
  - Initial results on multi-threading
- *PCC*.
  - Reflective Proof Carrying Code
  - Type-preserving and proof-transforming compilation

Software reliability

- Objective is to ensure that
    - software does not crash
    - software behaves as expected
    - software complies with resource constraints, e.g. timing
- Strong need for reliability in embedded and safety-critical software: bugs may cost lives and money

A famous example

Ariane 5:

- 10 years work,
- $7 billions,
- . . . for 39 seconds flight!

A famous example

Ariane 5:

- 10 years work,
- $7 billions,
- . . . for 39 seconds flight!

Software security

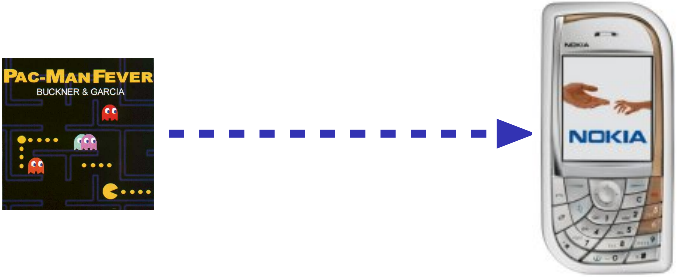
- Objective is to ensure that
  - Confidential information is not released
  - Integrity of data is preserved
  - Availability of resources is guaranteed
- Strong need for security in trusted personal devices: attacks may cost money, reputation, etc.

Mobile code

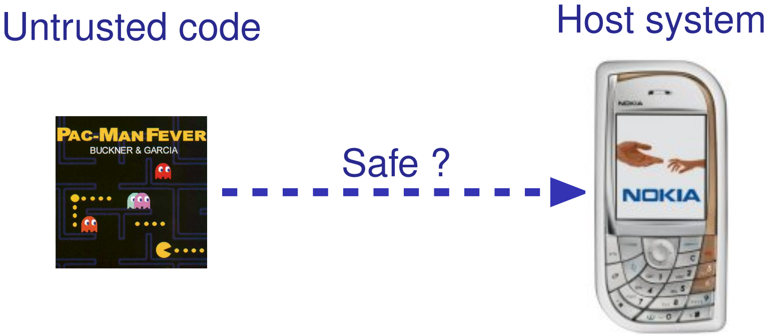
Execution of untrusted mobile code is increasingly ubiquitous:

- in web pages (JavaScript, Macromedia Flash)
- in trusted personal devices (mobile phones, smart cards)
- in embedded systems (aeronautics, automotive)

Mobile code dilemmas. . .



Mobile code dilemmas. . .

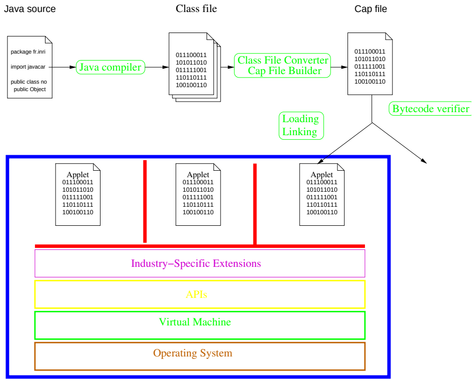


Mobile code risks

The untrusted code may

- cause damages on the system
  - destruction/corruption of files applications
- reveal sensitive information
  - passwords, financial information
- perform unauthorized actions
  - transactions, open sockets
- use too many resources
  - CPU, memory...
- be hostile towards other applications
  - (requiring applet isolation in multi-application smart cards)

Java(Card) runtime environment



Security challenge

Given an applet, how do we ensure that it is correct?

- Define correct: what does it mean?
  - secure: compliant with a policy
  - functionally correct
- Is the platform correct?
- Are the libraries correct?

Stack inspection: principles

- Each method has an associated set of permissions that determines the functionalities it can access; for example:
  - File permissions: read, write, delete;
  - Socket permissions: connect, accept connections
- Stack inspection regulates functionalities accessed by each method w.r.t. its set of permissions: an operation is performed only if all methods in the frame stack have permission to do.

Privileged code

- Stack inspection is effective when an untrusted principal calls an trusted principal: it protects the called method from the calling method.
- Code can be marked as privileged. This means that it grants all its permissions to its callers. It is useful to enable libraries to execute on behalf of untrusted code, e.g. applets may require reading fonts.

Stack inspection, informally

Repeat until the call stack is empty:

1. Check if current method has requested permission.
   If not, throw SecurityException.
2. Check if current method is privileged.
   If so, grant permission. (inspection stops)
3. Move on to calling method

History-based stack inspection

- Stack inspection is not effective to protect the calling method from the method called.
  1. If A calls B
  2. B returns
  3. A calls C

  Call to C depends on call to B: the dependency is missed by stack inspection
- Abadi and Fournet (2003): the run-time rights of a piece of code are determined by examining the attributes of any pieces of code that have run (including their origins) and any explicit requests to augment rights.

Beyond stack inspection

More expressive security policies can be expressed

- using e.g. security automata (Schneider)
- as temporal properties over control flow graphs (Jensen et al)

Both can account for history-based policies, but go beyond

Bytecode verification: goals

Bytecode verification aims to contribute to safe execution of programs by enforcing:

- Structural verification
- Values are used with the right types (no pointer arithmetic)
- Operand stack is of appropriate length (no overflow, no underflow)
- Subroutines are correct
- Object initialization

Bytecode verification: limits

Successful bytecode verification does not guarantee safety:

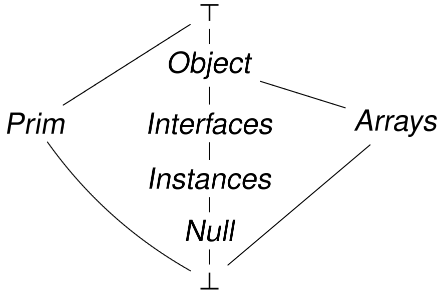
- array bound checks
- casts

must be enforced dynamically.

Bytecode verification: principle

- Bytecode verification is a data flow analysis based on a typed virtual machine. It is a conservative analysis (rejects all unsafe programs and potentially safe programs).
- Bytecode verification is modular: each method is analyzed on its own, using the method signature table to handle method calls.

The partial order of JCVM types



The typed abstract machine

- A JVM state consists of a heap and of a stack frame. Each frame contains a program counter, an operand stack and a register map
- A typed state contains a stack type *st* and a register type *rt*
- The typed abstract machine is defined by rules of the form

$$\frac{P[i] = instr \quad constraints}{i \vdash st, rt \Rightarrow st', rt'}$$

Sample rules

$$\frac{P[i] = iadd}{i \vdash int :: int :: st, rt \Rightarrow int :: st, rt}$$

$$\frac{P[i] = iconst\ n \quad |st| + 1 \leq Mstack}{i \vdash st, rt \Rightarrow int :: st, rt}$$

$$\frac{P[i] = aload\ n \quad rt(n) = \tau \quad \tau \prec Object \quad |st| + 1 \leq Mstack}{st, rt \Rightarrow \tau :: st, rt}$$

$$\frac{P[i] = astore\ n \quad \tau \prec Object \quad 0 \leq n < Mreg}{i \vdash \tau :: st, rt \Rightarrow st, rt[n \leftarrow \tau]}$$

$$\frac{P[i] = getfield\ C\ f\ \tau \quad \tau' \prec C}{i \vdash \tau' :: st, rt \Rightarrow \tau :: st, rt}$$

$$\frac{P[i] = putfield\ C\ f\ \tau \quad \tau_1 \prec \tau \quad \tau_2 \prec C}{i \vdash \tau_1 :: \tau_2 :: st, rt \Rightarrow st, rt}$$

Bytecode verification: implementation

- Different analyses are possible, with different assumptions to guarantee termination.
- Analyses use KILDALL algorithm.
    - Stackmaps collect execution history for each program point
    - Initially all program points except the entry point store uninitialized stackmaps
    - Unification “injects” a state into an history

Kildall algorithm: main loop

- Pick a program point `i` marked as true
- execute with the abstract virtual machine instruction `insi` on state `sti,lvi`
- For all successors `j` of `i`, merge the result `sti-res,lvi-res` with `stj,lvj`
- mark successors as true if the merge is not the identity.

Merging: monovariant verification

Original algorithm, does not handle polymorphic subroutines

history structure	=	one abstract state
merging	=	lub
termination	=	no infinite ascending chain

Verification example

PC	instruction	1st pass
0:	`jsr 10`	$(),[\top;\top]$
1:	`iconst_0`	
2:	`istore_0`	
3:	`jsr 10`	
4:	`iload_0`	
10:	`astore_1`	
11:	`ret 1`	

States : (operand stack), [local variables]

Verification example

PC	instruction	1st pass
0:	`jsr 10`	$()$,$[\top;\top]$
1:	`iconst_0`	
2:	`istore_0`	
3:	`jsr 10`	
4:	`iload_0`	
10:	`astore_1`	(`RA 0`),$[\top;\top]$
11:	`ret 1`	

States : (operand stack), [local variables]

Verification example

PC	instruction	1st pass
0:	`jsr 10`	(),[$\top$;$\top$]
1:	`iconst_0`	
2:	`istore_0`	
3:	`jsr 10`	
4:	`iload_0`	
10:	`astore_1`	(`RA 0`),[$\top$;$\top$]
11:	`ret 1`	(),[$\top$;`RA 0`]

States : (operand stack), [local variables]

Verification example

PC	instruction	1st pass
0:	`jsr 10`	(),[⊤;⊤]
1:	`iconst_0`	(),[⊤;RA 0]
2:	`istore_0`	
3:	`jsr 10`	
4:	`iload_0`	
10:	`astore_1`	(RA 0),[⊤;⊤]
11:	`ret 1`	(),[⊤;RA 0]

States : (operand stack), [local variables]

Verification example

PC	instruction	1st pass
0:	`jsr 10`	(),[⊤;⊤]
1:	`iconst_0`	(),[⊤;RA 0]
2:	`istore_0`	(int),[⊤;RA 0]
3:	`jsr 10`	
4:	`iload_0`	
10:	`astore_1`	(RA 0),[⊤;⊤]
11:	`ret 1`	(),[⊤;RA 0]

States : (operand stack), [local variables]

Verification example

PC	instruction	1st pass
0:	`jsr 10`	$()$,$[\top;\top]$
1:	`iconst_0`	$()$,$[\top;$`RA 0`$]$
2:	`istore_0`	$($`int`$)$,$[\top;$`RA 0`$]$
3:	`jsr 10`	$()$,$[$`int`$;$`RA 0`$]$
4:	`iload_0`	
10:	`astore_1`	$($`RA 0`$)$,$[\top;\top]$
11:	`ret 1`	$()$,$[\top;$`RA 0`$]$

States : (operand stack), [local variables]

Verification example

PC	instruction	1st pass	2nd pass
0:	`jsr 10`	(),[$\top$;$\top$]	
1:	`iconst_0`	(),[$\top$;RA 0]	
2:	`istore_0`	(int),[$\top$;RA 0]	
3:	`jsr 10`	(),[int;RA 0]	
4:	`iload_0`		
10:	`astore_1`	(RA 0),[$\top$;$\top$]	(RA 3),[int;RA 0]
11:	`ret 1`	(),[$\top$;RA 0]	

Choice of verification

Verification example

PC	instruction	1st pass	2nd pass
0:	`jsr 10`	(),[$\top$;$\top$]	
1:	`iconst_0`	(),[$\top$;RA 0]	
2:	`istore_0`	(int),[$\top$;RA 0]	
3:	`jsr 10`	(),[int;RA 0]	
4:	`iload_0`		
10:	`astore_1`	(RA 0),[$\top$;$\top$]	(RA 3),[int;RA 0]
11:	`ret 1`	(),[$\top$;RA 0]	

Failure of monovariant verification

Verification example

PC	instruction	1st pass	2nd pass
0:	`jsr 10`	(),[$\top$;$\top$]	
1:	`iconst_0`	(),[$\top$;`RA 0`]	
2:	`istore_0`	(`int`),[$\top$;`RA 0`]	
3:	`jsr 10`	(),[`int`;`RA 0`]	
4:	`iload_0`		
10:	`astore_1`	(`RA 0`),[$\top$;$\top$]	(`RA 3`),[`int`;`RA 0`]
11:	`ret 1`	(),[$\top$;`RA 0`]	

Polyvariant verification succeeds

Verification example

PC	instruction	1st pass	2nd pass
0:	`jsr 10`	(),[$\top$;$\top$]	
1:	`iconst_0`	(),[$\top$;`RA 0`]	
2:	`istore_0`	(`int`),[$\top$;`RA 0`]	
3:	`jsr 10`	(),[`int`;`RA 0`]	
4:	`iload_0`		
10:	`astore_1`	(`RA 0`),[$\top$;$\top$]	(`RA 3`),[`int`;`RA 0`]
11:	`ret 1`	(),[$\top$;`RA 0`]	(),[`int`;`RA 3`]

Verification example

PC	instruction	1st pass	2nd pass
0:	`jsr 10`	$()$,$[\top;\top]$	
1:	`iconst_0`	$()$,$[\top;$`RA 0`$]$	
2:	`istore_0`	(`int`),$[\top;$`RA 0`$]$	
3:	`jsr 10`	$()$,[`int`;`RA 0`]	
4:	`iload_0`	$()$,[`int`;`RA 3`]	
10:	`astore_1`	(`RA 0`),$[\top;\top]$	(`RA 3`),[`int`;`RA 0`]
11:	`ret 1`	$()$,$[\top;$`RA 0`$]$	$()$,[`int`;`RA 3`]

Handling subroutines with polyvariant verification

Does handle polymorphic subroutines, characterizes programs accepted by abstract VM

history structure	=	sets of abstract states
merging	=	set addition
termination	=	finite number of abstract states

Merging: hybrid verification

Limits memory consumption

history structure	=	sets of abstract states
merging	=	lub set addition
termination	=	finite number of abstract states + no infinite ascending chain

Other challenges with bytecode verification

- interfaces:

  ```
  interface I { ... }
  interface J { ... }
  class C implements I, J { ... }
  class D implements I, J { ... }
  ```

  Both I and J are upper bounds for C and D, but I and J are incomparable.
- initialization: enforced by side effect, not monotonic!

Bytecode verification vs. defensive virtual machine

- Defensive JCVM is closest to specification:
  - It manipulates typed values
  - Types are checked at run-time
- Offensive JCVM is closest to implementation:
  - It manipulates untyped values
  - Type correctness enforced by BCV
- Typed JCVM used in bytecode verification:
  - Manipulates types as values
  - Operates on a method-per-method basis

Cross Validation

Main property: Defensive and offensive VMs coincide on programs that are well-typed for the typed VM

- offensive and defensive VMs coincide on programs well-typed for the defensive VM
- programs that are ill-typed for the defensive VM are ill-typed with the typed VM

Best viewed as *some form* of correctness of abstractions

Offensive vs. Defensive

- Offensive VM is derived from defensive VM
- Derivation function removes types from typed values
- Diagram commutes

$$\begin{tikzcd}
\mathtt{dstate} \arrow[r, "\mathtt{dexec}"] \arrow[d, "\alpha_{do}"] & \mathtt{drstate} \arrow[d, "\alpha^{r}_{do}"] \\
\mathtt{ostate} \arrow[r, "\mathtt{oexec}"] & \mathtt{orstate}
\end{tikzcd}$$

Offensive vs. Defensive

- Offensive VM is derived from defensive VM
- Derivation function removes types from typed values
- Diagram commutes

$$\begin{tikzcd}
\texttt{dstate} \arrow[r, "\texttt{dexec}"] \arrow[d, "\alpha_{do}"] & \texttt{drstate} \arrow[d, "\alpha^{r}_{do}"] \\
\texttt{ostate} \arrow[r, "\texttt{oexec}"] & \texttt{orstate}
\end{tikzcd}$$

if defensive VM does not raise typing errors

Typed vs. Defensive

- Typed VM is derived from defensive VM
- Derivation function removes value from typed values
- Diagram commutes

$$\begin{tikzcd}
\mathtt{dstate} \arrow[r, "\mathtt{dexec}"] \arrow[d, "\alpha_{da}"] & \mathtt{drstate} \arrow[d, "\alpha^{r}_{da}"] \\
\mathtt{astate} \arrow[r, "\mathtt{aexec}"] & \mathtt{arstate}^{\star}
\end{tikzcd}$$

Typed vs. Defensive

- Typed VM is derived from defensive VM
- Derivation function removes value from typed values
- Diagram commutes

$$\begin{tikzcd}
\mathtt{dstate} \arrow[r, "\mathtt{dexec}"] \arrow[d, "\alpha_{da}"] & \mathtt{drstate} \arrow[d, hook, "\alpha^{r}_{da}"] \\
\mathtt{astate} \arrow[r, "\mathtt{aexec}"] & \mathtt{arstate}^{\star}
\end{tikzcd}$$

if “execution keeps in the same frame”

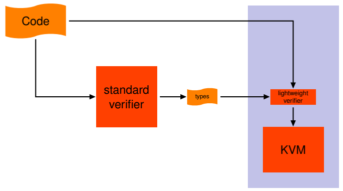
Variants of bytecode verification

In order to perform verification on-card, more resource-aware algorithms must be designed.

- Rose, KVM: Lightweight bytecode verification
- Leroy: Code rewriting and one-pass verification

Lightweight Bytecode Verifier

Part of KVM (JVM for embedded devices).



- safety policy = standard Java type safety
- the certificate is a type annotation
- verification is fast and requires very few memory

Discussing Rose's approach

Advantages:

- certificates are small,
- certificate verification is fast,
- certificate generation is automatic.

Limitations:

- security policy is simple and fixed

Beyond bytecode verification

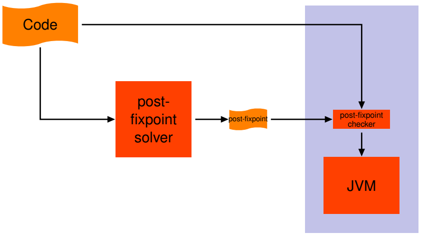
Abstract Interpretation [Cousot and Cousot 77] is a method for designing approximate semantics of programs.

- application to static analysis: static analysers are computable approximate semantics of programs
- a static analysis is presented as a post-fixpoint $[\![p]\!]^\sharp$ of a functional $F^\sharp$ in a lattice

$$F^\sharp\left([\![p]\!]^\sharp\right) \sqsubseteq [\![p]\!]^\sharp$$

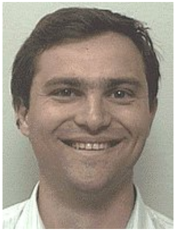
- $[\![p]\!]^\sharp$ is computed by complex iterative methods
- but checking a given post-fixpoint is fast

Abstract interpretation-based lightweight verification



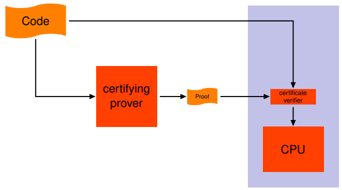
- no extra-annotation is required
- there are generic techniques for fixpoint compression
- but each post-fixpoint checker is ad-hoc

Pioneers



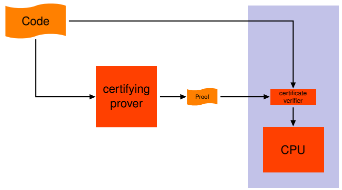
- Necula & Lee, *Safe Kernel Extensions Without Run-Time Checking*, OSDI'96
- Necula, *Proof-Carrying Code*, POPL'97
- Necula & Lee, *The Design and Implementation of a Certifying Compiler*, PLDI'98

Proof carrying code: standard framework



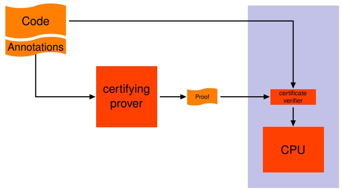
- based on axiomatic semantics,
- code is annotated (loop invariant),
- the VCGen computes logic formulae to be proved,
- the certifying prover computes *proof object*,
- the consumer builds formulae and checks proof objects against them

Proof carrying code: standard framework



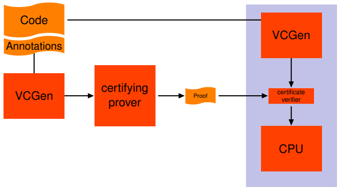
- based on axiomatic semantics,
- code is annotated (loop invariant),
- the VCGen computes logic formulae to be proved,
- the certifying prover computes *proof object*,
- the consumer builds formulae and checks proof objects against them

Proof carrying code: standard framework



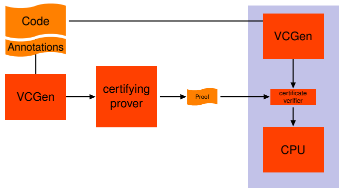
- based on axiomatic semantics,
- code is annotated (loop invariant),
- the VCGen computes logic formulae to be proved,
- the certifying prover computes *proof object*,
- the consumer builds formulae and checks proof objects against them

Proof carrying code: standard framework



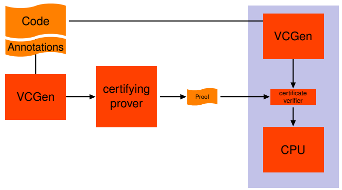
- based on axiomatic semantics,
- code is annotated (loop invariant),
- the VCGen computes logic formulae to be proved,
- the certifying prover computes *proof object*,
- the consumer builds formulae and checks proof objects against them

Proof carrying code: standard framework



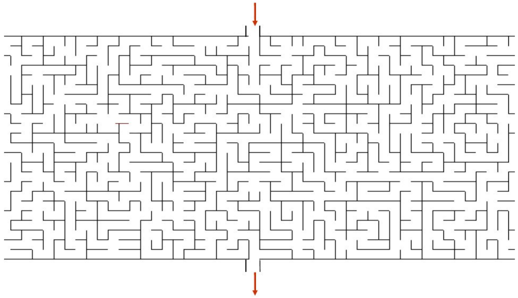
- based on axiomatic semantics,
- code is annotated (loop invariant),
- the VCGen computes logic formulae to be proved,
- the certifying prover computes *proof object*,
- the consumer builds formulae and checks proof objects against them

Proof carrying code: standard framework



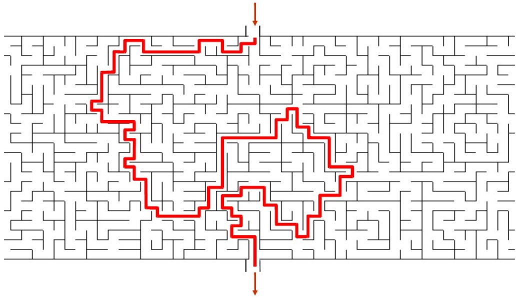
- based on axiomatic semantics,
- code is annotated (loop invariant),
- the VCGen computes logic formulae to be proved,
- the certifying prover computes *proof object*,
- the consumer builds formulae and checks proof objects against them

The maze metaphor (©G. Necula)



program = maze

The maze metaphor (©G. Necula)



program = maze

proof = red path

What is a certificate?

- are condensed and formalized mathematical proofs/hints
- are self-evident and unforgeable,
- can be checked efficiently.

PCC certificate is different from standard crypto-based certificate.

Certificates are tamper-proof

What happens if the program or the certificate are modified?

- the proof is no longer valid: the program is rejected
- the proof is valid: the program is accepted

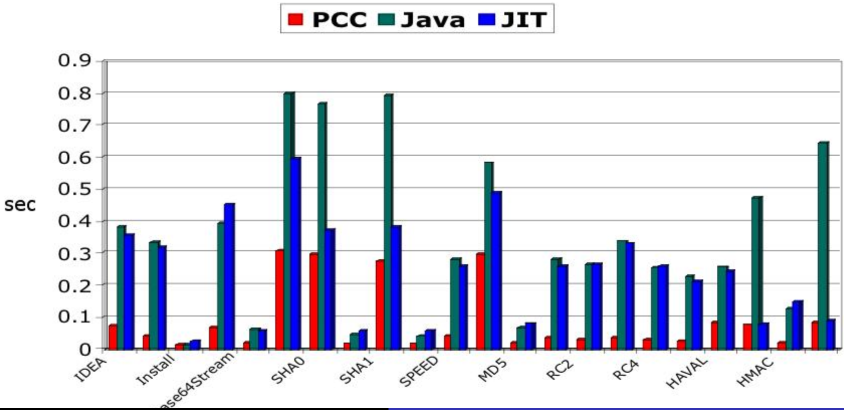
In the second case the guarantee still holds.

Advantages of PCC

- No need to trust the code producer nor the compiler
- No need for cryptography
- No runtime overhead
- Flexible policies can be supported

Optimizing native code using PCC

A successful application of standard PCC: compiling Java bytecode into optimised native code while ensuring type safety.



Pros and cons of standard PCC

Advantages:

- the verifier is efficient
- the verifier is (certainly) sound : well-understood techniques,
- the verifier is generic

Disadvantages:

- certificates are big
- soundness of the whole architecture relies on the VCGen
    - but VCGen is not bug-free!

What's a proof?

Follows Curry-Howard isomorphism.

Curry-Howard isomorphism

theorem	$\longleftrightarrow$	type
proof	$\longleftrightarrow$	λ-term
proof checker	$\longleftrightarrow$	type-checker

Other certificate formats

Trade-off between size of certificates and complexity of certificate checkers:

- Implicit λ-terms
- Tactic-based PCC
- Oracle-based PCC

Other approaches could be envisioned:

- probabilistic certificates

Foundational PCC

- Specify machine semantics directly in higher-order logic
  - avoids using a VCGen
- Prove typing rules as derived lemmas
  - removes type system from TCB

Pros and cons of FPCC

- Pros
  - increased security (smaller TCB)
  - increased flexibility (lesser commitment to a particular type system)
- Cons
  - machine semantics may be complex to formalize
  - formal verification of the typing rules is costly

The story so far. . .

It is very difficult to combine all PCC requirements:

- the certificates must be small,
- the verifier must be efficient,
- the verifier must be sound,
- the proof producer must exists,
- the safety policy must be expressive

Still an active research trend. . .

Mobius goals

- Scenarios
- Frameworks
- Requirements

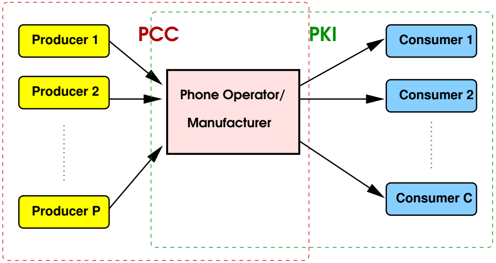
PCC with Intermediaries

- PCC with trusted intermediary
  - midlet that originates from untrusted party.
  - Signed by phone operator after checking the PCC certificate showing the midlet complies with the operator requirements.
  - Combines PCC and PKI.
  - Size of certificates is not an issue.
- PCC with untrusted intermediaries
  - application is transferred through intermediaries that modify its code
  - Each modification should be justified by certificate translation.

.

The Mobius scenario

- MOBIUS promotes *trust through verifiable evidence* (PCC). Can be combined with *trust by authority* or *trust by reputation*.
- Phone operators/manufacturers can act as trusted intermediaries:



- Provides an appropriate trade-off between feasibility and flexibility which will be exploited in the rest of the project.

Multiple Code Producers — Impact of PCC

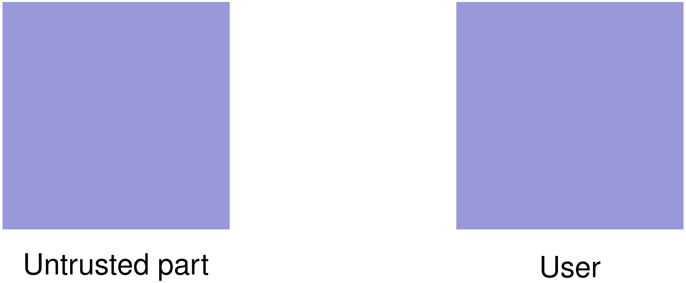
- (distributed) applications consist of compositions of many components from many sources
- fundamental changes to component-based development practices
    - component selection with PCC and generalised certificates
    - changes to the software development process
    - liability and culpability

Multiple Code Consumers

- In PCC, it is natural to have multiple code consumers.
- Scenarios with multiple consumers:
    - mutual trust among consumers: compositional verification
    - mutual distrust among consumers: result certification
    - large-scale distributed checking: evolving trust
    - dynamic composition of PCC components

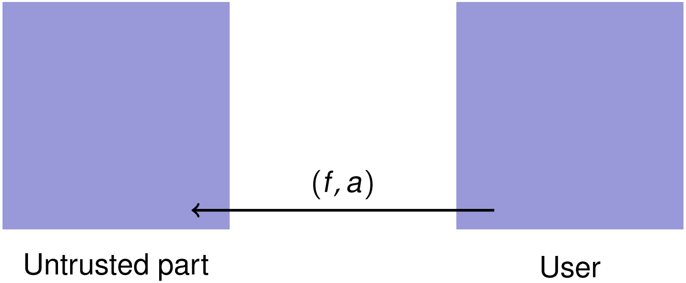
A parenthesis on result checking: principles

Given a function $f \in A \rightarrow B$ and an entry $a \in A$, the user delegates the computation $f(a) \in B$ to an untrusted part.



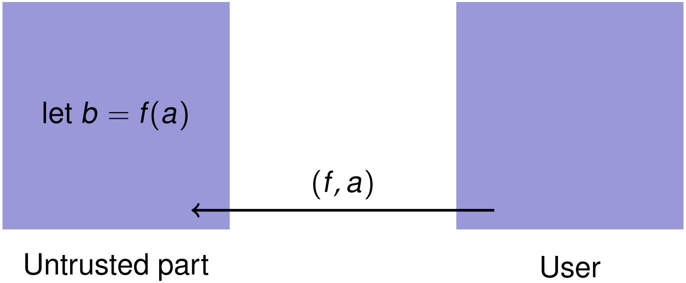
A parenthesis on result checking: principles

Given a function $f \in A \rightarrow B$ and an entry $a \in A$, the user delegates the computation $f(a) \in B$ to an untrusted part.



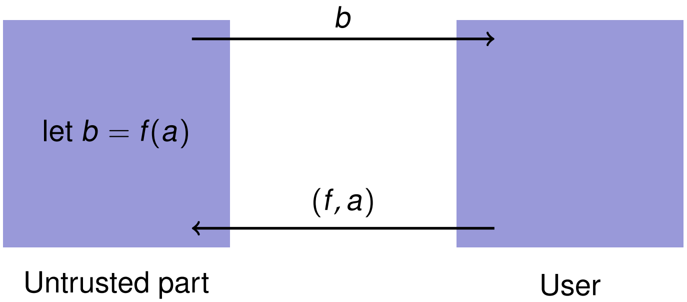
A parenthesis on result checking: principles

Given a function $f \in A \rightarrow B$ and an entry $a \in A$, the user delegates the computation $f(a) \in B$ to an untrusted part.



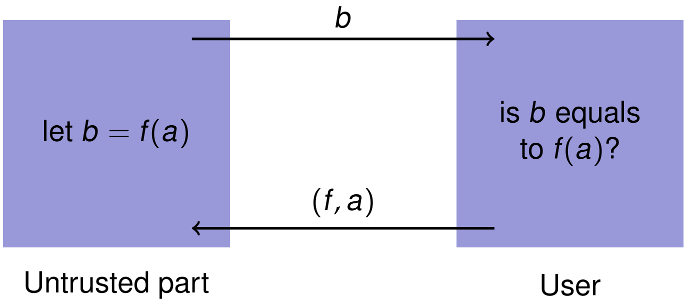
A parenthesis on result checking: principles

Given a function $f \in A \to B$ and an entry $a \in A$, the user delegates the computation $f(a) \in B$ to an untrusted part.



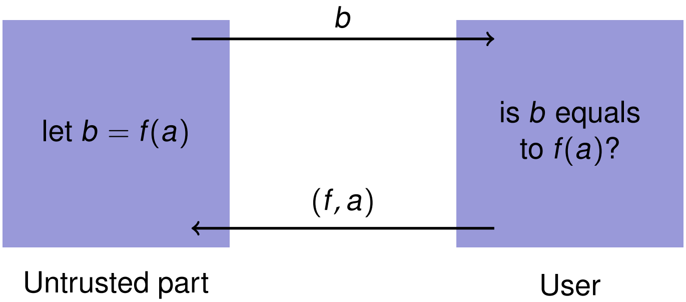
A parenthesis on result checking: principles

Given a function $f \in A \rightarrow B$ and an entry $a \in A$, the user delegates the computation $f(a) \in B$ to an untrusted part.



A parenthesis on result checking: principles

Given a function $f \in A \to B$ and an entry $a \in A$, the user delegates the computation $f(a) \in B$ to an untrusted part.



Idea : propose a checker $\mathtt{check}_f \in A \times B \to \mathtt{bool}$ s.t.

$$\forall a \in A,\ b \in B,\ \mathtt{check}_f(a, b) = \mathtt{true} \ \Rightarrow\ b = f(a)$$

Result-Checking requirements

Let $R \subseteq A \times B$ represents the expected behaviour of f.

$$\forall a \in A,\ (a, f(a)) \in R$$

Challenge in Result-Checking: find a checking function $\mathtt{check}_f$

1. correct : $\forall a \in A, b \in B,\ \mathtt{check}_f(a, b) = \mathtt{true} \Rightarrow (a, b) \in R$
2. efficient : $\mathrm{complexity}(f, a) < \mathrm{complexity}(\mathtt{check}_f, a)$

Efficiency can be relaxed if we focus on correctness of the computation.

- Result-Checking as a new proof technique : prove $\mathtt{check}_f$ instead of f

The naive checker

A naive checker always exists.

$$\mathrm{check}_f(a, b) := (b = f(a))$$

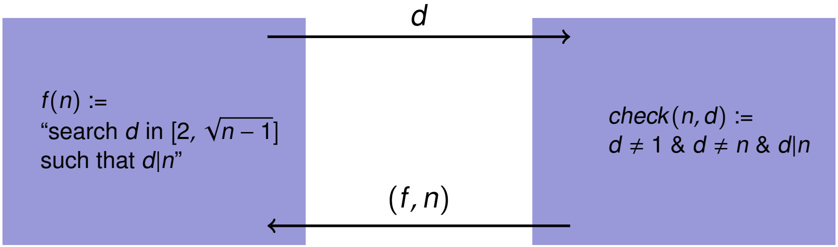
but, of course:

- correctness of $\mathtt{check}_f$ is then as hard as that of f!
- $\mathrm{complexity}(f, a) \sim \mathrm{complexity}(\mathrm{check}_f, a)$!

Hopefully, sometimes an efficient checker exists.

Result is sometimes enough

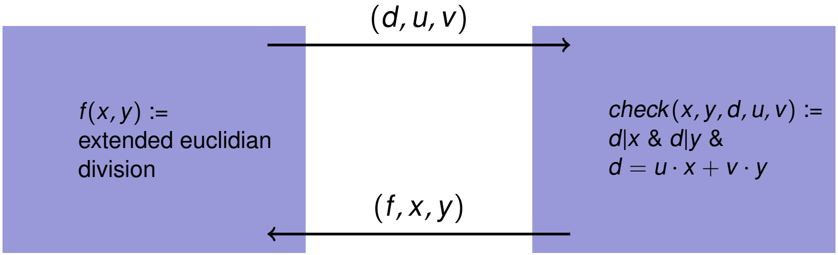
Example : finding a non trivial divisor of an integer


d
$f(n) :=$ "search d in $[2, \sqrt{n-1}]$ such that $d|n$"
$check(n,d) :=$ $d \neq 1$ & $d \neq n$ & $d|n$
(f,n)

But $f(a)$ is often not sufficient.

Result is not always enough

Example : compute a gcd of two integers x and y


(d,u,v)
$f(x,y) :=$ extended euclidian division
$check(x,y,d,u,v) :=$ $d|x$ & $d|y$ & $d = u \cdot x + v \cdot y$
(f,x,y)

gcd(x,y) is not enough!

Result is not enough

The checker takes an extra hint as argument:

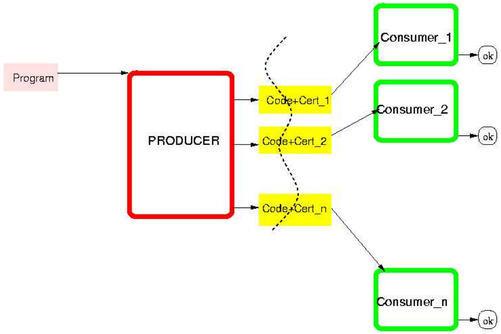
$$\mathtt{check}_f \in A \times B \times H \to \mathtt{bool}$$

Certificate = result + hint

The producer algorithm must do an extra job :

- in addition to compute the awaited result,
- to compute a hint for the checker

Generic Code and Certificates

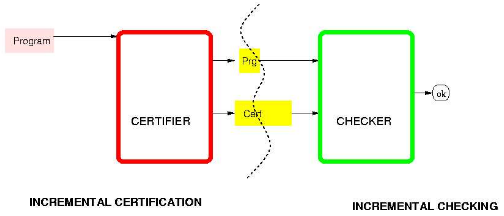


PCC with Generic Code and Certificates

- There are situations where the code+certificate generated by the producer is *generic*:
    - it may contain a large number of different uses. Often, only a small fraction of them is required in each consumer.
    - the code may be self tuning. Depending on the features of the platform, it may use different execution strategies.
    - sometimes, we may use partial evaluation techniques in order to optimize code for a particular context.
    - also, sometimes the certificate needs to be instantiated with platform-dependent information. A good example is certificates related to execution time
- Before being deployed in each consumer, the code+certificate bundle may have to be instantiated to the particular environment.

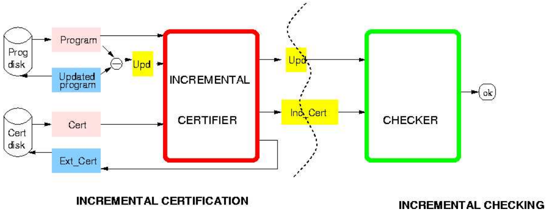
Incremental Proof-Carrying Code

- Standard PCC: Full program and certificate are submitted
- Incremental Certification: Only updates and incremental certificate are submitted. The original program and certificate must have been stored on disk.
- Incremental Checking: Only check the new updates (and indirectly affected information). Similar process to update the original program and certificate and store on disk.



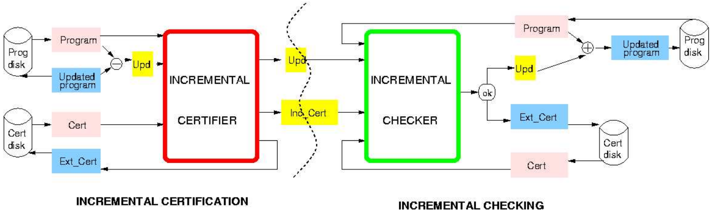
Incremental Proof-Carrying Code

- Standard PCC: Full program and certificate are submitted
- Incremental Certification: Only updates and incremental certificate are submitted. The original program and certificate must have been stored on disk.
- Incremental Checking: Only check the new updates (and indirectly affected information). Similar process to update the original program and certificate and store on disk.

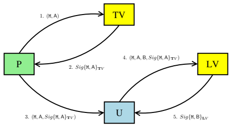


Incremental Proof-Carrying Code

- Standard PCC: Full program and certificate are submitted
- Incremental Certification: Only updates and incremental certificate are submitted. The original program and certificate must have been stored on disk.
- Incremental Checking: Only check the new updates (and indirectly affected information). Similar process to update the original program and certificate and store on disk.



PCC with Multiple Verifiers


TV
1. (M, A)
P
2. $Sig\{M, A\}_{TV}$
4. (M, A, B, $Sig\{M, A\}_{TV}$)
LV
U
3. (M, A, $Sig\{M, A\}_{TV}$)
5. $Sig\{M, B\}_{LV}$

TV: Trusted Verifier	**LV**: Local Verifier
P: Code Provider	**U**: Code User
A: Global Security Policy	**B**: Local Security Policy

- This framework allows different consumers to have different security policies.
- It requires the use of PKI for trusting non-local verifiers.

Application frameworks: overview

- Choice of the framework
- A threat model
- Security issues
- Application certification
- The actors

Framework selection criteria

- Global computing
  - Mobile code framework
  - Widely available on client devices
- Established standard
  - Accepted by the industry
  - Evolving and following the evolution of technology
- Open to reasoning on programs
  - Controlled execution environment
  - Code structured for analysis

Which frameworks?

- CLDC/MIDP
  - Available on many devices (over a billion)
  - Interesting subset of the Java language
  - Full framework, thousands of applications exist
- CDC
  - Only available on high-end phones (smart phones)
  - Supports the entire Java language
  - More recent standardization
- STIP
  - Available on very few devices
  - Dedicated to professional services

Frameworks and requirements

Focus on CLDC/MIDP

- Standardization is very active
- MIDP 2.0 has been available for a while
- Many additional API's have been defined
- MIDP is representative of our target
- Already includes PCC (lightweight bytecode verifier)
- Targets typical global computing client devices

A threat model for MIDP

- Goods to be protected
- Possible goals for attackers
- Common attack types
- Possible countermeasures

Assets to be protected

- Operator assets
  - Billable operations
  - Reputation
  - Network infrastructure
- End-user assets
  - Money (through operator billing)
  - Private information
  - In particular, confidential information

Goals of attackers

- Make money
  - Send messages to a premium number
- Steal sensitive information
  - Contact information (for spam)
  - Passwords, PIN codes, ...
- Attack an operator
  - Try to damage its reputation
- Perform a hacker stunt
  - Just showing off some attack skills
- Perform a terrorist act
  - Take the entire network down at a given time

Attack scenarios

- Many attack scenarios are possible
  - Attacking the phone's Java environment
  - Attacking the phone's operating system or hardware
  - Social engineering
- Not all of them are interesting
  - Some attacks are too sophisticated
  - Requirements are available from operators

Attacking the Java VM

- Identify bugs in the implementation
  - Bug in the VM or missing test in the verifier
  - Bug in the API implementation (buffer overflow, ...)
- Exploit the bugs through specifically designed applications

Still a big potential

- The quality of the implementations is not ideal
- The attacker/developer has unlimited access to the device
- The attack itself does not require any access (Trojan horse)

Attacking the OS

- Interfering with the OS has many interests
  - Gaining access to loaded applications
    - Reading/modifying an application's code
    - Reading/modifying an application's data
  - Modifying system data
    - Modifying/adding a public key used to certify applications
  - Modifying system code
    - Removing some checks in the Java implementation
- Was easy, becomes more difficult with time
  - More countermeasures are added (specific hardware)
  - Requires access to the device (out of scope)

Social engineering

- Manipulating the end-user to enable the attack
- Very interesting attack for MIDP
  - The user must authorize some operations
  - Some messages may be difficult to understand
- Good potential
  - The application may look perfectly legal
  - The application requires “normal” privileges (provide personal information to play with friends, authorize network connections to retrieve new levels)
  - No need to access the device directly

Some countermeasures

- Reactive countermeasures
  - Early detection (hotline monitoring)
- Proactive countermeasures
  - Extensive application testing
  - Static analysis (checking a security policy)

In current certification schemes:

- Java Verified is gaining ground
  - Functional testing, with standard requirements
  - Supported by manufacturers and operators
- Security certification is starting
  - Code review for sensitive operator applications (such applications have extended privileges)
  - Static analysis for some operators

Security issues in MIDP 2.0

Many mechanisms are now available to applications

- Making phone calls
- Sending/receiving SMS and MMS (messaging)
- Internet communication: http, https, sockets, datagram
- Application auto invocation
- Accessing Bluetooth and serial port
- Recording video and/or audio
- Accessing a user’s private information: contacts, appointments, notes, etc.
- Location information

MIDP 2.0 security

- Sensitive operations require a permission
  - The application must request the permission
  - The platform decides whether or not to grant it
- The decision process includes user interactions
  - The user decides on operations that incur a cost
    - Sending a SMS
    - Establishing a network connection
  - The user can manage its security policy, e.g. systematically allowing/forbidding some operations
- The decision process includes signature verification: some permissions can only be granted to signed applications

Are all issues addressed?

- Basic issues are addressed
  - All sensitive operations must be authorized
  - Interaction with the user is under control
- Typical threats are still present
  - Social engineering: always possible whenever there is an interaction
  - Design/implementation bugs and weaknesses
    - Java does not provide a complete protection
    - Proprietary features exist

The actors (1)

- Telecom operators
  - Support directly the cost of attacks
  - Ultimately responsible for the applications deployed
  - Define their own policies (portability, security, etc.)
- Application developers
  - Need to deploy their applications
  - Need to show that they follow operator policies
- End users
  - Want to protect their devices
  - Want to increase the choice of applications
  - Want to have a say in the security policy

The actors (2)

- Tool vendors
  - Their motto: automated is beautiful
- Testing houses
  - Their motto: manual is beautiful
- Technology providers and stakeholders
  - The main one: Sun Microsystems
    - Their motto: Java is secure
    - Application security is not on their agenda

Security requirements: overview

- Categories of properties
- Some properties of network connections
- Analyzing restrictions on method invocation
- Some challenges

Categories of properties

- Basic checks (methods used. . .)
- Safety policies (exceptions. . .)
- API usage (network connections, billable events. . .)
- Resource usage (memory and time consumption, restrictions on the values of some parameters)
- Mandatory sequences of operations
- Information flow (e.g. personal data must not be exported/must only be exported to known hosts/to the owner of the application)
- Functional correctness

Example of network properties

- Connection types
    - All network connections are HTTP(S) connections
    - All network connections are secure
- Connection destinations
    - All destination hosts are determined
    - The application only connects to its origin host

One example of network property

- All network connections are HTTP connections
- How are network connections initiated?
    - Connector.open(String url)
    - Connector.openInputStream(String url)
    - Manager.createPlayer(String url)
    - . . .
- What are network connections?
    - URL's starting with http, https, . . .
    - . . . and ssl, socket, datagram

The actual property

For all methods that open a network connection (there are many such methods) restrict the value of the URL parameter

- If the parameter value is unknown, property is violated or undetermined
- If the parameter starts with http://, property is satisfied
- If the parameter starts with https://, ssl://, socket://, datagram://, property is violated

API usage restrictions

Analyzing the values of arguments

- Analyzing integral values: very often constants, thus quite easy to analyze
- Analyzing Strings and URLs
    - Strings are heavily used in MIDP
    - In particular, essential APIs are based on URLs
    - Analyzing string values is essential and difficult

API usage: restrictions on string parameters

Back to example: all network connections must be HTTP connections

- The URL is a constant string

```
Connection conn;
conn = Connector.open("http://www.inria.fr");
```

  - The analysis is simple, both in source and bytecode
  - Strings are immutable, so threads definitely not a problem

- The URL is a concatenation, the required info is constant

```
Connection home;
home = Connector.open("http://" + host);
```

The source analysis is simple

Analyzing strings: problems at bytecode level

The bytecode analysis is more complex

- Concatenation is compiled with StringBuffer objects
- StringBuffer objects are not immutable, and can be used for other purposes

```
Connection home ;
StringBuffer sb = new StringBuffer("http://");
home = Connector.open(
sb.append(host).toString()) ;
```

Analyzing strings: method calls

- The URL is computed in another method
- The analysis cannot be local to a method any more
- The threads can become more of a problem
- All accessible code is known, provided we work on a fixed code base

```
Connection comm ;
// First sets the connection
url = myStuff.computeURL() ;
// Then opens the connection
comm = Connector.open(url) ;
```

Analyzing strings: arrays and containers

- The URL is stored in an array or in a container
- The analysis cannot be local to a method any more
- The threads become a real problem
- The content of the array/container must be analyzed
- Difficult with arrays, vectors, and hashtables
- Very difficult with record stores and files

```
Connection comm;
// Opens the connection
comm = Connector.open(urls[index]);
```

Analyzing strings: unknown input

Two typical cases

- The URL comes from another connection (web page)
- The URL (or part of it) is entered by the user

```
Connection comm ;
// Opens the connection if HTTP link
if (url.startsWith("http:"))
comm = Connector.open(url) ;
```

Information flow: which restrictions?

- Personal data must not be exported
    - Nothing personal should reach the network
- Personal data must not be exported in clear
    - Confidentiality protection against external attacks
- Personal data must only be exported to known hosts
    - Protecting the destination of the data
- Personal data must only be sent to the owner of the application
    - Forcing the owner to take responsibility
    - Forcing SSL, in a connection to the owner's server

Application-specific security requirements

- Security requirements for a particular application
  - MIDlet or platform component
  - In addition to platform-specific requirements
- Application-specific security requirements can be
  - Specialized versions of platform-specific requirements
  - Additional requirements to protect assets of the application
  - Detailed prescription of specific behavior, rather than generic restriction on behavior
  - Requires certification of functional properties

Examples considered

- Example MIDP applications
  - Instant messenger
  - Pocket Protector
  - E-voting Midlet
- Example platform components
  - Access controller
  - Bytecode verifier
  - TCP/IP
  - SSH Midlet

Pocket Protector: application

- MIDlet for storing valuable data protected by password: passwords, PIN code, credit card numbers, ...
- Uses the MIDP file system (Record store)

Example of security requirements:

- The record store cannot be accessed
- MIDlet creates only one file, with correct access rights
- No access to file without first giving the master password
- Second defense line: the data must remain secure
- All data in the file must be encrypted

Pocket Protector: analysis of requirements

- MIDlet creates only one file, with correct access rights: simple conditions on an API method invocation
- All data in the record store must be encrypted
  - Case 1: A standard crypto API is used. Simple data flow verification
  - Case 2: The crypto is inside the application. A formal specification must be provided
- No access to records without first giving the master password
  - Concepts are not defined in the framework
  - A formal specification must be provided

SSH MIDlet

- The SSH protocol provides some guarantees
  - e.g., confidentiality of traffic under some conditions
  - e.g., confidentiality/integrity of session (public) keys
- A SSH MIDlet provides similar guarantees, provided
  - It implements SSH correctly
  - All assumptions underlying SSH security (e.g. random generation) are valid

  These provisions are security requirements: the first one requires a detailed functional specification

Not to forget

Well-known causes of attacks:

- Exposed fields (i.e., any non-private field)
- Exposed aliases
- Subclassing (i.e., attacker defining a malicious subclass)

Simple countermeasures:

- Use final fields and immutable objects
- Use private fields
- Respect behavioral subtyping
- Use static analyses to detect uncaught exceptions, aliasing, shared references, etc.

Not to forge: safety vs. security

Standard requirement: exception freeness

- The application never exits by throwing an exception
- Sometimes restricted to some exceptions

This is not a security requirement

- It is a safety requirement
- It is related to the application working well

But

- Bad exception handling is a classical origin of vulnerability
- Good exception handling is a sign of quality

Expressing properties

- Which security properties?
  - Restrictions on API usage
  - Data/information flow within the application
  - Control flow within the application
  - Detailed functional specifications
  - Exception freeness
- How to express these properties
  - Using types
  - Using logic
    - JML, using annotations at the source code level
    - BML, using annotations at the bytecode level
- Generating annotations from purpose-specific formalisms

Restrictions on API usage using BML/JML

```
public class Connection{
//@ public ghost int _nr_of_SMS ;
//@ requires url.startsWith( sms://) &
_nr_of_SMS < 10 ;
//@ ensures _nr_of_SMS = \old(_nr_of_SMS)+1 ;
public open(String url) ...
}
```

Challenges

Framework-specific rules are well-defined, but . . .

- Their translation into programming rules is not obvious
- Programmers are difficult to control
- The checks often remain complex and/or incomplete/unsound
- The API is large and costly to model, whereas API behavior often requires specific modeling