**Motivation**

ICT infrastructures are evolving into huge distributed networks of computational devices:
- **flexibility**: aimed at providing seamless access to located services,
- **heterogeneity**: devices greatly vary in connectivity, computational power, libraries, etc.
- **extensibility**: frequent changes in the computational infrastructure (remote maintenance, evolvability…)
- **interactivity**: possible to delegate some tasks to other devices (computation, storing…) or entities (programming…)

How do we achieve security and reliability?
Certificates

- are condensed and formalized mathematical proofs/hints
- are self-evident and unforgeable
- can be checked efficiently...
- independent of difficulty of certificate generation

Proof carrying code: standard framework

- the program is annotated (loop invariants, function specifications).
- the VCCGen computes a logic formula $\phi$ that is true guarantees the program security.
- the certifying prover computes a proof object $\pi$ which establishes the validity of $\phi$.
- the consumer rebuilds the formula $\phi$ and checks that $\pi$ is a valid proof of $\phi$. 

Flavors of Proof Carrying Code

- Type-based PCC
  - Widely deployed in KVM
  - Application to JVM typing
  - On-device checking possible

- Logic-based PCC
  - Original scenario
  - Application to type safety and memory safety

Certificates
Proof carrying code: standard framework

- the program is annotated (loop invariants, function specifications),
- the VCGen computes a logic formula \( \phi \) that if true guarantees the program security,
- the certifying prover computes a proof object \( \pi \) which establishes the validity of \( \phi \),
- the consumer rebuilds the formula \( \phi \) and checks that \( \pi \) is a valid proof of \( \phi \).

Proof carrying code: standard framework

- the program is annotated (loop invariants, function specifications),
- the VCGen computes a logic formula \( \phi \) that if true guarantees the program security,
- the certifying prover computes a proof object \( \pi \) which establishes the validity of \( \phi \),
- the consumer rebuilds the formula \( \phi \) and checks that \( \pi \) is a valid proof of \( \phi \).
Simpler checkers?

Proof

Implementation

Trusted Computing Base (TCB)

The TCB of a program is the set of components that must be trusted to ensure the soundness of the program. Any bug in the others components will never affect the soundness.

What is the PCC TCB?
- the proof checker
- the VCGen
- the CPU

You don’t need to trust the compiler, the annotations, the prover, the proof...

TCB still too big?

“[…] there were errors in that code that escaped the thorough testing of the infrastructure.”

G.C. Necula and R.R. Schneck, LICS’03

The weak point was the VCGen (23,000 lines of C...). The size of the TCB can be reduced
- by removing the VCGen: Foundational Proof-Carrying Code
- by certifying the VCGen in a proof assistant
- by relying on simpler checkers
Machine-checking programming language semantics

Theorem
For all programs $p$, analyse($p$) computes a sound approximation of $\delta$.

Theorem
For all programs $p$, if $\phi(p, \phi, \psi)$ and $P, \mu \subseteq \nu, \psi$ then $\phi(\mu)$ implies $\psi(\mu, \nu, \psi)$

- depends on the definition of $\delta$ (a book!)
- and on the definition of analyse or po

TCB of certified PCC

In standard PCC
- If the VCCGen is proved correct
  - the proof checker of the VCCGen soundness proof (could be the same as for the code proof)
  - the formal definition of the language semantics
  - the formal definition of the security property

- This is also a large TCB, but we think it is better to have 2,000 lines of formal definitions than 20,000 lines of C code!

Mobius project

- Certified PCC
  - Certificate checkers must be machine-checked
  - We program executable checkers in the Coq proof assistant
- Basic technologies (type systems and logics) for static enforcement of expressive policies at application level
  - information flow: public outputs should not depend on confidential data
  - resource usage: memory usage, billable actions...
  - functional correctness: proof-transforming compilation
- Certificate generation by type-preserving compilation, certifying compilation, and proof-transforming compilation

Tools
Executable checkers

- In foundational PCC, certificates represent deductive proofs
  - Typing rules as lemmas
- We produce computational proofs by programming a type system and proving it correct!
  - Scalable and shorter proof terms
- Supports proof by reflection in wholesale PCC
- Allows extraction of certified checkers for retail PCC

Reflection

- A predicate $P : T \rightarrow \text{Prop}$
- A decision procedure $f : T \rightarrow \text{bool}$
- A correctness lemma $C : \forall x : T. f x = \text{true} \rightarrow P x$

If $f a$ reduces to true, then $C a \Rightarrow f = \text{true}$ is a proof of $P a$

Using executable checkers for retail PCC

In standard PCC:
- If the VCGen is proved correct
  - the proof checker of the VCGen soundness proof (could be the same as for the code proof)
  - the formal definition of the language semantics
  - the formal definition of the security property

This is also a large TCB, but we think it is better to have 2,000 lines of formal definitions than 20,000 lines of C code!
A program is an array of instructions:

\[
\text{instr} ::= \text{prim op} \mid \text{push } v \mid \text{load } x \mid \text{store } x \mid \text{if } j \mid \text{goto } j \mid \text{return}
\]

where:
- \( j \in \mathcal{P} \) is a program point
- \( v \in \mathcal{V} \) is a value
- \( x \in \mathcal{X} \) is a variable

**Semantics**

- States are of the form \( \langle i, p, s \rangle \) where:
  - \( i : \mathcal{P} \) is the program counter
  - \( p : \mathcal{X} \rightarrow \mathcal{V} \) maps variables to values
  - \( s : \mathcal{V}^\star \) is the operand stack
- Operational semantics is given by rules of the form
  \[ P[i] = \text{ins constraints} \]
  \[ s \leadsto s' \]
- Evaluation semantics: \( P, \mu \Downarrow v, v \) iff \( \langle \mu, c \rangle \leadsto^* \langle v, v \rangle \), where \( \leadsto^* \) is the reflexive transitive closure of \( \leadsto \)

**Type-based methods**

- Certified bytecode verification
- Non-interference
- Type-preserving compilation
Types for bytecode

- We fix a set of base types, typically: int, bool...
- Types are partially ordered by a relation $\sqsubseteq$
- Least upper bound of two types $\sqcup$
- Each operation is assigned a type:
  - $\neg b : \text{bool} \rightarrow \text{bool}$
  - $\iadd : (\text{int}, \text{int}) \rightarrow \text{int}$
  - $0 : \text{int}$
  - $\cdots$  
- Program $P$ has type $\sigma$ if it always returns a value of type $\sigma$

Type system for SBC

- Exhibit for each program point an abstraction of the operand stack, a.k.a. stack type, and verify that instructions are compatible with the abstraction

Informally

$$\Gamma \vdash \iadd : (\text{int} \sqsubseteq \text{int} : s) \Rightarrow (\text{int} \sqsubseteq s)$$

$$\Gamma \vdash \pop : (\alpha \sqsubseteq s) \Rightarrow s$$

- Compatibility w.r.t. stack types is formalized by transfer rules

$$P[i] = \text{ins} \quad \Gamma \vdash \sigma \Rightarrow \tau \quad \Rightarrow \quad \Gamma \vdash \sigma \Rightarrow \tau'$$

$$P[i] = \text{ins} \quad \Gamma \vdash \sigma \Rightarrow \tau \quad \Rightarrow \quad \Gamma \vdash \sigma' \Rightarrow \tau'$$

Program $P : \tau$ is type-safe if there exists $S : P \rightarrow T^*$ s.t.

- $S_1 = \epsilon$
- for all $i,j \in P$
  - $i \rightarrow j \Rightarrow \exists x.i \vdash S_i \Rightarrow x \sqsubseteq S_j$
  - $i \rightarrow i' \Rightarrow \exists x'.i \vdash x' \Rightarrow x' \sqsubseteq \tau$

Safety

Program execution may get stuck

- Stack underflow (and possibly overflow)
  - push 5
  - iadd

- Operations receive ill-typed arguments
  - push 5
  - push true
  - iadd

But well-typed programs do not go wrong
Consequences of type safety

- Programs do not go wrong
  - If $S \vdash P : \tau$ and $(i, p, s) \models \text{type-correct w.r.t. } S$ and $\Gamma$, then:
    - $P[i] = \text{return}$ and $i \vdash S_i \Rightarrow \tau$
    - $(i, p, s) \rightarrow (i', p', s')$ and $i \vdash S_i$ and $(i', p', s') \models \text{type-correct w.r.t. } S$ and $\Gamma$

- Run-time type checking is redundant
  - A typed state is of the form $(i, p, s)$, where $s$ is a stack of pairs of values and types
  - A defensive virtual machine checks types at execution, i.e.
    $\sim_{\text{def}} \subseteq \text{state} \times (\text{state} + \{\text{TypeError}\})$
  - If $P$ is type-safe w.r.t. $S$ and $\Gamma$, then executions of $\sim_{\text{def}}$ coincide

Type inference

Goal is to exhibit $S$.
- Entry point of program is typed with the empty stack
- Propagation
  - Pick an program point $i$ annotated with $s'$
  - Compute $s''$ such that $i \vdash S \Rightarrow s''$
  - If there is no $s''$, then reject program.
  - For all successors $j$ of $i$
    - if $j$ is not yet annotated, annotated it with $s''$
    - if $j$ is annotated with $s''''$, replace $s''''$ by $s''' \sqcup s''''$
  - Upon termination
    - accept program if no type error $\top$ in the computed $S$
- Termination is ensured by
  - tracking which states remain to be analyzed,
  - by ascending chain condition
- Fixpoint computation!

Typing rules

Relative to a typing environment $\Gamma : \mathcal{X} \rightarrow \mathcal{T}$

- $P[i] = \text{op add}$
  - $i \vdash \text{int} \Rightarrow \text{int} = \text{int}$
  - $P[i] = \text{pop}$
  - $i \vdash \text{st} \Rightarrow \text{st}$
  - $P[i] = \text{if} j$
  - $i \vdash \text{bool} \Rightarrow \text{st}$
  - $P[i] = \text{load x}$
  - $i \vdash \text{st} \Rightarrow \Gamma(x) \Rightarrow \text{st}$
  - $P[i] = \text{store x}$
  - $i \vdash \Gamma(x) \Rightarrow \text{st} \Rightarrow \text{st}$

Transfer rules are deterministic

Simplification: the type system is flow insensitive
Lightweight bytecode verification

Provide types of junction points

- Entry point and junction points are typed
  - the entry point of the program is typed with the empty stack
- Propagation
  - Pick an program point \( i \) annotated with \( st \)
  - Compute \( st' \) such that \( i \vdash st \Rightarrow st' \). If there is no \( st' \), then reject program.
  - For all successors \( j \) of \( i \)
    - if \( j \) is not yet annotated, annotated it with \( st' \)
    - if \( j \) is annotated with \( st'' \), check that \( st' \leq st'' \). If not, reject program

One pass verification, sound and complete wrt bytecode verification

Verified bytecode verification

A puzzle with 8 pieces,
- Each piece interacts with its neighbors

Verified bytecode verification

Example: JVM states

Frame: \( \langle h, \langle m, pc, l, v :: s \rangle, sf \rangle \)
Verified bytecode verification

- operational semantics \(\rightarrow\) between states
- Bicolano is the basis of all formalizations in Mobius

Bicolano

- Operational model of the (sequential fragment of) JVM
- Base block for all machine checked proofs in Mobius
- Already several formal semantics of JVM (Isabelle, ACL2, Coq)
- Particularities of Bicolano
  - targets the CLDC platform (Java for mobile devices)
  - uses intensively the Coq module system
  - some components are described as abstract types to be independent from any particular implementation choice
  - efficient implementations provided (using functional maps)

Semantics of Bicolano is not executable
- More difficult to test the implementation
- Easier to do proofs
- Still compatible with executable checkers
Examples of instructions

\[ P[(m, pc)] = \text{push } c \]
\[ \langle h, (m, pc + 1, l, c \cdot s), sf \rangle \]

\[ P[(m, pc)] = \text{invokevirtual } m_{id} \]
\[ m' = \text{methodLookup}(m_{id}, h(loc)) \]
\[ V = V_1 \cdot \cdots \cdot V_k \cdot \text{arguments}(m_{id}) \]
\[ \langle h, (m, pc, l, \text{loc} \cdot \text{V} \cdot c), (m, pc, l, s) \rangle \]

Method invocations

- Small-step semantics formalisation (basis of Bicolano)
- Intermediate semantics with big-step for method calls
  - Simplify verification of modular verification methods
  - Problematic with multi-threading
    - \[ \forall h, m, pc, s, l, v, h, sf, \]
    - \[ (h < m, pc, s, l) \cdot sf \]
    - \[ (h', < m, pc', v \cdot s', l'), sf \]
    - return(pc')
    - \[ m' \vdash \langle h', pc, s' \rangle \vdash \langle h', v \rangle \]

Verified bytecode verification

- the type system is specified by transfer rules
- ... written in a functional style (hence executable)
- a type is a solution of a post fixpoint problem \( f_{\sharp}^i(s^i) \sqsubseteq s^i \)
- equivalent to constraint system

\[
\begin{align*}
\begin{cases}
  f_{\sharp}(s^1, \ldots, s^k) \sqsubseteq_2 s^i \\
  f_{\sharp}(s^1, \ldots, s^k) \sqsubseteq_\triangle s^i
\end{cases}
\end{align*}
\]

If \( s \Rightarrow s' \) and \( s \) is type-correct, then \( s' \) is type-correct

- easy proof, but tedious: one proof by instruction
- uses intermediate semantics
- exceptions may be handled separately

Method invocations
Verified bytecode verification

- semantics domains
- abstraction relations
- abstract domains
- fixpoint solver/checker

- semantic rules
- soundness proof
- analysis specification
- analysis implem.

**Final results**

\[ \vdash P \quad s_{\text{init}} \Downarrow s_{\text{final}} \Rightarrow s_{\text{final}} \text{ type} \rightarrow \text{correct} \]

- progress
- commutation defensive and offensive machine

**Abstraction-Carrying Code**

- Powerful generalization of lightweight bytecode verification
- Programs come equipped with a partial solution
- One pass verification (decidable assuming \( \subseteq \) is decidable)
- Sound and complete w.r.t. fixpoint computation
- May embed a notion of certificate

**Verified abstraction carrying code**

It is possible to generalize verified bytecode verification to verified abstraction carrying code

- Resource control
- Array-out-of-bound exceptions

- Generic lattice library
- General lemmas about well-founded orders

**Verified bytecode verification**

- semantics domains
- abstraction relations
- abstract domains
- fixpoint solver/checker

- semantic rules
- soundness proof
- analysis specification
- analysis implem.

**Implement transfer rules: from relations to functions**

**Verified bytecode verification**

- Generic tools for fixpoint computation
- Fixpoint checker

In the two cases, efficient (purely functional) data structures
Non-interference

"Low-security behavior of the program is not affected by any high-security data." Goguen & Meseguer 1982

Examples of insecure programs

Direct flow
- load y
- store x
- return

Indirect flow
- load y
- if 5
- push 0
- store x
- return

Flow via return
- load y
- if 5
- push 1
- return
- push 0
- return

Flow via operand stack
- push 0
- push 1
- load y
- if 6
- swap
- store x
- return

Non-interference

"Low-security behavior of the program is not affected by any high-security data." Goguen & Meseguer 1982

∀s₁, s₂, s₁ ↘ L₁, s₂ ↘ L₂ ⇒ s₁ ↘ L₁ → s₂ ↘ L₂

High = confidential  Low = public
A program point \( j \) is in a control dependence region of a branching point \( i \) if:
- \( j \) is reachable from \( i \),
- there is a path from \( i \) to a return point which does not contain \( j \).

CDR can be computed using post-dominators of branching points.

Example:
- \( a \) must belong to \( \text{region}(i) \)
- \( b \) does not necessarily belong to \( \text{region}(i) \)

In a typical type system for a structured language:

\[
\begin{array}{c}
\vdash \text{exp} : k \\
[k_1] \vdash c_1 \\
[k_2] \vdash c_2 \\
k \leq k_1 \\
k \leq k_2
\end{array}
\]

In our context:
- \( \text{se} \): a security environment that attaches a security level to each program point
- for each branching point \( i \), we constrain \( \text{se}(j) \) for all \( j \in \text{region}(i) \)

\[
P[i] = \begin{cases} 
\text{if } & \forall j \in \text{region}(i), k \leq \text{se}(j) \\
\text{else } & 
\end{cases}
\]

Policy

- A lattice of security levels \( S = [H, L] \) with \( L \leq H \)
- Each program is given a security signature: \( \Gamma : X \rightarrow S \) and \( k_{\text{ret}} \)
- \( \Gamma \) determines an equivalence relation \( \sim_L \) on memories: \( \rho \sim_L \rho' \) iff \( \forall x \in X \Gamma(x) \leq L \Rightarrow \rho(x) = \rho'(x) \)
- Program \( P \) is non-interfering w.r.t. signature \( \Gamma, k_{\text{ret}} \) iff for every \( \mu, \mu', \nu, \nu', \nu' \):

\[
P, \mu \Downarrow \nu, \nu' \quad P, \mu' \Downarrow \nu', \nu'
\]

\[
\mu \sim_L \mu' \Rightarrow \nu \sim_L \nu' \land (k_{\text{ret}} \leq L \Rightarrow \nu = \nu')
\]

Type system

- Transfer rules of the form

\[
P[i] = \text{ins constraints} \quad P[i] = \text{ins constraints}
\]

\[
\begin{array}{c}
i : \text{st} \Rightarrow \text{st}' \\
i : \text{st} \Rightarrow
\end{array}
\]

where \( \text{st}, \text{st}' \in S^* \).
- Types assign stack of security levels to program points

\[
S : P \rightarrow S^*
\]

- \( S \vdash P \) iff \( S_i = \varepsilon \) and for all \( i, j \in \mathcal{P} \):

\[
\begin{array}{c}
i \mapsto j \Rightarrow 3a' : i \vdash S_i \Rightarrow \text{st}' \land \text{st}' \leq S_j; \\
i \mapsto j \Rightarrow i \vdash S_i
\end{array}
\]

The transfer rules and typability relation are implicitly parametrized by a signature \( \Gamma, k_{\text{ret}} \) and additional information (next slide)
CDR soundness
SOAP (Safe Over Approximation Properties)

CDR soundness is ensured by local conditions (instead of path properties) using \( \text{region} \in \mathcal{P} \rightarrow \wp(\mathcal{P}) \) and \( \text{jun} \in \mathcal{P} \rightarrow \mathcal{P} \).

**SOAP1:** for all program points \( i \) and all successors \( j, k \) of \( i \) (\( i \rightarrow j \) and \( i \rightarrow k \)) such that \( j \neq k \) (\( i \) is hence a branching point), \( k \in \text{region}(i) \) or \( k = \text{jun}(i) \);  

**SOAP2:** for all program points \( i, j, k \), if \( j \in \text{region}(i) \) and \( j \rightarrow k \), then either \( k \in \text{region}(i) \) or \( k = \text{jun}(i) \);  

**SOAP3:** for all program points \( i, j \), if \( j \in \text{region}(i) \) and \( j \rightarrow \) then \( \text{jun}(i) \) is undefined.
State equivalence

Unwinding lemmas focus on state equivalence $\sim_L$.

**State equivalence**

\[ \langle i, p, s \rangle \sim_L \langle i', p', s' \rangle \text{ if:} \]

- Memory equivalence $p \sim_L p'$
- Operand stack equivalence $s \sim_L s'$ (defined w.r.t. $S$)

Transfer rules

**Transfer rules**

State equivalence $\sim_L$ is ensured by local conditions (instead of path properties) using $\text{region} \in P \rightarrow \wp(P)$ and $\text{jun} \in P \rightarrow P$.

**CDR soundness**

**SOAP (Safe Over Approximation Properties)**

- **SOAP1:** for all program points $i$ and all successors $j, k$ of $i$ ($i \rightarrow j$ and $i \rightarrow k$) such that $j \neq k$ ($i$ is hence a branching point), $k \in \text{region}(i)$ or $k = \text{jun}(i)$;
- **SOAP2:** for all program points $i, j, k$, if $j \in \text{region}(i)$ and $j \rightarrow k$, then either $k \in \text{region}(i)$ or $k = \text{jun}(i)$;
- **SOAP3:** for all program points $i, j$, if $j \in \text{region}(i)$ and $j \rightarrow$ then $\text{jun}(i)$ is undefined.

**Operand stack equivalence** $s \sim_L s'$ (defined w.r.t. $S$):

- High stack positions in black
- Require that both stacks coincide, except in their lowest black portion

Operand stack equivalence $s \sim_L s'$ is defined w.r.t. $S_i$ and $S_{i'}$:

- High stack positions in black
- Require that both stacks coincide, except in their lowest black portion
Adding objects, exceptions and methods

We have formally proved in Coq the soundness of information flow type system for a sequential JVM-like language, and extracted an information flow checker.

Issues:
- Methods: signatures
- Objects: heap L-equivalence, allocator
- Exceptions: extreme loss of precision due to explosion of control flow, more complex signatures

Soundness

If $S \vdash P$ (wrt. se and adr) then $P$ is non-interfering.

Direct application of
- Low (locally respects) unwinding lemma:
  If $s \sim_L s'$ and $s \leadsto t$ and $s' \leadsto t'$, then $t \sim_L t'$, provided $s \cdot pc = s' \cdot pc$
- High (step consistent) unwinding lemma:
  If $s \sim_L s'$ and $s \leadsto t$ and then $t \sim_L t'$, provided $s \cdot pc = t$ is a high program point and $S_i$ is high and $se$ is well-formed
- Gluing lemmas for combining high and low unwinding lemmas (extensive use of SOAP properties)
- Monotonicity lemmas

Architecture

Three successive phases:
- the PA (pre-analyse) analyser computes information to reduce the control flow graph.
- the CDR analyser computes control dependence regions (to deal with implicit flows)
- the IF (Information Flow) analyser computes for each program point a security environment and a stack type

Compatibility with lightweight verification

The type system:
- is compatible with lightweight bytecode verification
- code provided with
  - regions (verified by a region checker)
  - security environment
  - type information at junction points
Pre-analyses

Branching is a major source of imprecision in an information flow static analysis.

The PA (pre-analyse) analyser computes information that is used to reduce the control flow graph and to detect branches that will never be taken.

- null pointers (to predict unthrowable null pointer exceptions),
- classes (to predict target of throws instructions),
- array accesses (to predict unthrowable out-of-bounds exceptions),
- exceptions (to over-approximate the set of throwable exceptions for each method)

Such analyses (and their respective certified checkers) can be developed using certified abstract interpretation.

Information flow type system

Type annotations required on programs:

- \( \beta : \mathcal{F} \rightarrow \mathcal{S} \) attaches security levels to fields,
- \( \alpha : \mathcal{M} \times \mathcal{F} \rightarrow \mathcal{S} \) attaches security levels to contents of arrays at their creation point
- each method possesses one (or several) signature(s):
  \[
  \vec{k}_o \xrightarrow{\vec{k}_m} \vec{k}_r
  \]

- \( \vec{k}_o \) provides the security level of the method parameters (and local variables),
- \( \vec{k}_m \) effect of the method on the heap,
- \( \vec{k}_r \) is a record of security levels of the form \( [r : k_r, e_1 : k_{e_1}, \ldots, e_n : k_{e_n}] \)
  - \( k_r \) is the security level of the return value (normal termination),
  - \( k_{e_i} \) is the security level of each exception \( e_i \) that might be propagated by the method

Architecture

- Each phase corresponds to a pair analyser/checker
- Trusted Computed Base (TCB) is reduced to the checkers
  Moreover, since we prove these checkers in Coq, TCB is in fact relegated to Coq and the formal definition of non-interference.
Typing judgment

General form

\[ P[i] = \text{ins. constraints} \]

\[ \Gamma, \mathbf{fI}, \text{region, sc, sgn, f} \vdash st \Rightarrow st' \]

Selected rules

\[ P[i] = \text{invokevirtual} \quad \Gamma, \sigma \vdash \text{region, sc, sgn, f} \quad k \sqsubseteq k' \quad k \in [k_1, k_2] \quad \forall i \in \mathbb{B} \quad \text{length}(st) = i \quad st[i] \in [k'_i + 1] \quad c \in \text{excAnalysis}(\mathbb{M}) \cup \mathbb{N} \quad \forall i \in \text{region}(c) \quad k \sqsubseteq k' \quad c \vdash \text{region, sc, sgn, f} \quad \Gamma, \text{region, sc, sgn, f} \quad k \sqsubseteq k' \quad k \sqsubseteq k' \quad k_1 \sqsubseteq k_2 \quad k_2 \sqsubseteq k_2 \quad st[i] \Rightarrow st[i] \]

See the Coq development for 63 others typing rules...

Remarks on machine-checked proofs

We have used the Coq proof assistant to
- to formally define non-interference definition,
- to formally define an information type system,
- to mechanically proved that typability enforces non-interference,
- to program a type checker and prove it enforces typability,
- to extract an Ocaml implementation of this type checker.

Structure of proofs

1. Intermediate semantics simplifies the intermediate definition of indistinguishability (call stacks).
2. Second intermediate semantics: annotated semantics with result of pre-analyses
   - the pre-analyse checker ensures that both semantics correspond
3. Implementation and correctness proof of the CDR checker
4. The information flow type system (and its corresponding type checker) enforce non-interference wrt. the annotated semantics.

Example

```java
int m(boolean x, y) throws C {
    if (x) { throw new C(); }
    else if (y) { x = 3; }
    return 1;
}
```

```java
try { z = m(x, y); } catch (NPE z) { t = 1; }
```

Fine grain exceptions handling: example

```java
try { z = m(x, y); } catch (NPE z) { t = 1; }
```

With only one level for all exceptions
- [4,5,6] is a high region (depends on y): t₂ = 1 is rejected
- With our signature
  - [4,5,6] is a low region: t₁ = 1 is accepted
  - a region is now associated to a branching point and a step kind (normal step or exception step)

```
```
We have defined an information flow policy that:
- supports controlled release of information (what and where),
- can be enforced efficiently,
- has a modular proof of soundness,
- is instantiable to bytecode
- allows reuse machine-checked proofs

Concurrency

- Mobile code applications often exploit concurrency
- Concurrent execution of secure sequential programs is not necessarily secure:

\[
\text{if}(h > 0)\{\text{skip; skip; skip}\} \quad \text{||} \quad \text{skip; skip; } l := 2
\]

- Security of multi-threaded programs can be achieved:
  - by imposing strong security conditions on programs
  - termination-sensitive, bisimulation-based
  - by relying on secure schedulers
- Modular type system and soundness proof for concurrent stack-based language via secure schedulers

Remarks on machine-checked proofs

We have used the Coq proof assistant to
- to formally define non-interference definition,
- to formally define an information type system,
- to mechanically proved that typability enforces non-interference,
- to program a type checker and prove it enforces typability,
- to extract an Ocaml implementation of this type checker.

About 20,000 lines of definitions and proofs. You only need to trust
- the JVM semantics (2000 lines)
- the definition of the security policy
- the statement of the theorem (but not the definition of the type system)

Declassification

- Baseline policies (i.e. non-interference) are too restrictive in practice. Declassification policies allow intentional information release.
- Main dimensions: what, where, who
Compiler correctness

The compiler is semantics-preserving (terminating runs, input/output behavior)

\[
P, \mu \Downarrow v, v \implies \{P\}, \mu \Downarrow v, v
\]

Thus source programs satisfy an input/output property if their compilation does

\[
\forall P, \phi, \psi, \mu, v, v
\]

\[
(\phi(\mu) \implies P, \mu \Downarrow v, v \implies \psi(\mu, v, v))
\]

But are typable programs compiled into typable programs?

\[
\forall P, \vdash P \implies \exists S, S \vdash \{P\}
\]

Yes for JVM typing and non-optimizing compiler, no in general

Secure schedulers

A secure scheduler selects the thread to be executed in function of the security environment:

- the thread pool is partitioned into low, high, and hidden threads
- if a thread is currently executing a high branch, then only high threads are scheduled
- if the program counter of the last executed thread becomes high (resp. low), then the thread becomes hidden or high (resp. low)
- the choice of a low thread only depends on low history

Round-robin schedulers are secure, provided they take over control when threads become high/low/hidden

Source language: While

A program is a command:

\[
\text{commands } \equiv \ x := e \quad \text{assignment} \\
\text{if } (c)[c] \quad \text{conditional} \\
\text{while } (c)[c] \quad \text{loop} \\
\text{c;c} \quad \text{sequence} \\
\text{skip} \quad \text{skip} \\
\text{return } e \quad \text{return value}
\]

Semantics is standard:

- States are pairs \( \langle c, \rho \rangle \)
- Small-step semantics \( \langle c, \rho \rangle \rightarrow \langle c', \rho' \rangle \) or \( \langle c, \rho \rangle \rightarrow* \langle v, \nu \rangle \)
- Evaluation semantics \( c, \mu \Downarrow \langle v, \nu \rangle \) if \( c, \mu \rightarrow* \langle v, \nu \rangle \)

Type-preserving compilation

- Source type systems offer tools for developing safe/secure applications, but does not directly address mobile code
- Bytecode verifiers provides safety/security assurance to users
- Relating both type systems ensure:
  - applications can be deployed in a mobile code architecture that delivers the promises of the source type system
  - enhanced safety/security architecture can benefit from tools for developing applications that meet the policy it enforces

Theoretical interest

Type-preserving compilation allows to derive soundness of source type system from soundness of target type system
Compiling security environment

```
load y \_L
if 6 \_L
push 1 H ∈ region(2)
store x H ∈ region(2)
goto 8 H ∈ region(2)
push 2 H ∈ region(2)
store x H ∈ region(2)
push 3 \_L jun(2)
store x’ \_L
push 2 \_L
return \_L
```

Information flow type system

- Security policy $\Gamma : X \rightarrow S$ and $k_{ret}$
- Volpano-Smith security type system

\[
\begin{align*}
\frac{e : k \quad k \sqsubseteq \Gamma(x)}{\exists c \quad k \sqsubseteq \Gamma(x)}
\end{align*}
\]

plus subtyping rules

\[
\begin{align*}
\frac{e : k \quad k \sqsubseteq k'}{e : k'}
\end{align*}
\]

Preservation of information flow types

If $P$ is typable, then the extended compiler generates security environment, regions, and stack types at junction points, such that:
- regions satisfy SOAP and can be checked by region checker
- $[P]$ can be verified by lightweight checker

The result also applies to:
- information flow type system for sequential Java (with exceptions)
- concurrency (using naive rule for parallel composition)
- declassification

Compiling statements

\[
\begin{align*}
[x] &= \text{load } x \\
[v] &= \text{push } v \\
[e_1 \text{ op } e_2] &= [e_1], [e_2]; \text{ binop } op \\
k : \{x := e\} &= [e]; \text{ store } x \\
k : \{k_1; k_2[\_i]\} &= [k_1]; [k_2[\_i]\_i] \\
&\quad \text{where } k_2 = k + [\_i]\_i \\
k : \{\text{return } e\} &= [e]; \text{ return} \\
k : \{\text{if}(e_1 \text{ cmp } e_2[\_i]; [\_i])\} &= [e_2], [e_1]; \text{ if } \text{ cmp } k_2; k_2[\_i]; \text{ goto } l; k_2[\_i] \\
&\quad \text{where } k_1 = k + [e_2][\_i] + [e_1][\_i] + 1 \\
&\quad k_2 = k_1 + [\_i]\_i + 1 \\
k : \{\text{while}(e_1 \text{ cmp } e_2[\_i])\} &= [e_2], [e_1]; \text{ if } \text{ cmp } k_2; k_2[\_i]; \text{ goto } k \\
&\quad \text{where } k_1 = k + [e_2][\_i] + [e_1][\_i] + 1 \\
&\quad k_2 = k_1 + [\_i]\_i + 1
\end{align*}
\]
Certifying compilation vs certificate translation

Source Code Verification

Certificate Translation

Traditional PCC

Compiling evidence from program verification

The missing link

- Program verification environments are being used successfully for proving properties of source programs
- Verification of source code does not address mobile code issues
- Our goal is to generate certificates from source code verification
Overview of results

- Non-optimizing compilation
- Program optimizations and certifying analyzers
- Certificate translation in abstract interpretation

Program Specification

- Assertions: formulae attached to a program point, characterizing the set of execution states at that point.
- Instructions are possibly annotated:

  ```
  [pre] ins_q {φ_q} 
  [post]
  ```

  Possibly annotated instructions
  ```
  ins ::= ins | (φ, ins)
  ```

- A partially annotated program is a triple \((P, Φ, Ψ)\) s.t.
  - \(Φ\) is a precondition and \(Ψ\) is a postcondition
  - \(P\) is a sequence of possibly annotated instructions

Certifying compilation vs certificate translation

<table>
<thead>
<tr>
<th>Conventional PCC</th>
<th>Certificate Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic</td>
<td>Specification</td>
</tr>
<tr>
<td>(Invariant inference)</td>
<td>Verification</td>
</tr>
<tr>
<td>Safety, inf. flow, resources (approximation)</td>
<td>Properties</td>
</tr>
</tbody>
</table>

Certified compilation aims at producing a proof term \(H\) such that

\[
H : ∀P \, μ \, ν \implies [P], μ \Downarrow ν
\]

Thus, we can build a proof term \(H' : \{φ\}[P\{ψ\}]\) from \(H\) and \(H_0 : \{φ\}[P\{ψ\}]\)

Certify Compiler

Certificates

Certificate translation vs certified compilation

1. Conventional PCC
2. Certificate Translation
3. Specification
5. Verification
6. Interactive (source code)
7. Properties
8. Security policies, Functional properties (precise)

Assertions: formulae attached to a program point, characterizing the set of execution states at that point.

Instructions are possibly annotated:

\[
ins ::= ins | (φ, ins)
\]

A partially annotated program is a triple \((P, Φ, Ψ)\) s.t.
- \(Φ\) is a precondition and \(Ψ\) is a postcondition
- \(P\) is a sequence of possibly annotated instructions

Thus, we can build a proof term \(H' : \{φ\}[P\{ψ\}]\) from \(H\) and \(H_0 : \{φ\}[P\{ψ\}]\)
Building a certificate

Certification of annotated programs is performed in three steps:

- A verification condition generator fully annotates the program, and extracts a set of verification conditions (a.k.a. proof obligations).
- Verification conditions are discharged interactively.
- A certificate is built from proofs of verification conditions.

Weakest precondition calculus

Computes an assertion for a given program node only if the corresponding assertion has been already computed for all successor nodes.

Requires programs to be sufficiently annotated: all infinite paths must go through an annotated program point.

Optimizing Compilers

Verification conditions

Proof obligations $PO(P, \phi, \psi)$

- Precondition implies the weakest precondition of entry point:
  \[ \phi \Rightarrow \text{wp}(\text{entry point}) \]
- For all annotated program points \([P][k] = (\phi, i)\), the annotation \(\phi\) implies the weakest precondition of the instruction at \(k\):
  \[ \phi \Rightarrow \text{wp}(i, k) \]

An annotated program is correct if its verification conditions are valid.

Soundness

If \([P, \phi, \psi] \) is correct, and \(P, \mu \models \nu, \) and \(\mu \models \phi\), then

- \(\mu, \nu \models \psi|_{\text{wp}}\)
- all intermediate assertions are verified.
For optimizations that are justified by arithmetic reasoning:
- Specifying and certifying automatically the result of the analysis
- Merging annotations (i.e. strengthen invariants)
- Merging certificates

Deployment of secure mobile code can benefit from:
- advanced verification mechanisms at bytecode level
- methods to “compile” evidence from producer to consumer
- machine checked proofs of certificate checkers

Many other topics are pursued in Mobius
- Type systems for alias and resources
- Certifying compilation
- Tools (verification tools, program analysers, on-device checkers)

For more information, see http://mobius.inria.fr

Proofs obligations might not be preserved
- annotations might need to be modified (e.g. constant propagation)
- certificates for analyzers might be needed (certifying analyzer)
- analyses might need to be modified (e.g. dead variable elimination)