Certificate translation

Gilles Barthe

INRIA Sophia-Antipolis, France
http://www-sop.inria.fr/everest

September 7th, 2005

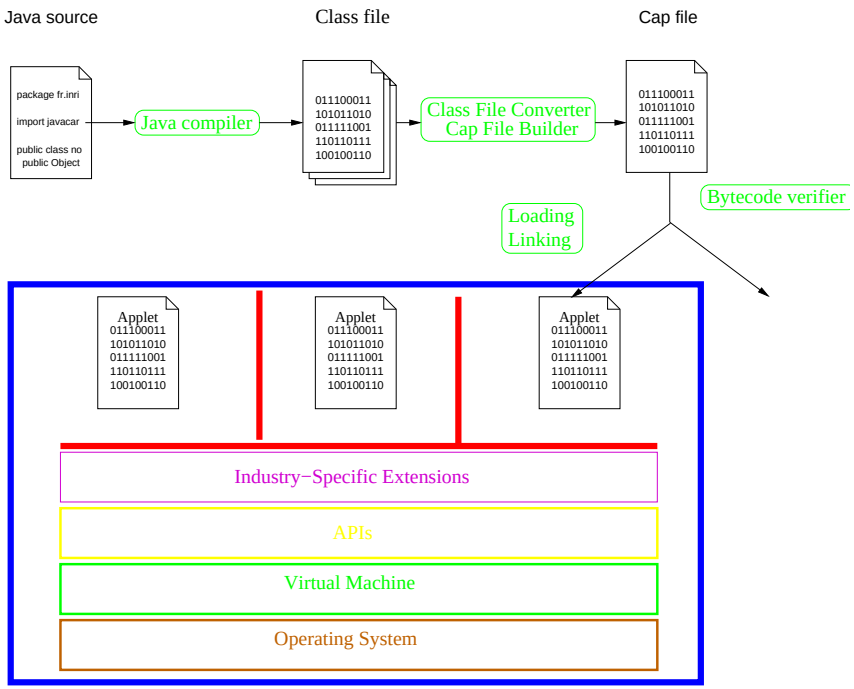
- EVEREST
- MOBIUS
- Certificate translation

EVEREST

- **Objective:**
  - design
  - implementation
  - applications

  of formal techniques for safety and security of software
- **Background:**
  - logic, type theory and proof assistants
  - programming language semantics and compilation
  - program analysis and language-based security
- **Application domains:**
  - trusted personal devices (smart cards, mobile phones)
  - ubiquitous computing

Example of Trusted Personal Device: Java SmartCard



- Widely deployed for banking and telecom applications; many upcoming applications such as e-ID, Pay-TV...
- Representative of tensions between flexibility and security
- Use of formal techniques supported by Common Criteria
- Ideal vector to experiment and transfer our technologies
- Security issues at the level of platforms (OS+VM) and applications

Some recent works

- Platform verification
  - Formal modeling of JavaCard VM, bytecode verifier and GlobalPlatform functional specifications and security requirements
  - Tool support for verified bytecode verifiers
  - Enhanced bytecode verification for secure information flow
- Verification environment for Java(Card) applications
  - Operates on annotated source code and bytecode, multi-prover
  - Annotation generation for high-level properties
  - Application to smartcard applets and OS components

Work performed in RNTL project CASTLES, FP6 IST project Inspired, and ACI Sécurité GECCOO and SPOPS

Zoom on automated security audit

- Security expert inspects code and checks that application obeys a given set of security rules
- Complex task that requires global analysis of program
- We have developed, implemented, and tested a method to verify mechanically security rules through synthesis and propagations of JML annotations from rules:

$$\frac{\begin{array}{c}\text{no run-time exception at top-level}\\ \text{no authentication within a transaction}\\ \text{no more memory consumption than X}\end{array}}{}$$

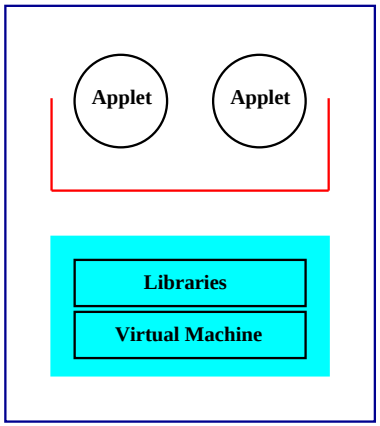
- Conclusion:
  - many applets do not obey the rules even after extensive inspection and testing!
  - our method ensures increased reliability with a limited overhead

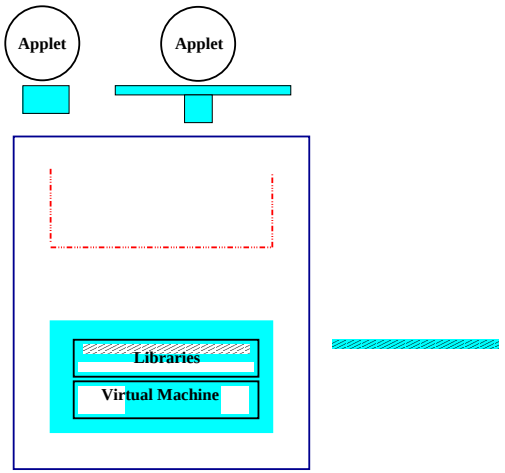
MOBIUS: Mobility, Ubiquity, Security

- Part of FET Global Computing II
- Integrated Project, Sept'05-Aug'09
- 16 partners (4 industrials) and EUP (12 partners initially). INRIA coordinator.
- Project objective: design a security architecture for next generation global computers, using Proof Carrying Code technology

Global computers

- Distributed computational infrastructure aiming at providing a global and uniform access to services.
- Large networks of *heterogeneous* devices hosting extensible computational infrastructures which can be updated remotely





Security issues

- Devices must be protected individually by means of static enforcement mechanisms
- No sharp separation between Trusted Computing Base and applications
- Trust infrastructures must allow verifiable evidence
- Need for expressive security policies and functional verification

Possible approaches

Three levels of ambition:

- 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

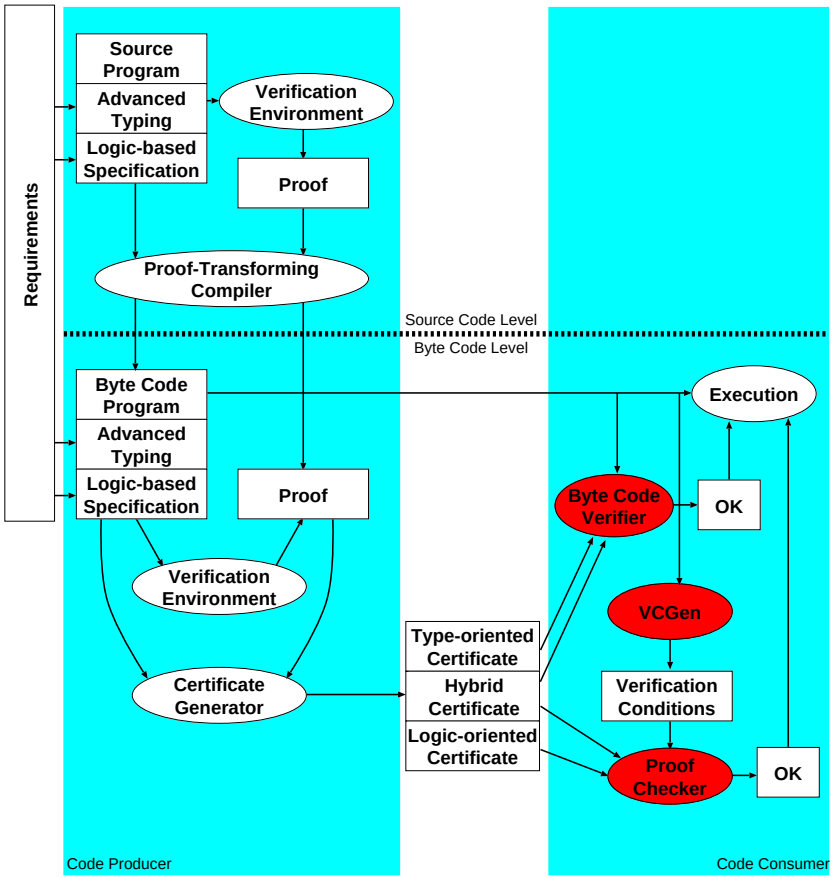
Useful to integrate different approaches

Proof Carrying Code

Programs are equipped with certificates, i.e. mathematical proofs that they obey their specification, which are verified automatically at the consumer side by a proof checker:

- No need to trust the code producer nor the compiler
- Transparent to the code consumer (no run-time penalty, no proof-search)
- Versatile (covers a wide range of safety policies)

Overall architecture



Certificate generation

PCC does not impose any mechanism to generate certificates. Yet the prime means of generating certificates is certifying compilation. We are interested in building certificates using program verification, which:

- provides a means to enforce a wide range of policies, including security properties and functional verification
- is supported by verification environments based on interactive proof assistants and automated theorem provers

Issue

- In the context of mobile and embedded code, correctness guarantees must be given for compiled programs
- There is currently no mechanism for bringing the benefits of source code verification to code consumers
- The objective of our work is to build a mechanism that enables to exploit the results of source code verification for checking compiled programs

Certificate Translation

Definition (Certificate Translation)

Mechanism that allows transferring evidence from source programs to compiled programs (i.e. translating certificate of source programs into certificates of compiled programs)

Remarks:

- Certificate translation is not certified compilation, nor certifying compilation.
- Certificate translation is relevant for interactive and automatic verification.

Formal definition

Certificate translators are given by two functions:

- a function f that maps for every program, proof obligations of the compiled program to proof obligations of the original program
- a function g that maps, for each proof obligation $\mathcal{E}$ of the compiled program, proofs of $f(\mathcal{E})$ to proofs of $\mathcal{E}$

Preservation of Proof Obligations

Source and compiled programs have syntactically equal proof obligations, so that the translation of certificates is the identity.

Languages, program logics, and certificates

We consider a simple imperative language and an intermediate RTL language with verification condition generators.

- Procedures annotated with their preconditions and postconditions.
- RTL instructions may be annotated with their preconditions (e.g. loop invariants).
- There is a well-formedness assumption on RTL programs, which establishes that a certain order on program points is well-founded.
- Certificates are left abstract, using the notion of proof algebra

RTL syntax

$$
\begin{array}{lrcl}
 & \star & ::= & < \mid \leq \mid = \mid \geq \mid > \\
\text{comparisons} & \mathsf{cmp} & ::= & r \star r \mid r \star n \\
\text{operators} & \mathsf{op} & ::= & n \mid r \mid \mathsf{cmp} \mid -r \mid r + r \mid n + r \\
 & & \mid & r - r \mid n - r \mid r * r \mid n * r \mid \neg r \\
 & & \mid & r \&\& r \mid r \| r \\
\text{instr. desc.} & \mathsf{id} & ::= & r_d := \mathsf{op},\ L \mid r_d := f(\vec{r}),\ L \\
 & & \mid & \mathsf{cmp}\ ?\ L_t : L_f \mid \mathsf{return}\ r \mid \mathsf{nop},\ L \\
\text{instructions} & I & ::= & (\phi,\ \mathsf{id}) \\
 & & \mid & \mathsf{id} \\
\text{graph} & G & ::= & L \mapsto I \\
 & P & ::= & L \mapsto \Lambda \\
\text{fun. decl} & F & ::= & \{\vec{r};\ \varphi;\ G;\ \psi;\ \Lambda;\ P\} \\
\text{program} & p & ::= & f \mapsto F
\end{array}
$$

VCGen

$$
\begin{aligned}
\mathrm{vcg}_f(L) &= \phi && \text{if } G_f(L) = (\phi,\ \mathrm{id}) \\
\mathrm{vcg}_f(L) &= \mathrm{vcg}_f^{\mathrm{id}}(\mathrm{id}) && \text{if } G_f(L) = \mathrm{id} \\
\mathrm{vcg}_f^{\mathrm{id}}(r_d := \mathrm{op},\ L) &= \mathrm{vcg}_f(L)\{r_d \leftarrow \langle \mathrm{op} \rangle\} \\
\mathrm{vcg}_f^{\mathrm{id}}(r_d := g(\vec{r}),\ L) &= \mathrm{pre}(g)\{\vec{r}_g \leftarrow \vec{r}\} \\
&\quad \wedge (\forall res.\ \mathrm{post}(g)\{\vec{r}^*_g \leftarrow \vec{r}\} \Rightarrow \mathrm{vcg}_f(L)\{r_d \leftarrow res\}) \\
\mathrm{vcg}_f^{\mathrm{id}}(\mathrm{cmp}\ ?\ L_t : L_f) &= (\langle \mathrm{cmp} \rangle \Rightarrow \mathrm{vcg}_f(L_t)) \\
&\quad \wedge (\neg \langle \mathrm{cmp} \rangle \Rightarrow \mathrm{vcg}_f(L_f)) \\
\mathrm{vcg}_f^{\mathrm{id}}(\mathrm{return}\ r) &= \mathrm{post}(f)\{\mathrm{res} \leftarrow r\} \\
\mathrm{vcg}_f^{\mathrm{id}}(\mathrm{nop},\ L) &= \mathrm{vcg}_f(f[L])
\end{aligned}
$$

Proof algebra

$$
\begin{array}{lcl}
\text{intro}_{\top} & : & \mathcal{P}(\Gamma \vdash \top) \\
\text{axiom} & : & \mathcal{P}(\Gamma; A; \Delta \vdash A) \\
\text{ring} & : & \mathcal{P}(\Gamma \vdash n_1 = n_2) \qquad \text{if } n_1 = n_2 \text{ is a ring equality} \\
\text{intro}_{\wedge} & : & \mathcal{P}(\Gamma \vdash A) \to \mathcal{P}(\Gamma \vdash B) \to \mathcal{P}(\Gamma \vdash A \wedge B) \\
\text{elim}^{\text{l}}_{\wedge} & : & \mathcal{P}(\Gamma \vdash A \wedge B) \to \mathcal{P}(\Gamma \vdash A) \\
\text{elim}^{\text{r}}_{\wedge} & : & \mathcal{P}(\Gamma \vdash A \wedge B) \to \mathcal{P}(\Gamma \vdash B) \\
\text{intro}_{\Rightarrow} & : & \mathcal{P}(\Gamma; A \vdash B) \to \mathcal{P}(\Gamma \vdash A \Rightarrow B) \\
\text{elim}_{\Rightarrow} & : & \mathcal{P}(\Gamma \vdash A \Rightarrow B) \to \mathcal{P}(\Gamma \vdash A) \to \mathcal{P}(\Gamma \vdash B) \\
\text{elim}_{=} & : & \mathcal{P}(\Gamma \vdash e_1 = e_2) \to \\
 & : & \quad \mathcal{P}(\Gamma \vdash A\{r \leftarrow e_1\}) \to \mathcal{P}(\Gamma \vdash A\{r \leftarrow e_2\}) \\
\text{subst} & : & \mathcal{P}(\Gamma \vdash A) \to \mathcal{P}(\Gamma\{r \leftarrow e\} \vdash A\{r \leftarrow e\}) \\
\text{weak} & : & \mathcal{P}(\Gamma \vdash A) \to \mathcal{P}(\Gamma; \Delta \vdash A) \\
\text{elim}_{\forall} & : & \mathcal{P}(\Gamma \vdash \forall r.A) \to \mathcal{P}(\Gamma \vdash A\{r \leftarrow e\}) \\
\text{intro}_{\forall} & : & \mathcal{P}(\Gamma \vdash A) \to \mathcal{P}(\Gamma \vdash \forall r.A) \qquad \text{if } r \text{ is not in } \Gamma
\end{array}
$$

Certified program

- A function f with declaration $\{\vec{r};\ \varphi;\ G;\ \psi;\ \Lambda;\ P\}$ is certified if:
  - Λ is a proof of $\vdash \varphi \Rightarrow \mathrm{vcg}_f(L_{\mathrm{sp}})\{\vec{r}^* \leftarrow \vec{r}\}$
  - $P(L)$ is a proof of $\vdash \phi \Rightarrow \mathrm{vcg}_f^{\mathtt{id}}(\mathrm{id})$ for all reachable labels L in f such that $f[L] = (\phi,\ \mathrm{id})$.
- A program is certified if all its functions are.

Compiler

We consider an optimizing compiler from a simple imperative language to an intermediate RTL language. The compiler proceeds in two phases:

- programs are translated into RTL in a standard fashion;
- common program optimizations are performed.

For each step, we built an appropriate certifcate translator and combine them to obtain the overall certificate translator.

From high-level to RTL programs

- The compiler is non-optimizing, and we have shown PPO.
- Remark: the result "almost" extends to Java:
  - Java compilers do not perform aggressive optimizations
  - PPO "almost" holds for Java
  - Minor difficulties: treatment of booleans, renamings, and of course definition of verification condition generator on bytecode (e.g. we must model the stack)
  - Prototype implementation in Jack

Optimizations

We have built certificate translators for

- Constant propagation
- Common subexpression elimination
- Dead register elimination (by ghost variable introduction)
- Inlining and unreachable code elimination
- etc.

For some optimizations, we must use certifying analyzers and weave certificates

Certifying Analyzer

A certifying analyzer for $\mathcal{A}$ is a function that outputs for each function f a certified function:

$$f_{\mathcal{A}} = \{\top; G_{\mathcal{A}}; \top; \Lambda_{\mathcal{A}}; P_{\mathcal{A}}\}$$

where $G_{\mathcal{A}}$ is a version of G_f annotated with results of $\mathcal{A}$:

$$G_{\mathcal{A}}(L) = (\mathrm{EQ}_{\mathcal{A}}(L),\ id)$$

where id is the instruction descriptor of $G_f(L)$.

Used for cp and cse (implementation in caml).

Optimized Graph Code

Definition

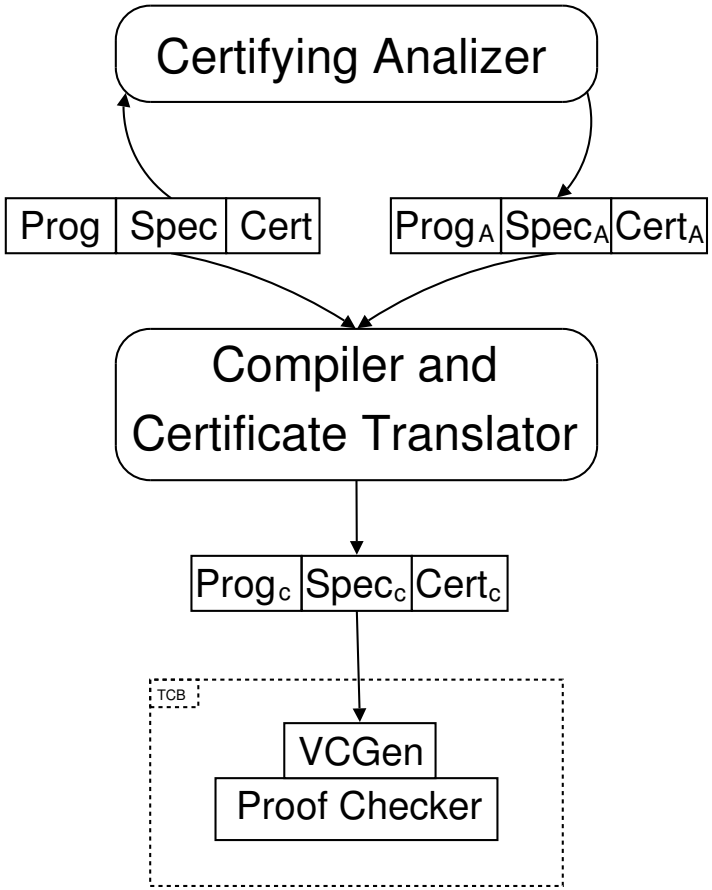
The optimized graph code of a function f is defined as follows:

$$G_{\bar{f}}(L) = \begin{cases} (\phi \wedge \mathrm{EQ}_{\mathcal{A}}(L),\ [\![\mathsf{id}]\!]) & \text{if } G_f(l) = (\phi,\ \mathsf{id}) \\ [\![\mathsf{id}]\!] & \text{if } G_f(l) = \mathsf{id} \end{cases}$$

where $[\![\mathsf{id}]\!]$ is the optimized version of instruction id.

The certificate for the optimized graph code is built by well-founded induction.

Overall Picture



Summary

Conclusion

We have given firm evidence that certificate translation provides a mean to bring the benefits of source code verification to the code consumers.

- Further work
  - richer language (objects, concurrency, aspects), more program optimizations, more advanced compilation (domain-specific languages)
  - size of certificates
  - certifying analyzers
  - implementation and experimentation