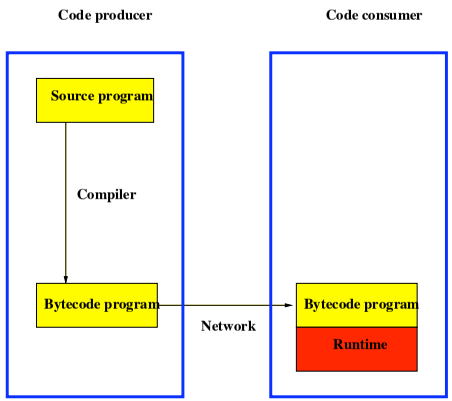
Security challenge

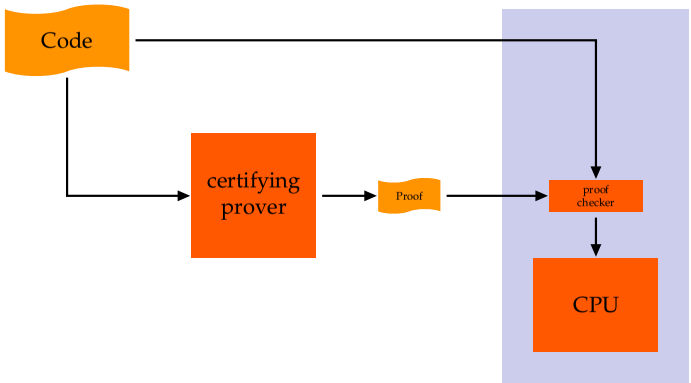


Evidence-based security

Gilles Barthe

IMDEA Software, Madrid, Spain

Proof carrying code: principles



Motivation

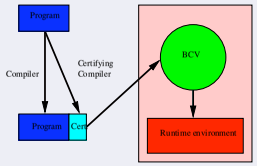
ICT infrastructures are evolving into huge distributed networks of computational devices:

- *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*?

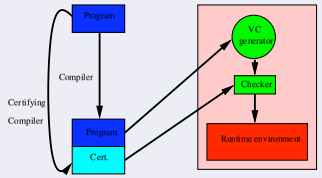
Flavors of Proof Carrying Code

Type-based PCC



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

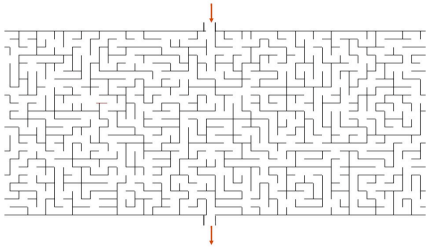
Logic-based PCC



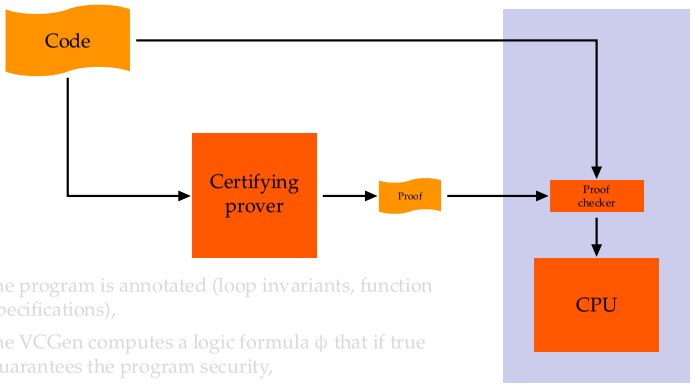
- Original scenario
- Application to type safety and memory safety

Certificates

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



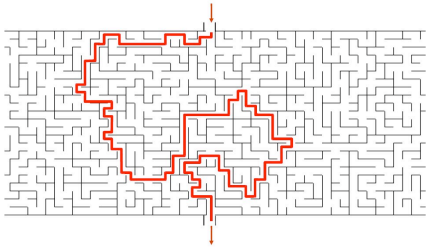
Proof carrying code: standard framework



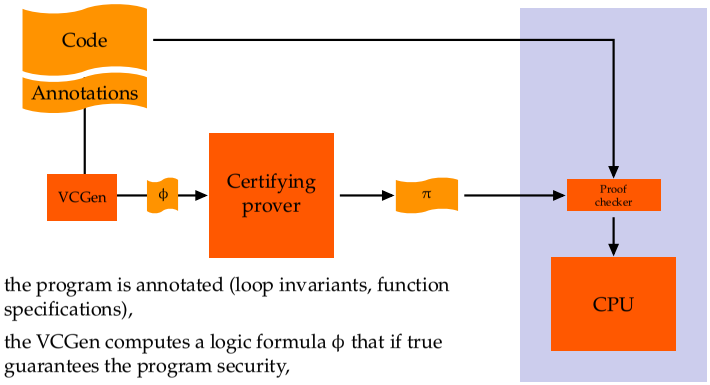
- the program is annotated (loop invariants, function specifications),
- the VCGen computes a logic formula ϕ that if true guarantees the program security,
- the certifying prover computes a *proof object* π which establishes the validity of ϕ,
- the consumer rebuilds the formula ϕ and checks that π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

Certificates

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

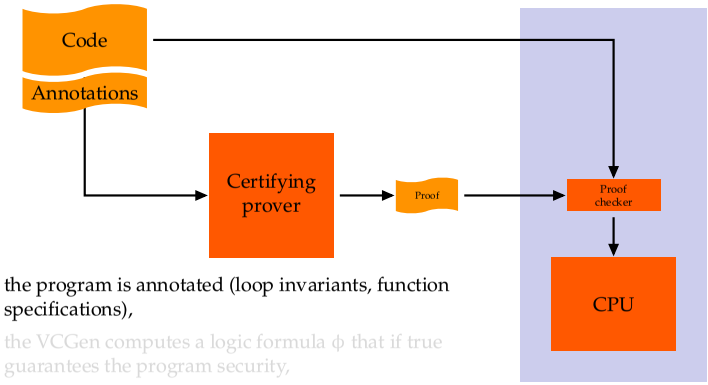


Proof carrying code: standard framework



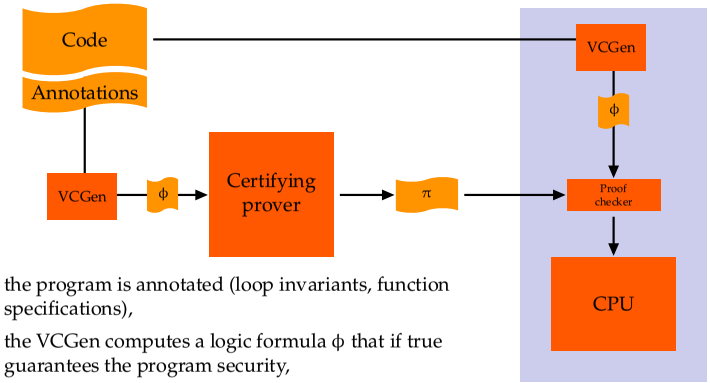
- the program is annotated (loop invariants, function specifications),
- the VCGen computes a logic formula ϕ that if true guarantees the program security,
- the certifying prover computes a *proof object* π which establishes the validity of ϕ,
- the consumer rebuilds the formula ϕ and checks that π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

Proof carrying code: standard framework



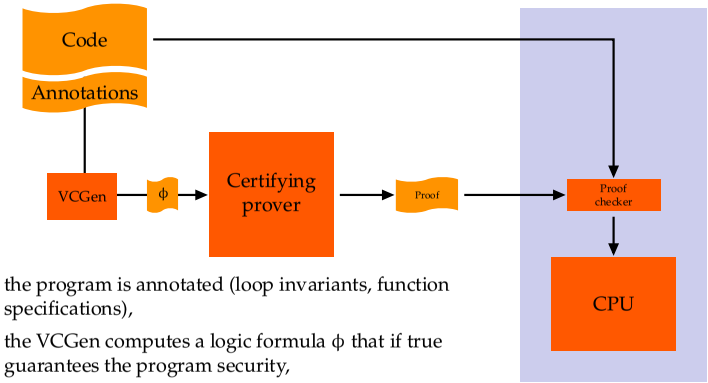
- the program is annotated (loop invariants, function specifications),
- the VCGen computes a logic formula ϕ that if true guarantees the program security,
- the certifying prover computes a *proof object* π which establishes the validity of ϕ,
- the consumer rebuilds the formula ϕ and checks that π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

Proof carrying code: standard framework



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

Proof carrying code: standard framework



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

Simpler checkers?

Proof

$\hat{a}[P](\mathrm{Post}[\mathbf{if}\ B\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f\ \mathbf{fi}])$

$= \quad$ {def. (110) of $\hat{a}[P]$}

$\hat{a}[P] \circ \mathrm{Post}[\mathbf{if}\ B\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f\ \mathbf{fi}] \circ \hat{\gamma}[P]$

$= \quad$ {def. (103) of Post}

$\hat{a}[P] \circ \mathrm{post}[\tau^*[\mathbf{if}\ B\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f\ \mathbf{fi}]] \circ \hat{\gamma}[P]$

$= \quad$ {big step operational semantics (93)}

$\hat{a}[P] \circ \mathrm{post}[(1_{\Sigma[P]} \cup \tau^B) \circ \tau^*[S_t] \circ (1_{\Sigma[P]} \cup \tau') \cup (1_{\Sigma[P]} \cup \tau^B) \circ \tau^*[S_f] \circ (1_{\Sigma[P]} \cup \tau')] \circ \hat{\gamma}[P]$

$= \quad$ {Galois connection (98) so that post preserves joins}

$\hat{a}[P] \circ (\mathrm{post}[(1_{\Sigma[P]} \cup \tau^B) \circ \tau^*[S_t]] \circ (1_{\Sigma[P]} \cup \tau')] \cup \mathrm{post}[(1_{\Sigma[P]} \cup \tau^B) \circ \tau^*[S_f] \circ (1_{\Sigma[P]} \cup \tau')]) \circ \hat{\gamma}[P]$

$= \quad$ {Galois connection (106) so that $\hat{a}[P]$ preserves joins}

$(\hat{a}[P] \circ \mathrm{post}[(1_{\Sigma[P]} \cup \tau^B) \circ \tau^*[S_t]] \circ (1_{\Sigma[P]} \cup \tau')] \circ \hat{\gamma}[P]) \sqcup (\hat{a}[P] \circ \mathrm{post}[(1_{\Sigma[P]} \cup \tau^B) \circ \tau^*[S_f] \circ (1_{\Sigma[P]} \cup \tau')] \circ \hat{\gamma}[P])$

$\Leftarrow \quad$ {lemma (5.3) and similar one for the **else** branch}

$\lambda J \cdot \mathrm{let}\ J' = \lambda l \in \mathrm{in}_P[P] \cdot \mathrm{if}\ l = \mathrm{at}_P[S_t]\ ?\ J_{\mathrm{at}_P[S_t]} \sqcup \mathrm{Abexp}[B](J_t) \downharpoonleft J_t)\ \mathrm{in}$ (120)

$\quad \mathrm{let}\ J'' = \mathrm{APost}[S_t](J')\ \mathrm{in}$

$\quad\quad \lambda l \in \mathrm{in}_P[P] \cdot \mathrm{if}\ l = \ell'\ ?\ J''_\ell \sqcup J''_{\mathrm{after}_P[S_t]} \downharpoonleft J'')$

$\quad \sqcup$

$\quad \mathrm{let}\ J'' = \lambda l \in \mathrm{in}_P[P] \cdot \mathrm{if}\ l = \mathrm{at}_P[S_f]\ ?\ J_{\mathrm{at}_P[S_f]} \sqcup \mathrm{Abexp}[T(\neg B)](J_f) \downharpoonleft J_f)\ \mathrm{in}$

$\quad\quad \mathrm{let}\ J'' = \mathrm{APost}[S_f](J'')\ \mathrm{in}$

$\quad\quad\quad \lambda l \in \mathrm{in}_P[P] \cdot \mathrm{if}\ l = \ell'\ ?\ J''_\ell \sqcup J''_{\mathrm{after}_P[S_f]} \downharpoonleft J'')$

$= \quad$ {by grouping similar terms}

$\lambda J \cdot \mathrm{let}\ J' = \lambda l \in \mathrm{in}_P[P] \cdot \mathrm{if}\ l = \mathrm{at}_P[S_t]\ ?\ J_{\mathrm{at}_P[S_t]} \sqcup \mathrm{Abexp}[B](J_t) \downharpoonleft J_t)$

$\quad \mathrm{and}\ J'' = \lambda l \in \mathrm{in}_P[P] \cdot \mathrm{if}\ l = \mathrm{at}_P[S_f]\ ?\ J_{\mathrm{at}_P[S_f]} \sqcup \mathrm{Abexp}[T(\neg B)](J_f) \downharpoonleft J_f)\ \mathrm{in}$

$\quad\quad \mathrm{let}\ J'' = \mathrm{APost}[S_t](J')$

$\quad\quad \mathrm{and}\ J'' = \mathrm{APost}[S_f](J'')\ \mathrm{in}$

$\quad\quad\quad \lambda l \in \mathrm{in}_P[P] \cdot \mathrm{if}\ l = \ell'\ ?\ J''_\ell \sqcup J''_{\mathrm{after}_P[S_t]} \sqcup J''_\ell \sqcup J''_{\mathrm{after}_P[S_f]} \downharpoonleft J''' \sqcup J'')$

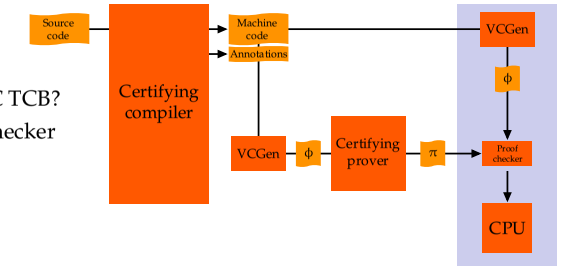
$= \quad$ {by locality (113) and labelling scheme (59) so that in particular $J''_\ell = J'''_\ell = J''_\ell = J''_\ell$ $= J''_\ell = J''_\ell$ and APost$[S_t]$ and APost$[S_f]$ do not interfere}

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...

Simpler checkers?

Proof

$\hat{a}[P](\mathrm{Post}[\mathbf{if}\ B\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f\ \mathbf{fi}])$

$= \quad$ {def. (110) of $\hat{a}[P]$}

$\hat{a}[P] \circ \mathrm{Post}[\mathbf{if}\ B\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f\ \mathbf{fi}] \circ \hat{\gamma}[P]$

$= \quad$ {def. (103) of Post}

$\hat{a}[P] \circ \mathrm{post}[\tau^*[\mathbf{if}\ B\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f\ \mathbf{fi}]] \circ \hat{\gamma}[P]$

$= \quad$ {big step operational semantics (93)}

$\hat{a}[P] \circ \mathrm{post}[(1_{\Sigma[P]} \cup \tau^B) \circ \tau^*[S_t] \circ (1_{\Sigma[P]} \cup \tau') \cup (1_{\Sigma[P]} \cup \tau^B) \circ \tau^*[S_f] \circ (1_{\Sigma[P]} \cup \tau')] \circ \hat{\gamma}[P]$

$= \quad$ {Galois connection (98) so that post preserves joins}

$\hat{a}[P] \circ (\mathrm{post}[(1_{\Sigma[P]} \cup \tau^B) \circ \tau^*[S_t]] \circ (1_{\Sigma[P]} \cup \tau')] \cup \mathrm{post}[(1_{\Sigma[P]} \cup \tau^B) \circ \tau^*[S_f] \circ (1_{\Sigma[P]} \cup \tau')]) \circ \hat{\gamma}[P]$

$= \quad$ {Galois connection (106) so that $\hat{a}[P]$ preserves joins}

$(\hat{a}[P] \circ \mathrm{post}[(1_{\Sigma[P]} \cup \tau^B) \circ \tau^*[S_t]] \circ (1_{\Sigma[P]} \cup \tau')] \circ \hat{\gamma}[P]) \sqcup (\hat{a}[P] \circ \mathrm{post}[(1_{\Sigma[P]} \cup \tau^B) \circ \tau^*[S_f] \circ (1_{\Sigma[P]} \cup \tau')] \circ \hat{\gamma}[P])$

$\Leftarrow \quad$ {lemma (5.3) and similar one for the **else** branch}

$\lambda J \cdot \mathrm{let}\ J' = \lambda l \in \mathrm{in}_P[P] \cdot \mathrm{if}\ l = \mathrm{at}_P[S_t]\ ?\ J_{\mathrm{at}_P[S_t]} \sqcup \mathrm{Abexp}[B](J_t) \downharpoonleft J_t)\ \mathrm{in}$ (120)

$\quad \mathrm{let}\ J'' = \mathrm{APost}[S_t](J')\ \mathrm{in}$

$\quad\quad \lambda l \in \mathrm{in}_P[P] \cdot \mathrm{if}\ l = \ell'\ ?\ J''_\ell \sqcup J''_{\mathrm{after}_P[S_t]} \downharpoonleft J'')$

$\quad \sqcup$

$\quad \mathrm{let}\ J'' = \lambda l \in \mathrm{in}_P[P] \cdot \mathrm{if}\ l = \mathrm{at}_P[S_f]\ ?\ J_{\mathrm{at}_P[S_f]} \sqcup \mathrm{Abexp}[T(\neg B)](J_f) \downharpoonleft J_f)\ \mathrm{in}$

$\quad\quad \mathrm{let}\ J'' = \mathrm{APost}[S_f](J'')\ \mathrm{in}$

$\quad\quad\quad \lambda l \in \mathrm{in}_P[P] \cdot \mathrm{if}\ l = \ell'\ ?\ J''_\ell \sqcup J''_{\mathrm{after}_P[S_f]} \downharpoonleft J'')$

$= \quad$ {by grouping similar terms}

$\lambda J \cdot \mathrm{let}\ J' = \lambda l \in \mathrm{in}_P[P] \cdot \mathrm{if}\ l = \mathrm{at}_P[S_t]\ ?\ J_{\mathrm{at}_P[S_t]} \sqcup \mathrm{Abexp}[B](J_t) \downharpoonleft J_t)$

$\quad \mathrm{and}\ J'' = \lambda l \in \mathrm{in}_P[P] \cdot \mathrm{if}\ l = \mathrm{at}_P[S_f]\ ?\ J_{\mathrm{at}_P[S_f]} \sqcup \mathrm{Abexp}[T(\neg B)](J_f) \downharpoonleft J_f)\ \mathrm{in}$

$\quad\quad \mathrm{let}\ J'' = \mathrm{APost}[S_t](J')$

$\quad\quad \mathrm{and}\ J'' = \mathrm{APost}[S_f](J'')\ \mathrm{in}$

$\quad\quad\quad \lambda l \in \mathrm{in}_P[P] \cdot \mathrm{if}\ l = \ell'\ ?\ J''_\ell \sqcup J''_{\mathrm{after}_P[S_t]} \sqcup J''_\ell \sqcup J''_{\mathrm{after}_P[S_f]} \downharpoonleft J'' \sqcup J'')$

$= \quad$ {by locality (113) and labelling scheme (59) so that in particular $J''_\ell = J'''_\ell = J''_\ell = J''_\ell$ $= J''_\ell = J''_\ell$ and APost$[S_t]$ and APost$[S_f]$ do not interfere}

Implementation

```
matrix_t* _matrix_alloc_int(const int mr, const int nc)
{
  matrix_t* mat = (matrix_t*)malloc(sizeof(matrix_t));
  mat->nbrows = mat->_maxrows = mr;
  mat->nbcolumns = nc;
  mat->_sorted = s;
  if (mr*nc>0){
    int i;
    pkint_t* q;
    mat->_pinit = _vector_alloc_int(mr*nc);
    mat->p = (pkint_t**)malloc(mr * sizeof(pkint_t*));
    q = mat->_pinit;
    for (i=0;i<mr;i++){
      mat->p[i]=q;
      q=q+nc;
    }}
  return mat;
}

void backsubstitute(matrix_t* con, int rank)
{
  int i,j,k;
  for (k=rank-1; k>=0; k--) {
    j = pk_chemi_intp[k];
    for (i=0; i<k; i++) {
      if (pkint_sgn(con->p[i][j]))
        matrix_combine_rows(con,i,k,i,j);
    }
    for (i=k+1; i<con->nbrows; i++) {
      if (pkint_sgn(con->p[i][j]))
        matrix_combine_rows(con,i,k,i,j);
    }}
}
```


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: *Foudational 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 $\Downarrow$.

Theorem

For all programs p, if $\mathsf{po}(P, \phi, \psi)$ and $P, \mu \Downarrow \nu, v$ then $\phi(\mu)$ implies $\psi(\mu, \nu, v)$

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

Simpler checkers?

Proof

$\hat{\alpha}[P](\mathrm{Post}[\mathbf{if}\ B\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f\ \mathbf{fi}])$
$=$ {def. (110) of $\hat{\alpha}[P]$}
$\hat{\alpha}[P] \circ \mathrm{Post}[\mathbf{if}\ B\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f\ \mathbf{fi}] \circ \hat{\gamma}[P]$
$=$ {def. (103) of Post}
$\hat{\alpha}[P] \circ \mathrm{post}[\tau^*[\mathbf{if}\ B\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f\ \mathbf{fi}]] \circ \hat{\gamma}[P]$
$=$ {big step operational semantics (9.3)}
$\hat{\alpha}[P] \circ \mathrm{post}(1_{\Sigma[P]} \cup \tau^B) \circ \tau^*[S_t] \circ (1_{\Sigma[P]} \cup \tau') \cup (1_{\Sigma[P]} \cup \tau^B) \circ \tau^*[S_f] \circ (1_{\Sigma[P]} \cup \tau')] \circ \hat{\gamma}[P]$
$=$ {Galois connection (98) so that post preserves joins}
$\hat{\alpha}[P] \circ (\mathrm{post}(1_{\Sigma[P]} \cup \tau^B) \circ \tau^*[S_t] \circ (1_{\Sigma[P]} \cup \tau')) \cup$
$\mathrm{post}(1_{\Sigma[P]} \cup \tau^B) \circ \tau^*[S_f] \circ (1_{\Sigma[P]} \cup \tau')] \circ \hat{\gamma}[P]$
$=$ {Galois connection (106) so that $\hat{\alpha}[P]$ preserves joins}
$(\hat{\alpha}[P] \circ \mathrm{post}(1_{\Sigma[P]} \cup \tau^B) \circ \tau^*[S_t] \circ (1_{\Sigma[P]} \cup \tau')] \circ \hat{\gamma}[P]) \cup (\hat{\alpha}[P] \circ$
$\mathrm{post}(1_{\Sigma[P]} \cup \tau^B) \circ \tau^*[S_f] \circ (1_{\Sigma[P]} \cup \tau')] \circ \hat{\gamma}[P])$
$\Rightarrow$ {Ind
λJ.let
k
$\cup$
let $J^{f'} = \lambda l \in \mathrm{in}_P[P]\backslash l = \mathrm{at}_P[S_f] \ ?\ J_{\mathrm{fv}_P[S_f]} \cup \mathrm{Abexp}[T(\neg B)](J_l) \ \&\ J_l)$ in
let $J^{f''} = \mathrm{APost}[S_f](J^{f'})$ in
$\lambda l \in \mathrm{in}_P[P]\backslash l = t' \ ?\ J_t^{f''} \cup J^{f''}_{\mathrm{aftr}_P[S_t]} \ \&\ J_t^{f''})$
$=$ {by grouping similar terms}
λJ.let $J^{t'} = \lambda l \in \mathrm{in}_P[P]\backslash l = \mathrm{at}_P[S_t] \ ?\ J_{\mathrm{fv}_P[S_t]} \cup \mathrm{Abexp}[B](J_l) \ \&\ J_l)$
and $J^{f'} = \lambda l \in \mathrm{in}_P[P]\backslash l = \mathrm{at}_P[S_f] \ ?\ J_{\mathrm{fv}_P[S_f]} \cup \mathrm{Abexp}[T(\neg B)](J_l) \ \&\ J_l)$ in
let $J^{t''} = \mathrm{APost}[S_t](J^{t'})$
and $J^{f''} = \mathrm{APost}[S_f](J^{f'})$ in
$\lambda l \in \mathrm{in}_P[P]\backslash l = t' \ ?\ J_t^{t''} \cup J^{t''}_{\mathrm{aftr}_P[S_t]} \cup J_t^{f''} \cup J^{f''}_{\mathrm{aftr}_P[S_f]} \ \&\ J_t^{t''} \cup J_t^{f''})$
$=$ {by locality (113) and labelling scheme (59) so that in particular $J_t^{t''} = J_t^{t'} = J_t^{f'} = J_t^{f''}$ and APost$[S_t]$ and APost$[S_f]$ do not interfere}

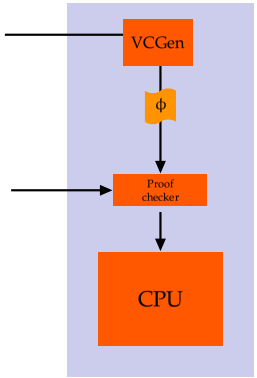
Implementation

```
matrix_t* _matrix_alloc_int(const int mr, const int nc)
{
  matrix_t* mat = (matrix_t*)malloc(sizeof(matrix_t));
  mat->nbrows = mat->_maxrows = mr;
  mat->nbcolumns = nc;
  mat->_sorted = s;
  if (mr*nc>0){
    int i;
    pkint_t* q;
    mat->_pinit = _vector_alloc_int(mr*nc);
    mat->p = (pkint_t**)malloc(mr * sizeof(pkint_t*));
    q = mat->_pinit;
    for (i=0;i<mr;i++){
      mat->p[i]=q;
      q=q+nc;
```

Do the two parts connect?

```
void backsubstitute(matrix_t* con, int rank)
{
  int i,j,k;
  for (k=rank-1; k>=0; k--) {
    j = pk_chernin_int[k];
    for (i=0; i<k; i++) {
      if (pkint_sgn(con->p[i][j]))
        matrix_combine_rows(con,i,k,i,j);
    }
    for (i=k+1; i<con->nbrows; i++) {
      if (pkint_sgn(con->p[i][j]))
        matrix_combine_rows(con,i,k,i,j);
    }}
}
```


TCB of certified PCC



1. In standard PCC
2. 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
3. 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 depends 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

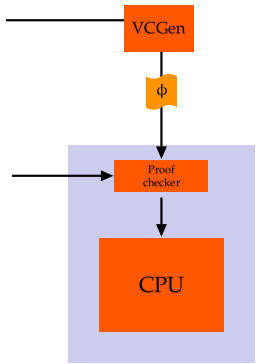
- In foundational PCC, certificates represent deductive proofs
  - Typing rules as lemmas
- We produce computational proofs by programming a type system and proving it correct!
  - Scalable and shorter proof terms
  - Supports proof by reflection in wholesale PCC
  - Allows extraction of certified checkers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

Reflection

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

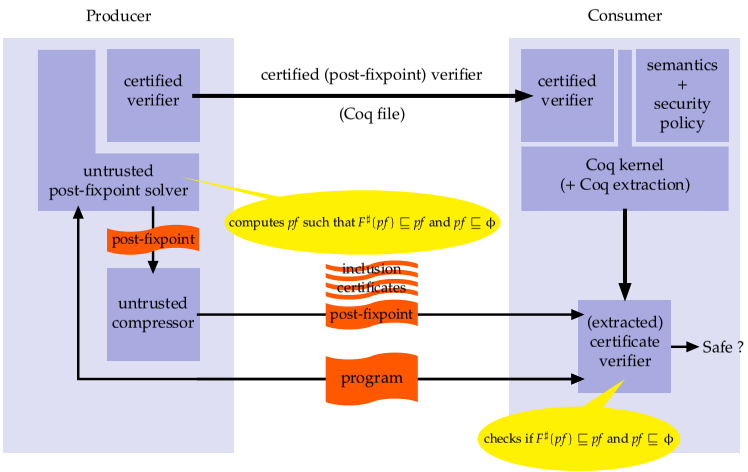
If $f\, a$ reduces to true, then $C\, a$ (`refl_eq` true) is a proof of $P\, a$

TCB of certified PCC

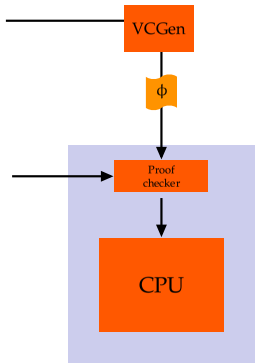


1. In standard PCC
2. If the VCGen is proved correct
   - the proof checker of the VCGen soundness proof (could be the same as for the code proof)
   - the formal definition of the language semantics
   - the formal definition of the security property
3. This is also a large TCB, but we think it is better to have 2,000 lines of formal definitions than 20,000 lines of C code!

Using executable checkers for retail PCC



TCB of certified PCC



1. In standard PCC
2. 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
3. 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!

Simple bytecode language SBC

A program is an array of instructions:

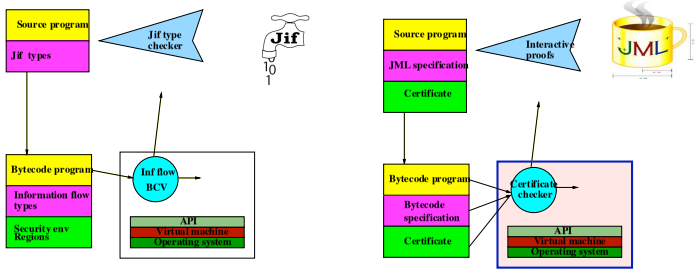
instr	::=	prim *op*	primitive operation
	|	push v	push value on top of stack
	|	load x	load value of x on stack
	|	store x	store top of stack in x
	|	if j	conditional jump
	|	goto j	unconditional jump
	|	return	return

where:

- $j \in \mathcal{P}$ is a program point
- $v \in \mathcal{V}$ is a value
- $x \in \mathcal{X}$ is a variable

Instances

- Type system for information flow based confidentiality policies
- Verification condition generator for logical specifications



Semantics

- States are of the form $\langle\langle i, \rho, s\rangle\rangle$ where:
  - $i : \mathcal{P}$ is the program counter
  - $\rho : \mathcal{X} \to \mathcal{V}$ maps variables to values
  - $s : \mathcal{V}^\star$ is the operand stack
- Operational semantics is given by rules are of the form

$$\frac{P[i] = ins \qquad constraints}{s \rightsquigarrow s'}$$

- Evaluation semantics: $P, \mu \Downarrow \nu, v$ iff $\langle\langle 1, \mu, \epsilon\rangle\rangle \rightsquigarrow^\star \langle\langle \nu, v\rangle\rangle$, where $\rightsquigarrow^\star$ is the reflexive transitive closure of $\rightsquigarrow$

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:

$$\begin{array}{lcl} negb & : & \text{bool} \to \text{bool} \\ iadd & : & (\text{int}, \text{int}) \to \text{int} \\ 0 & : & \text{int} \\ \ldots & & \ldots \end{array}$$

- Program P has type σ if it always returns a value of type σ

Semantics: rules

$$\frac{P[i] = \mathsf{prim}\ op \qquad n_1\ \underline{op}\ n_2 = n}{\langle\!\langle i, \rho, n_1 :: n_2 :: s \rangle\!\rangle \rightsquigarrow \langle\!\langle i+1, \rho, n :: s \rangle\!\rangle}$$

$$\frac{P[i] = \mathsf{push}\ n}{\langle\!\langle i, \rho, s \rangle\!\rangle \rightsquigarrow \langle\!\langle i+1, \rho, n :: s \rangle\!\rangle}$$

$$\frac{P[i] = \mathsf{load}\ x}{\langle\!\langle i, \rho, s \rangle\!\rangle \rightsquigarrow \langle\!\langle i+1, \rho, \rho(x) :: s \rangle\!\rangle}$$

$$\frac{P[i] = \mathsf{store}\ x}{\langle\!\langle i, \rho, v :: s \rangle\!\rangle \rightsquigarrow \langle\!\langle i+1, \rho(x := v), s \rangle\!\rangle}$$

$$\frac{P[i] = \mathsf{if}\ j}{\langle\!\langle i, \rho, \mathit{false} :: s \rangle\!\rangle \rightsquigarrow \langle\!\langle j, \rho, s \rangle\!\rangle}$$

$$\frac{P[i] = \mathsf{if}\ j}{\langle\!\langle i, \rho, \mathit{true} :: s \rangle\!\rangle \rightsquigarrow \langle\!\langle i+1, \rho, s \rangle\!\rangle}$$

$$\frac{P[i] = \mathsf{goto}\ j}{\langle\!\langle i, \rho, s \rangle\!\rangle \rightsquigarrow \langle\!\langle j, \rho, s \rangle\!\rangle}$$

$$\frac{P[i] = \mathsf{return}}{\langle\!\langle i, \rho, v :: s \rangle\!\rangle \rightsquigarrow \langle\!\langle \rho, v \rangle\!\rangle}$$

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

$$\vdash iadd : (\text{int} :: \text{int} :: s) \Rightarrow (\text{int} :: s) \qquad \nvdash iadd : (\text{bool} :: \text{int} :: s) \Rightarrow (\text{int} :: s)$$
$$\vdash pop : (\alpha :: s) \Rightarrow s \qquad \nvdash pop : s \Rightarrow s$$

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

$$\frac{P[i] = ins}{i \vdash st \Rightarrow st'} \qquad \frac{P[i] = ins}{i \vdash st \Rightarrow}$$

- Program $P : \tau$ is type-safe if there exists $S : \mathcal{P} \to \mathcal{T}^*$ s.t.
  - $S_1 = \epsilon$
  - for all $i, j \in \mathcal{P}$
    - $i \mapsto j \Rightarrow \exists \sigma.\, i \vdash S_i \Rightarrow \sigma \sqsubseteq S_j$;
    - $i \mapsto \Rightarrow \exists \tau'.\, i \vdash S_i \Rightarrow \tau' \sqsubseteq \tau$

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 $\langle\langle i, \rho, s\rangle\rangle$ is type-correct w.r.t. S and Γ, then:

- $P[i] = \mathsf{return}$ and $i \vdash S_i \Rightarrow \tau$
- $\langle\langle i, \rho, s\rangle\rangle \rightsquigarrow \langle\langle i', \rho', s'\rangle\rangle$ and $i \vdash S_i \Rightarrow S_{i'}$ and $\langle\langle i', \rho', s'\rangle\rangle$ is type-correct w.r.t. S and Γ

Run-time type checking is redundant

- A typed state is of the form $\langle\langle i, \rho, swt\rangle\rangle$, where swt is a stack of pairs of values and types
- A defensive virtual machine checks types at execution, i.e. $\rightsquigarrow_{\text{def}} \subseteq \mathsf{tstate} \times (\mathsf{tstate} + \{\textbf{TypeError}\})$
- If P is type-safe w.r.t. S and Γ, then executions of $\rightsquigarrow$ and $\rightsquigarrow_{\text{def}}$ coincide

Typing rules

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

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

$$\frac{P[i] = \mathsf{pop}}{i \vdash \alpha :: st \Rightarrow st}$$

$$\frac{P[i] = \mathsf{if}\ j}{i \vdash \mathsf{bool} :: st \Rightarrow st}$$

$$\frac{P[i] = \mathsf{load}\ x}{i \vdash st \Rightarrow \Gamma(x) :: st}$$

$$\frac{P[i] = \mathsf{store}\ x}{i \vdash \Gamma(x) :: st \Rightarrow st}$$

Transfer rules are deterministic

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 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'', replace st'' by $st' \sqcup st''$
  - 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 $\Gamma : \mathcal{X} \to \mathcal{T}$

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

$$\frac{P[i] = \mathsf{pop}}{i \vdash \alpha :: st \Rightarrow st}$$

$$\frac{P[i] = \mathsf{if}\ j}{i \vdash \mathsf{bool} :: st \Rightarrow st}$$

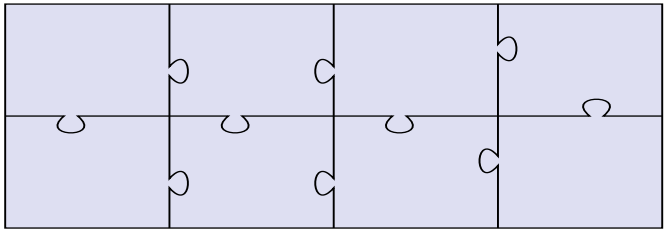
$$\frac{P[i] = \mathsf{load}\ x}{i \vdash st \Rightarrow \Gamma(x) :: st}$$

$$\frac{P[i] = \mathsf{store}\ x}{i \vdash \Gamma(x) :: st \Rightarrow st}$$

Transfer rules are deterministic

Simplification: the type system is flow insensitive

Verified bytecode verification



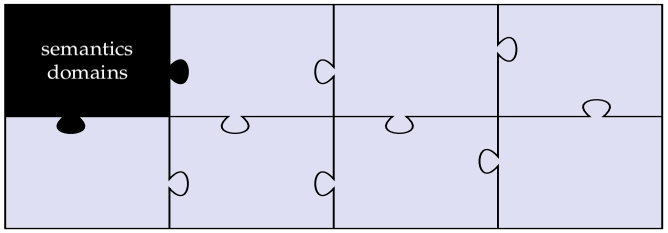
- A puzzle with 8 pieces,
- Each piece interacts with its neighbors

Lightweight bytecode verification

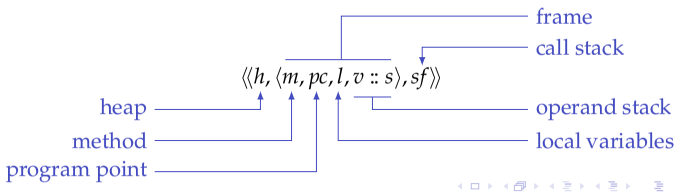
Provide types of junction points

- 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 $st' \leqslant st''$. If not, reject program

Verified bytecode verification



Example: JVM states



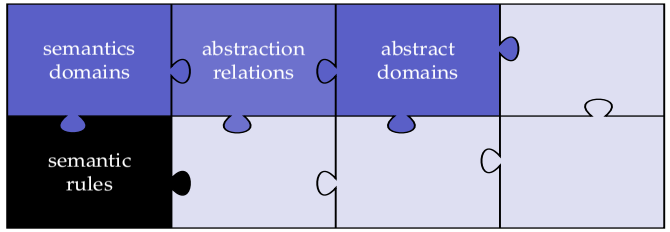
Lightweight bytecode verification

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 $st' \leqslant st''$. If not, reject program

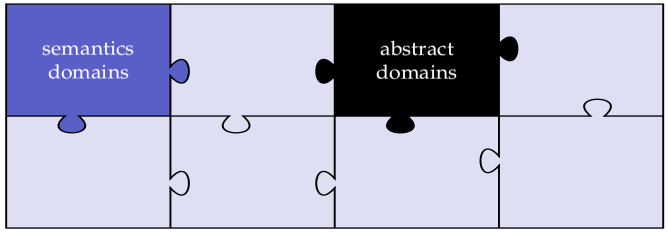
One pass verification, sound and complete wrt bytecode verification

Verified bytecode verification



- operational semantics $\rightsquigarrow$ between states
- Bicolano is the basis of all formalizations in Mobius

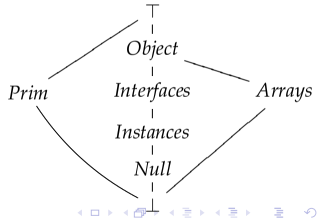
Verified bytecode verification



Type structure

- is partially ordered,
- with a top element $\top$ for errors,
- and a "lub" operator $\vee$
- w/o infinite increasing chains

$$x_0 \sqsubset x_1 \sqsubset \cdots \sqsubset \cdots$$



Bicolano

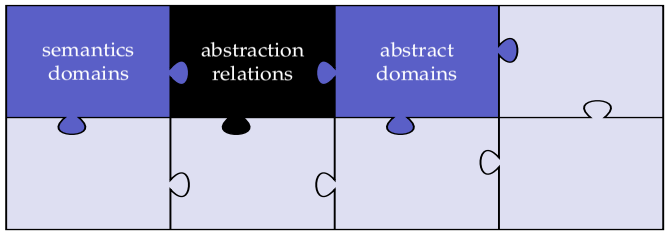
- Operational model of the (sequential fragment of) JVM
- Base block for all machine checked proofs in Mobius

- 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

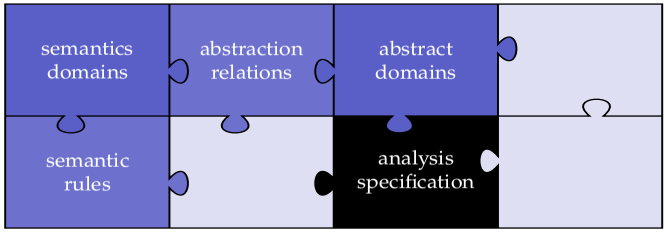
- More difficult to test the implementation
- Easier to do proofs
- Still compatible with executable checkers

Verified bytecode verification



- Each type represents a property on concrete values
- This correspondence is formalised by the relation *value* : *type* (that respects subtyping)

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 $F_P^\sharp(s^\sharp) \sqsubseteq^\sharp s^\sharp$
- equivalent to constraint system

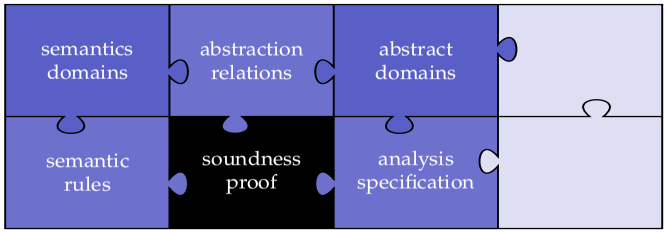
$$\begin{cases} f_1^\sharp(s_1^\sharp, \ldots, s_n^\sharp) & \sqsubseteq^\sharp & s_{i_1}^\sharp \\ & \ldots & \\ f_n^\sharp(s_1^\sharp, \ldots, s_n^\sharp) & \sqsubseteq^\sharp & s_{i_n}^\sharp \end{cases}$$

Examples of instructions

$$\frac{P[(m,pc)] = \textsf{push}\ c}{\langle\!\langle h, \langle m, pc, l, s\rangle, sf\rangle\!\rangle \rightsquigarrow \langle\!\langle h, \langle m, pc+1, l, c :: s\rangle, sf\rangle\!\rangle}$$

$$\frac{\begin{array}{rcl} P[(m,pc)] & = & \textsf{invokevirtual}\ m_{id} \\ m' & = & \text{methodLookup}(m_{id}, h(loc)) \\ V & = & v_1 :: \cdots :: v_{\text{nbArguments}(m_{id})} \end{array}}{\langle\!\langle h, \langle m, pc, l, loc :: V :: s\rangle, sf\rangle\!\rangle \rightsquigarrow \langle\!\langle h, \langle m', 1, V, \varepsilon\rangle, \langle m, pc, l, s\rangle :: sf\rangle\!\rangle}$$

Verified bytecode verification



If $s \rightsquigarrow s'$ 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

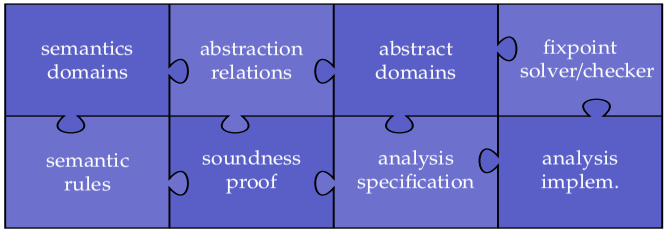
Method invokations

1. Small-step semantics formalisation (basis of Bicolano)
2. Intermediate semantics with big-step for method calls.
   - Simplify verification of modular verification methods
   - Problematic with multi-threading

$$\forall h, m, pc, s, l, v, h, sf,$$
$$\left(\begin{array}{l} \exists pc', s', l', \\ \quad \langle\!\langle h, < m, pc, s, l >, sf\rangle\!\rangle \Rightarrow^*_{\geq sf} \langle\!\langle h', < m, pc', v :: s', l' >, sf\rangle\!\rangle \\ \quad \wedge\ return(pc') \end{array}\right)$$
$$\Leftrightarrow$$
$$m \vdash \langle\!\langle h, k, pc, s, l\rangle\!\rangle \Rightarrow^* \langle\!\langle h', v\rangle\!\rangle$$

A similar equivalence holds for exceptional termination

Verified bytecode verification

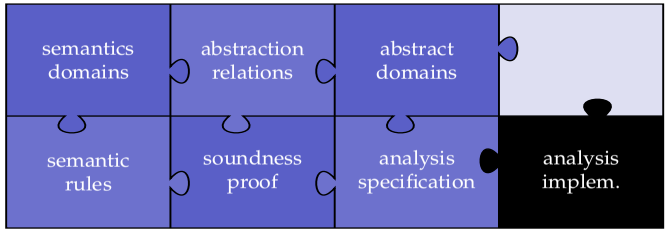


Final results

$$\left.\begin{array}{c} \vdash P \\ s_{\text{init}} \Downarrow s_{\text{final}} \\ s_{\text{final}}\ \text{type} - \text{correct} \end{array}\right\} \Rightarrow s_{\text{final}}\ \text{type} - \text{correct}$$

- progress
- commutation defensive and offensive machine

Verified bytecode verification



Implement transfer rules: from relations to functions

Abstraction-Carrying Code

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

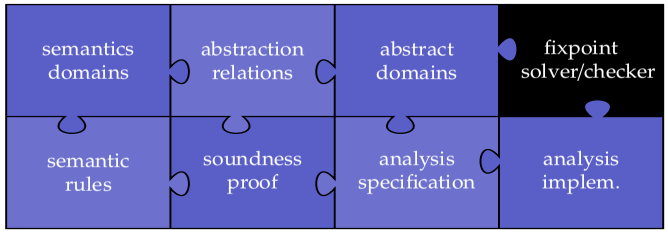
Verified abstraction carrying code

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

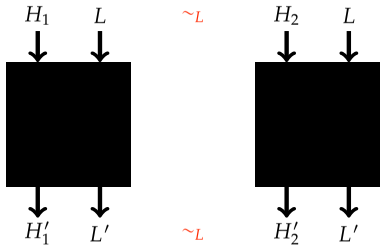


- Generic tools for fixpoint computation
- Fixpoint checker

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

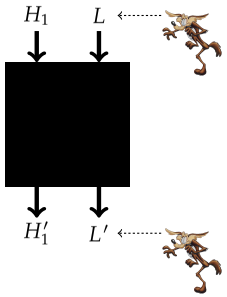


$$\forall s_1, s_2,\ s_1 \sim_L s_2 \wedge P, s_1 \Downarrow s_1' \wedge P, s_2 \Downarrow s_2' \implies s_1' \sim_L s_2'$$

High = confidential Low = public

Non-interference

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



High = confidential Low = public

Examples of insecure programs

Direct flow

load y_H
store x_L
return

Indirect flow

load y_H
if 5
push 0
store x_L
return

Flow via return

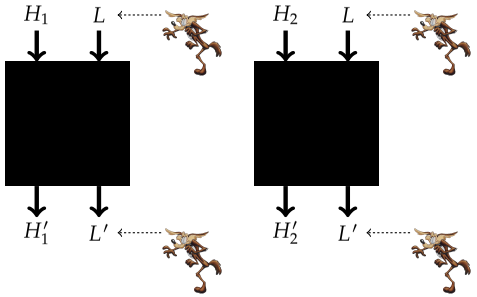
load y_H
if 5
push 1
return
push 0
return

Flow via operand stack

push 0
push 1
load y_H
if 6
swap
store x_L
return 0

Non-interference

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



High = confidential Low = public

Control dependence regions

Approximating the scope of branching statements

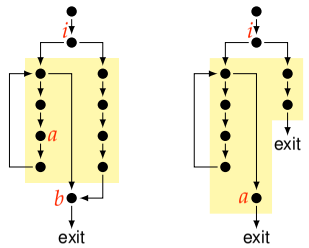
A program point j is in a *control dependence region* of a branching point i if

- 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 $region(i)$
- b does not necessary belong to $region(i)$



Policy

- A lattice of security levels $\mathcal{S} = \{H, L\}$ with $L \leqslant H$
- Each program is given a security signature: $\Gamma : \mathcal{X} \to \mathcal{S}$ and k_{ret}.
- Γ determines an equivalence relation $\sim_L$ on memories: $\rho \sim_L \rho'$ iff

$$\forall x \in \mathcal{X}.\Gamma(x) \leqslant L \Rightarrow \rho(x) = \rho'(x)$$

- Program P is *non-interfering* w.r.t. signature Γ, k_{ret} iff for every $\mu, \mu', \nu, \nu', v, v'$,

$$\left.\begin{array}{c} P, \mu \Downarrow \nu, v \\ P, \mu' \Downarrow \nu', v' \\ \mu \sim_L \mu' \end{array}\right\} \Rightarrow \nu \sim_L \nu' \wedge (k_{\text{ret}} \leqslant L \Rightarrow v = v')$$

CDR usage : tracking implicit flows

In a typical type system for a structured language:

$$\frac{\vdash exp : k \qquad [k_1] \vdash c_1 \qquad [k_2] \vdash c_2 \qquad k \leqslant k_1 \qquad k \leqslant k_2}{[k] \vdash \text{if } exp \text{ then } c_1 \text{ else } c_2}$$

In our context

- se: a security environment that attaches a security level to each program point
- for each branching point i, we constrain $se(j)$ for all $j \in region(i)$

$$\frac{P[i] = \text{if } i' \qquad \forall j \in region(i),\ k \leqslant se(j)}{i \vdash k :: st \Rightarrow \cdots}$$

Type system

- Transfer rules of the form

$$\frac{P[i] = ins \qquad constraints}{i \vdash st \Rightarrow st'} \qquad \frac{P[i] = ins \qquad constraints}{i \vdash st \Rightarrow}$$

  where $st, st' \in \mathcal{S}^\star$.
- Types assign stack of security levels to program points

$$S : \mathcal{P} \to \mathcal{S}^\star$$

- $S \vdash P$ iff $S_1 = \epsilon$ and for all $i, j \in \mathcal{P}$
  - $i \mapsto j \Rightarrow \exists st'.\ i \vdash S_i \Rightarrow st' \wedge st' \leqslant S_j$;
  - $i \mapsto\Rightarrow\ i \vdash S_i \Rightarrow$

The transfer rules and typability relation are implicitly parametrized by a signature Γ, k_{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 $region \in \mathcal{P} \to \wp(\mathcal{P})$ and $jun \in \mathcal{P} \rightharpoonup \mathcal{P}$.

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

SOAP2: for all program points i, j, k, if $j \in region(i)$ and $j \mapsto k$, then either $k \in region(i)$ or $k = jun(i)$;

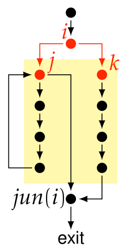
SOAP3: for all program points i, j, if $j \in region(i)$ and $j \mapsto$ then $jun(i)$ is undefined.

CDR soundness

SOAP (Safe Over Approximation Properties)

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

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



SOAP2: for all program points i, j, k, if $j \in region(i)$ and $j \mapsto k$, then either $k \in region(i)$ or $k = jun(i)$;

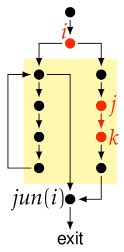
SOAP3: for all program points i, j, if $j \in region(i)$ and $j \mapsto$ then $jun(i)$ is undefined.

CDR soundness

SOAP (Safe Over Approximation Properties)

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

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



SOAP2: for all program points i, j, k, if $j \in region(i)$ and $j \mapsto k$, then either $k \in region(i)$ or $k = jun(i)$;

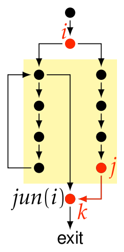
SOAP3: for all program points i, j, if $j \in region(i)$ and $j \mapsto$ then $jun(i)$ is undefined.

CDR soundness

SOAP (Safe Over Approximation Properties)

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

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



SOAP2: for all program points i, j, k, if $j \in region(i)$ and $j \mapsto k$, then either $k \in region(i)$ or $k = jun(i)$;

SOAP3: for all program points i, j, if $j \in region(i)$ and $j \mapsto$ then $jun(i)$ is undefined.

State equivalence

Unwinding lemmas focus on state equivalence $\sim_L$.

State equivalence

$\langle\!\langle i, \rho, s \rangle\!\rangle \sim_L \langle\!\langle i', \rho', s' \rangle\!\rangle$ if:

- Memory equivalence $\rho \sim_L \rho'$
- Operand stack equivalence $s \overset{i,i'}{\sim}_L s'$ (defined w.r.t. S)

CDR soundness

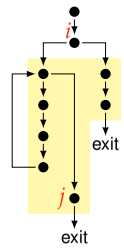
SOAP (Safe Over Approximation Properties)

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

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

SOAP2: for all program points i, j, k, if $j \in region(i)$ and $j \mapsto k$, then either $k \in region(i)$ or $k = jun(i)$;

SOAP3: for all program points i, j, if $j \in region(i)$ and $j \mapsto$ then $jun(i)$ is undefined.



State equivalence

Unwinding lemmas focus on state equivalence $\sim_L$.

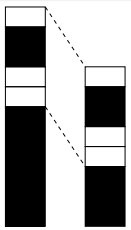
State equivalence

$\langle\!\langle i, \rho, s \rangle\!\rangle \sim_L \langle\!\langle i', \rho', s' \rangle\!\rangle$ if:

- Memory equivalence $\rho \sim_L \rho'$
- Operand stack equivalence $s \overset{i,i'}{\sim}_L s'$ (defined w.r.t. S)

Operand stack equivalence $s \overset{i,i'}{\sim}_L s'$ 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



Transfer rules

$$\frac{P[i] = \mathsf{push}\ n}{i \vdash st \Rightarrow se(i) :: st} \qquad \frac{P[i] = \mathsf{binop}\ op}{i \vdash k_1 :: k_2 :: st \Rightarrow (k_1 \sqcup k_2) :: st}$$

$$\frac{P[i] = \mathsf{load}\ x}{i \vdash st \Rightarrow (\Gamma(x) \sqcup se(i)) :: st} \qquad \frac{P[i] = \mathsf{store}\ x \qquad se(i) \sqcup k \leqslant \Gamma(x)}{i \vdash k :: st \Rightarrow st}$$

$$\frac{P[i] = \mathsf{goto}\ j}{i \vdash st \Rightarrow st} \qquad \frac{P[i] = \mathsf{return} \qquad se(i) \sqcup k \leqslant k_r}{i \vdash k :: st \Rightarrow}$$

$$\frac{P[i] = \mathsf{if}\ j \qquad \forall j' \in region(i),\ k \leqslant se(j')}{i \vdash k :: \epsilon \Rightarrow \epsilon}$$

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 $S \vdash P$ (w.r.t. *se* and *cdr*) then P is non-interfering.

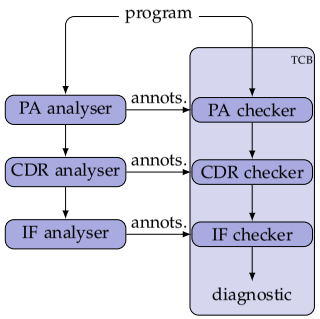
Direct application of

- Low (locally respects) unwinding lemma:
  If $s \sim_L s'$ and $s \rightsquigarrow t$ and $s' \rightsquigarrow t'$, then $t \sim_L t'$, provided $s \cdot pc = s' \cdot pc$
- High (step consistent) unwinding lemma:
  If $s \sim_L s'$ and $s \rightsquigarrow t$ and then $t \sim_L t'$, provided $s \cdot pc = i$ 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:

1. the PA (pre-analyse) analyser computes information to reduce the control flow graph.
2. the CDR analyser computes *control dependence regions* (to deal with implicit flows)
3. 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 lighweight 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.

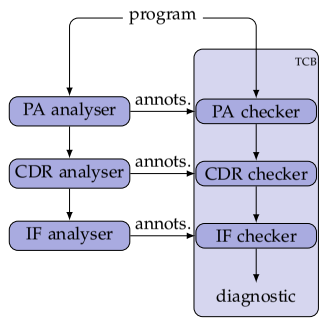
The PA (pre-analyse) analyser computes information that is used to reduce the control flow graph and to detect branches that will never be taken.

- 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*.

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.



Information flow type system

Type annotations required on programs:

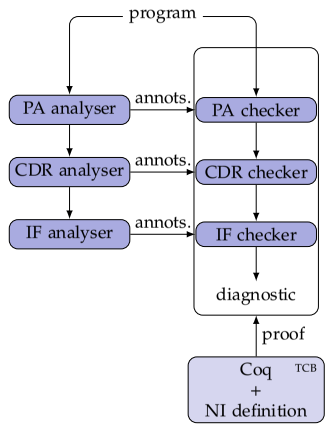
- $ft : \mathcal{F} \rightarrow \mathcal{S}$ attaches security levels to fields,
- $at : \mathcal{M} \times \mathcal{P} \rightharpoonup \mathcal{S}$ attaches security levels to contents of arrays at their creation point
- each method posseses one (or several) signature(s):

$$\vec{k_v} \xrightarrow{k_h} \vec{k_r}$$

  - $\vec{k_v}$ provides the security level of the method parameters (and local variables),
  - k_h: effect of the method on the heap,
  - $\vec{k_r}$ is a record of security levels of the form $\{n : k_n, e_1 : k_{e_1}, \ldots e_n : k_{e_n}\}$
    - k_n is the security level of the return value (normal termination),
    - k_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

$$\frac{P[i] = ins \quad constraints}{\Gamma, ft, region, se, sgn, i \vdash^{\tau} st \Rightarrow st'}$$

Selected rules

$$\frac{\begin{array}{c} P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad \Gamma_{m_{\mathrm{ID}}}[k] = \vec{k_a'} \xrightarrow{k_h'} \vec{k_r'} \\ k \sqcup k_h \sqcup se(i) \leqslant k_h' \qquad k \leqslant \vec{k_a'}[0] \qquad \forall i \in [0, \mathsf{length}(\mathsf{st}_1) - 1],\ \mathsf{st}_1[i] \leqslant \vec{k_a'}[i+1] \\ e \in \mathsf{excAnalysis}(m_{\mathrm{ID}}) \cup \{\mathbf{np}\} \qquad \forall j \in region(i,e),\ k \sqcup \vec{k_r'}[e] \leqslant se(j) \qquad \mathsf{Handler}(i,e) = t \end{array}}{\Gamma, region, se, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, i \vdash^{e} \mathsf{st}_1 :: k :: \mathsf{st}_2 \Rightarrow (k \sqcup \vec{k_r'}[e]) :: \varepsilon}$$

$$\frac{P[i] = \mathsf{xastore} \qquad k_1 \sqcup k_2 \sqcup k_3 \leqslant k_e \qquad \forall j \in region(i, \emptyset),\ k_e \leqslant se(j)}{\Gamma, region, se, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, i \vdash^{\emptyset} k_1 :: k_2 :: k_3[k_e] :: \mathsf{st} \Rightarrow \mathsf{lift}_{k_e}(\mathsf{st})}$$

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

Example

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

1 load x
2 if 5
3 new C
4 throw
5 load y
6 push 3
7 putfield f
8 push 1
9 return

$$m : (x : L,\ y : H) \xrightarrow{H} \{\mathsf{n} : H,\ C : L,\ \mathbf{np} : H\}$$

- $k_h = H$: no side effect on low fields ,
- $\vec{k_r}[n] = H$: result depends on y,
- termination by an exception C doesn't depend on y,
- but termination by a null pointer exception does.

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.

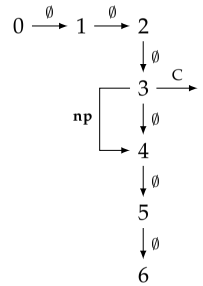
Structure of proofs

1. Itermediate semantics simplifies the intermediate definition of indistinguishability (call stacks),
2. Second intermediate semantics : annotated semantics with result of pre-analyses
   - the pre-analyse checker enforces 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.

Fine grain exceptions handling : example

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

0 : load o_L
1 : load y_H
2 : load x_L
3 : invokevirtual m
4 : store z_H
5 : push 1
6 : store t_L
handler : $[0,3]$, NullPointer $\rightarrow$ 4



With only one level for all exceptions

- [4,5,6] is a high region (depends on y_H): $t_L = 1$ is rejected

With our signature

- [4,5,6] is a low region: $t_L = 1$ is accepted
- a region is now associated to a branching point and a step kind (normal step or exception step)

Information release for JVM

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

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)

Concurrency

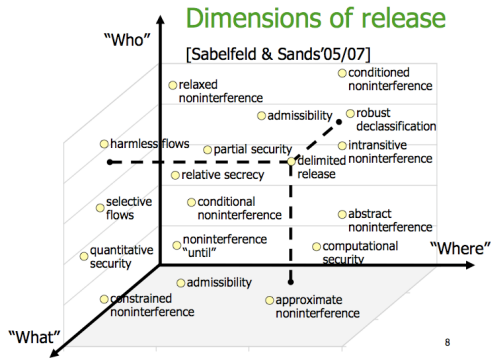
- Mobile code applications often exploit concurrency
- Concurrent execution of secure sequential programs is not necessarily secure:

$$\mathsf{if}(h > 0)\{\mathsf{skip};\mathsf{skip}\}\{\mathsf{skip}\};l := 1 \quad || \quad \mathsf{skip};\mathsf{skip};\mathsf{l} := 2$$

- 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

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)

$P, \mu \Downarrow \nu, v$ implies $[\![P]\!], \mu \Downarrow \nu, v$ Thus source programs satisfy an input/output property iff their compilation does

$$\begin{array}{l}\forall P, \phi, \psi, \mu, \nu, v. \\ (\phi(\mu) \Rightarrow P, \mu \Downarrow \nu, v \Rightarrow \psi(\mu, \nu, v)) \\ \Rightarrow (\phi(\mu) \Rightarrow [\![P]\!], \mu \Downarrow \nu, v \Rightarrow \psi(\mu, \nu, v))\end{array}$$

But are typable programs compiled into typable programs?

$$\forall P, \vdash P \Longrightarrow \exists S.\ S, \vdash [\![P]\!]$$

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:

commands	c	$::=$	$x := e$	assignment
		$\mid$	$\mathsf{if}(e)\{c\}\{c\}$	conditional
		$\mid$	$\mathsf{while}(e)\{c\}$	loop
		$\mid$	$c; c$	sequence
		$\mid$	skip	skip
		$\mid$	$\mathsf{return}\ e$	return value

Semantics is standard:

- States are pairs $\langle\langle c, \rho\rangle\rangle$
- Small-step semantics $\langle\langle c, \rho\rangle\rangle \rightsquigarrow \langle\langle c', \rho'\rangle\rangle$ or $\langle\langle c, \rho\rangle\rangle \rightsquigarrow \langle\langle \nu, v\rangle\rangle$
- Evaluation semantics $c, \mu \Downarrow \langle\langle \nu, v\rangle\rangle$ iff $c, \mu \rightsquigarrow^{\star} \langle\langle \nu, v\rangle\rangle$

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

$\mathsf{if}(y_H)\{x := 1\}\{x := 2\};$
$x' := 3;$
return 2

load y_H	L	
if 6	L	
push 1	H	$\in region(2)$
store x	H	$\in region(2)$
goto 8	H	$\in region(2)$
push 2	H	$\in region(2)$
store x	H	$\in region(2)$
push 3	L	$jun(2)$
store x'	L	
push 2	L	
return	L	

Information flow type system

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

$$\frac{e : k \quad k \sqcup pc \leqslant \Gamma(x)}{[pc] \vdash x := e} \qquad \frac{[k] \vdash c \qquad [k] \vdash c'}{[pc] \vdash c; c'}$$

$$\frac{e : k \quad [k] \vdash c_1 \quad [k] \vdash c_2}{[pc] \vdash \mathsf{if}(e)\{c_1\}\{c_2\}} \qquad \frac{e : k \quad [k] \vdash c}{[pc] \vdash \mathsf{while}(e)\{c\}}$$

$$\frac{e : k \quad k \sqcup pc \leqslant k_{\text{ret}}}{[pc] \vdash \mathsf{return}\ e} \qquad \frac{}{[pc] \vdash \mathsf{skip}}$$

plus subtyping rules

$$\frac{[pc] \vdash c \quad pc' \leqslant pc}{[pc'] \vdash c'} \qquad \frac{e : k \quad k \leqslant k'}{e : k'}$$

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

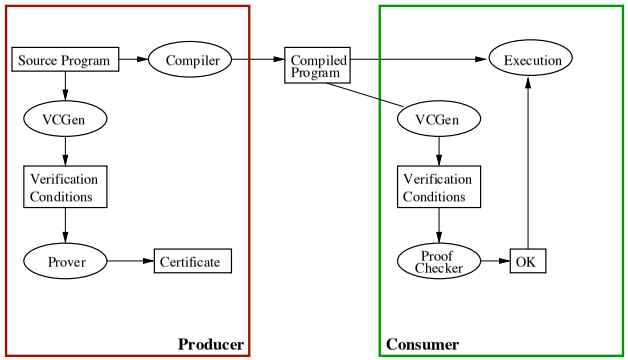
- information flow type system for sequential Java (with exceptions!)
- concurrency (using naive rule for parallel composition)
- declassification

Compiling statements

$$\begin{array}{rcl}
[\![x]\!] & = & \mathsf{load}\ x \\
[\![v]\!] & = & \mathsf{push}\ v \\
[\![e_1\ op\ e_2]\!] & = & [\![e_2]\!];\ [\![e_1]\!];\ \mathsf{binop}\ op \\
\\
k{:}[\![x := e]\!] & = & [\![e]\!];\ \mathsf{store}\ x \\
k{:}[\![i_1; i_2]\!] & = & k{:}[\![i_1]\!]; k_2{:}[\![i_2]\!] \\
\text{where } k_2 & = & k + |[\![i_1]\!]| \\
\\
k{:}[\![\mathsf{return}\ e]\!] & = & [\![e]\!];\ \mathsf{return} \\
\\
k{:}[\![\mathsf{if}(e_1\ cmp\ e_2)\{i_1\}\{i_2\}]\!] & = & [\![e_2]\!];\ [\![e_1]\!];\ \mathsf{if}\ cmp\ k_2; k_1{:}[\![i_1]\!];\ \mathsf{goto}\ l;\ k_2{:}[\![i_2]\!] \\
\text{where } k_1 & = & k + |[\![e_2]\!]| + |[\![e_1]\!]| + 1 \\
k_2 & = & k_1 + |[\![i_1]\!]| + 1 \\
l & = & k_2 + |[\![i_2]\!]| \\
\\
k{:}[\![\mathsf{while}(e_1\ cmp\ e_2)\{i\}]\!] & = & [\![e_2]\!];\ [\![e_1]\!];\ \mathsf{if}\ cmp\ k_2; k_1{:}[\![i]\!];\ \mathsf{goto}\ k \\
\text{where } k_1 & = & k + |[\![e_2]\!]| + |[\![e_1]\!]| + 1 \\
k_2 & = & k_1 + |[\![i]\!]| + 1
\end{array}$$

Certifying compilation vs certificate translation

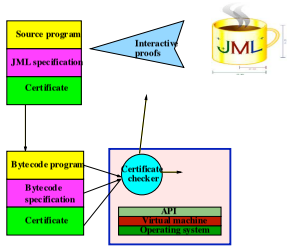
Source Code Verification



Compiling evidence from program verification

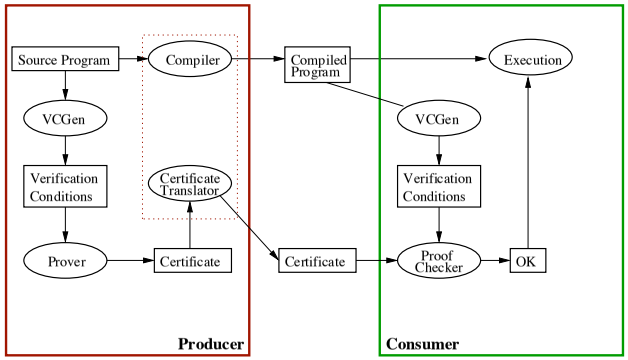
The missing link

- 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



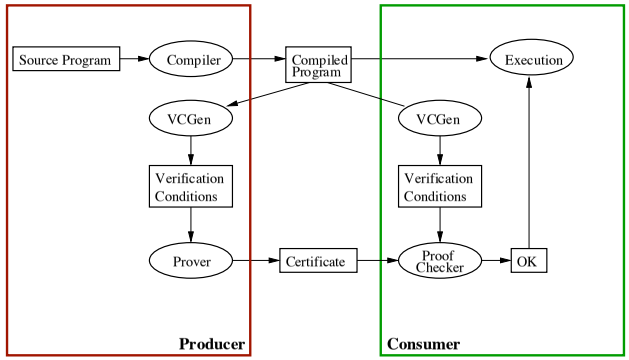
Certifying compilation vs certificate translation

Certificate Translation



Certifying compilation vs certificate translation

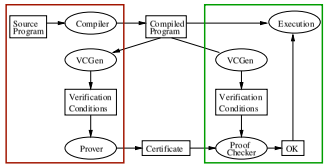
Traditional PCC

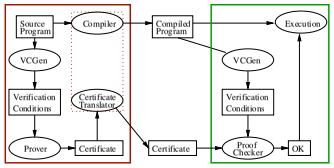


Overview of results

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

Certifying compilation vs certificate translation





Conventional PCC		Certificate Translation
Automatic (Invariant inference)	**Specification**	Manual
Automatic	**Verification**	Interactive (source code)
Safety, inf. flow, resources (approximation)	**Properties**	Security policies Functional properties (precise)

Program Specification

$\{pre\}$
ins_1
$\{\varphi_1\}$
ins_2
$\vdots$
$\{\varphi_2\}$
ins_k
$\{post\}$

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

Possibly annotated instructions

$\overline{ins} ::= ins \mid \langle \varphi, ins \rangle$

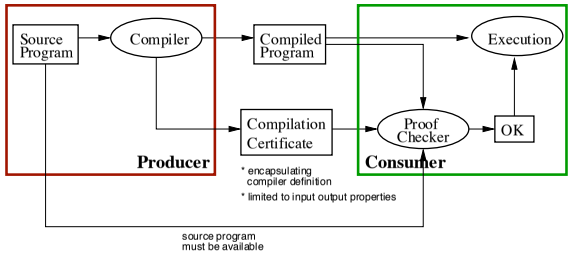
- A partially annotated program is a triple $\langle P, \Phi, \Psi \rangle$ s.t.
  - Φ is a precondition and Ψ is a postcondition
  - P is a sequence of possibly annotated instructions

Certificate translation vs certified compilation

Certified compilation aims at producing a proof term H such that

$$H : \forall P\ \mu\ \nu,\ P, \mu \Downarrow \nu \implies [\![P]\!], \mu \Downarrow \nu$$

Thus, we can build a proof term $H' : \{\phi\}[\![P]\!]\{\psi\}$ from H and $H_0 : \{\phi\}P\{\psi\}$

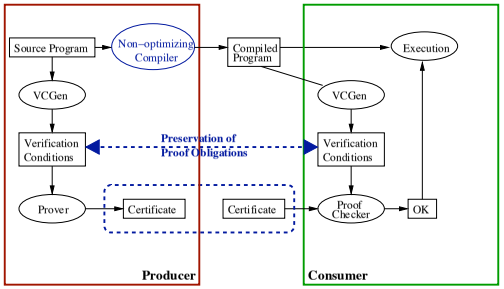


Preservation of proof obligations

Non-optimizing compiler

Proof obligations are syntactically equal

$$\mathsf{PO}(P, \phi, \psi) = \mathsf{PO}([\![P]\!], \phi, \psi)$$



Building a certificate

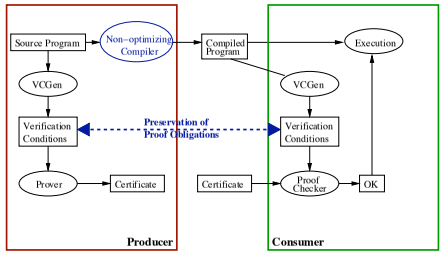
Certification of annotated programs is performed in three steps

1. A verification condition generator fully annotates the program, and extracts a set of verification conditions (a.k.a. proof obligations)
2. verification conditions are discharged interactively
3. 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 $\mathsf{PO}(P, \phi, \psi)$

- Precondition implies the weakest precondition of entry point:

$$\phi \Rightarrow \mathsf{wp}_i(1)$$

- For all annotated program points ($P[k] = \langle \varphi, i \rangle$), the annotation φ implies the weakest precondition of the instruction at k:

$$\varphi \Rightarrow \mathsf{wp}_i(k)$$

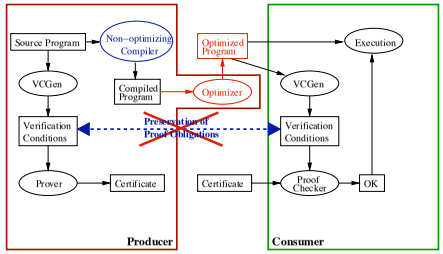
An annotated program is correct if its verification conditions are valid.

Soundness

If (P, ϕ, ψ) is correct, and $P, \mu \Downarrow \nu, v$, and $\mu \models \phi$, then

- $\mu, \nu \models \psi[^{v}/_{res}]$
- all intermediate assertions are verified

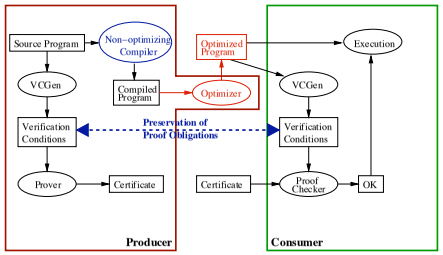
Optimizing Compilers



For optimizations that are justified by arithmetic reasoning:

1. Specifying and certifying automatically the result of the analysis
2. Merging annotations (i.e. strengthen invariants)
3. Merging certificates

Optimizing Compilers



Conclusions

Deployment of secure mobile code can benefit from:

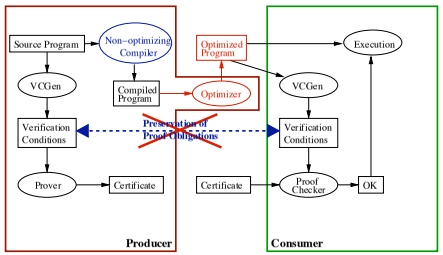
- advanced verification mechanisms at bytecode level
- methods to "compile" evidence from producer to consumer
- machine checked proofs of certificate checkers

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`

Optimizing Compilers



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)