Quis Custodiet Ipsos Custodes?

Gregor Snelting, Daniel Wasserrab, Andreas Lochbihler, Denis Lohner
funded by DFG Sn11/10, Sn11/12

LEHRSTUHL PROGRAMMIERPARADIGMEN, KIT

```
theorem nonInterferenceSecurity:
  assumes "[cf_1] \approx_L [cf_2]" and "(\text{-High-}) \notin \text{HRB-slice (CFG-node \text{-Low-})}_\text{CFG}" and "valid-edge a"
  and "sourcenode a = (\text{-High-})" and "targetnode a = n" and "kind a = (\text{ls. True})\nu" and "n \neq c"
  and "final c" and "\langle c, [cf_1]\rangle \Rightarrow \langle c', s_1\rangle" and "\langle c, [cf_2]\rangle \Rightarrow \langle c', s_2\rangle"
  shows "s_1 \approx_L s_2"

proof -
  from High-target-Entry-edge obtain ax where "valid-edge ax" and "sourcenode ax = (\text{-Entry-})" and "targetnode ax = (\text{-High-})" and "kind ax = (\text{ls. True})\nu" by blast
  from \text{"n \neq c\" \langle c, [cf_1]\rangle \Rightarrow \langle c', s_1\rangle" obtain n_1 as_1 cf_1 where \text{"n \neg \text{as}_1 \rightarrow \sqrt{\neg} n_1\" and "n_1 \neq c\" and "preds (kinds as_1) [(cf_1, undefined)]" and "transfers (kinds as_1) [(cf_1, undefined)] = cf_1" and "map fst cf_1 = s_1" by(fastsimp dest:fundamental-property)
  from \text{"n \neg \text{as}_1 \rightarrow \sqrt{\neg} n_1\" \text{valid-edge a\" \text{sourcenode a = (\text{-High-})\" \text{targetnode a = n\" \text{kind a = (\text{ls. True})\nu}\" have \text{"(\text{-High-}) -a\neg\neg \text{as}_1 \rightarrow \sqrt{\neg} n_1\" by(fastsimp intro:Cons-path simp:(vp-def valid-path-def)
  from \text{"final c\" \text{"n \neq c\" \text{obtain a_1 where \text{valid-edge a_1\" and \text{sourcenode a_1 = n_1\" and \text{targetnode a_1 = (\text{-Low-})\" and \text{kind a_1 = \text{id}}\" by(fastsimp dest:final-edge-Low)
```

http://pp.info.uni-karlsruhe.de/
Who will guard the Guards?
Many software security analysis algorithms are published without soundness proof, some with a manual proof only
Who will guard the Guards?
Many software security analysis algorithms are published without soundness proof, some with a manual proof only

**Vision of our Project:**
- provide machine-checked proofs for IFC algorithms
- reaching a new level of reliability in Language Based Security
- developing new techniques to validate the underlying language description
- integrating semantics, theorem provers and program analysis with Language Based Security

**Ultimate Goal:** automatically generate an executable, completely machine-verified, PDG-based IFC tool
Starting Point and Goals

Developed in earlier, long-standing projects:

- TUM: Jinja, Isabelle formalization of realistic Java subset includes type system, operational semantics, type safety proof, verified JVM, verified compiler all proofs machine checked
- KIT: Joana, program dependence graph for full Java flow-sensitive, context-sensitive, object-sensitive scales to 100kLOC; Eclipse plug in GUI + IFC algorithm based on PDGs + manual correctness proof

Project Idea

1. verify the PDG-based IFC algorithm using Isabelle
2. support verification by innovative counter example generators
A tiny PDG

```java
1  a = input();
2  while (n>0) {
3      x = input();
4      if (x>0)
5          b = a;
6      else
7          c = b;
8  }
9  z = c;
```

**Slicing theorem:**

No path $x \rightarrow^* y \implies$ no information flow $x \rightarrow y$ guaranteed

∃ Path $x \rightarrow^* y \implies$ information flow $x \rightarrow y$ possible

Backward slice: $BS(y) = \{x \mid x \rightarrow^* y\}$

Precise PDG construction for full Java is very complex

requires precise points-to analysis

Scalability: ca 100kLOC
Flow equations (intraprocedural)

$S(x)$: security level for statement/variable $x$

- Confidentiality: $S(x) \geq \bigcup_{y \in \text{pred}(x)} S(y)$
- Integrity: $S(x) \leq \bigcap_{y \in \text{pred}(x)} S(y)$
- required and provided levels $R(x), P(x)$ (for I/O only): $R(x) \geq S(x)$ and

$$S(x) = \begin{cases} P(x) \sqcup \bigcup_{y \in \text{pred}(x)} S(y) & \text{if } P(x) \text{ defined} \\ \bigcup_{y \in \text{pred}(x)} S(y) & \text{otherwise} \end{cases}$$

- for given PDG, $P(x), R(x), S$ is computed by standard fixpoint iteration
Implementation

JOANA Eclipse Plugin: slicing, definition of $P(x)$, $R(x)$, declassifications displays security violations, flow through the program
Results in Karlsruhe

- precise PDGs for full Java bytecode [PASTE ’04, Hamm ’09]
- precise slicing of multithreaded programs
  [FSE ’03, SCAM ’07, Hamm ’09, JASE ’09a]
- path conditions in PDGs: precise, necessary conditions for
  information flow, “witnesses”
  [SAS ’96, ICSE ’02, TOSEM ’06, SCAM ’07, PLAS ’08, JASE ’09b]
- IFC for full Java, based on PDGs and path conditions
  [ISSSE ’06, ISOLA ’06, PLAS ’08, IJIS ’09]
Results in Karlsruhe

- precise PDGs for full Java bytecode [PASTE ’04, Hamm ’09]
  precise slicing of multithreaded programs
  [FSE ’03, SCAM ’07, Hamm ’09, JASE ’09a]
- path conditions in PDGs: precise, necessary conditions for
  information flow, “witnesses”
  [SAS ’96, ICSE ’02, TOSEM ’06, SCAM ’07, PLAS ’08, JASE ’09b]
- IFC for full Java, based on PDGs and path conditions
  [ISSSE ’06, ISOLA ’06, PLAS ’08, IJIS ’09]

Quis Custodiet: Isabelle proofs

1. Multiple Inheritance in C++ is Type Safe [OOPSLA ’06, AFP ’06]
2. PDG-based IFC is correct [TPHOLS ’08, PLAS ’09, VERIFY ’10]
3. Verified Compiler for Java Threads [FOOL ’08, ESOP ’10]
C++ Multiple Inheritance is Type Safe
Multiple Inheritance in C++

A valid C++ program:

```cpp
class A { int x; };
class B { int x; };
class C : virtual A, virtual B { int x; };
class D : virtual A, virtual B, C { };
...
D* d = new D();
d->x = 42;
```
Multiple Inheritance in C++

A valid C++ program:

```cpp
class A { int x; };
class B { int x; };
class C : virtual A, virtual B { int x; };
class D : virtual A, virtual B, C { };
...
D* d = new D();
d->x = 42;
```

- but: gcc rejects it as ambiguous!
- yet, other compilers (z.B. Intel) do accept it
- problem: subobject-domination far from trivial
Subobjects and Domination

- necessary due to multiple inherits of the same super class
- Subobject: entity with the fields of the resp. class
- accessed via class path

```cpp
class A { int x; };
class B { int x; };
class C : virtual A, virtual B { int x; };
class D : virtual A, virtual B, C { };
...
D* d = new D();
d->x = 42;
```

one-step-“smaller”-relation on subobjects (reflexive transitive closure $\subseteq$):

- repeated: smaller subobj. contains bigger one physically in the store
- shared: smaller subobj. has pointer to bigger one
Subobjects and Domination

- necessary due to multiple inherits of the same super class
- Subobject: entity with the fields of the resp. class
- accessed via class path

```cpp
class A { int x; }
class B { int x; }
class C : virtual A, virtual B { int x; }
class D : virtual A, virtual B, C { }
...
D* d = new D();
d->x = 42;
```

one-step-“smaller”-relation on subobjects (reflexive transitive closure \( \sqsubseteq \)):

- repeated: smaller subobj. contains bigger one physically in the store
- shared: smaller subobj. has pointer to bigger one

Domination: subobject “smaller” (w.r.t. \( \sqsubseteq \)) than all others
Subobject Formalization

Label within a class: subobject identified via class and path:
  types path = cname list
  types subobj = cname \times path

Object on the heap: path selects fields of the resp. subobject:
  types subo = path \times (var \rightarrow val)
  types obj = cname \times subo set

this-pointer: path denotes the subobject on which it points:
  types reference = addr \times path
  may be changed via explicit and implicit casts

⊑-Relation: compares path w.r.t. a class: \( P, C \vdash Cs \sqsubseteq Cs' \)
Dynamic Lookup

- collecting all subobjects (paths) of a class with method declaration:
  \((Cs, mthd) \in MethodDefs P C M\), where \(mthd\) body of \(M\) in subobj. \((C, Cs)\)

- resolve domination:
  \(P \vdash C\) has least \(M = mthd\) via \(Cs\) ⇔ \((Cs, mthd) \in MethodDefs P C M \land \forall (Cs', mthd') \in MethodDefs P C M. P, C \vdash Cs \sqsubseteq Cs')\)

Multiple Inheritance problem: ambiguities possible at runtime!

A code example

```plaintext
class Top { int f(); }
class Left : Top { }
class Right : Top { }
class Bottom : Left, Right { }
...
Left* l = new Bottom();
l->f();
```

statically everything ok

At runtime:

2 Top -subobjects (via Left and Right)

implicit cast of the `this`-pointer at call impossible!
Dynamic Lookup

- collecting all subobjects (paths) of a class with method declaration:
  \[(Cs, mthd) \in \text{MethodDefs } P \ C \ M, \text{ where } mthd \text{ body of } M \text{ in subobj. } (C, Cs)\]

- resolve domination:
  \[P \vdash C \text{ has least } M = mthd \text{ via } Cs \equiv (Cs, mthd) \in \text{MethodDefs } P \ C \ M \land \]
  \[(\forall (Cs’, mthd’) \in \text{MethodDefs } P \ C \ M. \ P, C \vdash Cs \sqsubseteq Cs’)\]

Multiple Inheritance problem: ambiguities possible at runtime!
Dynamic Lookup

- collecting all subobjects (paths) of a class with method declaration:
  \[(Cs,mthd) \in \text{MethodDefs } P \ C \ M, \text{ where } mthd \text{ body of } M \text{ in subobj. } (C,Cs)\]
- resolve domination:
  \[P \vdash C \text{ has least } M = mthd \text{ via } Cs \equiv (Cs,mthd) \in \text{MethodDefs } P \ C \ M \land \]
  \[(\forall (Cs’,mthd’) \in \text{MethodDefs } P \ C \ M. \ P,C \vdash Cs \sqsubseteq Cs’)\]

Multiple Inheritance problem: ambiguities possible at runtime!

A code example

```c
class Top { int f(); }
class Left : Top { }
class Right : Top { }
class Bottom: Left, Right { }
...
Left* l = New Bottom();
l->f();
```

statically everything ok
At runtime:
- 2 Top-subobjects (via Left and Right)
- implicit cast of the this-pointer at call impossible!
Dynamic Lookup

If lookup ambiguous at runtime, static information is used (as C++ does)

- collect minimal elements:
  \[
  \text{MinimalMethodDefs } P \ C \ M \equiv (Cs,mthd) \in \text{MethodDefs } P \ C \ M \land \\
  \left( \forall (Cs’,mthd’)) \in \text{MethodDefs } P \ C \ M. \ P,C \vdash Cs \subseteq Cs’ \rightarrow Cs = Cs’ \right)
  \]

- determine minimal subobjects smaller than static lookup subobject:
  \((Cs,mthd) \in \text{MethodDefs } P \ S \ M, \text{ where } S \text{ is the subobject of the caller}\)

- guarantee uniqueness of the minimal subobject:
  \[
  \vdash S \text{ has overrider } M = mthd \text{ via } Cs \equiv \\
  (Cs,mthd) \in \text{MethodDefs } P \ S \ M \land |\text{MethodDefs } P \ S \ M| = 1
  \]
Dynamic Lookup

If lookup ambiguous at runtime, **static information** is used (as C++ does)

- collect minimal elements:
  \[
  \text{MinimalMethodDefs } P \ C \ M \equiv (Cs,mthd) \in \text{MethodDefs } P \ C \ M \land \\
  (\forall (Cs’,mthd’) \in \text{MethodDefs } P \ C \ M. P,C \vdash Cs \subseteq Cs’ \rightarrow Cs = Cs’)
  \]

- determine minimal subobjects smaller than static lookup subobject:
  
  \[(Cs,mthd) \in \text{MethodDefs } P \ S \ M, \text{ where } S \text{ is the subobject of the caller}\]

- guarantee uniqueness of the minimal subobject:
  
  \[P \vdash S \text{ has overrider } M = mthd \text{ via } Cs \equiv \\
  (Cs,mthd) \in \text{MethodDefs } P \ S \ M \land |\text{MethodDefs } P \ S \ M| = 1\]

Real dynamic lookup: \(P \vdash (C,Cs) \text{ selects } M = mthd \text{ via } Cs’\)

- dyn. lookup unique: \(P \vdash C \text{ has least } M = mthd \text{ via } Cs\)

- dyn. lookup ambiguous: \(P \vdash (C,Cs) \text{ has overrider } M = mthd \text{ via } Cs’\)
Type Safety: Execution of a program statement $e$ of type $T$ in state $s$

- either fully evaluated value $v$ of type smaller than $T$
- or controlled exception

**Type Safety Theorem**

$\text{wf}_C\text{prog } P \quad P,E \vdash s \sqrt{\quad} P,E \vdash e :: T \quad \mathcal{D} e [\text{dom (lcl } s)]$

$P,E \vdash \langle e, s \rangle \rightarrow^{*} \langle e', s' \rangle \quad \not\exists e'', s''. P,E \vdash \langle e', s' \rangle \rightarrow \langle e'', s'' \rangle$

$(\exists v. e' = \text{Val } v \land P,hp \ s' \vdash v : \leq T) \lor$

$(\exists r. e' = \text{Throw } r \land \text{the_addr (Ref } r) \in \text{dom (hp } s'))$
Type Safety Proof

**Type Safety**: Execution of a program statement \( e \) of type \( T \) in state \( s \)
- either fully evaluated value \( v \) of type smaller than \( T \)
- or controlled exception

### Type Safety Theorem

\[
\begin{align*}
\text{wf}_C \text{ prog} & \quad P, E \vdash s \vdash \quad P, E \vdash e :: T \quad \mathcal{D} e \left[ \text{dom} \ (lcl \ s) \right] \\
\vdash P, E \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \quad \nexists \ e'' \ s''. \ P, E \vdash \langle e', s' \rangle \rightarrow \langle e'', s'' \rangle \quad \\
(\exists v. \ e' = \text{Val} \ v \land P, hp \ s' \vdash v : \leq T) \lor \\
(\exists r. \ e' = \text{Throw} \ r \land \text{the_addr} \ (\text{Ref} \ r) \in \text{dom} \ (hp \ s'))
\end{align*}
\]

**Standard proof technique:**

**Progress**: “the semantics cannot get stuck”
**Preservation**: “evaluating a well-typed statement results in another well-typed statement with smaller type”

Proof invariant formulated as run-time type system
CoreC++ Outline

- object-oriented core language with C++ multiple inheritance and exceptions, bases on Jinja
- big-step and small-step operational semantics with equivalence proof
- type system with compiler checks
- type safety proof of semantics w.r.t. type system
- semantics and type system executable, i.e., we have an interpreter for CoreC++ programs basing on the formal semantics
  a small tool translates simple C++ programs in CoreC++ programs

<table>
<thead>
<tr>
<th>Formalization Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoC</td>
</tr>
<tr>
<td>14,727</td>
</tr>
</tbody>
</table>
Proving Slicing Correct
Slicing bases on graphs
- graphs independent of underlying concrete program syntax
- Slicing itself reachability analysis
- hence, basic slicing algorithm is language independent

**Correctness of Slicing**
At slicing node, all used variables have same value, regardless if original or sliced program executed
Slicing

- Slicing bases on graphs
- graphs independent of underlying concrete program syntax
- Slicing itself reachability analysis
- hence, basic slicing algorithm is language independent

**Correctness of Slicing**
At slicing node, all used variables have same value, regardless if original or sliced program executed

**Goal:** correctness proof also language independent!
- language independent framework for slicing
- instantiantable with different (formal) language semantics
- ideal starting point: abstract control flow graph
Abstract Control Flow Graph

- defined in a context of function specifications and axioms
- language instantiations provide concrete function definitions and proofs that those fulfil axioms
Abstract Control Flow Graph

- defined in a context of function specifications and axioms
- language instantiations provide concrete function definitions and proofs that those fulfil axioms
- functions the instantiations have to provide:
Abstract Control Flow Graph

- defined in a context of function specifications and axioms
- language instantiations provide concrete function definitions and proofs that those fulfil axioms
- functions the instantiations have to provide:
  - valid edges of the graph
  - valid nodes are source and target nodes of valid edges
Abstract Control Flow Graph

- defined in a context of function specifications and axioms
- language instantiations provide concrete function definitions and proofs that those fulfil axioms
- functions the instantiations have to provide:
  - valid edges of the graph
    - valid nodes are source and target nodes of valid edges
  - semantic information of edges
    - two kinds, different effect when traversing this edge in a state
      - update edge: updates state
      - predicate edge: checks that predicate holds in state
Abstract Control Flow Graph

- defined in a context of function specifications and axioms
- language instantiations provide concrete function definitions and proofs that those fulfil axioms
- functions the instantiations have to provide:
  - valid edges of the graph
    - valid nodes are source and target nodes of valid edges
  - semantic information of edges
    - two kinds, different effect when traversing this edge in a state
      - update edge: updates state
      - predicate edge: checks that predicate holds in state
  - def and use sets of nodes
    - which variables are defined and used in a node (statement)
Abstract Control Flow Graph

- defined in a context of function specifications and axioms
- language instantiations provide concrete function definitions and proofs that those fulfil axioms
- functions the instantiations have to provide:
  - valid edges of the graph
    - valid nodes are source and target nodes of valid edges
  - semantic information of edges
    - two kinds, different effect when traversing this edge in a state
      - update edge: updates state
      - predicate edge: checks that predicate holds in state
  - def and use sets of nodes
    - which variables are defined and used in a node (statement)
- axiomatization of control flow graph properties
  - structural properties: e.g., no multi-edges
  - well-formedness properties: e.g., semantic effect and def/use agree
Program Dependence Graph

defined in proof context of abstract CFG

data dependence: “variable defined at one statement and used in a subsequent one, without being redefined in between”

\[ n \text{ influences } V \text{ in } n' \equiv \exists a' \text{ as}'. V \in \text{Def } n \land V \in \text{Use } n' \land n -a' \cdot \text{as'} \rightarrow^* n' \land (\forall n'' \in \text{set } (\text{srcs as'}). V \notin \text{Def } n'') \]

control dependence: “a statement controls whether another statement is executed” (e.g., if-branches or while-body)

needs postdominator: “every terminating execution at the parameter statement has to execute the postdominating statement”

\[ n' \text{ postdominates } n \equiv \text{valid_node } n \land \text{valid_node } n' \land (\forall as. \ n -as \rightarrow^* \text{Exit} \rightarrow n' \in \text{set } (\text{srcs as})) \]

\[ n \text{ controls } n' \equiv \exists a \ a' \text{ as}. n -a \cdot \text{as} \rightarrow^* n' \land n' \notin \text{set}(\text{srcs } (a' \cdot \text{as})) \land \text{valid_edge } a' \land \text{src } a = n \land n' \text{ postdominates } (\text{trg } a) \land \text{src } a' = n \land \neg n' \text{ postdominates } (\text{trg } a') \]
Slicing

- **Backward Slice**: $\rightarrow_{d*}$ reflexive transitive closure of control $\rightarrow_{cd}$ and $\rightarrow_{dd}$

  
  $$BS \ n_c \equiv \text{if valid_node} \ n_c \ \text{then} \ \{n' \mid n' \rightarrow_{d*} n_c\} \ \text{else} \ \emptyset$$

- **Sliced CFG**: not eliminating nodes, but invalidating semantic effects!
  
  if source node of an edge not in slice, no-op as semantic effect:
  
  - update with identity
  - predicates *True* or *False*

  hence, traversing edge no effect, as if it were not there

- **Program execution**: traversing control flow paths from *Entry* to *Exit*
  
  - in original CFG for executions in original program
  - in sliced CFG for executions in sliced program
Correctness Proof

Following Ranganath et al. [TOPLAS ’07] and Amtoft [IPL ’08]:
Weak Simulation Property between original and sliced CFG

- graphs as labelled transition systems (LTS)
  - LTS state: (node,state) tuple
  - LTS label: edges with source node in slice
  - LTS transition: silent and observable moves

\[
\begin{align*}
\text{src } a \notin BS n_c \quad &\Rightarrow \quad \text{valid_edge } a \quad \text{pred} \ (f a) \ s \\
\text{transfer} \ (f a) \ s = s' \\
\hline \\
\end{align*}
\]

\[
\begin{align*}
\text{src } a \in BS n_c \quad &\Rightarrow \quad \text{valid_edge } a \quad \text{pred} \ (f a) \ s \\
\text{transfer} \ (f a) \ s = s' \\
\hline \\
\end{align*}
\]

- Weak Simulation $\sim$ relation between (node,state) tuples
Correctness Proof

Following Ranganath et al. [TOPLAS ’07] and Amtoft [IPL ’08]:
**Weak Simulation Property** between original and sliced CFG

- graphs as labelled transition systems (LTS)
  - **LTS state:** (node,state) tuple
  - **LTS label:** edges with source node in slice
  - **LTS transition:** silent and observable moves

\[
\begin{align*}
\text{src} \ a \notin BS \ n_c \\
\text{valid_edge} \ a \quad \text{pred} \ (f \ a) \ s \\
\text{transfer} \ (f \ a) \ s = s' \\
n_c,f \vdash (\text{src} \ a,s) \xrightarrow{a} \tau \ (\text{trg} \ a,s')
\end{align*}
\]

\[
\begin{align*}
\text{src} \ a \in BS \ n_c \\
\text{valid_edge} \ a \quad \text{pred} \ (f \ a) \ s \\
\text{transfer} \ (f \ a) \ s = s' \\
n_c,f \vdash (\text{src} \ a,s) \xrightarrow{a} (\text{trg} \ a,s')
\end{align*}
\]

- **Weak Simulation** \(\sim\) relation between (node,state) tuples
- **Proof:** show that moves fulfil following simulation diagrams
Fundamental Property of Slicing

Correctness of Slicing
At slicing node, all used variables have same value, regardless if original or sliced program executed

weak simulation property says nothing about executions!
Fundamental Property of Slicing

Correctness of Slicing

At slicing node, all used variables have same value, regardless if original or sliced program executed.

weak simulation property says nothing about executions!

When we have a semantics which agrees to executing the CFG:

**Fundamental Property of Slicing**

\[
\begin{align*}
 n \triangleq c & \quad \langle c, s \rangle \Rightarrow \langle c', s' \rangle \\
\exists n' \text{ as. } n \xrightarrow{\text{-as}}^* n' & \land \text{preds (slice_kinds } n' \text{ as)} s \land n' \triangleq c' \land \\
(\forall V \in \text{Use } n'. \text{ state_val (transfers (slice_kinds } n' \text{ as}) s) V & = \text{state_val } s' \text{ V})
\end{align*}
\]

transfers (slice_kinds \( n' \) as) s:
execution of the sliced program of  \( n' \) in state  \( s \)
- new nodes for formal (in callee) and actual parameters (in caller)
- new edges (dotted) in dependence graph:
  - call edges for calling procedures and
  - parameter-in and -out edges for argument passing
- yet, simple reachability includes spurious nodes!
new nodes for *formal* (in callee) and *actual parameters* (in caller)
new edges (dotted) in dependence graph:
- call edges for calling procedures and
- parameter-in and -out edges for argument passing
yet, simple reachability includes spurious nodes!
context-sensitivity can eliminate such spurious nodes
Algorithm of Horwitz, Reps, Binkley (HRB)

standard for interprocedural context-sensitive slicing [TOPLAS ’90]

- 2 phases: first only ascends to callee, second only descends to callers
- context-sensitivity via summary edges (bold) efficient computable [Reps et al.: SIGSOFT ’94]
- but no correctness proof!
Summary Edges and HRB Slice

- in actual algorithm: complex algorithm $\mathcal{O}(n^3)$
- in formalization: simple declarative description

**Summary Edge**

If $m$ formal in-parameter and $m'$ formal out-parameter node, $m \rightarrow_{d*} m'$ and $n$ and $n'$ corresponding actual parameter nodes at call site, then $n \rightarrow_{sum} n'$

Formalizing the two phases of the HRB algorithm as sets:

$$\text{sum}_\text{SDG}_\text{slice1 } n = \{n'. n' \rightarrow \{\text{cd,dd,call,in,sum}\} \ast n\}$$

$$\text{sum}_\text{SDG}_\text{slice2 } n = \{n'. n' \rightarrow \{\text{cd,dd,out,sum}\} \ast n\}$$

HRB slice as combination of this two sets:

$$n' \in \text{sum}_\text{SDG}_\text{slice1 } n \Rightarrow n' \in \text{HRB}_\text{slice } n$$

$$n'' \in \text{sum}_\text{SDG}_\text{slice1 } n \Rightarrow n'' \text{ is actual out-parameter node}$$

$$n' \in \text{sum}_\text{SDG}_\text{slice2 } n'' \Rightarrow n' \in \text{HRB}_\text{slice } n$$
Summary Edges and HRB Slice

- in actual algorithm: complex algorithm $\mathcal{O}(n^3)$
- in formalization: simple declarative description

**Summary Edge**

If $m$ formal in-parameter and $m'$ formal out-parameter node, $m \rightarrow_d m'$ and $n$ and $n'$ corresponding actual parameter nodes at call site, then $n \rightarrow_{sum} n'$

Formalizing the two phases of the HRB algorithm as sets:

- $\text{sum}_\text{SDG_slice1 } n = \{n'. n' \rightarrow_{\{cd,dd,call,in,sum\}* n}\}$
- $\text{sum}_\text{SDG_slice2 } n = \{n'. n' \rightarrow_{\{cd,dd,out,sum\}* n}\}$
Summary Edges and HRB Slice

- in actual algorithm: complex algorithm $O(n^3)$
- in formalization: simple declarative description

**Summary Edge**

| If $m$ formal in-parameter and $m'$ formal out-parameter node, $m \rightarrow_d^* m'$ and $n$ and $n'$ corresponding actual parameter nodes at call site, then $n \rightarrow_{\text{sum}} n'$ |

Formalizing the two phases of the HRB algorithm as sets:

| $\text{sum}_\text{SDG_slice1} n$ = \{n’. n’ $\rightarrow$ \{cd,dd,call,in,sum\}* n\} |
| $\text{sum}_\text{SDG_slice2} n$ = \{n’. n’ $\rightarrow$ \{cd,dd,out,sum\}* n\} |

HRB slice as combination of this two sets:

| $n' \in \text{sum}_\text{SDG_slice1} n$ |
| $n' \in \text{HRB_slice} n$ |

| $n'' \in \text{sum}_\text{SDG_slice1} n$ |
| $n''$ is actual out-parameter node |

| $n' \in \text{sum}_\text{SDG_slice2} n''$ |

| $n' \in \text{HRB_slice} n$ |
Correctness Proof

- using the same **Weak Simulation Property**
- but: due to context-sensitivity we need **call history**
  - remembers call sites previously visited, but not returned to
  - we use a **node stack**
- LTS state now (node stack, state) tuple
- much more complicated definition of moves and simulation relation
Correctness Proof

- using the same **Weak Simulation Property**
- but: due to context-sensitivity we need **call history**
  - remembers call sites previously visited, but not returned to
  - we use a **node stack**
- **LTS state now** (node stack, state) tuple
- much more complicated definition of moves and simulation relation

But finally, same result as for intraprocedural slicing:

**Fundamental Property of Slicing**

\[
\begin{align*}
\exists n' \text{ as. } n \xrightarrow{\text{as}} n' & \land \text{preds (slice_kinds } n' \text{ as) } s \land n' \triangleq c' \land \\
\forall V \in \text{Use } n'. \text{ state_val (transfers (slice_kinds } n' \text{ as) } s) V = & \text{state_val } s' V)
\end{align*}
\]

But much more effort...
Instantiations

While: standard while language with procedures
- source code language
- complex CFG construction (label semantics)
- proving conditions mainly by inductive reasoning

Jinja byte code: quite sophisticated object-oriented language
- features exception throwing and catching
- fully object oriented
- but: no points-to analysis yet
  \[\rightarrow\text{far from precise ("heap as a whole")}\]
- byte code language
- "simple" CFG construction
- proving conditions mainly by reasoning by case distinction
Application to Information Flow Control

IFC: check if secret information may leak to public output
- variables partitioned in $H$ (secret) and $L$ (public)
- Low Equality $=_{L}$: two states agree in values of all $L$ variables
- Classical Noninterference: $\forall s s'. s =_{L} s' \rightarrow [c]s =_{L} [c]s'$
  differing values in initial $H$ variables no effect on final $L$ values
Application to Information Flow Control

IFC: check if secret information may leak to public output
- variables partitioned in \( H \) (secret) and \( L \) (public)
- Low Equality \( =_L \): two states agree in values of all \( L \) variables
- Classical Noninterference: \( \forall s \ s' . s =_L s' \implies [c]s =_L [c]s' \)
  differing values in initial \( H \) variables no effect on final \( L \) values

Proof that Slicing guarantees Classical Noninterference:
- enhance CFG by adding two nodes:
  - \( \text{High} \) immediately after \( \text{Entry} \), defines all \( H \) variables
  - \( \text{Low} \) immediately before \( \text{Exit} \), uses all \( L \) variables
- additional nodes also appear in Dependence Graph
- if \( \text{High} \notin BS \text{ Low} \), no influence from \( \text{High} \) to \( \text{Low} \)
Application to Information Flow Control

No influence from High to Low.

Slicing Guarantees Noninterference

\[ s_1 = L \]
\[ HRB_{\text{slice}} \text{Low initial } n \]
\[ \text{final } n' \]
\[ \text{c'} \]
\[ \langle c',s_1' \rangle \Rightarrow \langle c,s_2 \rangle \Rightarrow \langle c',s_2' \rangle \]

\[ s_1' = L \]

\[ L s_2' \]

Proof mainly by Correctness of Slicing
No influence from \textit{High} to \textit{Low}. Noninterferent?
No influence from High to Low. **Noninterferent!**

**Slicing Guarantees Noninterference**

\[
\begin{array}{c}
s_1 =_L s_2 \\
\text{High } \notin \text{ HRB_slice Low} \\
\text{initial } n \quad n \triangleq c \\
\text{final } n' \quad n' \triangleq c' \\
\langle c, s_1 \rangle \Rightarrow \langle c', s_1' \rangle \\
\langle c, s_2 \rangle \Rightarrow \langle c', s_2' \rangle \\
s_1' =_L s_2'
\end{array}
\]

Proof mainly by Correctness of Slicing
Slicing Outline

- language-independent framework for slicing via dependence graphs
- dynamic, static intra- and interprocedural slicing proved correct
- two instantiations:
  - a simple While source code language and
  - a sophisticated object-oriented byte code language
- first proof that slicing can guarantee classical noninterference

<table>
<thead>
<tr>
<th>Formalization Size Intraprocedural Slicing</th>
<th>LoC</th>
<th>Lemmas</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Framework</strong></td>
<td>6,872</td>
<td>209</td>
<td>43</td>
</tr>
<tr>
<td><strong>Instantiations</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>While</td>
<td>3,177</td>
<td>51</td>
<td>17</td>
</tr>
<tr>
<td>Jinja</td>
<td>5,517</td>
<td>100</td>
<td>27</td>
</tr>
<tr>
<td><strong>IFC Noninterference</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proof</td>
<td>558</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>CFG lifting</td>
<td>1,470</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>17,594</td>
<td>387</td>
<td>92</td>
</tr>
</tbody>
</table>
## Formalization Size Interprocedural Slicing

<table>
<thead>
<tr>
<th></th>
<th>LoC</th>
<th>Lemmas</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Framework</strong></td>
<td>18,988</td>
<td>579</td>
<td>104</td>
</tr>
<tr>
<td><strong>Instantiations</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(w/o semantics)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>While</td>
<td>6,758</td>
<td>127</td>
<td>29</td>
</tr>
<tr>
<td>Jinja</td>
<td>3,429</td>
<td>64</td>
<td>30</td>
</tr>
<tr>
<td><strong>IFC Noninterference</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proof</td>
<td>1,502</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>CFG lifting</td>
<td>2,025</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>32,702</td>
<td>798</td>
<td>175</td>
</tr>
</tbody>
</table>
Ongoing and future work

Points-to analysis:
- language: Jinja (byte code)
- bases on abstract dataflow framework [Kildall ’73] formalization
- Goals:
  1. verify Correctness (machine checked)
  2. improve precision of PDG formalization

IFC:
- formalization of
  - suitable noninterference definition (supporting I/O)
  - the PDG-based IFC algorithm [IJIS ’09]
    (without declassification)
- language independent
- bases on slicing framework
- Goal: verify Correctness of the algorithm
JinjaThreads
**Java features**
- classes, objects & fields
- inheritance & late binding
- exceptions
- imperative features

**not modelled**
- reflection & class loading
- interfaces
- threads
Verifying a Compiler for Java Threads

Java concurrency features:
- dynamic thread creation
- synchronisation
- wait / notify
- join & thread interruption

not modelled:
- java.util.concurrent
- Java Memory Model
Interleaving small-step semantics

\[ t \mapsto \langle x, h \rangle \xrightarrow{ta} \langle x', h' \rangle \]

single-thread semantics

\[ \text{NewThread} \ x \]
\[ \text{Lock} \ l / \text{Unlock} \ l \]
\[ \text{Wait} \ w / \text{notify} \ w / ... \]

\[ t \mapsto \langle (\text{addr} \ a).\text{start}(), h \rangle \xrightarrow{[\text{NewThread} \ \text{body}]} \langle \text{Unit}, h \rangle \]

multithreaded semantics

locks

thread-local states

wait sets

\[ \langle \sigma, h \rangle \xrightarrow{ta} \langle \sigma', h' \rangle \]

\[ \text{typeof}_h \ a = \text{Class} \ C \]
\[ P \vdash C \leq \text{Thread} \]
\[ P \vdash C \text{ sees } \text{run()} = \text{body} \]
Type safety

progress

\[
P \vdash (\sigma, h) \checkmark \quad \neg \text{final } \sigma
\]

\[
\exists t \ a \ \sigma' \ h'. \ (\sigma, h) \xrightarrow{t \ a} (\sigma', h')
\]

preservation

\[
P \vdash (\sigma, h) \checkmark \quad (\sigma, h) \xrightarrow{t \ a} (\sigma', h')
\]

\[
P \vdash (\sigma', h') \checkmark
\]
Type safety

progress

\[ P \vdash (\sigma, h) \checkmark \quad \neg \text{final } \sigma \]

\[ \exists t \; \tau \; \sigma' \; h'. \quad \sigma, h \xrightarrow{t} \sigma', h' \]

preservation

\[ P \vdash (\sigma, h) \checkmark \quad \sigma, h \xrightarrow{\tau} \sigma', h' \]

\[ P \vdash (\sigma', h') \checkmark \]

Generic preservation lemma

If single-thread semantics preserves prop. thread-locally,

\[ \Rightarrow \] multithreaded semantics preserves property globally.
Type safety

progress

\[ P(\sigma, h) \\not\!
\vdash \neg \text{final } \sigma \ (\sigma, h) \not\in \text{deadlock} \]

\[ \exists t \ ta \ \sigma' h'. \ \langle \sigma, h \rangle \xrightarrow{t} \langle \sigma', h' \rangle \]

preservation

\[ P(\sigma, h) \\not\!
\vdash \langle \sigma, h \rangle \xrightarrow{t \ ta} \langle \sigma', h' \rangle \]

\[ P(\sigma', h') \\not\!
\]

Deadlock

- all unfinished threads wait for
  - locks held by other threads
  - unfinished other threads
  - notification from wait set
- independent of concrete single-thread semantics
- coinductive characterisation

Generic preservation lemma

If single-thread semantics preserves prop. thread-locally,

\[ \Rightarrow \] multithreaded semantics preserves property globally.
Deadlock characterisation

\[ \text{thr } \sigma t = \lfloor x \rfloor \quad t \vdash \langle x, h \rangle \rightarrow \]
\[ \forall ta. \ t \vdash \langle x, h \rangle \overset{ta}{\rightarrow} \exists lt \in ta. \ \exists t' \in \text{deadlocked } (\sigma, h). \ \text{must-wait } \sigma t t' lt \]
\[ t \in \text{deadlocked } (\sigma, h) \]

\[ \sigma t = \lfloor x \rfloor \quad t \in \text{wait-sets } \sigma \quad \forall t \not\in \text{deadlocked } (\sigma, h). \ \text{final } (\sigma t) \]
\[ t \in \text{deadlocked } (\sigma, h) \]

\[ \text{deadlock} = \{ (\sigma, h) | \forall t. \ \text{final } (\sigma t) \lor t \in \text{deadlocked } (\sigma, h) \} \]
Compiler correctness

correctness statement

source code

result states
nontermination
deadlock

iff

compiled code

B

B

The correctness proof

source code

byte code

stage 1

intermed. language

stage 2

byte code
Compiler correctness

Correctness statement

source code \( \Downarrow \) compiled code
\[ \text{result states nontermination} \iff \text{iff} \]
\[ B \Downarrow B \]

Delay bisimulation \( \approx \)

source code \( \Downarrow \) stage 1
intermed. language \( \Downarrow \) stage 2
byte code
The correctness proof

\[ (\sigma_1, h) \approx (\sigma_2, h) \]

where \( \tau \) is a delay bisimulation.

\[ (\sigma'_1, h') \]

result states:
- nontermination
- deadlock

iff compiled code

source code

B

compiled code

B

Verifying a Compiler for Java Threads

IPD, programming paradigms group

23 March 2010
Compiler correctness

Correctness statement

Source code \rightarrow\downarrow B

Result states nontermination iff deadlock

Compiled code \rightarrow\downarrow B

Delay bisimulation \approx

(\sigma_1, h) \approx (\sigma_2, h)

\tau \downarrow \tau

(\sigma_1', h') \approx (\sigma_2', h')

38
Compiler correctness

correctness statement

source code \[\downarrow\]
result states
nontermination
iff
B

compiled code \[\downarrow\]
deadlock

\[B\]

\[\uparrow\]

delay bisimulation \[\approx\]

\[(σ_1, h) \approx (σ_2, h)\]

\[σ_1, h' \approx (σ'_2, h')\]

\[σ'_1, h'\]

\[\tau\]

\[σ'_1, h'\]

\[\tau\]

delay bisimulation \[\approx\]

\[(σ_1, h) \approx (σ_2, h)\]

\[(σ_1, h) \approx (σ_2, h)\]

\[σ'_1, h'\]

\[\ast\]

\[o\]

\[σ'_1, h'\]
The correctness proof uses delay bisimulation $\approx$ to relate source code $(\sigma_1, h)$ to compiled code $(\sigma_2, h)$. The states of the source code and compiled code must be compared under the operations $\tau$, $\Theta$, $\ast$, and $\Theta^*$ to ensure correctness.

Correctness statement:
- nontermination
- deadlock

Source code $\Rightarrow^B$ Compiled code $\Rightarrow^B$ if and only if:
- $(\sigma_1, h) \approx (\sigma_2, h)$
- $(\sigma_1', h') \approx (\sigma_2', h')$
Compiler correctness

**Observable steps**
- heap access
- synchronisation
- thread creation
- external method calls

**Correctness statement**

<table>
<thead>
<tr>
<th>delay bisimulation ( \approx )</th>
<th>((\sigma_1, h) \approx (\sigma_2, h))</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau \rightarrow \tau )**</td>
<td>( (\sigma_1', h') \approx (\sigma_2', h') )</td>
</tr>
<tr>
<td>( (\sigma_1, h) \approx (\sigma_2, h) )</td>
<td>( (\sigma_1', h') \approx (\sigma_2', h') )</td>
</tr>
</tbody>
</table>

**Define** \((\sigma_1, h) \approx (\sigma_2, h)\):
- locks and wait sets of \(\sigma_1\) and \(\sigma_2\) are the same
- thread-local states \(x_1\) and \(x_2\) satisfy: \((x_1, h) \approx_t (x_2, h)\)

**Stage diagram**:
- Source code \(\approx\)
- Stage 1 \(\approx_t\)
- Intermediate language \(\approx\)
- Stage 2 \(\approx_t\)
- Byte code
The correctness proof:

If \( \approx_t \) is a single-thread delay bisimulation, then \( \approx \) is a multithreaded delay bisimulation.

Define \((\sigma_1, h) \approx (\sigma_2, h)\):
- locks and wait sets of \(\sigma_1\) and \(\sigma_2\) are the same
- thread-local states \(x_1\) and \(x_2\) satisfy:
  \((x_1, h) \approx_t (x_2, h)\)

Observable steps:
- heap access
- synchronisation
- thread creation
- external method calls
### Statistics

<table>
<thead>
<tr>
<th>Formalisation</th>
<th>LoC</th>
<th>lemmas</th>
<th>definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>60,225</td>
<td>2812</td>
<td>463</td>
<td></td>
</tr>
</tbody>
</table>

⇒ 3 times the size of Jinja
## Statistics

<table>
<thead>
<tr>
<th>LoC</th>
<th>lemmas</th>
<th>definitions</th>
<th>Build times:</th>
</tr>
</thead>
<tbody>
<tr>
<td>60,225</td>
<td>2812</td>
<td>463</td>
<td>4GB, 1×2.6GHz, x86: 3:30h</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>40GB, 1×2.53GHz, x86_64: 1:30h</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>40GB, 8×2.53GHz, x86_64: 0:30h</td>
</tr>
</tbody>
</table>

⇒ 3 times the size of Jinja
## Statistics

### Formalisation

<table>
<thead>
<tr>
<th>LoC</th>
<th>lemmas</th>
<th>definitions</th>
<th>Build times:</th>
</tr>
</thead>
<tbody>
<tr>
<td>60,225</td>
<td>2812</td>
<td>463</td>
<td>4GB, 1×2.6GHz, x86: 3:30h</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>40GB, 1×2.53GHz, x86_64: 1:30h</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>40GB, 8×2.53GHz, x86_64: 0:30h</td>
</tr>
</tbody>
</table>

⇒ 3 times the size of Jinja

### Essential Isabelle features:

- Isar
- locales as a module system
- (co-)inductive definitions and proofs by (co-)induction
### Statistics

#### Formalisation

<table>
<thead>
<tr>
<th>LoC</th>
<th>lemmas</th>
<th>definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>60,225</td>
<td>2812</td>
<td>463</td>
</tr>
</tbody>
</table>

⇒ 3 times the size of Jinja

**Build times:**

- 4GB, 1×2.6GHz, x86: 3:30h
- 40GB, 1×2.53GHz, x86_64: 1:30h
- 40GB, 8×2.53GHz, x86_64: 0:30h

---

**Essential Isabelle features:**

- Isar
- locales as a module system
- (co-)inductive definitions and proofs by (co-)induction

**JinjaThreads hits the limits**

- locales and parallelisation devour lots of memory
- very little support for refactoring
JinjaThreads summary

- formal small-step semantics for multithreaded Java source code and byte code
- type system and type safety proof
- verified compiler from source code to byte code
- available in the Archive of Formal Proofs
  http://afp.sourceforge.net/entries/JinjaThreads.shtml
JinjaThreads summary

- formal small-step semantics for multithreaded Java source code and byte code
- type system and type safety proof
- verified compiler from source code to byte code
- available in the Archive of Formal Proofs
  http://afp.sourceforge.net/entries/JinjaThreads.shtml

Current and future work:

- Java Memory Model
- extract executable Java interpreter
Conclusion
Ongoing work in Quis Custodiet

- Isabelle proof for full algorithm from [IJIS ’09] incl. points-to, threads requires generalized noninterference (cmp. [Askarov ’08]) proof will require $> 100000$ LOC Isabelle text
- extend compiler formalization/proof with memory model
Ongoing work in Quis Custodiet

- Isabelle proof for full algorithm from [IJIS ’09] incl. points-to, threads requires generalized noninterference (cmp. [Askarov ’08]) proof will require $> 100000$ LOC Isabelle text
- extend compiler formalization/proof with memory model
- automatically generate an executable, completely machine-verified, PDG-based IFC tool (?)
Ongoing work in Quis Custodiet

- Isabelle proof for full algorithm from [IJIS ’09] incl. points-to, threads requires generalized noninterference (cmp. [Askarov ’08]) proof will require > 100000 LOC Isabelle text
- extend compiler formalization/proof with memory model
- automatically generate an executable, completely machine-verified, PDG-based IFC tool (?!)

*Quis Custodiet Ipsos Custodes?*
Ongoing work in Quis Custodiet

- Isabelle proof for full algorithm from [IJIS ’09] incl. points-to, threads requires generalized noninterference (cmp. [Askarov ’08]) proof will require > 100000 LOC Isabelle text
- extend compiler formalization/proof with memory model
- automatically generate an executable, completely machine-verified, PDG-based IFC tool (?!)

Quis Custodiet Ipsos Custodes?
Isabelle!