Machine Checked Conditional Information Flow

Joey Dodds, Andrew Appel (Advisor)

Collaborators: Lennart Beringer (Princeton), Torben Amtoft, Zhi Zhang, John Hatcliff, Simon Ou, Andrew Cousino (Kansas State University)

To appear in POST 2012
Outline

1. What is Information flow?
2. SPARK programming language
3. Agreement Logic
4. SIFL Static Analyzer
5. Evidence Formalization
6. Soundness Proof
Outline

1. What is Information flow?
2. SPARK programming language
3. Agreement Logic
4. SIFL Static Analyzer
5. Evidence Formalization
6. Soundness Proof
Information Flow

Controlling where confidential data flows

Health records: My doctors can have them, release parts to specialists, everything invisible to public

More than access control, access control is too restrictive

Sabelfeld, Myers [7]
Information Flow

Information Flow is a system-wide property

Controlling where information flows in a program is a part of the end-to-end picture (this talk’s part)

Our goal is to allow for more expressive policies

Sabelfeld, Myers [7]
Analyzing information flow

Early attempts at analysis were dynamic methods like access control

Many dynamic methods fail to capture implicit flows, but static analysis excels at it

```plaintext
low := 0;
if hi=1 then low := 1;
else skip;
```
Old

SPARK Code with annotations

Type-Based SPARK analyzer

Paper Proof

YES  NO
Newer

SPARK Code with annotations

SIFL Static analyzer

Paper Proof

YES  NO
SPARK Code with annotations

SIFL Static Analysis

Outputs

Coq Evidence for Condition

Instance Of

Coq Evidence Type

Proven sound Wrt.

SPARK Operational Semantics

= My Contribution

Newest
Outline

1. What is Information flow?
2. SPARK programming language
3. Agreement Logic
4. SIFL Static Analyzer
5. Evidence Formalization
6. Soundness Proof
SPARK Annotations
SPARK annotations allow developers to specify information flow

Procedure ex (hi : in out integer; low : in out integer)
--# derives hi from *
--# derives low from low
begin
  hi := low + hi;
  low := low+1;
end

This passes the SPARK checker so we know hi doesn't flow into low
SPARK

No pointers or heap-based data

Uses arrays and for loops for complex data structures

Allows automatically checked specifications for information flow using a type based system (Chapman, Hilton [1])

Useful for development in the MILS architecture (Boettcher et al. [8])
Limitations of the SPARK checker

A checker for information flow currently ships with spark

SPARK Checker and annotations don't allow for conditions in flows

SPARK checker is imprecise when arrays are involved
Industrial Interest

Rockwell Collins has requested a way to make more precise specifications.

AdaCore is very interested in using the more precise analysis I will be presenting for the rest of this talk.
Limitation Examples

Conditional

if secure then
  hi:=low
else
  low:=hi
end

It would be useful to state that high only flows into low when secure=false
Limitation Examples

Using an array `pair` of size 2 to represent a pair.

```plaintext
low := pair[0]
hi := pair[1]
```

In this example, it would be good to say `pair[0]` flows into `low` and `pair[1]` flows into `hi`.

We can only say that `pair` flows into both
Outline

1. What is Information flow?
2. SPARK programming language
3. Agreement Logic
4. SIFL Static Analyzer
5. Evidence Formalization
6. Soundness Proof
The Agreement Operator

We use assertions containing the agreement operator \( \# (\bowtie) \) (Amtoft, Banerjee [2])

\( E\# \) is true iff \( E \) evaluates to the same value in two states

\[ \text{In } S = [x=1, y=3] \quad S_1 = [x=3, y=3] \]

\( y\# \) holds but \( x\# \) does not
Agreement in Hoare Triples

We can use the agreement operator in Hoare Triples to specify information flow properties

```plaintext
{low#}
high := low
low := low + 1
{low#}
```

We can see that no more variables than low will flow into low.
2-Assertions

We extend the agreement with boolean conditions

\{\text{secure}, \text{secure} \Rightarrow \text{low}, \text{!secure} \Rightarrow \text{hi}\}

if secure then
  hi := low
else
  low := hi
end

{low}
2-Assertion

A 2-Assertion $\Phi \Rightarrow E#$
holds iff for all states $s$ and $s_1$

$s \not\models \Phi$ and $s_1 \not\models \Phi$ then $[[E]]_s = [[E]]_{s_1}$

$\Phi$ evaluates to true in $s$ and $s_1$

and

$E$ evaluates to the same value in $s$ and $s_1$

or

$\Phi$ does not evaluate to true in $s$ or $s_1$
2-Assertion Example

s = [x = true, y = false,  z = 3, w = 4]

s₁ = [x = true, y = true, z = 3, w = 5]

over states s and s₁

x ⇒ z#  holds

y ⇒ w#  holds

x ⇒ y#  does not hold
Syntactic Rules

\[ \text{SkipE}(\Theta) : \{\Theta\} \text{ skip } \{\Theta\} \]

\[ \text{SeqE}(\eta_1, \eta_2) : \{\Theta_1\} C_1 ; C_2 \{\Theta_2\} \]

\[ \text{AssignE}(\Theta, x, A) : \{\Theta[A/x]\} \ x := A \{\Theta\} \]

\[ \text{HAssignE}(\Theta, h, H) : \{\Theta[H/h]\} \ h := H \{\Theta\} \]

\[ \text{AssertE}(\Theta, B) : \{\text{add}^\wedge_B(\Theta)\} \text{ assert}(B) \{\Theta\} \]

\[ \Theta = \{\phi \Rightarrow E \times\} \quad \vdash \eta_i : \{\Theta_i\} C_i \{\Theta\} \quad (i = 1, 2) \quad \vdash \nu : \phi_0 \xleftarrow{C} \phi \]

\[ \text{CondE}(\eta_1, \eta_2, \nu, B) : \{\text{add}^\wedge_B(\Theta_1) \cup \text{add}^\wedge_B(\Theta_2) \cup \{\phi_0 \Rightarrow B \times\}\} \ C \{\Theta\} \]

Examples soon!
Outline

1. What is Information flow?
2. SPARK programming language
3. Agreement Logic
4. SIFL Static Analyzer
5. Evidence
SPARK Code with annotations

SIFL Static analyzer

YES  NO
SIFL Precondition Generator

Generates a precondition given a postcondition and a command

Also generates evidence the precondition is sound

Example of proofs in proof carrying code (Necula [6])
Form of evidence

Evidence is a text file containing a Coq definition

If the form of evidence is proven sound, evidence represents a complete proof of correctness for the generated precondition
Notation

\{a_1, a_2, a_3 \ldots a_n\} is equivalent to the conjunction of a_1 - a_n

an assertion with no antecedent (x#) is equivalent to true \Rightarrow x#
The Language

\[ C ::= \text{skip} | \ C ; \ C | \text{assert}(B) | \ x := A | \ h := H \]
\[ \quad | \quad \text{while } B \text{ do } C \text{ od} | \text{for } q \gets 1 \text{ to } m \text{ do } C \]

\[ A ::= c | x | A \text{ op } A | H[A] \]
\[ B ::= A \text{ bop } A | \text{true} | \text{false} | B \land B | B \lor B | \neg B \]
\[ H ::= h | Z | H\{A : A\} \]
Evidence

Previous work presented an algorithm to generate preconditions given a program and a postcondition.

Running the algorithm should generate a precondition and evidence for the triple.

If the evidence is defined correctly, the precondition must be correct.
Form of Evidence

Primary goal is evidence whose types correspond to triples

Where $\eta$ is evidence

Assume that $\vdash \eta : \{\Theta\} C \{\Theta'\}$. Then $\models \{\Theta\} C \{\Theta'\}$.

for all $s$ and $s_1$ if $\theta$ holds on both and $s'$ and $s_1'$ are the states transformed by $C$ then $\theta'$ holds on $s'$ and $s_1'$
Assignment

3 Assign rules, one for each type.

\[ \vdash \text{AssignE}(\Theta, x, A) : \{\Theta[A/x]\} \ x := A \ \{\Theta\} \]

Substitution in 2-assertion is standard substitution on each part of the assertion

\( \{ y\# \} \)

\( \ x := y \)

\( \{ x\# \} \)
Don’t panic!

Here is the rule but things are clearer if we work through an example!
Conditional Example

\{\text{secure#, secure } \Rightarrow \text{ low#}, \text{ !secure } \Rightarrow \text{ hi#}\}

\text{if secure then}
  \\{\text{low#}\}
  \text{hi} \ := \ \text{low}
\text{else}
  \{\text{hi#}\}
  \text{low} \ := \ \text{hi}
\text{fi}
\{\text{low#}\}
While Loops

Rule for while loops requires a loop invariant, 
true can always be used for a less precise analysis

Evidence for while loop requires calls to a SMT solver to verify

We lose information about array indices if they are accessed in while loops
For Loops

For certain for loops we can use an interesting rule

This rule generates a polymorphic triple

Polymorphic triples contain variables $u$ that can be replaced with constants
For Loop Example

for \( q \leftarrow 1 \) to \( m \) do
    \( t := h[q] \);
    \( h[q] := h[q+m] \);
    \( h[q+m] := t \);
od

For the analysis we need
- Linear array accesses
- No loop-carried dependencies

Otherwise we do analysis as a \texttt{while} loop, losing information about indices
For Loop Example

{m#, (u < 1 || u > 2m) ⇒ h[u]#,
1 <= u <= m ⇒ h[u + m]#, 
m + 1 <= u <= 2m ⇒ h[u - m]#}
for q <- 1 to m do
  t := h[q];
  h[q] := h[q+m];
  h[q+m] := t;
od
{h[u]#}
For Loop Example

$$\{m\#, \ (u < 1 \lor u > 2m) \Rightarrow h[u]\#, \ 1 \leq u \leq m \Rightarrow h[u+m]\#, \ m + 1 \leq u \leq 2m \Rightarrow h[u - m]\#\}$$

for q <- 1 to m do
  t := h[q];
  h[q] := h[q+m];
  h[q+m] := t;
od

$$\{h[u]\#\}$$
For Loop Example

\{m\#, (u < 1 \text{ or } u > 2m) \Rightarrow h[u]\#, \\
1 \leq u \leq m \Rightarrow h[u + m]\#, \\
m + 1 \leq u \leq 2m \Rightarrow h[u - m]\#\}

for q <- 1 to m do
    t := h[q];
    h[q] := h[q+m];
    h[q+m] := t;
od

\{h[u]\#\}
For Loop Example

\{m\#, (u < 1 \text{ or} u > 2m) \Rightarrow h[u]\#,  \\
1 \leq u \leq m \Rightarrow h[u + m]\#,  \\
m + 1 \leq u \leq 2m \Rightarrow h[u - m]\#\}

\text{for } q \leftarrow 1 \text{ to } m \text{ do}
  \begin{align*}
    &t := h[q]; \\
    &h[q] := h[q+m]; \\
    &h[q+m] := t;
  \end{align*}
\text{od}

\{h[u]\#\}
Outline

1. What is Information flow?
2. SPARK programming language
3. Agreement Logic
4. SIFL Static Analyzer
5. Evidence Formalization
   1. Justification for Formalization
6. Soundness proof
SPARK Code with annotations

SIFL Static Analysis

Outputs

Coq Evidence for Condition

Instance Of

Coq Evidence Type

Proven sound Wrt.

SPARK Operational Semantics

= My Contribution

My Contribution
Coq Evidence Representation

We represent evidence as a Coq inductive datatype.

We prove the evidence sound wrt. operational semantics.

Evidence that type checks guarantees a correct triple corresponding to the type of evidence.
Coq Language Representation

Language is defined in coq as three types of expressions \( \text{AExpr}, \text{BExpr}, \text{and} \ \text{HEExpr} \) grouped together as \( \text{Expr} \).

Commands are also declared as expected from the language definition.

Operational Semantics and Expression evaluation are also defined and equivalent to the language defined.
2-Assertion Representation

We represent two assertions as a pair

Definition TwoAssn := prod BExpr Expr.

A list of TwoAssn is called TwoAssns

Duplicates aren't a correctness issue as they can easily be removed
2-Assertion Representation

We define pre/postconditions as

```
Inductive assns :=
  | Assns : TwoAssns -> TwoAssns -> assns
  | APoly : (AExpr -> assns)-> assns.
```

Assns is a standard assertion pair

APoly is a pair parameterized by a shared variable, like conditions generated by for loop
Evidence Representation

Evidence is an inductive proposition

\textbf{Inductive} \ TEvid (X: list SkalVar) :

\begin{itemize}
  \item Command -> assns -> Prop :=
  \item TSkipE ...
  \item TAAssignE ...
  \item TCondE ...
\end{itemize}

...
Outline

1. What is Information flow?
2. SPARK programming language
3. Agreement Logic
4. SIFL Static Analyzer
5. Evidence Formalization
   1. Justification for Formalization
6. Soundness proof
A justification for formalization

If we already have a paper proof, why add a Coq proof?

When we formalize, we must make all assumptions explicit

Notice the $X$ parameter of evidence

**Inductive** TEvid ($X$: list Var) :
Command $\rightarrow$ assns $\rightarrow$ Prop :=
Conditional Problem

(B && npc(post, c1)) || (!B && npc(post, c2))

We get to a point in the soundness proof where we know both B and npc(post, c1) evaluate to true
Conditional Problem

true || (!B && npc(post, c2))

The paper proof assumes this simplifies to true, which seems reasonable.

These are program expressions, not logic expressions!

Could evaluate to None
Conditional Solution!

The only time an expression evaluates to None is when a lookup fails.

Our soundness property only holds on states that have certain variables in their domain.

We use the parameter $X$ to hold a superset of these variables.
What does that mean?

The paper proof doesn't worry about this because it knows it isn't a problem in SPARK.

What if we wanted to use this analysis for a different language?

Formalization ensures that any assumptions we make are explicit.
Assignment rule

| TAAssignE : forall Theta x A,
  TEvid X (Assign x (AExp A))
  (Assns(TwoAssnsSubstA Theta x A) Theta)

exactly matches

\[ \Gamma \vdash AssignE(\Theta, x, A) : \{\Theta[A/x]\} \ x := A \ \{\Theta\} \]
Conditional Rule

| TCondE: forall {Theta1 phi E Theta2 Theta' C1 C2 phi0
(eta1: TEvid X C1 (Assns Theta1 [(phi, E)]))
(eta2: TEvid X C2 (Assns Theta2 [(phi, E)])) B
(nu : NPCEvid X phi0 (Cond B C1 C2) phi),
andIntoTheta Theta1 B ++ andIntoTheta Theta2 (NotExp B)
  ++ [(phi0, BExp B)] = Theta' ->
allVarsIn (BFv phi0) X =true ->
TEvid X (Cond B C1 C2) (Assns Theta' [(phi, E)])

\[
\begin{align*}
\Theta &= \{ \phi \Rightarrow E \times \} \\
\vdash \eta_i : \{ \Theta_i \} C_i \{ \Theta \} (i = 1, 2) \\
\vdash \nu : \phi_0 \leftrightarrow \phi \\
\vdash \text{CondE}(\eta_1, \eta_2, \nu, B) : \{ \text{add}_B^\land(\Theta_1) \cup \text{add}_B^\lor(\Theta_2) \cup \{ \phi_0 \Rightarrow B \times \} \} C \{ \Theta \}
\end{align*}
\]
Outline

1. What is Information flow?
2. SPARK programming language
3. Agreement Logic
4. SIFL Static Analyzer
5. Evidence Formalization
   1. Justification for Formalization
6. Soundness Proof
**Soundness**

Inductive twoSatisfies(s1 s2:State): TwoAssn -> Prop :=
|TwoSatisfies: forall phi E, BEval phi s1 = Some true ->
 BEval phi s2 = Some true ->
 (Eval E s1 = Eval E s2) ->
twoSatisfies s1 s2 (phi, E).

Definition twoAssnsInterpretation (s1 s2: State)
(a:TwoAssns):= Forall (twoSatisfies s1 s2) a.

Fixpoint validHoareTriple (X: list SkalVar) C (asns: assns):=
(forall s s' t t', lookupsComplete X s s' ->
 (twoAssnsInterpretation s s' pre) ->
 Opsem s C t -> Opsem s' C t' ->
 (twoAssnsInterpretation t t' post))
Soundness

Definition soundness :=
forall X C assns,
TEvid X C assns ->
validHoardTriple X C assns

I have completed this proof for all but loops
Code count

Soundness proof is 2300 lines of coq code

Only 590 lines of trusted Coq code (code that needs to be human verified)
  Language definition
  Operational semantics
  Soundness definition

Other trusted code is SMT Solver
Precondition Generator

Evidence can be large

We can generate precondition in Coq and prove the generator sound by reflection

We prove that evidence exists for all generated conditions

Generator can be extracted to ML so it runs quickly
Precondition Generator

Fixpoint generatePrecondition
  (c:Command) (post:TwoAssns)
  (X: list SkalVar):
  (TwoAssns * list SkalVar) := ...

Theorem generatePreconditionEvidence:
  forall C Y X Z post pre,
  (pre, X)=generatePrecondition C post Z ->
  allVarsIn X Y = true->
  TEvid Y C (Assns pre post).
Current System

SPARK Code with annotations

SIFL Precondition Generator (Java)

Outputs

Coq Evidence for Precondition

Instance Of

Coq Evidence Type

Proven sound Wrt.

SPARK Operational Semantics

= My Contribution
System with Coq Static Analysis

SPARK Code with annotations

SIFL Precondition Generator (Coq)

Proven to have some valid instance of

Coq Evidence Type

Proven sound Wrt.

SPARK Operational Semantics

= My Contribution

My Contribution
Future Work

Soundness proofs for loops

Implications by decision procedure are currently trusted. Techniques such as Armand et al. [3] may help

Coq Precondition Generator with loops

Connection to CompCert [4] for an end-to-end guarantee of information flow
Questions
Sources


**Conclusion**

SIFL analysis improves the precision with which we can make information flow specifications.

SIFL analysis generates evidence.

I have defined and proven the evidence generated sound in Coq.

AdaCore is very interested in.
Arrays in the Language

\[ H ::= h \mid Z \mid H\{A : A\} \]

- total mapping
- 0 except for in a finite number of places where it has been updated
- \( Z \) is the array that is 0 for all indices
- Safe assumptions because SPARK examiner checks array bounds