Secure Information Flow Analysis using KeY

Jing Pan[1]   Philipp Rümmer[2]

[1]jinpan@mdstud.chalmers.se

[2]philipp@cs.chalmers.se

4th KeY Symposium, 2005

Secure Information Flow Overview

Definition

The attacker cannot learn about initial value of high variable h from final value of low variable `l`. (*confidentiality*)

Example

program	*secure?*
`h=l;`	Yes
`l=6;`	Yes
`l=h; l=l-h;`	Yes (though insecure parts)
`l=h;`	No (direct flow)
`if(h>=0) l=1; else l=0;`	No (indirect flow)

Equivalent A variation of high input does not cause a variation of low output. (*non-interference*)

Previous Work in KeY

"a variation of high input does not cause a variation of low output"

$$\forall \mathtt{l}, \mathtt{l}'.\exists \mathtt{h}.\forall \mathtt{h}'.(\mathtt{l} \doteq \mathtt{l}' \rightarrow \langle\{\mathtt{p}(\mathtt{l}, \mathtt{h}); \mathtt{p}(\mathtt{l}', \mathtt{h}')\}\rangle \mathtt{l} \doteq \mathtt{l}')$$

Problems:

- doubled program size
- instances of h might need other variables

New approach:

- Do not double the program but need to record final state of $\mathtt{l}$.
- Symbolically execute program until the final value for $\mathtt{l}$ is obvious.
- Using the idea from previous approach, derive a second proof obligation from information of open goals .

Formalizing Secure Information Flow in KeY

- Definition in .key

```
sorts {
  TermList;
}

functions { //a list for arbitrary terms
  TermList nil;
  Termlist cons(TermList, any);
}

predicates { //to hold final value of low variables
  secure(TermList);
}
```

- Proof obligation

$$\langle\{\alpha\}\rangle\ \mathtt{secure}(L)$$

where L is the complete TermList of *prog vars* that are supposed to be low

Verifying Secure Information Flow in KeY I

- If α terminates properly (not abruptly), after symbolically executing program or applying induction rules correctly on loops we get open goals:

$$\Gamma_0 \texttt{==>} \Delta_0, \texttt{secure}(\text{cons}\ldots(\text{cons}(\text{cons}(\text{nil}, t_{00}), t_{01}), \ldots t_{0m}))$$
$$\ldots \qquad\qquad \ldots$$
$$\Gamma_i \texttt{==>} \Delta_i, \texttt{secure}(\text{cons}\ldots(\text{cons}(\text{cons}(\text{nil}, t_{i0}), t_{i1}), \ldots t_{im}))$$
$$\ldots \qquad\qquad \ldots$$
$$\Gamma_n \texttt{==>} \Delta_n, \texttt{secure}(\text{cons}\ldots(\text{cons}(\text{cons}(\text{nil}, t_{n0}), t_{n1}), \ldots t_{nm}))$$

- Each column represents the final values for a low variable.

$$fin(l_j) = \begin{cases} t_{0j} & iff \quad \Theta_0 \\ \ldots & \ldots \\ t_{ij} & iff \quad \Theta_i \\ \ldots & \ldots \\ t_{nj} & iff \quad \Theta_n \end{cases}$$

where $j \in \{0..m\}$ $\quad fin(v) \stackrel{def}{=}$ final value of v $\quad \Theta_i \stackrel{def}{=} (\bigwedge \Gamma_i) \wedge \neg(\bigvee \Delta_i)$

Verifying Secure Information Flow in KeY II

- Derive a function from open goals for each l_j. KeY uses *conditional term* "`if` (ϵ) (t_{then}) (t_{else})"

$$
\begin{aligned}
fin(l_j) \quad = \quad & \text{if } (\Theta_0)\ (t_{0j})\ (\\
& \quad \ldots \\
& \qquad \text{if } (\Theta_i)\ (t_{ij})\ (\\
& \qquad\quad \ldots \\
& \qquad\qquad \text{if } (\Theta_n)\ (t_{nj})\ (defaultTerm) \ldots\ ) \ldots)
\end{aligned}
$$

defaultTerm can be any term; may use t_{nj} to make proof simpler.

- *"a variation of high input does not cause a variation of low output"*

$$\forall \bar{H}.\ \forall \bar{H}'.\ \bigwedge_{0 \le j \le m} (fin(l_j) = fin(l'_j))$$

$$\bar{H} \stackrel{def}{=} h_1, h_2, \ldots, h_k \qquad \bar{H}' \stackrel{def}{=} h'_1, h'_2, \ldots, h'_k \qquad fin(l'_j) \stackrel{def}{=} fin(l_j)[\bar{H}/\bar{H}']$$

Example(1) I

program "`if (h>0) l=1; else l=0;`" (`l` – low, `h` – high)

- Create `.key` file:

```
sorts {
  TermList;
}

functions {
  TermList nil;
  Termlist cons(TermList, any);
}

predicates {
  secure(TermList);
}

program {
  <{if (h>0) l=1; else l=0;}> secure(cons(nil, l))
}
```

Example(1) II

- After symbolically run proof in KeY it stops with two open goals:

```
==> 0 < h, secure(cons(nil, 0))
0 < h ==> secure(cons(nil, 1))
```

- To continue proof
  1. Click button "Extract security proof" in Toolbar.
  2. Select the variables that are supposed to be secret (In this case, h).
- A new proof is generated:

```
==>
 all h1:int. all h2:int.
      ({h:=h1} if (!0 <  h) (0)
                     (if (0 <  h) (1) (1)))
    = ({h:=h2} if (!0 <  h) (0)
                     (if (0 <  h) (1) (1)))
```

Example(1) III

- Run prover, proof stops with two open goals:

```
0 <  h2 ==> 0 <  h1
0 <  h1 ==> 0 <  h2
```

- Conclusion: *program* ”`if (h>0) l=1; else l=0;`” leaks information of the sign of high variable h

Abstracting Programs

Motivation: Sometime computation that program performs is not really interesting for us, such as in Secure Information Flow Analysis study.

Example

$$h = l++; \tag{1}$$

$$h = x/y; \tag{2}$$

`l, x, y` — low, `h` — high

Idea: Remove unnecessary knowledge about program state (values of variables). (*Abstraction*)
eg. Example (1)

Abstraction of Programs in KeY

KeY translates a piece of program into simultaneous update

$$\nu = \{l_1 := t_1, \ldots, l_n := t_n\}$$

to describe states of variables $l_1, ..$ $, ...l_n$.

- Carry out from update?
  Var state is clear but unnecessary work in computing results . Not good.
- Carry out from program?
  in only one step. Sounds right!

Abstraction of Programs in KeY

KeY translates a piece of program into simultaneous update

$$\nu = \{l_1 := t_1, \ldots, l_n := t_n\}$$

to describe states of variables $l_1, ..$ $, ...l_n$.

- Carry out from update?
  Var state is clear but unnecessary work in computing results . Not good.
- Carry out from program?
  in only one step. Sounds right!

Abstraction of Programs in KeY

KeY translates a piece of program into simultaneous update

$$\nu = \{l_1 := t_1, \ldots, l_n := t_n\}$$

to describe states of variables $l_1, ..\quad, ...l_n$.

- Carry out from update?
  Var state is clear but unnecessary work in computing results . Not good.
- Carry out from program?
  in only one step. Sounds right!

Rule For Abstraction – Attempt 1

- Example 1: What KeY does

$$
\begin{aligned}
\langle\{h = l{+}{+};\}\rangle\,\Phi &\quad \leadsto \\
\langle\{h = l;\ l = (int)(l+1);\}\rangle\,\Phi &\quad \leadsto \\
\dots &\quad \leadsto \\
\{h := l,\ l := l+1\}\ \langle\{\}\rangle\,\Phi &
\end{aligned}
$$

- Wanted
  - locations whose states may change after execution of program
  - new state of those locations
- Abstracting program

$$
\begin{aligned}
\langle\{h = l{+}{+};\}\rangle\,\Phi &\quad \leadsto \\
\{h := l,\ l := f(l)\}\ \langle\{\}\rangle\,\Phi &
\end{aligned}
$$

- A rule can be

$$
\frac{\bar{\nu}\ \langle\{..\ ...\}\rangle\ \Phi}{\langle\{..\ \alpha\ ...\}\rangle\ \Phi}
$$

Rule For Abstraction – Attempt 1

- Example 1: What KeY does

$$\langle\{h = l{+}{+};\}\rangle\,\Phi \quad \rightsquigarrow$$
$$\langle\{h = l;\ l = (int)(l+1);\}\rangle\,\Phi \quad \rightsquigarrow$$
$$\ldots \quad \rightsquigarrow$$
$$\{h := l,\ l := l+1\}\,\langle\{\}\rangle\,\Phi$$

- Wanted
  - locations whose states may change after execution of program
  - new state of those locations
- Abstracting program

$$\langle\{h = l{+}{+};\}\rangle\,\Phi \quad \rightsquigarrow$$
$$\{h := l,\ l := f(l)\}\,\langle\{\}\rangle\,\Phi$$

- A rule can be

$$\frac{\bar{\nu}\,\langle\{\ldots\ \ldots\}\rangle\,\Phi}{\langle\{\ldots\ \alpha\ \ldots\}\rangle\,\Phi}$$

Rule For Abstraction – Attempt 1

- Example 1: What KeY does

$$\langle\{h = l{+}{+};\}\rangle\,\Phi \quad \rightsquigarrow$$
$$\langle\{h = l;\ l = (int)(l+1);\}\rangle\,\Phi \quad \rightsquigarrow$$
$$\ldots \quad \rightsquigarrow$$
$$\{h := l,\ l := l+1\}\,\langle\{\}\rangle\,\Phi$$

- Wanted
  - locations whose states may change after execution of program
  - new state of those locations
- Abstracting program

$$\langle\{h = l{+}{+};\}\rangle\,\Phi \quad \rightsquigarrow$$
$$\{h := l,\ l := f(l)\}\,\langle\{\}\rangle\,\Phi$$

- A rule can be

$$\frac{\bar{\nu}\,\langle\{..\ ...\}\rangle\,\Phi}{\langle\{..\ \alpha\ ...\}\rangle\,\Phi}$$

Rule For Abstraction – Attempt 1

- Example 1: What KeY does

$$\begin{aligned} \langle\{h = l{+}{+};\}\rangle\, \Phi &\quad \leadsto \\ \langle\{h = l;\ l = (int)(l+1);\}\rangle\, \Phi &\quad \leadsto \\ \ldots &\quad \leadsto \\ \{h := l,\ l := l+1\}\ \langle\{\}\rangle\, \Phi & \end{aligned}$$

- Wanted
  - locations whose states may change after execution of program
  - new state of those locations
- Abstracting program

$$\begin{aligned} \langle\{h = l{+}{+};\}\rangle\, \Phi &\quad \leadsto \\ \{h := l,\ l := f(l)\}\ \langle\{\}\rangle\, \Phi & \end{aligned}$$

- A rule can be

$$\frac{\bar{\nu}\ \langle\{\ldots\ \ldots\}\rangle\, \Phi}{\langle\{\ldots\ \alpha\ \ldots\}\rangle\, \Phi}$$

Rule For Abstraction – Attempt 1

- Example 1: What KeY does

$$\langle\{h = l{+}{+};\}\rangle\ \Phi \quad \rightsquigarrow$$
$$\langle\{h = l;\ l = (int)(l + 1);\}\rangle\ \Phi \quad \rightsquigarrow$$
$$\ldots \quad \rightsquigarrow$$
$$\{h := l,\ l := l + 1\}\ \langle\{\}\rangle\ \Phi$$

- Wanted
  - locations whose states may change after execution of program
  - new state of those locations
- Abstracting program

$$\langle\{h = l{+}{+};\}\rangle\ \Phi \quad \rightsquigarrow$$
$$\{h := l,\ l := f(l)\}\ \langle\{\}\rangle\ \Phi$$

- A rule can be

$$\frac{\bar{\nu}\ \langle\{..\ ...\}\rangle\ \Phi}{\langle\{..\ \alpha\ ...\}\rangle\ \Phi}$$

Rule For Abstraction – Attempt 2

Is it suitable for general case? NO!

- Example 2: What KeY does

$$\langle\{h = x/y;\}\rangle\, \Phi \quad \rightsquigarrow$$
$$(\neg(y \doteq 0) \rightarrow \{h := \mathtt{jdiv}(x,y)\})\langle\{\}\rangle\, \Phi) \wedge (y \doteq 0 \rightarrow$$
$$\langle\{\mathtt{throw\ new\ java.lang.ArithmeticException\ ();}\}\rangle\, \Phi) \quad \rightsquigarrow$$

- Need to handle exception(s) as well. We want it to be

$$\langle\{h = x/y;\}\rangle\, \Phi \quad \rightsquigarrow$$
$$(\neg(y \doteq 0) \rightarrow \{h := f(x,y)\})\langle\{\}\rangle\, \Phi) \wedge (y \doteq 0 \rightarrow$$
$$\langle\{\mathtt{throw\ new\ java.lang.ArithmeticException\ ();}\}\rangle\, \Phi) \quad \rightsquigarrow$$

Rule For Abstraction – Attempt 2

Is it suitable for general case? NO!

- Example 2: What KeY does

$$\langle\{h = x/y;\}\rangle\, \Phi \quad \rightsquigarrow$$
$$(\neg(y \doteq 0) \rightarrow \{h := \texttt{jdiv}(x,y)\})\langle\{\}\rangle\, \Phi) \wedge (y \doteq 0 \rightarrow$$
$$\langle\{\texttt{throw new java.lang.ArithmeticException ();}\}\rangle\, \Phi) \quad \rightsquigarrow$$

- Need to handle exception(s) as well. We want it to be

$$\langle\{h = x/y;\}\rangle\, \Phi \quad \rightsquigarrow$$
$$(\neg(y \doteq 0) \rightarrow \{h := f(x,y)\})\langle\{\}\rangle\, \Phi) \wedge (y \doteq 0 \rightarrow$$
$$\langle\{\texttt{throw new java.lang.ArithmeticException ();}\}\rangle\, \Phi) \quad \rightsquigarrow$$

Rule For Abstraction – Attempt 2

Is it suitable for general case? NO!

- Example 2: What KeY does

$$\langle\{h = x/y;\}\rangle\, \Phi \quad \rightsquigarrow$$
$$(\neg(y \doteq 0) \rightarrow \{h := \mathtt{jdiv}(x, y)\})\langle\{\}\rangle\, \Phi) \wedge (y \doteq 0 \rightarrow$$
$$\langle\{\mathtt{throw\ new\ java.lang.ArithmeticException\ ();}\}\rangle\, \Phi) \quad \rightsquigarrow$$

- Need to handle exception(s) as well. We want it to be

$$\langle\{h = x/y;\}\rangle\, \Phi \quad \rightsquigarrow$$
$$(\neg(y \doteq 0) \rightarrow \{h := f(x, y)\})\langle\{\}\rangle\, \Phi) \wedge (y \doteq 0 \rightarrow$$
$$\langle\{\mathtt{throw\ new\ java.lang.ArithmeticException\ ();}\}\rangle\, \Phi) \quad \rightsquigarrow$$

Rule For Abstraction – Attempt 2

Is it suitable for general case? NO!

- Example 2: What KeY does

$$\langle\{h = x/y;\}\rangle\,\Phi \;\rightsquigarrow$$
$$(\neg(y \doteq 0) \rightarrow \{h := \mathtt{jdiv}(x,y)\})\langle\{\}\rangle\,\Phi) \wedge (y \doteq 0 \rightarrow$$
$$\langle\{\mathtt{throw\ new\ java.lang.ArithmeticException\ ();}\}\rangle\,\Phi) \;\rightsquigarrow$$

- Need to handle exception(s) as well. We want it to be

$$\langle\{h = x/y;\}\rangle\,\Phi \;\rightsquigarrow$$
$$(\neg(y \doteq 0) \rightarrow \{h := f(x,y)\})\langle\{\}\rangle\,\Phi) \wedge (y \doteq 0 \rightarrow$$
$$\langle\{\mathtt{throw\ new\ java.lang.ArithmeticException\ ();}\}\rangle\,\Phi) \;\rightsquigarrow$$

Rule For Abstraction

Looks complicated ...

$$
\frac{
\begin{array}{l}
\nu_0 \; (\neg\psi_0 \rightarrow \\
\quad \nu_1 \; (\neg\psi_1 \rightarrow \\
\qquad \ldots \\
\qquad\quad \nu_m \; (\neg\psi_m \rightarrow \nu \; \langle\{ .. \quad ... \}\rangle\Phi \\
\qquad\qquad \wedge \; (\psi_m \rightarrow \langle\{ .. \; \mathtt{throw} \; E_m \; ... \}\rangle\Phi \;) \;) \\
\qquad \ldots \\
\qquad \wedge \; (\psi_1 \rightarrow \langle\{ .. \; \mathtt{throw} \; E_1 \; ... \}\rangle\Phi \;) \;) \\
\quad \wedge \; (\psi_0 \rightarrow \langle\{ .. \; \mathtt{throw} \; E_0 \; ... \}\rangle\Phi \;) \;)
\end{array}
}{
\langle\{ .. \; \alpha \; ... \}\rangle\Phi
}
$$

E_i	—	exception classes that may be thrown in the execution of α
ψ_i	—	condition to throw E_i
ν_i	—	update which occurs before E_i is thrown but after E_{i-1} is thrown (if there are any)
ν	—	update which occurs when no exception is thrown

Taclet

```
schema variables {
  program statement #concreteStatement;
  formula post;
}

rules {
  abstract {
    find ( <{.. #concreteStatement ...}> post)
    varcond ( #concreteStatement isAbstractable )
    replacewith ( #abstract(<{..  ...}> post) )
  };
}
```

Only certain statements can be treated so far, so we use `varcond` to identify here.

Implementation Issues

- Something we need to construct abstraction of program

$$([(\nu_i, \psi_i, E_i) : i = 1, \ldots, m], \nu) \tag{3}$$

- Deductively extract (3) from program statement. Some samples:

$$\frac{}{\vdash \ \mathtt{v} = \mathtt{v_0} -- \Downarrow (\varnothing, \{ \mathtt{v} := \mathtt{v_0}, \mathtt{v_0} := \mathtt{f}(\mathtt{v_0}) \})}$$

$$\frac{\vdash \ \mathtt{v} = \mathtt{v} + (\mathtt{e}) \Downarrow (\sigma, \nu)}{\vdash \ \mathtt{v} += \mathtt{e} \Downarrow (\sigma, \nu)}$$

$$\frac{\vdash \ \mathtt{v_0} = \mathtt{e_0} \Downarrow [\sigma_0, \nu_0] \qquad \vdash \ \mathtt{v_1} = \mathtt{e_1} \Downarrow [\sigma_1, \nu_1]}{\vdash \ \mathtt{v} = \mathtt{e_0}/\mathtt{e_1} \Downarrow ([\sigma_0, \sigma_1, (\nu_0 \nu_1, \mathtt{v_1} \doteq \mathtt{0}, E)], \mathtt{v} := \mathtt{f}(\mathtt{v_0}, \mathtt{v_1}))}$$

where $E \stackrel{def}{=}$ java.lang.ArithmeticException

Implementation Issues

- Something we need to construct abstraction of program

$$([(\nu_i, \psi_i, E_i) : i = 1, \ldots, m], \nu) \tag{3}$$

- Deductively extract (3) from program statement. Some samples:

$$\frac{}{\vdash \mathtt{v} = \mathtt{v}_0 \, -- \Downarrow (\emptyset, \{ \mathtt{v} := \mathtt{v}_0, \mathtt{v}_0 := \mathtt{f}(\mathtt{v}_0) \})}$$

$$\frac{\vdash \mathtt{v} = \mathtt{v} + (\mathtt{e}) \Downarrow (\sigma, \nu)}{\vdash \mathtt{v} {+}{=} \mathtt{e} \Downarrow (\sigma, \nu)}$$

$$\frac{\vdash \mathtt{v}_0 = \mathtt{e}_0 \Downarrow [\sigma_0, \nu_0] \qquad \vdash \mathtt{v}_1 = \mathtt{e}_1 \Downarrow [\sigma_1, \nu_1]}{\vdash \mathtt{v} = \mathtt{e}_0 / \mathtt{e}_1 \Downarrow ([\sigma_0, \sigma_1, (\nu_0 \nu_1, \mathtt{v}_1 \doteq 0, E)], \mathtt{v} := \mathtt{f}(\mathtt{v}_0, \mathtt{v}_1))}$$

where $E \stackrel{def}{=}$ java.lang.ArithmeticException

Example(2)

- Given proof obligation

```
==>
  <{int h; int l;}>
    <{h = l++ + ++l;}>
      secure(cons(nil, l))
```

- After applying `abstract` rule, proof turns to

```
==>
 {h:=f4(l),
   l:=f2(l)}
   <{}> secure(cons(nil, l))
```

Example(2)

- Given proof obligation

```
==>
  <{int h; int l;}>
    <{h = l++ + ++l;}>
      secure(cons(nil, l))
```

- After applying `abstract` rule, proof turns to

```
==>
 {h:=f4(l),
   l:=f2(l)}
   <{}> secure(cons(nil, l))
```

Example(2) more complicated one

- Given proof obligation

```
==>
  <{int x; int y; int z;}>
    <{x = y /= z++;}>
      secure(cons(cons(cons(nil, x),y),z))
```

- After applying `abstract` rule, proof turns to

```
==>
   (   !z = 0
    -> {x:=f5(y, z),
         y:=f5(y, z),
         z:=f4(z)}
         <{}> secure(cons(cons(cons(nil, x), y), z)))
  & (   z = 0
    -> {z:=f4(z)}
         <{
            throw new  java.lang.ArithmeticException ();
          }> <{}> secure(cons(cons(cons(nil, x), y), z)))
```

Example(2) more complicated one

- Given proof obligation

```
==>
  <{int x; int y; int z;}>
    <{x = y /= z++;}>
      secure(cons(cons(cons(nil, x),y),z))
```

- After applying `abstract` rule, proof turns to

```
==>
   (   !z = 0
    -> {x:=f5(y, z),
          y:=f5(y, z),
          z:=f4(z)}
          <{}> secure(cons(cons(cons(nil, x), y), z)))
 & (   z = 0
    -> {z:=f4(z)}
          <{
             throw new  java.lang.ArithmeticException ();
           }> <{}> secure(cons(cons(cons(nil, x), y), z)))
```

Contribution and Future Work

- Adapt for arrays and attribute variables.

  new structures can be easily extended

- Leakage by program termination behavior.

  analyze open goals

- Formalize insecurity property.

  negation of new proof obligation

- Formalize declassification (intended leakage).

  probably non-trivial

- Result
  - Approach works fine on small examples.
  - Able to treat about 40 operators in Java including those ones with side-effects.
- Future Work
  - Generalize our approach for more statements, such as `if` and `while`.
  - Formalize application of program abstraction, *i.e*, when abstracting program is meaningful.

Contribution and Future Work

- Adapt for arrays and attribute variables.

  new structures can be easily extended
- Leakage by program termination behavior.

  analyze open goals
- Formalize insecurity property.

  negation of new proof obligation
- Formalize declassification (intended leakage).

  probably non-trivial
- Result
  - Approach works fine on small examples.
  - Able to treat about 40 operators in Java including those ones with side-effects.
- Future Work
  - Generalize our approach for more statements, such as `if` and `while`.
  - Formalize application of program abstraction, *i.e*, when abstracting program is meaningful.