Secure Information Flow as a Safety Problem

Overview

- Introduction to secure information flow
- Type-Based approach
- Self composition
- Downgrading
- Self composition with downgrading
- Type directed transformation
- Conclusion

Introduction

The termination insensitive secure information flow problem (non-interference) can be reduced to solving a safety problem via a simple program transformation.

The transformation is called Self-composition.

This paper generalizes this self-compositional approach with a form of information downgrading.

The authors combine this with a type-based approach to achieve a better way to analyse software.

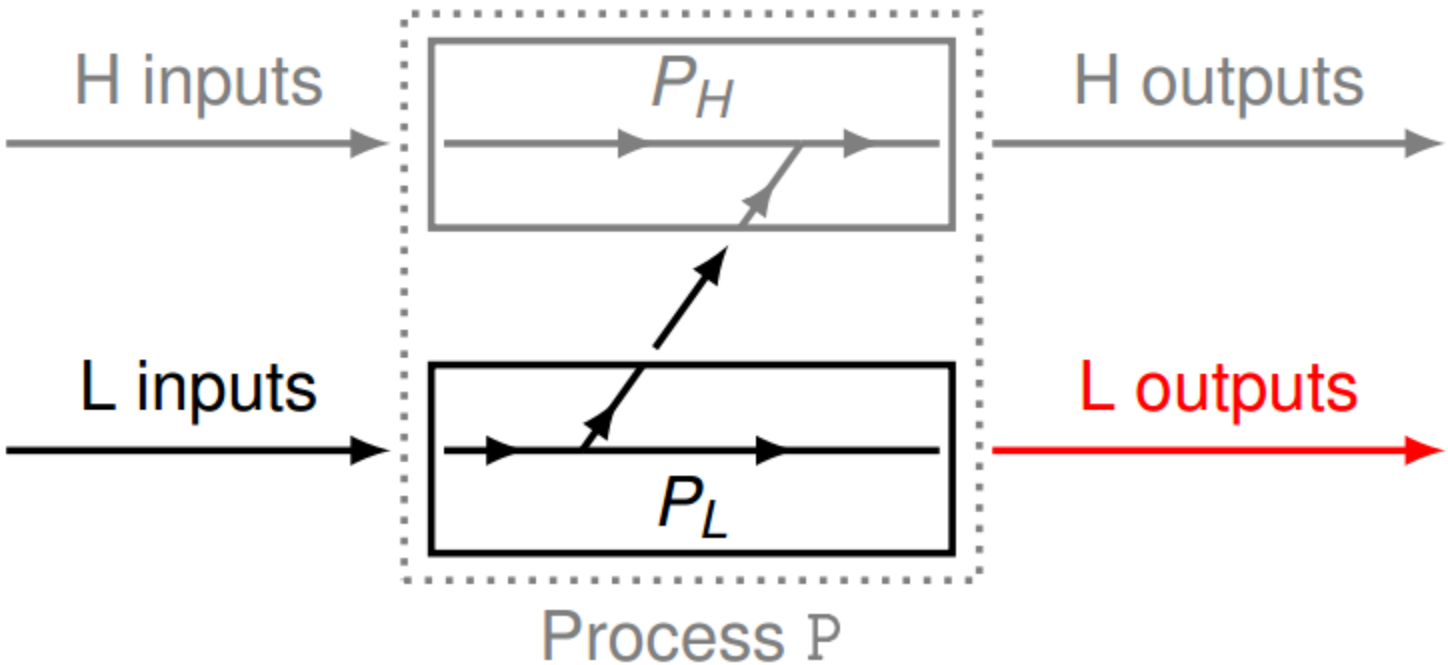
Secure Information Flow

Definition

Given a program P whose variables $H = \{h_1, . . . , h_n\}$ are high security variables and $L = \{l_1, . . . , l_n\}$ are low-security variables, P is said to be secure if and only if for any stores M_1 and M_2 such that $M_1 =_{Hc} M_2$,

$(\langle M_1, P\rangle \neq \perp \wedge \langle M_2, P\rangle \neq \perp) \Rightarrow \langle M_1, P\rangle =_L \langle M_2, P\rangle$

Non-Interference (Vanilla)



Safety Problem

A safety property is a property of a program that can be refuted by observing a finite path

Non-interference is almost a safety problem

The 2-safety property is defined similarly but the program can be refuted by observing two finite paths

Type-Based approach

Evaluates statically if the low security variables is dependent of the high security variables.

if(b) then x:=1 else skip
l:=l+x; ***SAFE***

if(h) then x:=1 else skip
l:=l+x; ***UNSAFE***

Type-based limitation

Type-based cannot show that the example is safe

$$
\begin{array}{l}
z \ := \ 1; \\
\texttt{if} \ (h) \ \texttt{then} \ x \ := \ 1 \ \texttt{else} \ \texttt{skip}; \\
\texttt{if} \ (\neg h) \ \texttt{then} \ x \ := \ z \ \texttt{else} \ \texttt{skip}; \\
l \ := \ x \ + \ y
\end{array}
$$

Self-Composition

Type Based can't verify the previous figure, that's why we use Self-Composition because?

1. let $V(P)$ be all variables in P
2. $C(P)$ is a copy of P where $x \in V(P)$ is replaced by $C(x)$
3. For any stores M_1 and M_2 such that $domain(M_1) = V(P)$ and $domain(M_2) = V(C(P))$, let $M_1 =_L M_2$ before execution
4. Run $P;C(P)$
5. Check if $\langle M_1,P;C(P)\rangle =_L \langle M_2,P;C(P)\rangle$

Self-Composition

$$
\begin{array}{l}
z\ \texttt{:=}\ 1\texttt{;}\\
\texttt{if}\ (h)\ \texttt{then}\ x\ \texttt{:=}\ 1\ \texttt{else}\ \texttt{skip;}\\
\texttt{if}\ (\neg h)\ \texttt{then}\ x\ \texttt{:=}\ z\ \texttt{else}\ \texttt{skip;}\\
l\ \texttt{:=}\ x\ \texttt{+}\ y\texttt{;}\\
z'\ \texttt{:=}\ 1\texttt{;}\\
\texttt{if}\ (h')\ \texttt{then}\ x'\ \texttt{:=}\ 1\ \texttt{else}\ \texttt{skip;}\\
\texttt{if}\ (\neg h')\ \texttt{then}\ x'\ \texttt{:=}\ z'\ \texttt{else}\ \texttt{skip}\\
l'\ \texttt{:=}\ x'\ \texttt{+}\ y'
\end{array}
$$

Downgrading 1

Vanilla secure information flow is too strict.

For example:

```
if(hashfunc(input)=hash)
        then
            l:=secret
         else
            skip;
```

Downgrading 2

In order to ease on the restrictions, we need a downgrading function f_{hi} for each high security variable h_i that defines when and how a high security variable can be leaked.

Example (same as last page):

$f = \lambda x.if(hashfunc(input)=hash)\ then\ x\ else\ c$

More examples:

$f = \lambda x.length(x)$

$f = \lambda x.0$ *(Vanilla)*

Downgrading 3

A program F can be expressed as
$F(f(h_1) \ldots f(h_n)) = F(e_1 \ldots e_n)$ *and agree with P on low-security variables at termination.*
where e_i is a security policy, that associates each high-security variable h_i to a downgrading function f_h
The program F first evaluates the downgrading functions $f(h_1) \ldots f(h_n)$ so the $(h_1,\ldots,h_n)$ are not mentioned in the running of the rest of the program.

At termination $<M,P> =_L <M,F(e)>$

Downgrading and self composition

$$\texttt{if}\ (hashfunc(input)\ \texttt{=}\ hash)\ \texttt{then}$$
$$t\ \texttt{:=}\ t\ \texttt{+}\ 1;\ l\ \texttt{:=}\ l\ \texttt{+}\ secret$$
$$\texttt{else skip}$$

Above does not work with type based

But it works with self composition

Because type based is dependent on structure of downgrading operations

Self-Composition Problem

$\texttt{while}\ (n > 0)\ \texttt{do}$

$\quad f_1 := f_1 + f_2;\ f_2 := f_1 - f_2;\ n := n - 1;$

$\texttt{if}\ (f_1 > k)\ \texttt{then}\ l := 1\ \texttt{else}\ l := 0;$

$\texttt{while}\ (n > 0)\ \texttt{do}$

$\quad f_1 := f_1 + f_2;\ f_2 := f_1 - f_2;\ n := n - 1;$

$\texttt{if}\ (f_1 > k)\ \texttt{then}\ l := 1\ \texttt{else}\ l := 0;$

$\texttt{while}\ (n' > 0)\ \texttt{do}$

$\quad f_1' := f_1' + f_2';\ f_2' := f_1' - f_2';\ n' := n' - 1;$

$\texttt{if}\ (f_1' > k')\ \texttt{then}\ l' := 1\ \texttt{else}\ l' := 0;$

Can't be verified with self-composition, but works with type-based.

Type-directed Transformation

Both the type-based and the self-composition approach have their downsides.

Type-directed transformation combines the best of two worlds.
Using the WHILE-language to illustrate how it works.

While-language

$$P ::= x \texttt{ := } e \mid \texttt{if } e \texttt{ then } P_1 \texttt{ else } P_2 \mid \texttt{while } e \texttt{ do } P \mid P_1; P_2 \mid \texttt{skip}$$

$$\begin{aligned}\varepsilon ::= [\,] \mid x \texttt{ := } \varepsilon \mid \texttt{if } \varepsilon \texttt{ then } P_1 \texttt{ else } P_2 \mid \texttt{if } e \texttt{ then } \varepsilon \texttt{ else } P \mid \texttt{if } e \texttt{ then } P \texttt{ else } \varepsilon \mid \\ \texttt{while } \varepsilon \texttt{ do } P \mid \texttt{while } e \texttt{ do } \varepsilon \mid \varepsilon; P \mid P; \varepsilon\end{aligned}$$

Type-directed translation

$$\frac{\Gamma \vdash e : \tau \text{ where } \tau \text{ is a low-security type}}{x \mathrel{\texttt{:=}} e \rightarrow_\Gamma x \mathrel{\texttt{:=}} e; C(x) \mathrel{\texttt{:=}} x}$$

$$\frac{\Gamma \nvdash e : \tau \text{ where } \tau \text{ is a low-security type}}{x \mathrel{\texttt{:=}} e \rightarrow_\Gamma x \mathrel{\texttt{:=}} e; C(x) \mathrel{\texttt{:=}} C(e)}$$

$$\frac{\Gamma \vdash e : \tau \text{ where } \tau \text{ is a low-security type} \qquad P_1 \rightarrow_\Gamma P_1^* \qquad P_2 \rightarrow_\Gamma P_2^*}{\texttt{if } e \texttt{ then } P_1 \texttt{ else } P_2 \rightarrow_\Gamma \texttt{if } e \texttt{ then } P_1^* \texttt{ else } P_2^*}$$

$$\frac{\Gamma \nvdash e : \tau \text{ where } \tau \text{ is a low-security type}}{\texttt{if } e \texttt{ then } P_1 \texttt{ else } P_2 \rightarrow_\Gamma \texttt{if } e \texttt{ then } P_1 \texttt{ else } P_2; \texttt{if } C(e) \texttt{ then } C(P_1) \texttt{ else } C(P_2)}$$

$$\frac{\Gamma \vdash e : \tau \text{ where } \tau \text{ is a low-security type} \qquad P \rightarrow_\Gamma P^*}{\texttt{while } e \texttt{ do } s \rightarrow_\Gamma \texttt{while } e \texttt{ do } P^*}$$

$$\frac{\Gamma \nvdash e : \tau \text{ where } \tau \text{ is a low-security type}}{\texttt{while } e \texttt{ do } P \rightarrow_\Gamma \texttt{while } e \texttt{ do } P; \texttt{while } C(e) \texttt{ do } C(P)}$$

$$\frac{P_1 \rightarrow_\Gamma P_1^* \qquad P_2 \rightarrow_\Gamma P_2^*}{P_1; P_2 \rightarrow_\Gamma P_1^*; P_2^*} \qquad \frac{}{\texttt{skip} \rightarrow_\Gamma \texttt{skip}}$$

Type-directed translation Example 1

Before:

```
while (n > 0) do
  f1 := f1 + f2; f2 := f1 - f2; n := n - 1;
if (h) then x := 1 else skip;
if (¬h) then x := 1 else skip;
while (i < f1) do
  l := l + x; i := i + 1
```

$$\boxed{f_1 := f_1 + f_2;}\ f_2 := f_1 - f_2;\ n := n - 1;$$

Rule:

$$\frac{\Gamma \vdash e : \tau \text{ where } \tau \text{ is a low-security type}}{x := e \rightarrow_{\Gamma} x := e; C(x) := x}$$

After:

```
while (n > 0) do
  f1 := f1 + f2; f1' := f1; f2 := f1 - f2; f2' := f2;
  n := n - 1; n' := n;
if (h) then x := 1 else skip; if (h') then x' := 1 else skip;
if (¬h) then x := 1 else skip; if (¬h') then x' := 1 else skip;
while (i < f1) do
  l := l + x; l' := l' + x'; i := i + 1; i' := i
```

$$\boxed{f_1 := f_1 + f_2;\ f_1' := f_1;}\ f_2 := f_1 - f_2;\ f_2' := f_2;$$

Type-directed translation Example 2

Before:

$$\begin{array}{l}\texttt{while } (n > 0) \texttt{ do}\\ \quad f_1 \texttt{ := } f_1 \texttt{ + } f_2\texttt{; } f_2 \texttt{ := } f_1 \texttt{ - } f_2\texttt{; } n \texttt{ := } n \texttt{ - } 1\texttt{;}\\ \boxed{\texttt{if } (h) \texttt{ then } x \texttt{ := } 1 \texttt{ else skip;}}\\ \texttt{if } (\neg h) \texttt{ then } x \texttt{ := } 1 \texttt{ else skip;}\\ \texttt{while } (i < f_1) \texttt{ do}\\ \quad l \texttt{ := } l \texttt{ + } x\texttt{; } i \texttt{ := } i \texttt{ + } 1\end{array}$$

Rule:

$$\frac{\Gamma \not\vdash e : \tau \text{ where } \tau \text{ is a low-security type}}{\texttt{if } e \texttt{ then } P_1 \texttt{ else } P_2 \rightarrow_\Gamma \texttt{if } e \texttt{ then } P_1 \texttt{ else } P_2\texttt{; if } C(e) \texttt{ then } C(P_1) \texttt{ else } C(P_2)}$$

After:

$$\begin{array}{l}\texttt{while } (n > 0) \texttt{ do}\\ \quad f_1 \texttt{ := } f_1 \texttt{ + } f_2\texttt{; } f_1' \texttt{ := } f_1\texttt{; } f_2 \texttt{ := } f_1 \texttt{ - } f_2\texttt{; } f_2' \texttt{ := } f_2\texttt{;}\\ \quad n \texttt{ := } n \texttt{ - } 1\texttt{; } n' \texttt{ := } n\texttt{;}\\ \boxed{\texttt{if } (h) \texttt{ then } x \texttt{ := } 1 \texttt{ else skip; if } (h') \texttt{ then } x' \texttt{ := } 1 \texttt{ else skip;}}\\ \texttt{if } (\neg h) \texttt{ then } x \texttt{ := } 1 \texttt{ else skip; if } (\neg h') \texttt{ then } x' \texttt{ := } 1 \texttt{ else skip;}\\ \texttt{while } (i < f_1) \texttt{ do}\\ \quad l \texttt{ := } l \texttt{ + } x\texttt{; } l' \texttt{ := } l' \texttt{ + } x'\texttt{; } i \texttt{ := } i \texttt{ + } 1\texttt{; } i' \texttt{ := } i\end{array}$$

Type-directed translation Example 3

Before:

```
while (n > 0) do
   f1 := f1 + f2; f2 := f1 - f2; n := n - 1;
if (h) then x := 1 else skip;
if (¬h) then x := 1 else skip;
while (i < f1) do
   l := l + x; i := i + 1
```

Rule:

$$\frac{\Gamma \vdash e : \tau \text{ where } \tau \text{ is a low-security type} \qquad P \rightarrow_\Gamma P^*}{\texttt{while}\ e\ \texttt{do}\ s \rightarrow_\Gamma \texttt{while}\ e\ \texttt{do}\ P^*}$$

After:

```
while (n > 0) do
   f1 := f1 + f2; f1' := f1; f2 := f1 - f2; f2' := f2;
   n := n - 1; n' := n;
if (h) then x := 1 else skip; if (h') then x' := 1 else skip;
if (¬h) then x := 1 else skip; if (¬h') then x' := 1 else skip;
while (i < f1) do
   l := l + x; l' := l' + x'; i := i + 1; i' := i
```

Conclusion

- Type-directed transformation is better than the type based approach.
- But not much different to self-composed approach for a hypothetical analysis tool
- More digestible than self-composed
- Still not perfect.