Secure Information Flow by Self-Composition

Paper by Gilles Barthe, Pedro R. D’Argenio, and Tamara Rezk

Slides by Isaac Sheff

\[ S \cong_{\xi, \mathcal{I}, \mathcal{I}'} S[\xi] \]
Background

• Noninterference

• Type Systems
  • new features and policies require extensions to Type System and proof

• Logical Verification and Proof-Carrying Code

• Hoare Logic
  • \{Precondition\} Code \{Postcondition\}
Program Logic

• Idea: let’s encode “Secure Information Flow” as a logically provable property
• Andrews and Reitman extended Hoare Logic
• Darvas, Hähnle and Sands used dynamic logic
• Problem: reasoning about only one process
• Pottier’s Pi Calculus work: 2 processes reduced to 1
2-Safety

- Prove that if you run program $P$ twice, and the starting conditions each time are “low-equivalent,” then the finishing conditions are “low-equivalent.”

\[
\begin{array}{c}
\text{high:7} \\
\text{low:3}
\end{array}
\quad \Rightarrow \quad
\begin{array}{c}
\text{high:2} \\
\text{low:0}
\end{array}
\]

\[
\begin{array}{c}
\text{high:6} \\
\text{low:3}
\end{array}
\quad \Rightarrow \quad
\begin{array}{c}
\text{high:1} \\
\text{low:0}
\end{array}
\]
Self-Composition

- From program $P$, create $P'$
- just like $P$, but all new variable names
- New Program: $P;P'$
- Need only consider single program
Notation

• Memory state: $\mu$

• Read variable foo from memory $\mu$: $\mu(\text{foo})$

• Program Termination: $\sqrt{\text{✓}}$

• Configuration: (Program, Memory state)

• Stick two (non-overlapping) memories together: $\oplus$

• S, starting with memory $\mu$, doesn’t terminate: $(S, \mu) \perp$

• Get all variable names in a memory: $\text{var}(\mu)$

• Get abstract data structure at variable foo: $\nu(\mu, \text{foo})$

• “Small Step” transition: $\leadsto$

• Any number of steps: $\leadsto^*$
\[ S ::= x := e \mid \text{if } b_0 \rightarrow S_0 \cdots b_n \rightarrow S_n \text{ fi} \]
\[ \mid S_1 ; S_2 \mid \text{while } b \text{ do } S \text{ od} \mid S_1 || S_2 \]

**Par**

\[(x := e, \mu) \leadsto (\sqrt, \mu[x \mapsto \mu(e)])\]

\[\begin{array}{l}
(S_1, \mu) \leadsto (S_1', \mu') \\
(S_1 ; S_2, \mu) \leadsto (S_1', S_2, \mu')
\end{array}\]

**nondeterminism!**

\[\frac{(S_j, \mu) \leadsto (S_j', \mu')}{(\text{if } b_0 \rightarrow S_0 \cdots b_n \rightarrow S_n \text{ fi, } \mu) \leadsto (S_j', \mu') \quad 0 \leq j \leq n} \]

**Parallelism!**

\[\begin{array}{l}
(S, \mu) \leadsto (S', \mu') \\
(\text{while } b \text{ do } S \text{ od, } \mu) \leadsto (S'; \text{while } b \text{ do } S \text{ od, } \mu')
\end{array}\]

\[\frac{-\mu(b) \text{ holds}}{(\text{while } b \text{ do } S \text{ od, } \mu) \leadsto (\sqrt, \mu)}\]

\[\frac{(S_1, \mu) \leadsto (S_1', \mu')}{(S_1 ; S_2, \mu) \leadsto (S_1', S_2, \mu')}\]

\[\frac{(S_2, \mu) \leadsto (S_2', \mu')}{(S_1 || S_2, \mu) \leadsto (S_1 || S_2', \mu')}\]

\[\frac{(\sqrt || S_2, \mu) \leadsto (S_2, \mu)}{(S_1 || \sqrt, \mu) \leadsto (S_1, \mu)}\]
Assumption 1. Transitions preserve the set of variables of a program. Moreover, if the part of the memory that is affected by the program is separated from the rest, transitions do not affect the values of other variables than those appearing in the program.

Formally, for all $S, S', \mu_1, \mu_2,$ and $\mu'$, if $\text{var}(S) = \text{var}(\mu_1)$ and $(S, \mu_1 \oplus \mu_2) \leadsto (S', \mu')$, then $\text{var}(S) \supseteq \text{var}(S')$ and $\exists \mu'_1 : \mu' = \mu'_1 \oplus \mu_2 \land \text{var}(\mu'_1) = \text{var}(S)$. In addition, if $(S, \mu_1 \oplus \mu_2) \leadsto (S', \mu'_1 \oplus \mu_2)$, then for all $\mu_3$ s.t. $\mu_1 \oplus \mu_3$ is defined, $(S, \mu_1 \oplus \mu_3) \leadsto (S', \mu'_1 \oplus \mu_3)$.

- Implies “var($S$)” is “deep.”
Assumption 2. Apart from its syntax, the semantics of a program depends only on the abstract value of its own variables.

Formally, we assume that for all configurations \((S, \mu_1)\) and \((S, \mu_2)\) such that \(\forall x \in \text{var}(S) : v(\mu_1, x) = v(\mu_2, x)\) then for all \((S', \mu'_1), (S, \mu_1) \Rightarrow^* (S', \mu'_1)\) \(\Rightarrow\) \(\exists(S', \mu'_2) : (S, \mu_2) \Rightarrow^* (S', \mu'_2)\) and \(\forall x \in \text{var}(S) : v(\mu'_1, x) = v(\mu'_2, x)\).

- Definitely not true for all programs (or languages)
- Pointer logic messes with this

Assumption 3. The operational semantics of the language \text{Lang} is independent of variable names. Formally, if \(y \notin \text{var}(S)\) and \((S, \mu) \Rightarrow^* (S', \mu')\) then \((S[y/x], \mu[y \mapsto v(\mu, x)]) \Rightarrow^* (S'[y/x], \mu'[x \mapsto d][y \mapsto v(\mu', x)])\) for some \(d\).
Given These Assumptions . . .

- Program S won’t change anything not in var(S)
- Stuff not in var(S) won’t affect S’s termination
- Changing a variable name doesn’t affect termination
- S, run on an identical set of values in different memories, has the same termination
More Notation

- $\varphi : \text{Var} \rightarrow \text{Var}$ injective functions from one set of variables to another

- “Indistinguishable” variable sets: $\mathcal{I} \subseteq \mathcal{V}^n \times \mathcal{V}^n$

- “Indistinguishable” memories:
  \[ \text{dom}(\varphi) = \{x_1, \ldots, x_n\} \]
  \[ \mu \sim_{\mathcal{I}} \mu', \text{ if } \langle (v(\mu, x_1), \ldots, v(\mu, x_n)), (v(\mu', \varphi(x_1)), \ldots, v(\mu', \varphi(x_n))) \rangle \in \mathcal{I} \]

- things that act like “$S_1;S_2$” (including parallelism): $S_1 \triangleleft S_2$
Non-Interference, Formalized

• $S_1$ is termination sensitive (TS) non interferent with program $S_2$

$$S_1 \cong_{\phi, \mathcal{I}} S_2 \text{ if for all } \mu_1, \mu_2, \mu'_1 \in \mathcal{M},$$

$$(\mu_1 \sim_{\phi} \mu_2 \land (S_1, \mu_1) \rightsquigarrow^{*} (\sqrt{\mathcal{I}}, \mu'_1)) \Rightarrow \exists \mu'_2 \in \mathcal{M} : (S_2, \mu_2) \rightsquigarrow^{*} (\sqrt{\mathcal{I}}, \mu'_2) \land \mu'_1 \sim_{\phi} \mu'_2.$$

• $S_1$ is termination insensitive (TI) non interferent with program $S_2$

$$S_1 \cong_{\phi, \mathcal{I}} S_2 \text{ if for all } \mu_1, \mu_2, \mu'_1 \in \mathcal{M},$$

$$(\mu_1 \sim_{\phi} \mu_2 \land (S_1, \mu_1) \rightsquigarrow^{*} (\sqrt{\mathcal{I}}, \mu'_1)) \Rightarrow$$

$$(S_2, \mu_2) \perp \lor \exists \mu'_2 \in \mathcal{M} : (S_2, \mu_2) \rightsquigarrow^{*} (\sqrt{\mathcal{I}}, \mu'_2) \land \mu'_1 \sim_{\phi} \mu'_2.$$
Non-Interference, Formalized

- Different pre-condition and post-condition variable renaming and indistinguishability
- Extremely flexible
- Security: a program is non-interferent with itself
- Maybe indistinguishability is just low-equivalence

Let $\mathcal{I}, \mathcal{I}' \subseteq \mathcal{V}^n \times \mathcal{V}^n$ with $n = \# \text{var}(S)$.

(a) $S$ is termination sensitive (TS) $(\mathcal{I}, \mathcal{I}')$-secure iff $S \approx_{\text{id}, \mathcal{I}, \mathcal{I}'}^{\text{id}, \mathcal{I}, \mathcal{I}'} S$.

(b) $S$ is termination insensitive (TI) $(\mathcal{I}, \mathcal{I}')$-secure iff $S \approx_{\text{id}, \mathcal{I}, \mathcal{I}'}^{\text{id}, \mathcal{I}, \mathcal{I}'} S$. 
Definition 2. Let $S_1$, $S_2$ be two programs such that $\text{var}(S_1) \cap \text{var}(S_2) = \emptyset$. We define $S_1 \triangleright_{\phi,\mathcal{I}} S_2$ (and $S_1 \triangleright_{\phi',\mathcal{I}'} S_2$ for the TI case) if for all $\mu_1$, $\mu_2$, $\mu'_1$, $\mu'_2$, $\text{var}(\mu_1) = \text{var}(\mu'_1) = \text{var}(S_1)$ and $\text{var}(\mu_2) = \text{var}(S_2)$,

$$\left( \mu_1 \oplus \mu_2 \sim^{\mathcal{I}}_{\phi} \mu_1 \oplus \mu_2 \land (S_1 \triangleright S_2, \mu_1 \oplus \mu_2) \sim^{\star} (S_2, \mu'_1 \oplus \mu'_2) \right)$$

$$\Rightarrow (\exists \mu'_2 : \text{var}(\mu'_2) = \text{var}(S_2) : (S_2, \mu'_1 \oplus \mu_2) \sim^{\star} (\sqrt{\mu_1} \oplus \mu'_2) \land \mu'_1 \oplus \mu'_2 \sim^{\mathcal{I}'}_{\phi} \mu'_1 \oplus \mu'_2)$$

$$\left( \lor (S_2, \mu'_1 \oplus \mu_2) \perp \text{for the TI case}. \right)$$

- If the memory state was “indistinguishable from itself” to begin with, then after running $S_1 \triangleleft S_2$, it remains “indistinguishable from itself”
The Big One

Theorem 1. Let $S_1$ and $S_2$ such that $\text{var}(S_1) \cap \text{var}(S_2) = \emptyset$ and let $\phi : \text{var}(S_1) \rightarrow \text{var}(S_2)$. Then

(a) $S_1 \approx_{\phi, I} S_2$ if and only if $S_1 \bowtie_{\phi, I} S_2$, and

(b) $S_1 \approx_{\phi, I} S_2$ if and only if $S_1 \bowtie_{\phi, I} S_2$.

- For programs with non-overlapping sets of variables, our non-interference for two programs on different memories is the same as for two composed programs.
Proof: TS

Proposition 1. For all $\mu_1, \mu_2, \mu_1'', \mu_2'', I$, and $\phi : \text{var}(\mu_1) \rightarrow \text{var}(\mu_2)$, $\mu_1 \sim^I_{\phi} \mu_2$ iff $\mu_1 \oplus \mu_1'' \sim^I_{\phi} \mu_2 \oplus \mu_2''$.

- $\oplus$ is commutative, so
  $$\mu_1 \sim^I_{\phi} \mu_2 \iff \mu_1 \oplus \mu_2 \sim^I_{\phi} \mu_1 \oplus \mu_2$$

- Since programs don’t affect variables not in the program:
  $$(S_1, \mu_1) \Rightarrow^* (\sqrt{}, \mu'_1) \iff (S_1, \mu_1 \oplus \mu_2) \Rightarrow^* (\sqrt{}, \mu'_1 \oplus \mu_2)$$

- Therefore
  $$(S_1, \mu_1 \oplus \mu_2) \Rightarrow^* (\sqrt{}, \mu'_1 \oplus \mu_2) \iff (S_1 \triangleright S_2, \mu_1 \oplus \mu_2) \Rightarrow^* (S_2, \mu'_1 \oplus \mu_2)$$

- And so:
  $$\exists \mu'_2 : (S_2, \mu_2) \Rightarrow^* (\sqrt{}, \mu'_2) \land \mu'_1 \sim^I_{\phi'} \mu'_2$$

iff
  $$\exists \mu'_2 : (S_2, \mu'_1 \oplus \mu_2) \Rightarrow^* (\sqrt{}, \mu'_1 \oplus \mu'_2) \land \mu'_1 \oplus \mu'_2 \sim^I_{\phi'} \mu'_1 \oplus \mu'_2$$
Proof: TI

- Because variables not in \( \text{var}(S) \) don't affect the termination of \( S \):

\[
(\exists \mu'_2 : (S_2, \mu_2) \xrightarrow{*} (\sqrt{}, \mu'_2) \land \mu'_1 \xrightarrow{\bar{\phi}'} \mu'_2) \lor (S_2, \mu'_1) \bot
\]

iff

\[
(\exists \mu'_2 : (S_2, \mu'_1 \oplus \mu_2) \xrightarrow{*} (\sqrt{}, \mu'_1 \oplus \mu'_2) \land \mu'_1 \oplus \mu_2 \xrightarrow{\bar{\phi}'} \mu'_1 \oplus \mu'_2) \lor (S_2, \mu'_1 \oplus \mu_2) \bot
\]
Theorem 2. Let $\xi : \text{var}(S_2) \rightarrow V$ be a bijective function on a set of variables $V$. Then

(a) $S_1 \cong^{\phi, I}_{\phi, I} S_2$ iff $S_1 \cong^{\xi \circ \phi, I}_{\xi \circ \phi, I} S_2[\xi]$, and

(b) $S_1 \cong^{\phi, I}_{\phi, I} S_2$ iff $S_1 \cong^{\xi \circ \phi, I}_{\xi \circ \phi, I} S_2[\xi]$.

where $S_2[\xi]$ is program $S_2$ whose variables have been renamed according to function $\xi$.

- Let $\xi$ be a function mapping the variables of one program to a new set.

- Noninterference of two programs is the same as noninterference of those programs, with one of their variables all renamed via $\xi$. 
Theorem 2 Proof?

- Idea: prove that changing one variable’s name does not alter noninterference
- Induct over all the variables in the program
- It’s not very concise.
Corollary 1. Let $\xi : \text{var}(S) \to \text{Var}$. Define $\text{var}(S)' = \{\xi(x) \mid x \in \text{var}(S)\}$ so that $\text{var}(S) \cap \text{var}(S)' = \emptyset$ and $x \mapsto \xi(x)$ is a bijection from $\text{var}(S)$ to $\text{var}(S)'$. Then, the following statements are equivalent

1. $S$ is TS (resp. TI) $(\mathcal{I}, \mathcal{I}')$-secure.
2. $S \approx_{\xi, \mathcal{I}, I'} S[\xi]$ (resp. $S \approx_{\xi, \mathcal{I}, I'} S[\xi]$)
3. $S \triangleright_{\xi, \mathcal{I}, I'} S[\xi]$ (resp. $S \triangleright_{\xi, \mathcal{I}, I'} S[\xi]$)

• Now we can check if a program $S$ is secure, by analyzing single executions of the program $S \triangleleft S[\xi]$.
“Analyzing Single Executions”

• This is what verification logics are for
• Sections 5-9 are characterizations of security with some such logics
A Neat Example (5)

- $x_l$: public
- $y_h$: private
- We can show $x_l := x_l + y_h$; $x_l := x_l - y_h$ is non-interferent
Deterministic

• We can actually check the security of a program by analyzing only the I/O (start and finish memories) of a self-composed program, e.g. $S;S[\xi]$.

**Theorem 3.** Let $S$ be a deterministic program and $\xi : \text{var}(S) \rightarrow \text{Var}$ and $\text{var}(S)'$ as in Corollary 1.

1. $S$ is TS $(I_1, I_2)$-secure if and only if

$$
\forall \mu_1, \mu_2 : \text{var}(\mu_1) = \text{var}(S) \land \text{var}(\mu_2) = \text{var}(S)' : \\
\mu_1 \oplus \mu_2 \sim_{\xi}^I \mu_1 \oplus \mu_2 \land \exists \mu'_1 : (S, \mu_1 \oplus \mu_2) \sim^* (\sqrt{\text{, }}, \mu'_1 \oplus \mu_2) \\
\Rightarrow \exists \mu''_1, \mu''_2 : \text{var}(\mu''_1) = \text{var}(S) \land \text{var}(\mu''_2) = \text{var}(S)' : \\
(S \triangleright S[\xi], \mu_1 \oplus \mu_2) \sim^* (\sqrt{\text{, }}, \mu''_1 \oplus \mu''_2) \land \mu''_1 \oplus \mu''_2 \sim_{\xi}^{I_2} \mu'_1 \oplus \mu'_2
$$

2. $S$ is TI $(I_1, I_2)$-secure if and only if

$$
\forall \mu_1, \mu_2, \mu'_1, \mu'_2 : \text{var}(\mu_1) = \text{var}(\mu'_1) = \text{var}(S) \land \text{var}(\mu_2) = \text{var}(\mu'_2) = \text{var}(S)' : \\
(\mu_1 \oplus \mu_2 \sim_{\xi}^I \mu_1 \oplus \mu_2 \land (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \sim^* (\sqrt{\text{, }}, \mu'_1 \oplus \mu'_2)) \Rightarrow \mu'_1 \oplus \mu'_2 \sim_{\xi}^{I_2} \mu'_1 \oplus \mu'_2
$$
Deterministic Language

• While: a version of Par without nondeterminism (if can have only if \ldots else \ldots fi and get rid of parallelism)

• Consider memory that only stores integers, which is conveniently separable by ⊕

Corollary 2. Let $S \triangleright S[\xi]$ be a deterministic program with memory as defined in Example 1. Let $\xi : \text{var}(S) \rightarrow \text{Var}$ and $\text{var}(S)'$ as in Corollary 1.

1. $S$ is $\text{TS} (I_1, I_2)$-secure if and only if
   \[ \forall \mu : (\mu \sim_{\xi}^{I_1} \mu \land \exists \mu'. (S, \mu) \sim^{*} (\sqrt{,} \mu')) \Rightarrow (\exists \mu''. (S \triangleright S[\xi], \mu) \sim^{*} (\sqrt{,} \mu'') \land \mu'' \sim_{\xi}^{I_2} \mu'') \]

2. $S$ is $\text{TI} (I_1, I_2)$-secure if and only if
   \[ \forall \mu, \mu' : (\mu \sim_{\xi}^{I_1} \mu \land (S \triangleright S[\xi], \mu) \sim^{*} (\sqrt{,} \mu') ) \Rightarrow \mu' \sim_{\xi}^{I_2} \mu' \]
Hoare Logic

- {Precondition} Code {Postcondition}

$\{P[e/x]\} \ x := e \ \{P\}$

$\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi } \{Q\}$

$\{P\} S_1 \{R\} \quad R \ S_2 \{Q\}$

$\{P\} \ S_1 \ ; \ S_2 \{Q\}$

$P' \Rightarrow P \quad \{P\} \ S \{Q\} \quad Q \Rightarrow Q'$

$\{P'\} \ S \{Q'\}$

$\{P \land b\} \ S_1 \{Q\} \quad \{P \land \neg b\} \ S_2 \{Q\}$

$\{P\} \ \text{while } b \ \text{do } S \ \text{od } \{P \land \neg b\}$
Indistinguishability Criterion Shorthand

\[ \mu \models I(I) \iff \mu \sim^I \mu \quad (\text{iff} \quad (v(\mu, \overline{x}), v(\mu, \xi(\overline{x}))) \in I) \]
Proposition 2. Termination insensitive \((\mathcal{I}_1, \mathcal{I}_2)\)-security can be characterized in Hoare logic as follows:

\[ S \text{ is TI (} \mathcal{I}_1, \mathcal{I}_2 \text{)-secure iff } \{\mathbf{I}(\mathcal{I}_1)\} S ; S[\xi] \{\mathbf{I}(\mathcal{I}_2)\} \text{ is provable.} \]

Proof (recall that we’re in a deterministic setting, and have a sequential composition operator ; )

\[ S \text{ is TI (} \mathcal{I}_1, \mathcal{I}_2 \text{)-secure iff } \{\text{Corollary 2.2}\} \]
\[ \forall \mu, \mu' : ( \mu \triangleright\triangleright_{\xi} \mu \land (S ; S[\xi], \mu) \triangleright\triangleright^* (\sqrt{ }, \mu') ) \Rightarrow \mu' \triangleleft\triangleleft_{\xi} \mu' \]
\[ \text{iff } \{\text{Def. of I}\} \]
\[ \forall \mu, \mu' : ( \mu \models \mathbf{I}(\mathcal{I}_1) \land (S ; S[\xi], \mu) \triangleright\triangleright^* (\sqrt{ }, \mu') ) \Rightarrow \mu' \models \mathbf{I}(\mathcal{I}_2) \]
\[ \text{iff } \{\text{Soundness and completeness, provided } \mathbf{I} \text{ is definable}\} \]
\[ \{\mathbf{I}(\mathcal{I}_1)\} S ; S[\xi] \{\mathbf{I}(\mathcal{I}_2)\} \text{ is provable} \]
Example Proof

- **Original Program:**
  \[
  x_l := x_l + y_h;
  x_l := x_l - y_h
  \]

- **Self-Composed Version:**
  \[
  x_l := x_l + y_h;
  x_l := x_l - y_h;
  x_l' := x_l' + y_h';
  x_l' := x_l' - y_h'
  \]

- **\(\xi\):** \(x_l \mapsto x_l', \ y_h \mapsto y_h'\)

- **Indistinguishability:** \(=_{L}\)

  Meaning the low (really just \(x_l\)) values must be the same
Example Proof

\{x_l = x_l'\} \{x_l + y_h - y_h = x_l'\}

\begin{align*}
x_l &:= x_l + y_h; \\
\{x_l - y_h = x_l'\} \\
x_l &:= x_l - y_h;
\end{align*}

\begin{align*}
\{x_l = x_l'\} \{x_l = x_l' + y_h' - y_h'\} \\
x_l' &:= x_l' + y_h'; \\
\{x_l = x_l' - y_h'\} \\
x_l' &:= x_l' - y_h' \\
\{x_l = x_l'\}
\end{align*}

• \((v(\mu,x_l), v(\mu,y_h)), (v(\mu,x_l'), v(\mu,y_h'))) \in =_L\)

In both the “before” and “after” states, since the “x_l” values are equal.
Conclusions

• General Noninterference Formulation
• Self-composition for Security Analysis
  • Need only analyze one program
  • Determinism: only I/O analysis
• Various logics applied
• Future Work:
  • characterizations for other non-interference notions
  • other security properties
  • prove secure type systems with these logics
  • automation of such proofs in “real” languages