

Applications of Formal Verification

Verification of Information Flow Properties

Bernhard Beckert · Mattias Ulbrich | SS 2019

KIT – INSTITUT FÜR THEORETISCHE INFORMATIK



Security is everywhere ...

INFORMATION LEAK BIG DATA
UBIQUITOUS COMPUTING NSA
WIKILEAKS
WEB 2.0 **SECURITY**
CROSS-SITE-SCRIPTING
CLOUD COMPUTING
INDUSTRIE 4.0 PRISM
SMART GRID



- published in April 2014
- security bug in the OpenSSL TLS library
- heartbeat protocol (“ping”)
- vulnerability classified as a buffer over-read (read more data than should be allowed.)
- some 17% (around half a million) of certified secure web servers believed vulnerable to the attack
- fixed by adding one `if` statement.
- known data theft: hackers stole security keys from community health systems, compromising the confidentiality of 4.5 million patient records.

Heartbleed – Information Flow

OpenSSL's memory: Contains secret information

?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

Heartbleed – Information Flow

OpenSSL Heartbeat Request ('PING', 12)

				?	?	?	?	?	?	?	?	?	?	?
P	I	N	G	?	?	?	?	?	?	?	?	?	?	?

Heartbleed – Information Flow

OpenSSL Heartbeat Request ('PING', 12)

				?	?	?	?	?	?	?	?	?	?	?
P	I	N	G	?	?	?	?	?	?	?	?	?	?	?

OPENSSL with 

Heartbleed – Information Flow

OpenSSL Heartbeat Request ('PING', 12)

				?	?	?	?	?	?	?	?	?	?	?
P	I	N	G	?	?	?	?	?	?	?	?	?	?	?

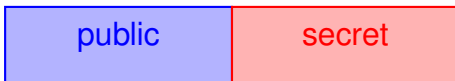
OPENSSL with 

				?	?	?	?	?	?	?	?	?	?	?
P	I	N	G	p	r	i	v	=	1	5	7	?	?	?

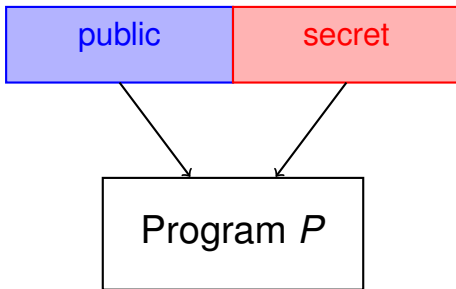
Information Flow Model



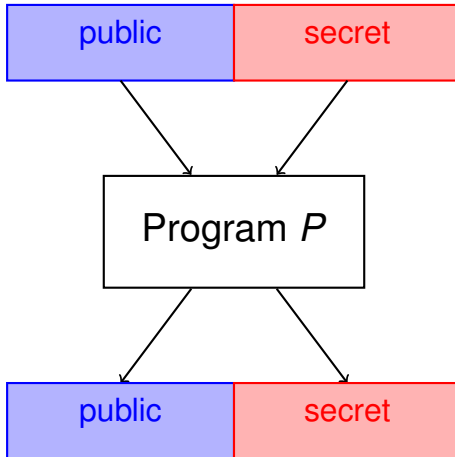
Information Flow Model



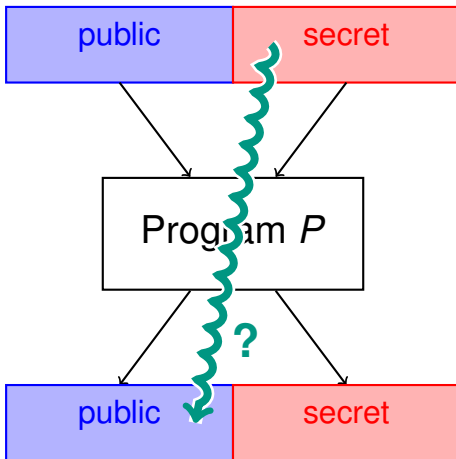
Information Flow Model

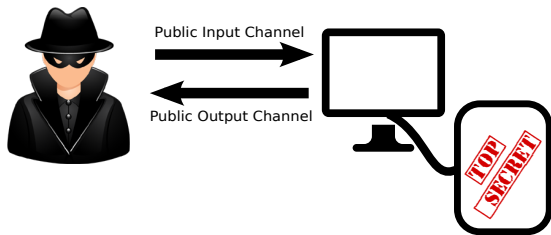


Information Flow Model

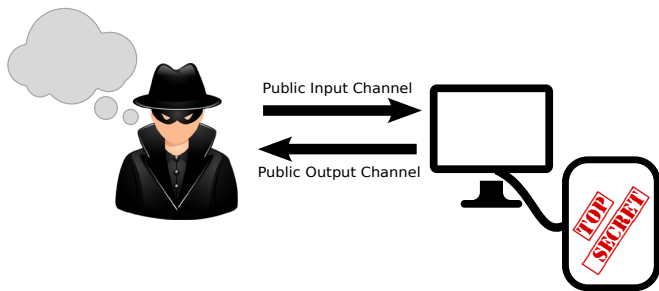


Information Flow Model

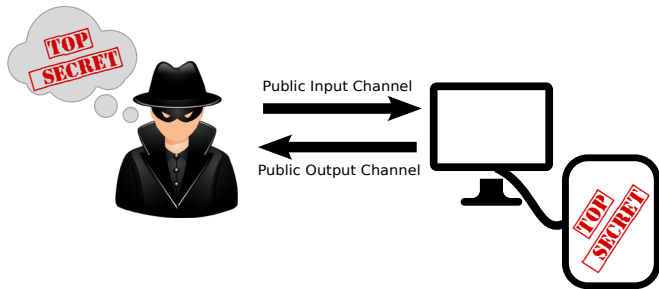




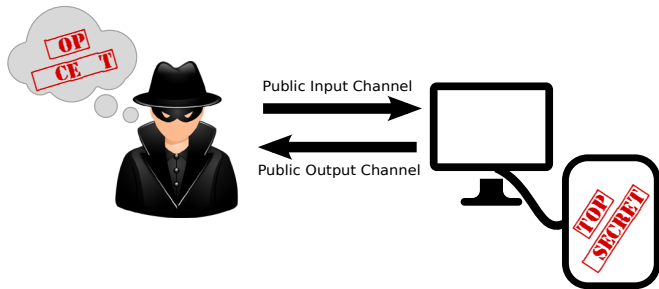
- Attacker communicates with system over public channels



- Attacker communicates with system over public channels
- ... tries to learn the secret which is kept inside the system



- Attacker communicates with system over public channels
- ... tries to learn the secret which is kept inside the system



- Attacker communicates with system over public channels
- ... tries to learn the secret which is kept inside the system
- ... or at least parts of the secret

Attacker is ...	Public channels are ...
an agent over the network	network traffic
another application on same device	shared resources (files), interprocess comm.
program using a library	shared memory, method calls

In models:

Attacker's capabilities expressed by the public channels.

Every program is a function

$$P : \textit{SecretInput} \times \textit{PublicInput} \rightarrow \textit{SecretOutput} \times \textit{PublicOutput}$$

Decomposition into two functions $P = (s, p)$

$$\begin{aligned} s & : \textit{SecretInput} \times \textit{PublicInput} \rightarrow \textit{SecretOutput} \\ p & : \textit{SecretInput} \times \textit{PublicInput} \rightarrow \textit{PublicOutput} \\ P(h, \ell) & = (s(h, \ell), p(h, \ell)) \end{aligned}$$

We will define security properties for such programs and analyse them.

Convention

Variables with high security status are named h (h_1 etc.) and variables with low (public) security status are named ℓ (ℓ_1 etc.).

Example

Java method

```
int h;  
int l;  
void f() {  
    if(h > 5) {  
        l ++;  
    } else {  
        h --;  
    }  
}
```

`h` and `l` serve as input
and output variables.

Java method

```
int h;  
int l;  
void f() {  
    if(h > 5) {  
        l ++;  
    } else {  
        h --;  
    }  
}
```

h and l serve as input
and output variables.

Model

$$s_f(h, l) = \begin{cases} h & \text{if } h > 5 \\ h - 1 & \text{if } h \leq 5 \end{cases}$$
$$p_f(h, l) = \begin{cases} l + 1 & \text{if } h > 5 \\ l & \text{if } h \leq 5 \end{cases}$$

Java method

```
int h;  
int l;  
void f() {  
    if(h > 5) {  
        l ++;  
    } else {  
        h --;  
    }  
}
```

h and l serve as input and output variables.

Model

$$s_f(h, l) = \begin{cases} h & \text{if } h > 5 \\ h - 1 & \text{if } h \leq 5 \end{cases}$$
$$p_f(h, l) = \begin{cases} l + 1 & \text{if } h > 5 \\ l & \text{if } h \leq 5 \end{cases}$$

Attacker model

- Attacker can see l .
- Attacker cannot see h .

Java method

```
private int h;  
public int l;  
void f() {  
    if(h > 5) {  
        l ++;  
    } else {  
        h --;  
    }  
}
```

h and l serve as input and output variables.

Model

$$s_f(h, l) = \begin{cases} h & \text{if } h > 5 \\ h - 1 & \text{if } h \leq 5 \end{cases}$$
$$p_f(h, l) = \begin{cases} l + 1 & \text{if } h > 5 \\ l & \text{if } h \leq 5 \end{cases}$$

Attacker model

- Attacker can see l .
- Attacker cannot see h .
- (e.g. by visibility modifiers)

Secure information flow as a game

Parties: the attacker and the system

Secure information flow as a game

Parties: the attacker and the system

Assume: Atacker knows program P

Secure information flow as a game

Parties: the attacker and the system

Assume: Attacker knows program P

Secure information flow as a game

Parties: the attacker and the system

Assume: Attacker knows program P

Protocol: ① Attacker chooses
 $x, y \in \textit{SecretInput}$,
 $z \in \textit{PublicInput}$

Secure information flow as a game

Parties: the attacker and the system

Assume: Attacker knows program P

Protocol:

- 1 Attacker chooses
 $x, y \in \text{SecretInput}$,
 $z \in \text{PublicInput}$
- 2 System selects $a \in \{x, y\}$ randomly (i.i.d.).

Secure information flow as a game

Parties: the attacker and the system

Assume: Attacker knows program P

Protocol:

- 1 Attacker chooses
 $x, y \in \text{SecretInput}$,
 $z \in \text{PublicInput}$
- 2 System selects $a \in \{x, y\}$ randomly (i.i.d.).
- 3 Attacker receives public output $p(a, z)$.

Secure information flow as a game

Parties: the attacker and the system

Assume: Attacker knows program P

- Protocol:**
- 1 Attacker chooses
 $x, y \in \text{SecretInput}$,
 $z \in \text{PublicInput}$
 - 2 System selects $a \in \{x, y\}$ randomly (i.i.d.).
 - 3 Attacker receives public output $p(a, z)$.
 - 4 Attacker guesses whether $a = x$ or $a = y$.

Secure information flow as a game

Parties: the attacker and the system

Assume: Attacker knows program P

Protocol:

- 1 Attacker chooses
 $x, y \in \text{SecretInput}$,
 $z \in \text{PublicInput}$
- 2 System selects $a \in \{x, y\}$ randomly (i.i.d.).
- 3 Attacker receives public output $p(a, z)$.
- 4 Attacker guesses whether $a = x$ or $a = y$.

Winner: Attacker wins this game if they guess a correctly

Secure information flow as a game

Parties: the attacker and the system

Assume: Attacker knows program P

Protocol:

- 1 Attacker chooses
 $x, y \in \text{SecretInput}$,
 $z \in \text{PublicInput}$
- 2 System selects $a \in \{x, y\}$ randomly (i.i.d.).
- 3 Attacker receives public output $p(a, z)$.
- 4 Attacker guesses whether $a = x$ or $a = y$.

Winner: Attacker wins this game if they guess a correctly

→ Program has **secure information flow** if best guessing strategy has winning probability 0.5.

Secure information flow is a hard condition:

- **Attacker may freely choose the secret**
 - even if that value may be unlikely to occur
 - (→ chosen plaintext in crypto)

Secure information flow is a hard condition:

- **Attacker may freely choose the secret**
 - even if that value may be unlikely to occur
 - (→ chosen plaintext in crypto)
- **The winning probability must not deviate from 50%.**
 - 50% are the winning odds for blind guessing.
 - Information gained from public channels still leaves the attacker with same chance.
 - information theoretical security
 - stricter than computational security
(increasing winning probability within negligible polynomial bounds, → IND-CPA in cryptography)

(Goguen and Meseguer, 1982)

Semantic definition

A program $P = (s, p)$ satisfies **noninterference** if a user cannot learn anything about secret input from inspecting public outputs.

Mathematical condition

$$\forall h_1, h_2, l. \quad p(h_1, l) = p(h_2, l)$$

The **public result** p of program P is **independent of** the **secret input**.

Have the following programs the noninterference property?

Quiz

```
class MiniExamples {  
    public int l;  
    private int h;
```

Quiz

```
class MiniExamples {  
    public int l;  
    private int h;  
  
    void m1() {  
        l = h;  
    }  
}
```

Quiz

```
class MiniExamples {  
    public int l;  
    private int h;  
  
    void m1() {  
        l = h;  
    }  
  
    void m2() {  
        if (l > 0) {  
            h=1;  
        } else {  
            h=2;  
        }  
    }  
}
```

Quiz

```
class MiniExamples {  
    public int l;  
    private int h;
```

```
    void m1() {  
        l = h;  
    }
```

```
    void m2() {  
        if (l > 0) {  
            h=1;  
        } else {  
            h=2;  
        }  
    }
```

```
    void m3() {  
        if (h>0) {l=1;}  
        else {l=2;};  
    }
```

Quiz

```
class MiniExamples {  
    public int l;  
    private int h;
```

```
    void m1() {  
        l = h;  
    }
```

```
    void m2() {  
        if (l > 0) {  
            h=1;  
        } else {  
            h=2;  
        }  
    }
```

```
    void m3() {  
        if (h>0) {l=1;}  
        else {l=2;};  
    }
```

```
    void m4() {  
        h=0; l=h;  
    }
```


Quiz

```
class MiniExamples {  
    public int l;  
    private int h;
```

```
    void m1() {  
        l = h;  
    }
```

```
    void m2() {  
        if (l > 0) {  
            h=1;  
        } else {  
            h=2;  
        }  
    }
```

```
    void m3() {  
        if (h>0) {l=1;}  
        else {l=2;};  
    }
```

```
    void m4() {  
        h=0; l=h;  
    }
```

```
    void m5() {  
        while(h == 0) { }  
    }
```

Quiz

```
class MiniExamples {  
    public int l;  
    private int h;
```

```
    void m1() {  
        l = h;  
    }
```

```
    void m2() {  
        if (l > 0) {  
            h=1;  
        } else {  
            h=2;  
        }  
    }
```

```
    void m3() {  
        if (h>0) {l=1;}  
        else {l=2;};  
    }
```

```
    void m4() {  
        h=0; l=h;  
    }
```

```
    void m5() {  
        while(h == 0) { }  
    }
```

```
    void m6() {  
        Thread.sleep(h * 1000);  
    }  
}
```

Sometimes it is ok to leak a bit ... or two

```
private int secretPIN;  
int checkPIN(int triedPIN) {  
    if(secretPIN == triedPIN) {  
        return 1;  
    } else {  
        return 0;  
    }  
}
```

- 1 This method leaks information.
- 2 How much?
- 3 Can this be used to learn about the secret?

Noninterference is often too strict.

Relaxations:

Declassification

Allow particular data to flow

Quantitative analysis

Analyse the amount of secret information that flows

Situation

The attacker must not learn anything but the value of an expression $ex(h, l)$.

$ex(h, l)$ is called **declassified** and no longer secret.

Situation

The attacker must not learn anything but the value of an expression $ex(h, l)$.

$ex(h, l)$ is called **declassified** and no longer secret.

Mathematical condition

$$\forall h_1, h_2, \ell.$$

$$p(h_1, \ell) = p(h_2, \ell)$$

Situation

The attacker must not learn anything but the value of an expression $ex(h, l)$.

$ex(h, l)$ is called **declassified** and no longer secret.

Mathematical condition

$$\forall h_1, h_2, l. ex(h_1, l) = ex(h_2, l) \rightarrow p(h_1, l) = p(h_2, l)$$

Secure information flow as a game (again)

Parties: the attacker and the system

Assume: Attacker knows program P

Protocol:

- 1 Attacker chooses
 $x, y \in \text{SecretInput}$,
 $z \in \text{PublicInput}$,

- 2 System selects $a \in \{x, y\}$ randomly (i.i.d.).
- 3 Attacker receives public output $p(a, z)$.
- 4 Attacker guesses whether $a = x$ or $a = y$.

Winner: Attacker wins this game if they guess a correctly

→ Program has **secure information flow** if best guessing strategy has winning probability 0.5.

Secure information flow as a game (again)

Parties: the attacker and the system

Assume: Attacker knows program. Attacker cannot use ex to discern x and y .

Protocol:

- 1 Attacker chooses $x, y \in \text{SecretInput}$, $z \in \text{PublicInput}$, such that $ex(x, z) = ex(y, z)$
- 2 System selects $a \in \{x, y\}$ randomly (i.i.d.).
- 3 Attacker receives public output $p(a, z)$.
- 4 Attacker guesses whether $a = x$ or $a = y$.

Winner: Attacker wins this game if they guess a correctly

→ Program has **secure information flow** if best guessing strategy has winning probability 0.5.

Declassification in the example

Code

```
private int sec;  
int checkPIN(int try) {  
    if(sec == try) return 1; else return 0;  
}
```

Code

```
private int sec;  
int checkPIN(int try) {  
    if(sec == try) return 1; else return 0;  
}
```

Declassification

It is declassified whether PIN is correct: $ex := sec = try$
(Admissible to learn that PIN is correct if the attacker already has the number.)

Code

```
private int sec;  
int checkPIN(int try) {  
    if(sec == try) return 1; else return 0;  
}
```

Declassification

It is declassified whether PIN is correct: $ex := sec = try$
(Admissible to learn that PIN is correct if the attacker already has the number.)

Proof obligation:

$$\forall sec, sec', try. ((sec = try) \leftrightarrow (sec' = try)) \rightarrow \\ p_{\text{checkPIN}}(sec, try) = p_{\text{checkPIN}}(sec', try)$$

... is valid

Analyse *how much information* flows
not only whether or not it flows.

Examples

```
l = h & 0b0111 /*7*/;
```

Analyse *how much information* flows
not only whether or not it flows.

Examples

```
l = h & 0b0111 /*7*/; leaks 3 bits (of 32).
```

Analyse *how much information* flows
not only whether or not it flows.

Examples

```
l = h & 0b0111 /*7*/; leaks 3 bits (of 32).  
l = h1 ^ h2 ^ h3;
```

Analyse *how much information* flows
not only whether or not it flows.

Examples

```
l = h & 0b0111 /*7*/;   leaks 3 bits (of 32).  
l = h1 ^ h2 ^ h3;      leaks 32 bits (of 96).
```


Analyse *how much information* flows
not only whether or not it flows.

Examples

$l = h \ \& \ 0b0111 \ /*7*/;$ leaks 3 bits (of 32).
 $l = h1 \wedge h2 \wedge h3;$ leaks 32 bits (of 96).

One metric to compute amount of information:
Shannon Entropy H :

$$Pr(r) := \{h \mid p(h) = r\} / SecretSize$$
$$H(L) = \sum_r Pr(r) \cdot \log_2(Pr(r))$$

(other metrics exist and have use cases)

Verification of Noninterference Properties

- ① Dynamic checking
- ② Static verification
 - ① Precise: deductive verification
 - ② Approximative: type systems
 - ③ Approximative: program graph analyses

Semantics of Dynamic Logic

$$s \models [P]\varphi \iff s' \models \varphi \text{ for all } s \text{ with } (s, s') \in \rho_P$$

$[P]\varphi$ means “ φ holds after the execution of P ”.

Deductive verification: Self-composition

Variant P' Let P' be a variant of program P in which every occurrence of every variable x is replaced by x' .

Assumption P has one secret variable h and one public variable ℓ (used for input and output).

Noninterference condition

A program P satisfies noninterference if and only if the formula

$$\forall h, h', \ell, \ell'. \quad \ell = \ell' \rightarrow [P; P']\ell = \ell'$$

is valid.

- Different variable sets, executions independent
- “*Self-composition*”: Sequentially composing (;) the same program (modulo variant) twice.

Loops are difficult to verify: Invariants are needed.

Let $P = \text{beforeLoop}; \text{while}(c) \{ \text{body} \}; \text{afterLoop}$.

The self-composition

```
P;P' = beforeLoop; while(c) { body }; afterLoop ;  
      beforeLoop'; while(c') { body' }; afterLoop'
```

has *two loops*.

Reorder statements to reduce complexity:

```
beforeLoop; beforeLoop';  
while(...) { body; body' };  
afterLoop; afterLoop'
```

is equivalent problem with a single loop.

Coupling invariant (\rightarrow Event-B) is easier to find

(Darvas, Hähnle, Sands 2005)

An alternative condition

A program P satisfies noninterference if and only if the formula

$$\forall \ell. \exists r. \forall h. p(h, \ell) = r$$

is valid.

- Equivalent to $\forall h_1, h_2, \ell. p(h_1, \ell) = p(h_2, \ell)$
(\rightarrow exercise: prove it!)
- Dynamic Logic Proof Obligation: $\forall \ell. \exists r. \forall h. [P](r = \ell)$
- + Only one program execution, reduce complexity.
- How to instantiate the existential quantifier?
(\rightarrow example)

Goal:

Define programming language in which syntactically correct programs have noninterference property.

Goal:

Define programming language in which syntactically correct programs have noninterference property.

Language Grammar:

Variable: $l_1, l_2, \dots, h_1, h_2, \dots$
(fixed security-levels by name)

Goal:

Define programming language in which syntactically correct programs have noninterference property.

Language Grammar:

Variable: $l_1, l_2, \dots, h_1, h_2, \dots$
(fixed security-levels by name)

Expression: Variable | Expression '+' Expression

Goal:

Define programming language in which syntactically correct programs have noninterference property.

Language Grammar:

Variable: $l_1, l_2, \dots, h_1, h_2, \dots$
(fixed security-levels by name)

Expression: Variable | Expression '+' Expression

Command: Variable ':=' Expression

Goal:

Define programming language in which syntactically correct programs have noninterference property.

Language Grammar:

Variable: $l_1, l_2, \dots, h_1, h_2, \dots$
(fixed security-levels by name)

Expression: Variable | Expression '+' Expression

Command: Variable ':=' Expression
| Command ';' Command

Goal:

Define programming language in which syntactically correct programs have noninterference property.

Language Grammar:

Variable: $l_1, l_2, \dots, h_1, h_2, \dots$
(fixed security-levels by name)

Expression: Variable | Expression '+' Expression

Command: Variable ':=' Expression
| Command ';' Command
| if Expression = 0 then Command
else Command end

Goal:

Define programming language in which syntactically correct programs have noninterference property.

Language Grammar:

Variable: $l_1, l_2, \dots, h_1, h_2, \dots$
(fixed security-levels by name)

Expression: Variable | Expression '+' Expression

Command: Variable ':=' Expression
| Command ';' Command
| if Expression = 0 then Command
 else Command end
| while Expression = 0 do Command end

Problem:

Assignment can leak information

For instance: $l_1 := h_1$

Problem:

Assignment can leak information

For instance: $l_1 := h_1$

Solution

Assignments to low variables are forbidden if high variables occur in the expression.

Problem:

Conditinal/Loop can leak information

For instance:

```
if  $h_1 = 0$   
then  $l_1 := 0$   
else  $l_1 := 1$   
end
```

Problem:

Conditinal/Loop can leak information

For instance:

```
if  $h_1 = 0$   
then  $l_1 := 0$   
else  $l_1 := 1$   
end
```

Solution

Assignments to low variables are forbidden in a conditional (if) command if a high variable occurs in the branching condition.

(Similar applies to while loops.)

Type rules

$\overline{exp : high}$

Type rules

$$\overline{exp : high}$$
$$\frac{h_i \notin Vars(exp)}{exp : low}$$

$$\frac{}{exp : high}$$

$$\frac{h_i \notin Vars(exp)}{exp : low}$$

$$\frac{pc \in \{low, high\}}{[pc] \vdash h_i := exp}$$

$$\overline{exp : high}$$

$$\frac{h_i \notin \text{Vars}(exp)}{exp : low}$$

$$\frac{pc \in \{low, high\}}{[pc] \vdash h_i := exp}$$

$$\frac{exp : low}{[low] \vdash l_i := exp}$$

$$\frac{}{exp : high}$$

$$\frac{[high] \vdash comm}{[low] \vdash comm}$$

$$\frac{h_i \notin Vars(exp)}{exp : low}$$

$$\frac{pc \in \{low, high\}}{[pc] \vdash h_i := exp}$$

$$\frac{exp : low}{[low] \vdash \perp_i := exp}$$

$$\frac{}{exp : high}$$

$$\frac{[high] \vdash comm}{[low] \vdash comm}$$

$$\frac{h_i \notin Vars(exp)}{exp : low}$$

$$\frac{[pc] \vdash comm_1 \quad [pc] \vdash comm_2}{[pc] \vdash comm_1 ; comm_2}$$

$$\frac{pc \in \{low, high\}}{[pc] \vdash h_i := exp}$$

$$\frac{exp : low}{[low] \vdash \perp_j := exp}$$

$$\frac{}{exp : high}$$

$$\frac{[high] \vdash comm}{[low] \vdash comm}$$

$$\frac{h_i \notin Vars(exp)}{exp : low}$$

$$\frac{[pc] \vdash comm_1 \quad [pc] \vdash comm_2}{[pc] \vdash comm_1; comm_2}$$

$$\frac{pc \in \{low, high\}}{[pc] \vdash h_i := exp}$$

$$\frac{exp : pc \quad [pc] \vdash th \quad [pc] \vdash el}{[pc] \vdash \text{if } exp = 0 \text{ then } th \text{ else } el}$$

$$\frac{exp : low}{[low] \vdash l_i := exp}$$

$$\frac{}{exp : high}$$

$$\frac{[high] \vdash comm}{[low] \vdash comm}$$

$$\frac{h_i \notin Vars(exp)}{exp : low}$$

$$\frac{[pc] \vdash comm_1 \quad [pc] \vdash comm_2}{[pc] \vdash comm_1; comm_2}$$

$$\frac{pc \in \{low, high\}}{[pc] \vdash h_i := exp}$$

$$\frac{exp : pc \quad [pc] \vdash th \quad [pc] \vdash el}{[pc] \vdash \text{if } exp = 0 \text{ then } th \text{ else } el}$$

$$\frac{exp : low}{[low] \vdash l_i := exp}$$

$$\frac{exp : pc \quad [pc] \vdash comm}{[pc] \vdash \text{while } exp = 0 \text{ do } comm}$$

$$\frac{}{exp : high}$$

$$\frac{[high] \vdash comm}{[low] \vdash comm}$$

$$\frac{h_i \notin Vars(exp)}{exp : low}$$

$$\frac{[pc] \vdash comm_1 \quad [pc] \vdash comm_2}{[pc] \vdash comm_1 ; comm_2}$$

$$\frac{pc \in \{low, high\}}{[pc] \vdash h_i := exp}$$

$$\frac{exp : pc \quad [pc] \vdash th \quad [pc] \vdash el}{[pc] \vdash \text{if } exp = 0 \text{ then } th \text{ else } el}$$

forbid explicit flow

$$\frac{exp : low}{[low] \vdash l_i := exp}$$

$$\frac{exp : pc \quad [pc] \vdash comm}{[pc] \vdash \text{while } exp = 0 \text{ do } comm}$$

$$\frac{}{exp : high}$$

$$\frac{[high] \vdash comm}{[low] \vdash comm}$$

$$\frac{h_i \notin Vars(exp)}{exp : low}$$

$$\frac{[pc] \vdash comm_1 \quad [pc] \vdash comm_2}{[pc] \vdash comm_1; comm_2}$$

$$\frac{pc \in \{low, high\}}{[pc] \vdash h_i := exp}$$

$$\frac{exp : pc \quad [pc] \vdash th \quad [pc] \vdash el}{[pc] \vdash \text{if } exp = 0 \text{ then } th \text{ else } el}$$

$$\frac{exp : low}{[low] \vdash l_i := exp}$$

$$\frac{exp : pc \quad [pc] \vdash comm}{[pc] \vdash \text{while } exp = 0 \text{ do } comm}$$

forbid explicit flow

forbid implicit flow

Type rules

A program P is correctly typed if

$$[pc] \vdash P$$

can be inferred for $pc = low$ or $pc = high$.

A program P is correctly typed if

$$[pc] \vdash P$$

can be inferred for $pc = low$ or $pc = high$.

Theorem

Every correctly typed program has noninterference property.

A program P is correctly typed if

$$[pc] \vdash P$$

can be inferred for $pc = low$ or $pc = high$.

Theorem

Every correctly typed program has noninterference property.

Incompleteness

There are programs which have noninterference property that cannot be typed.

For instance: $\perp_1 := h_1 - h_1$

`http://ifc-challenge.appspot.com`

© 2012 Andrei Sabelfeld and Arnar Birgisson



J O A N A

<http://pp.ipd.kit.edu/projects/joana/>

Some interesting extensions

- more than 2 security levels
(e.g., “public” < “internal” < “secret”)
 - pointers / objects / records / heap data structures
 - exceptions
 - reactive systems (more than one input, one output)
 - termination / timing analysis
 - concurrency
- All research challenges in their own right!

Information flow can be analysed and noninterference verified using formal methods.

- Type systems / graph-based systems scale well (up to 100 kLOC)
- Deductive systems are more precise, can prove more cases
- Declassification of expressions in deductive verification
- Declassification of variables in type systems