Applications of Formal Verification

Verification of Information Flow Properties

Bernhard Beckert · Mattias Ulbrich | SS 2017
Security is everywhere ...
Heartbleed Disaster

- 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)
**Heartbleed – Information Flow**

OpenSSL Heartbeat Request (‘PING’, 12)

|------|---|---|---|---|---|---|---|---|---|---|---|

OPENSSL with 🔥
Heartbleed – Information Flow

OpenSSL Heartbeat Request ('PING', 12)

PING

OPENSSL with 🐜

PING priv = 157
Information Flow Model
Information Flow Model

public

secret
Information Flow Model

Program $P$

- public
- secret
Information Flow Model

Program $P$

public

secret

public

secret
Information Flow Model

Program $P$

public

secret

?
Attacker model

- Attacker communicates with system over public channels
Attacker model

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

- 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 scenarios

<table>
<thead>
<tr>
<th>Attacker is</th>
<th>Public channels are</th>
</tr>
</thead>
<tbody>
<tr>
<td>an agent over the network</td>
<td>network traffic</td>
</tr>
<tr>
<td>another application on same device</td>
<td>shared resources (files),</td>
</tr>
<tr>
<td></td>
<td>interprocess comm.</td>
</tr>
<tr>
<td>program using a library</td>
<td>shared memory,</td>
</tr>
<tr>
<td></td>
<td>method calls</td>
</tr>
</tbody>
</table>

In models:

Attacker’s capabilities expressed by the public channels.
Mathematical model

Every program is a function

\[ P : \text{SecretInput} \times \text{PublicInput} \rightarrow \text{SecretOutput} \times \text{PublicOutput} \]

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

\[
\begin{align*}
    s &: \text{SecretInput} \times \text{PublicInput} \rightarrow \text{SecretOutput} \\
    p &: \text{SecretInput} \times \text{PublicInput} \rightarrow \text{PublicOutput} \\
    P(h, \ell) &= (s(h, \ell), p(h, \ell))
\end{align*}
\]

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 \( \ell \) (\( \ell_1 \) etc.).
Java method

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

h and l serve as input and output variables.
**Example**

**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 can see `l`.
Attacker cannot see `h`.
Example

Java method

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

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

Model

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

Attacker model

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

Java method

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

h and l serve as input and output variables.

Model

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

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:** Attacker 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 $x, y \in \text{SecretInput}$, $z \in \text{PublicInput}$
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: Atacker 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: Atacker 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 $a = y$. 

Program has secure information flow if best guessing strategy has winning probability 0.5.
Secure information flow as a game

Parties: the attacker and the system

Assume: Atacker 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

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:

- Attackers 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)
Noninterference

(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.
Quiz

Have the following programs the noninterference property?
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);
    }
}
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);
    }
}
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);
    }
}
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);
    }
}
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() {
    }

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

```java
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?
Information flow control

Noninterference is often too strict.

Relaxations:

Declassification
- Allow particular data to flow

Quantitative analysis
- Analyse the amount of secret information that flows
Declassification

**Situation**

The attacker must not learn anything but the value of an expression \( \text{ex}(h, l) \).

\( \text{ex}(h, l) \) is called **declassified** and no longer secret.
Declassification

**Situation**

The attacker must not learn anything but the value of an expression $\text{ex}(h, l)$. $\text{ex}(h, l)$ is called **declassified** and no longer secret.

**Mathematical condition**

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

Situation

The attacker must not learn anything but the value of an expression \( \text{ex}(h, l) \).

\( \text{ex}(h, l) \) is called declassified and no longer secret.

Mathematical condition

\[ \forall h_1, h_2, \ell. \text{ex}(h_1, \ell) = \text{ex}(h_2, \ell) \rightarrow p(h_1, \ell) = p(h_2, \ell) \]
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

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

$\rightarrow$ Program has secure information flow if best guessing strategy has winning probability 0.5.
Declassification in the example

**Code**

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

It is declassified whether PIN is correct: 

\[
\text{ex} := \text{sec} = \text{try}
\]

(Admissible to learn that PIN is correct if the attacker already has the number.)

Proof obligation:

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

... is valid
Declassification in the example

Code

```java
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.)
Declassification in the example

Code

```java
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:

\[
\forall sec, sec', try. ((sec = try) \iff (sec' = try)) \rightarrow p_{\text{checkPIN}}(sec, try) = p_{\text{checkPIN}}(sec', try)
\]

...is valid
Quantitative information flow analysis

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

Examples

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

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

$$H(L) = \sum_r \Pr(r) \cdot \log_2(\Pr(r))$$

(Other metrics exist and have use cases)
Quantitative information flow analysis

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

### Examples

\[ l = h \& 0b0111 \ /*7*/; \]

leaks 3 bits (of 32).
Quantitative information flow analysis

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

Examples

\[
l = h \& 0b0111 /*7*/; \text{ leaks 3 bits (of 32).}
l = h1 ^ h2 ^ h3;
\]
Quantitative information flow analysis

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

<table>
<thead>
<tr>
<th>Examples</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>l = h &amp; 0b0111 /*7*/;</code></td>
<td>leaks 3 bits (of 32).</td>
</tr>
<tr>
<td><code>l = h1 ^ h2 ^ h3;</code></td>
<td>leaks 32 bits (of 96).</td>
</tr>
</tbody>
</table>
Quantitative information flow analysis

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

Examples

\[
\begin{align*}
l &= h \& 0b0111 /*7*/; & \text{leaks 3 bits (of 32).} \\
l &= h1 \oplus h2 \oplus h3; & \text{leaks 32 bits (of 96).}
\end{align*}
\]

One metric to compute amount of information:

**Shannon Entropy** \( H \):

\[
Pr(r) := \{h \mid p(h) = r\} / \text{SecretSize}
\]

\[
H(L) = \sum_r Pr(r) \cdot \log_2(Pr(r))
\]

(other metrics exist and have use cases)
Verification of Noninterference Properties
Enforcing Noninterference

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 “\( \varphi \) 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 $\ell$ (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.
Better self-composition

Loops are difficult to verify: Invariants are needed.

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

The self-composition

\[ P;P' = \text{beforeLoop; while}(c) \{ \text{body} \}; \text{afterLoop} ; \]
\[ \text{beforeLoop'; while}(c') \{ \text{body'} \}; \text{afterLoop'} \]

has two loops.

Reorder statements to reduce complexity:

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

is equivalent problem with a single loop.
Coupling invariant (→ Event-B) is easier to find
Alternating Quantifiers

(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.
Security type systems

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

**Language Grammar:**

Variable: \( l_1, l_2, \ldots, h_1, h_2, \ldots \)

(fixed security-levels by name)
Security type systems

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

Language Grammar:

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

Expression: Variable | Expression ‘+’ Expression
Security type systems

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

Language Grammar:

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

Expression:  Variable | Expression ‘+’ Expression

Command:  Variable ‘:=’ Expression
Security type systems

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

Language Grammar:

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

Expression: Variable $|$ Expression ‘+’ Expression

Command: Variable ‘:=’ Expression $|$ Command ‘;’ Command
Security type systems

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

Language Grammar:

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

Expression: Variable \( \mid \) Expression ‘+’ Expression

Command: Variable ‘:=’ Expression \( \mid \) Command ‘;’ Command \( \mid \) if Expression = 0 then Command else Command end
Security type systems

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

Language Grammar:

Variable: $l_1, l_2, \ldots, h_1, h_2, \ldots$
(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:

Conditional/Loop can leak information

For instance:

```plaintext
if h₁ = 0
then l₁ := 0
else l₁ := 1
end
```
Security type system: Implicit flow

Problem:
Conditional/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

\[ exp : \text{high} \]
Type rules

\[
\begin{align*}
\text{exp : high} \\
\ h_i \notin \mathit{Vars(\text{exp})} & \quad \text{exp : low}
\end{align*}
\]
Type rules

\[
\begin{align*}
\text{exp} : \text{high} \\
\text{hi} \notin \text{Vars}(\text{exp}) \\
\text{exp} : \text{low} \\
\text{pc} \in \{\text{low, high}\} \\
[\text{pc}] \vdash \text{hi} := \text{exp}
\end{align*}
\]
Type rules

\[ \frac{}{\text{exp : high}} \]

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

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

\[ \frac{\text{exp : low}}{[\text{low}] l_i := \text{exp}} \]
Type rules

\[
\begin{align*}
\text{exp} : \text{high} & \quad \frac{}{[\text{high}] \vdash \text{comm}} \quad \frac{}{[\text{low}] \vdash \text{comm}} \\
\text{h}_i \not\in Vars(\text{exp}) & \quad \frac{}{\text{exp} : \text{low}} \\
\text{pc} \in \{\text{low}, \text{high}\} & \quad \frac{}{[\text{pc}] \vdash \text{h}_i := \text{exp}} \\
\text{exp} : \text{low} & \quad \frac{}{[\text{low}] \text{l}_i := \text{exp}}
\end{align*}
\]
Type rules

\[
\begin{align*}
\frac{}{\text{exp} : \text{high}} \\
\frac{h_i \notin \text{Vars(\text{exp})}}{\text{exp} : \text{low}} \\
\frac{pc \in \{\text{low, high}\}}{[pc] \vdash \text{comm} \\
\frac{\text{exp} : \text{low}}{[low] \vdash \text{comm}}
\end{align*}
\]

\[
\begin{align*}
\frac{}{[\text{high}] \vdash \text{comm}} \\
\frac{}{[\text{low}] \vdash \text{comm}} \\
\frac{[pc] \vdash \text{comm}_1 \quad [pc] \vdash \text{comm}_2}{[pc] \vdash \text{comm}_1; \text{comm}_2}
\end{align*}
\]
Type rules

\[
\frac{\text{exp} : \text{high}}{
\text{exp} : \text{low}}
\]

\[
\frac{\text{hi} \notin \text{Vars(exp)}}{
\text{exp} : \text{low}}
\]

\[
\frac{\text{pc} \in \{\text{low, high}\}}{
[\text{pc}] \vdash \text{hi} := \text{exp}}
\]

\[
\frac{\text{exp} : \text{low}}{
[\text{low}] \text{l}_i := \text{exp}}
\]

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

\[
\frac{[\text{pc}] \vdash \text{comm}_1 \quad [\text{pc}] \vdash \text{comm}_2}{
[\text{pc}] \vdash \text{comm}_1 ; \text{comm}_2}
\]

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

\[
\frac{\text{exp} : \text{high}}{\text{exp} : \text{low}}
\]
\[
\frac{\text{hi} \notin Vars(\text{exp})}{\text{exp} : \text{low}}
\]
\[
\frac{\text{pc} \in \{\text{low}, \text{high}\}}{[\text{pc}] \vdash \text{hi} := \text{exp}}
\]
\[
\frac{\text{exp} : \text{low}}{[\text{low}] \text{l}_i := \text{exp}}
\]
\[
\frac{[\text{high}] \vdash \text{comm}}{[\text{low}] \vdash \text{comm}}
\]
\[
\frac{[\text{pc}] \vdash \text{comm}_1 \quad [\text{pc}] \vdash \text{comm}_2}{[\text{pc}] \vdash \text{comm}_1; \text{comm}_2}
\]
\[
\frac{\text{exp} : \text{pc} \quad [\text{pc}] \vdash \text{th} \quad [\text{pc}] \vdash \text{el}}{[\text{pc}] \vdash \text{if exp} = 0 \text{ then th else el}}
\]
\[
\frac{\text{exp} : \text{pc} \quad [\text{pc}] \vdash \text{comm}}{[\text{pc}] \vdash \text{while exp} = 0 \text{ do comm}}
\]
Type rules

\[
\begin{align*}
\text{exp : high} & \quad \frac{}{[high] \vdash \text{comm}} \\
\text{exp : low} & \quad \frac{\text{hi} \not\in Vars(\text{exp})}{[low] \vdash \text{comm}} \\
\text{pc} \in \{\text{low, high}\} & \quad \frac{}{[pc] \vdash \text{hi} := \text{exp}} \\
\text{exp : low} & \quad \frac{}{[low]\text{l}_i := \text{exp}} \\
\text{exp : pc} & \quad \frac{[pc] \vdash \text{th} \quad [pc] \vdash \text{el}}{[pc] \vdash \text{if exp = 0 then th else el}} \\
\text{exp : pc} & \quad \frac{[pc] \vdash \text{comm}}{[pc] \vdash \text{while exp = 0 do comm}} \\
\end{align*}
\]
Type rules

\[
\begin{align*}
\text{exp} : \text{high} & \quad \frac{\text{exp} : \text{high}}{} \\
\text{exp} : \text{low} & \quad \frac{\text{hi} \notin \text{Vars}(\text{exp})}{\text{exp} : \text{low}} \\
\text{pc} \in \{\text{low}, \text{high}\} & \quad \frac{\text{pc} \vdash \text{hi} := \text{exp}}{} \\
\text{exp} : \text{low} & \quad \frac{\text{exp} : \text{low}}{\text{[low]}l_i := \text{exp}} \\
\text{[high]} & \vdash \text{comm} \\
\text{[low]} & \vdash \text{comm} \\
\text{[pc]} & \vdash \text{comm}_1 \quad \text{[pc]} & \vdash \text{comm}_2 \\
\text{[pc]} & \vdash \text{comm}_1 ; \text{comm}_2 \\
\text{exp} : \text{pc} & \quad \text{[pc]} \vdash \text{th} \quad \text{[pc]} \vdash \text{el} \\
\text{[pc]} & \vdash \text{if exp} = 0 \text{ then } \text{th} \text{ else } \text{el} \\
\text{exp} : \text{pc} & \quad \text{[pc]} \vdash \text{comm} \\
\text{[pc]} & \vdash \text{while } \text{exp} = 0 \text{ do } \text{comm}
\end{align*}
\]

forbid explicit flow

forbid implicit flow
Type rules

A program $P$ is correctly typed if

$$[pc] \vdash P$$

can be inferred for $pc = \text{low}$ or $pc = \text{high}$.
Type rules

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.
Type rules

A program $P$ is correctly typed if

$$[pc] \vdash P$$

can be inferred for $pc = \text{low}$ or $pc = \text{high}$.

Theorem

Every correctly typed program has noninterference property.

Incompleteness

There are programs which have noninterference property that cannot be typed.
For instance: $l_1 := h_1 - h_1$
Online Challenge

http://ifc-challenge.appspot.com

© 2012 Andrei Sabelfeld and Arnar Birgisson
Graph-based information flow control

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