# Applications of Formal Verification

## Verification of Information Flow Properties

Bernhard Beckert · Mattias Ulbrich | SS 2017

KIT – Institut für Theoretische Informatik

# 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)`

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |
| P | I | N | G | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |

# Heartbleed – Information Flow

OpenSSL Heartbeat Request `('PING', 12)`

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |
| P | I | N | G | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |

OPENSSL with 🐞

# Heartbleed – Information Flow

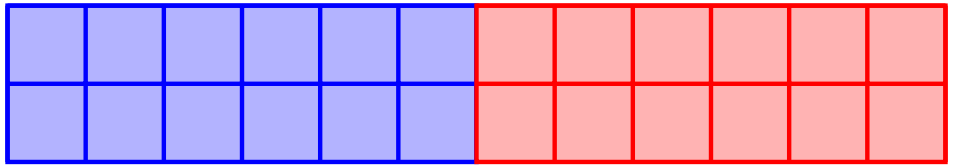
OpenSSL Heartbeat Request `('PING', 12)`

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |
| P | I | N | G | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |

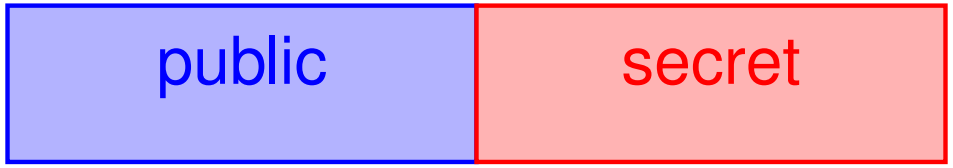
OPENSSL with 🐞

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

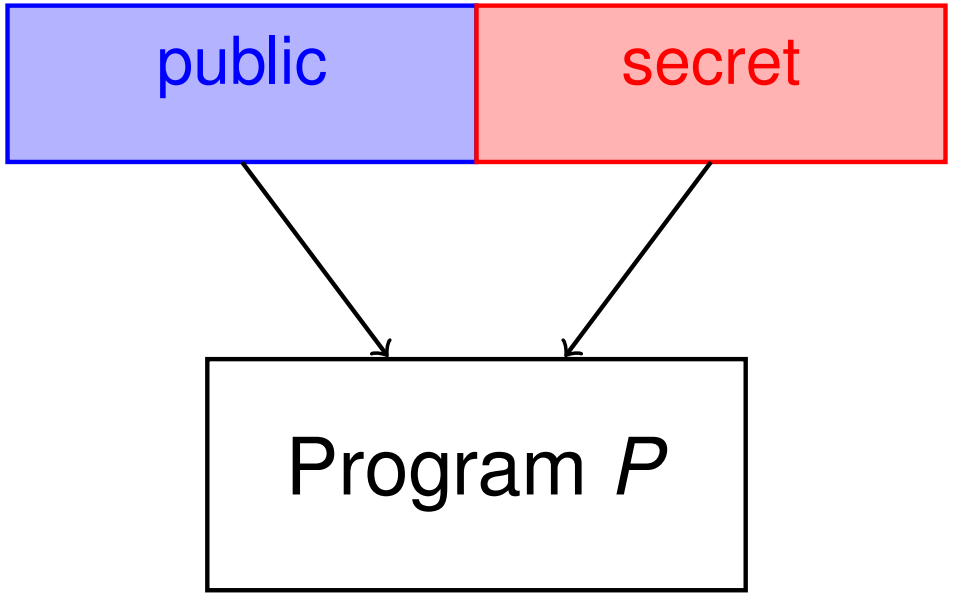
# Information Flow Model



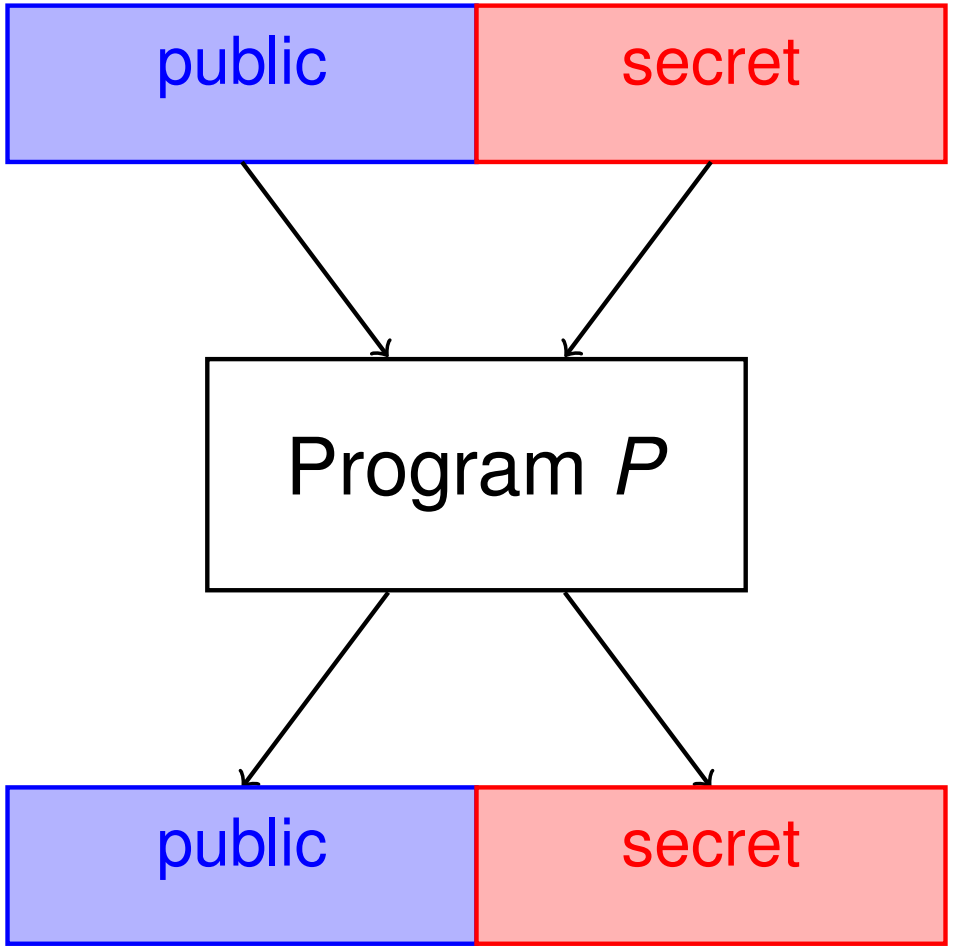
# Information Flow Model



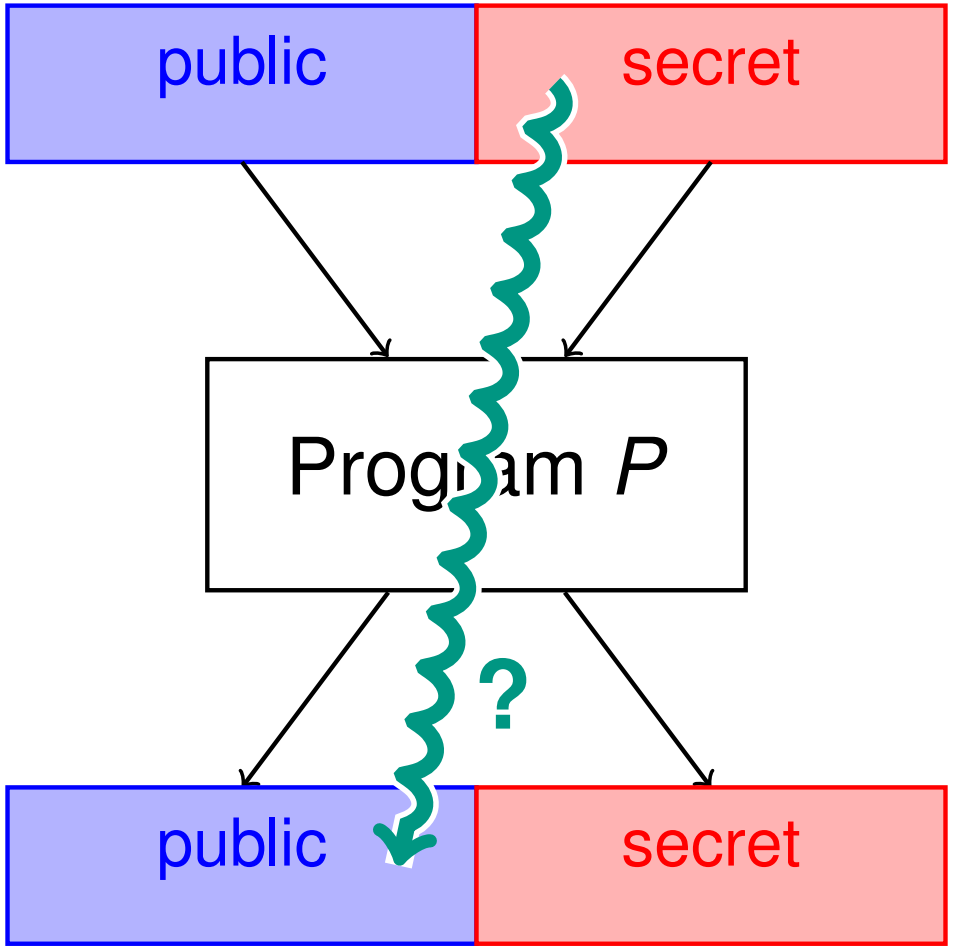
# Information Flow Model



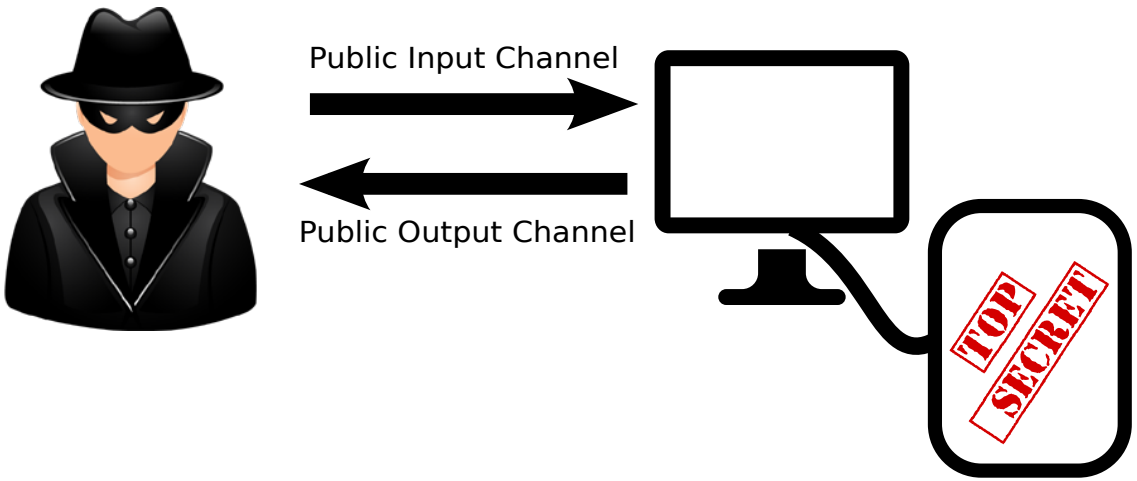
# Information Flow Model



# Information Flow Model

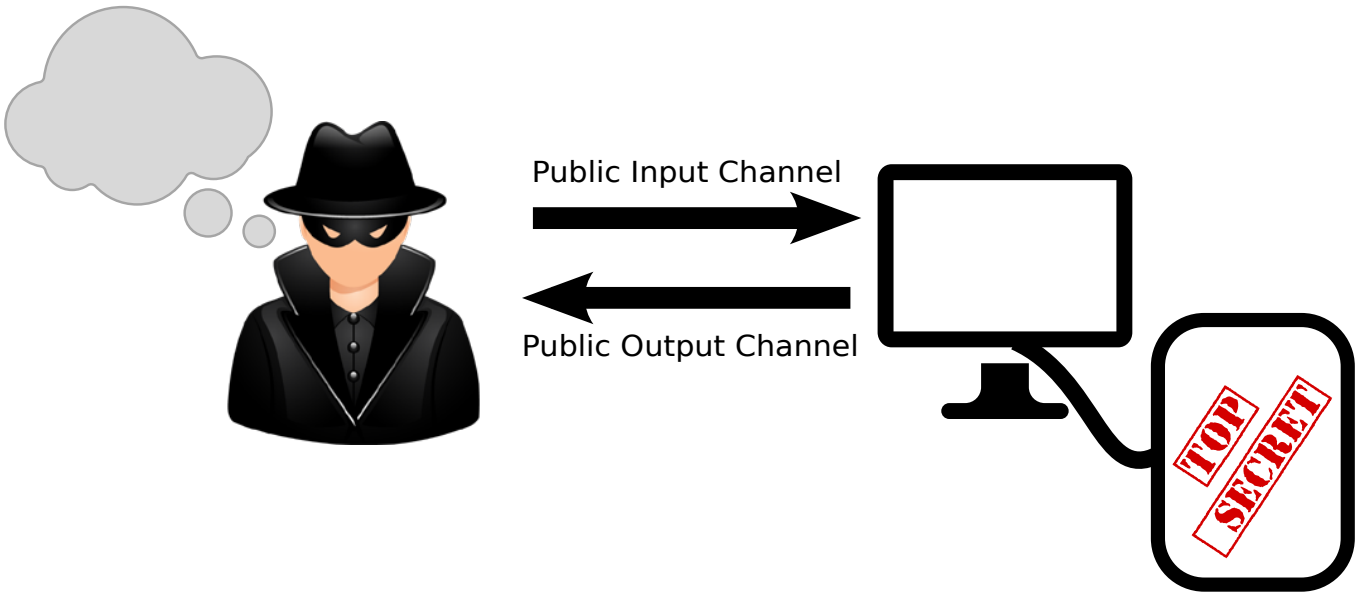


# Attacker model



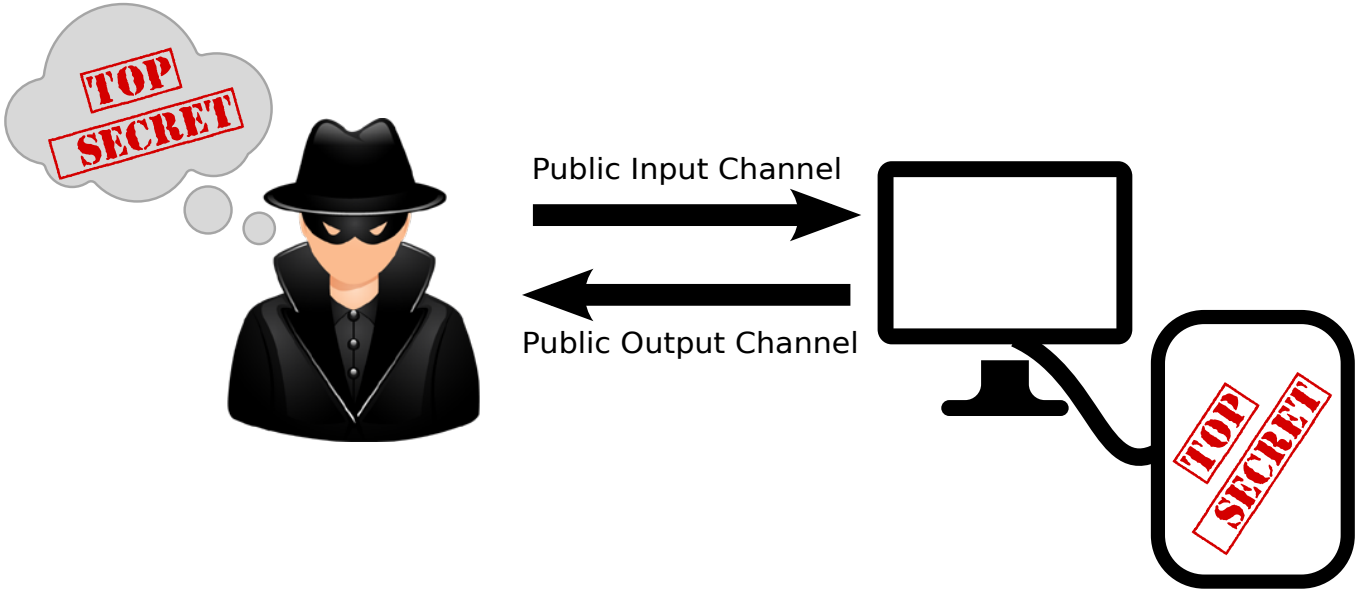

- Attacker communicates with system over public channels

# Attacker model



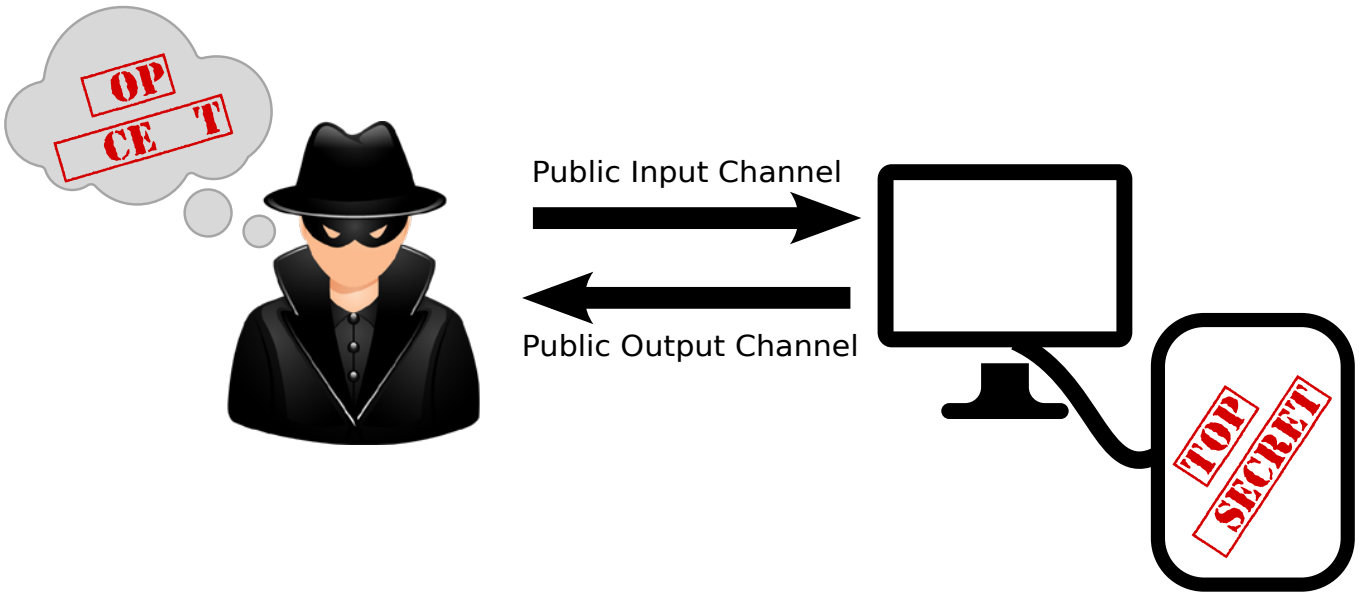

- Attacker communicates with system over public channels
- . . . tries to learn the secret which is kept inside the system

# Attacker model




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

# Attacker model




- 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

| 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*.

# Mathematical model

## Every program is a function

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

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

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

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 $\ell$ ($\ell_1$ 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.

# 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

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

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

# 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 model

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

# Example

## 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: Atacker knows program $P$

# Secure information flow as a game

Parties: the attacker and the system

Assume: Atacker knows program $P$

Protocol:

1. Attacker chooses
   $x, y \in SecretInput$,
   $z \in PublicInput$

# Secure information flow as a game

Parties: the attacker and the system

Assume: Atacker knows program $P$

Protocol:

1. Attacker chooses
   $x, y \in SecretInput$,
   $z \in PublicInput$
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
   $x, y \in SecretInput$,
   $z \in PublicInput$
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
   $x, y \in SecretInput$,
   $z \in PublicInput$
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: Atacker knows program $P$

Protocol:

1. Attacker chooses
   $x, y \in SecretInput$,
   $z \in PublicInput$
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: Atacker knows program $P$

Protocol:

1. Attacker chooses
   $x, y \in SecretInput$,
   $z \in PublicInput$
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 (II)

## Secure information flow is a hard condition:

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

# Secure information flow as a game (II)

## Secure information flow is a hard condition:

- **Attacker may freely choose the secret**
  - even if that value may be unlikely to occur
  - ($\rightarrow$ chosen plaintext in crypto)
- **The winning probablity 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, $\rightarrow$ 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?

# 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?

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

# Declassification

## 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

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

# Declassification

## 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

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

# Secure information flow as a game (again)

Parties: the attacker and the system

Assume: Atacker knows program $P$

Protocol:

1. Attacker chooses
   $x, y \in SecretInput$,
   $z \in PublicInput$,
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: Atacker knows pro

Attacker cannot use *ex* to discern $x$ and $y$.

Protocol:

1. Attacker choo
   $x, y \in SecretInput$,
   $z \in PublicInput$,
   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;
}
```

# Declassification in the example

## 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.)

# Declassification in the example

## 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:

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

...is valid

# Quantitative information flow analysis

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

## Examples

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

# 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*/;
```
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.

## Examples

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

# 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). |
| `l = h1 ^ h2 ^ h3;` | leaks 32 bits (of 96). |

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

$$Pr(r) := \{h \mid p(h) = r\} / SecretSize$$

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

(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

# Dynamic Logic (recap)

## Semantics of Dynamic Logic

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

$[P]\varphi$ 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' = 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

# 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)

# Security type systems

**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: $\mathtt{l}_1, \mathtt{l}_2, \ldots, \mathtt{h}_1, \mathtt{h}_2, \ldots$
(fixed security-levels by name)

# Security type systems

**Goal:**

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

## Language Grammar:

Variable: $\mathtt{l}_1, \mathtt{l}_2, \ldots, \mathtt{h}_1, \mathtt{h}_2, \ldots$
(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: $\mathtt{l}_1, \mathtt{l}_2, \ldots, \mathtt{h}_1, \mathtt{h}_2, \ldots$
(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: | $\mathtt{l}_1, \mathtt{l}_2, \ldots, \mathtt{h}_1, \mathtt{h}_2, \ldots$ (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: $\mathtt{l}_1, \mathtt{l}_2, \ldots, \mathtt{h}_1, \mathtt{h}_2, \ldots$
(fixed security-levels by name)

Expression: Variable | Expression '+' Expression

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

# Security type systems

**Goal:**

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

## Language Grammar:

Variable: $\mathtt{l}_1, \mathtt{l}_2, \ldots, \mathtt{h}_1, \mathtt{h}_2, \ldots$
(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`

# Security type system: Explicit flow

**Problem:**

Assignment can leak information

For instance: $\mathtt{l}_1 := \mathtt{h}_1$

# Security type system: Explicit flow

**Problem:**

Assignment can leak information

For instance: $\mathtt{l}_1 := \mathtt{h}_1$

**Solution**

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

# Security type system: Implicit flow

**Problem:**

Conditinal/Loop can leak information

For instance:

```
if h1 = 0
then l1 := 0
else l1 := 1
end
```

# Security type system: Implicit flow

**Problem:**

Conditinal/Loop can leak information

For instance:

```
if h1 = 0
then l1 := 0
else l1 := 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

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

# Type rules

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

$$\frac{\mathtt{h}_i \notin Vars(exp)}{exp : low}$$

# Type rules

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

$$\frac{\mathtt{h}_i \notin Vars(exp)}{exp : low}$$

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

# Type rules

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

$$\frac{\mathtt{h}_i \notin Vars(exp)}{exp : low}$$

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

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

# Type rules

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

$$\frac{\mathtt{h}_i \notin Vars(exp)}{exp : low}$$

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

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

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

# Type rules

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

$$\frac{\mathtt{h}_i \notin Vars(exp)}{exp : low}$$

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

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

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

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

# Type rules

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

$$\frac{\mathtt{h}_i \notin Vars(exp)}{exp : low}$$

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

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

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

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

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

# Type rules

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

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

$$\frac{\mathtt{h}_i \notin Vars(exp)}{exp : low}$$

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

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

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

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

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

# Type rules

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

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

$$\frac{\mathtt{h}_i \notin Vars(exp)}{exp : low}$$

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

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

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

forbid explicit flow

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

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

# Type rules

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

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

$$\frac{\mathtt{h}_i \notin Vars(exp)}{exp : low}$$

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

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

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

forbid explicit flow

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

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

forbid implicit flow

# Type rules

A program $P$ is correctly typed if

$$[pc] \vdash P$$

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

# 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 $pc = low$ 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: $\mathtt{l}_1 := \mathtt{h}_1 - \mathtt{h}_1$

# Online Challenge

http://ifc-challenge.appspot.com

# Graph-based information flow control

**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!

# Summary

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