LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Bachelor Seminar "Logic for Security"
Winter Semester 2026/27

A Short Introduction to Logic for Security

Dr. Michael Kirsten

Chair of Theoretical Computer Science and
Theorem Proving

(Partially based on slides by Bruno Blanchet, Christoph Scheben, and Vladimir Klebanov)

Version of October 27, 2026



Why verify security protocols ?

The verification of security protocols has been and is still a very active research area.

- Their design is error prone.
- Security errors are not detected by testing:
  they appear only in the presence of an adversary.
- Errors can have serious consequences.

Models of protocols

Active attacker:

- the attacker can intercept all messages sent on the network
- he can compute messages
- he can send messages on the network

Models of protocols: the symbolic model

The symbolic model or "Dolev-Yao model" is due to Needham and Schroeder [1978] and Dolev and Yao [1983].

- The cryptographic primitives are blackboxes.
- The messages are terms on these primitives.
  $\hookrightarrow \{m\}_k$ encryption of the message m with key k,
  $\hookrightarrow (m_1, m_2)$ pairing of messages m_1 and m_2, ...
- The attacker is restricted to compute only using these primitives.
  $\Rightarrow$ perfect cryptography assumption

One can add equations between primitives, but in any case, one makes the hypothesis that the only equalities are those given by these equations.

This model makes automatic proofs relatively easy
(Tamarin, ProVerif, AVISPA, ...).

Models of protocols: the computational model

The computational model has been developed at the beginning of the 1980's by Goldwasser, Micali, Rivest, Yao, and others.

- The messages are bitstrings.
- The cryptographic primitives are functions on bitstrings.
- The attacker is any probabilistic (polynomial-time) Turing machine.

This model is much more realistic than the symbolic model, but until recently proofs were only manual.

Models of protocols: side channels

The computational model is still just a model, which does not exactly match reality.

In particular, it ignores side channels:

- timing
- power consumption
- noise
- physical attacks against smart cards

which can give additional information.

Security properties: trace and equivalence properties

- Trace properties: properties that can be defined on a trace.
  - Symbolic model: they hold when they are true for all traces.
  - Computational model: they hold when they are true except for a set of traces of negligible probability.
- Equivalence (or indistinguishability) properties: the attacker cannot distinguish two protocols (with overwhelming probability)
  - Give compositional proofs.
  - Hard to prove in the symbolic model.

Security properties: secrecy

The attacker cannot obtain information on the secrets.

- Symbolic model:
  - (syntactic) secrecy: the attacker cannot obtain the secret (trace property)
  - strong secrecy: the attacker cannot distinguish when the value of the secrecy changes (equivalence property)
- Computational model: the attacker can distinguish the secret from a random number only with negligible probability (equivalence property)

Security properties: authentication

If A thinks she is talking to B, then B thinks he is talking to A, with the same protocol parameters.

- Symbolic model: formalized using correspondence assertions of the form "if some event has been executed, then some other events have been executed" (trace property).
- Computational model: matching conversations or session identifiers, which essentially require that the messages exchanged by A and B are the same up to negligible probability (trace property).

Verifying protocols in the symbolic model

Main idea (for most verifiers):

- Compute the knowledge of the attacker.

Difficulty: security protocols are infinite state.

- The attacker can create messages of unbounded size.
- Unbounded number of sessions of the protocol.

Relevance of the symbolic model

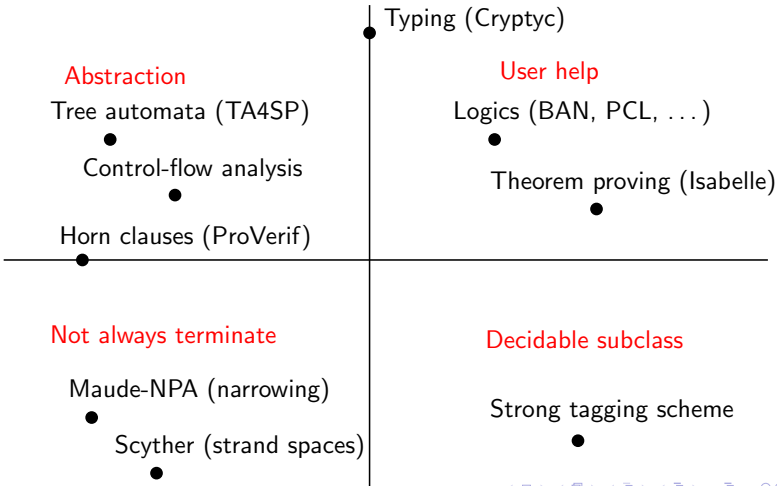
- Numerous attacks have already been obtained.
- An attack in the symbolic model immediately implies one in the computational model (and a practical attack).
  - A proof in the symbolic model does not always imply a proof in the computational model (see next).
- Allows us to perform automatic verification.

Solutions to undecidability

To solve an undecidable problem, we can

- Use approximations, abstraction.
- Not always terminate.
- Rely on user interaction or annotations.
- Consider a decidable subclass.

Solutions to undecidability



Proofs in the computational model

Proofs by reduction:

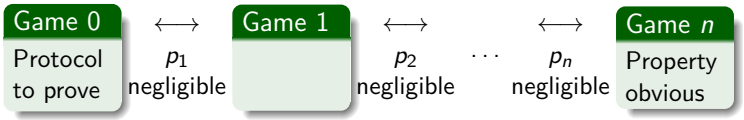
> *from an adversary that breaks the protocol,*
> *one can build an adversary that solves a difficult mathematical problem.*

Difficulty: protocols typically rely on several cryptographic primitives, each relying on the difficulty of a different problem.

Proofs in the computational model

Proofs by sequences of games [Shoup, Bellare&Rogaway]:

- The first game is the real protocol.
- One goes from one game to the next by syntactic transformations or by reducing to a security assumption on a cryptographic primitive. The difference of probability between consecutive games is negligible.
- The last game is "ideal": the security property is obvious from the form of the game.
  (The advantage of the adversary is 0 for this game.)



Link between the two models

- Computational soundness theorems:

  Proof in the symbolic model $\Rightarrow$ proof in the computational model

  modulo additional assumptions.

  Approach pioneered by Abadi&Rogaway [2000]; many works since then.

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)



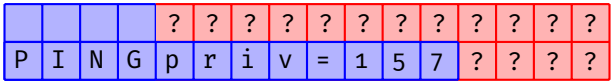
OPENSSL with

Heartbleed – Information Flow

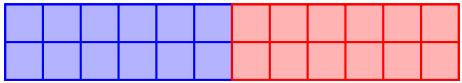
OpenSSL Heartbeat Request (‘PING’, 12)



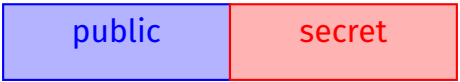
OPENSSL with 



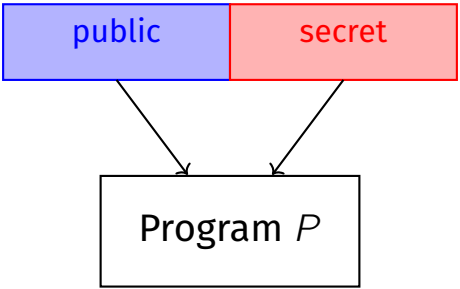
Information Flow Model



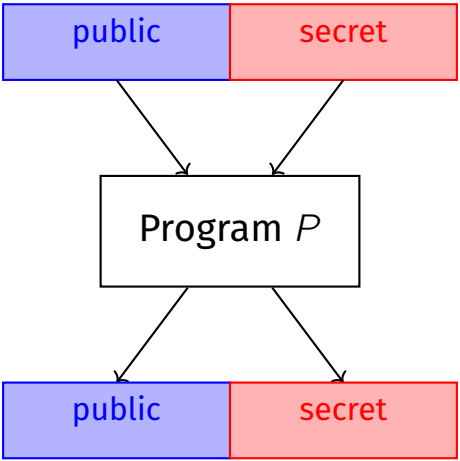
Information Flow Model



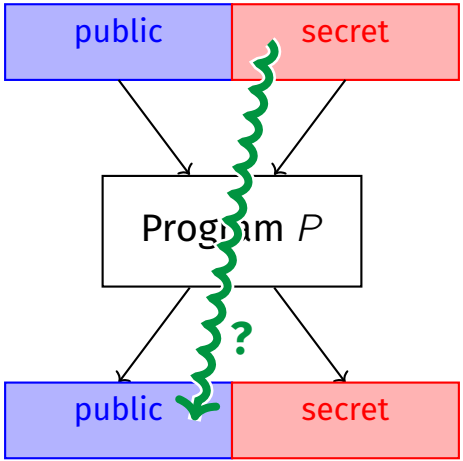
Information Flow Model



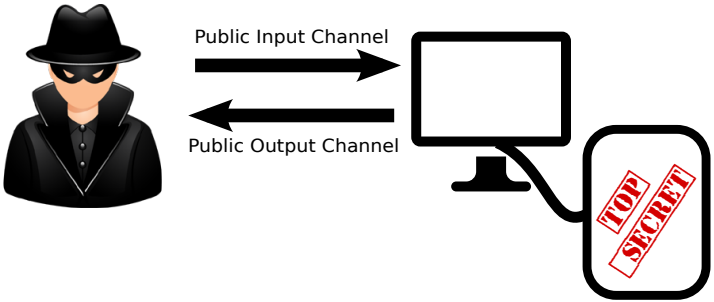
Information Flow Model



Information Flow Model

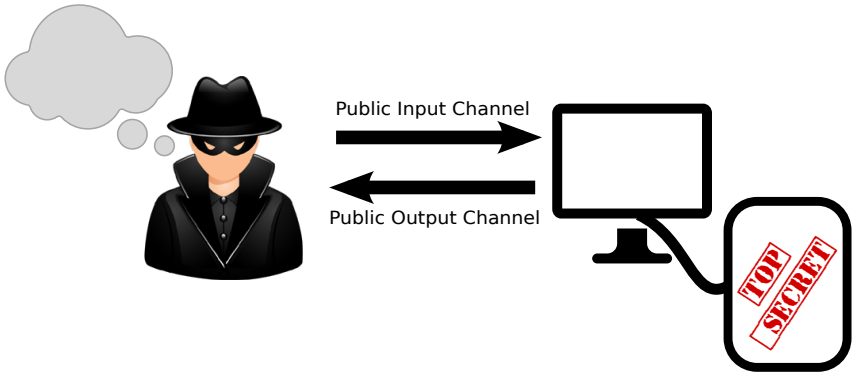


Attacker model



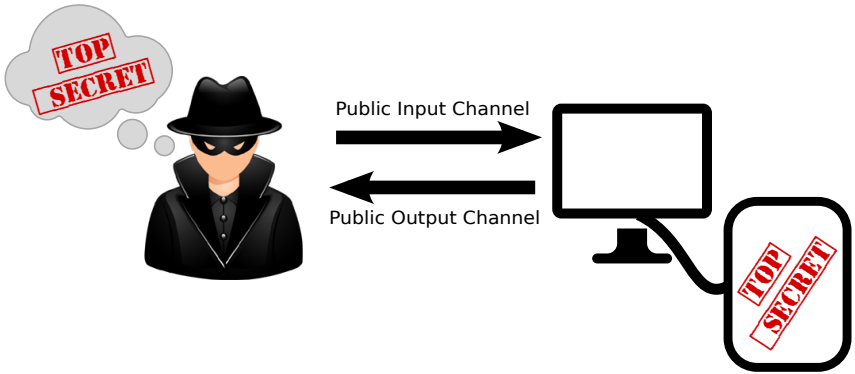
- Attacker communicates with system over public channels

Attacker model



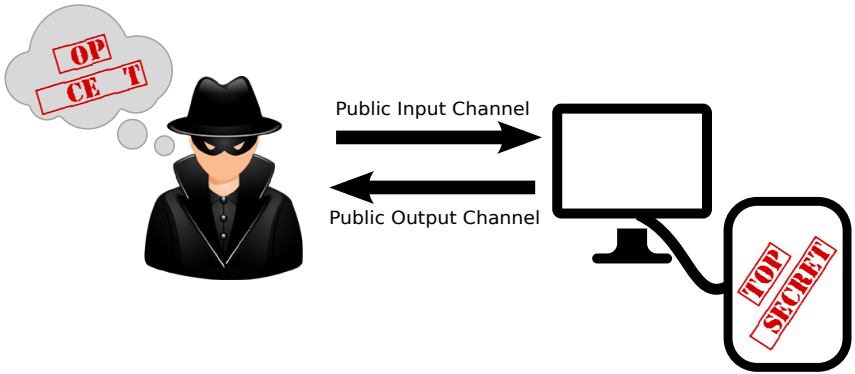
- Attacker communicates with system over public channels
- …tries to learn the secret which is kept inside the system

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
a user at a console	monitor and keyboard

In models:
Attacker's capabilities expressed by the *public channels*.

Mathematical model

Every program is a function

$$P : \mathsf{SecretInput} \times \mathsf{PublicInput} \to \mathsf{SecretOutput} \times \mathsf{PublicOutput}$$

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

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

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 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.

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: 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
   $x, y \in \mathsf{SecretInput}$,
   $z \in \mathsf{PublicInput}$

Secure information flow as a game

Parties: the attacker and the system

Assume: Attacker knows program P

Protocol:

1. Attacker chooses
   $x, y \in \mathsf{SecretInput}$,
   $z \in \mathsf{PublicInput}$
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
   $x, y \in \mathsf{SecretInput}$,
   $z \in \mathsf{PublicInput}$
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
   $x, y \in \mathsf{SecretInput}$,
   $z \in \mathsf{PublicInput}$
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
   $x, y \in \mathsf{SecretInput}$,
   $z \in \mathsf{PublicInput}$
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 can tell a correctly.

Secure information flow as a game (II)

Attacker knowledge:
Attacker knows all relevant information: P, x, y, z

Secure information flow as a game (II)

Attacker knowledge:
Attacker knows all relevant information: P, x, y, z

Two possibilities:

$p(x, z) = p(y, z)$

Attacker *cannot know* if $a = x$ or $a = y$.

They must guess.

Secure information flow as a game (II)

Attacker knowledge:
Attacker knows all relevant information: P, x, y, z

Two possibilities:

$p(x, z) = p(y, z)$	$p(x, z) \neq p(y, z)$
Attacker *cannot know* if $a = x$ or $a = y$. They must guess.	Attacker *knows* if $a = x$ or $a = y$ (by computing both). They can answer correctly.

Secure information flow as a game (II)

Attacker knowledge:
Attacker knows all relevant information: P, x, y, z

Two possibilities:

$p(x,z) = p(y,z)$	$p(x,z) \neq p(y,z)$
Attacker *cannot know* if $a = x$ or $a = y$. They must guess.	Attacker *knows* if $a = x$ or $a = y$ (by computing both). They can answer correctly.

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

Secure information flow as a game (III)

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 (III)

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

Do the following programs have 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
   $x, y \in \mathsf{SecretInput}$,
   $z \in \mathsf{PublicInput}$,
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

Protocol:

1. Attacker choo
   $x, y \in \mathsf{SecretInput}$,
   $z \in \mathsf{PublicInput}$,
   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 $a = y$.

Attacker cannot use ex to discern x and 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: $\mathsf{ex} := \mathsf{sec} = \mathsf{try}$
(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: $\mathsf{ex} := \mathsf{sec} = \mathsf{try}$
(Admissible to learn that PIN is correct if the attacker already has the number.)

Proof obligation:

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

…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:

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

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

Noninterference Definition

Definition (Low-Equivalence on States)

Two states are low-equiv. if they assign the same values to low variables.

Noninterference Definition

Definition (Low-Equivalence on States)

Two states are low-equiv. if they assign the same values to low variables.

Definition (Noninterference)

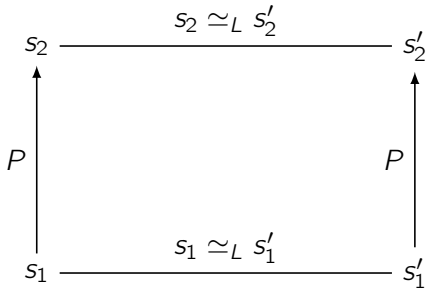
Starting a program P in two arbitrary low-equivalent states results in two final states that are also low-equivalent.

Noninterference Semantics

- P a program
- L the set of low variables
- s_1, s_1' two low-equiv. ($\simeq_L$) program states
- s_2, s_2' the two final states

Noninterference Semantics

- P a program
- L the set of low variables
- s_1, s_1' two low-equiv. ($\simeq_L$) program states
- s_2, s_2' the two final states



where

$$s_i \simeq_L s_i' \quad \Leftrightarrow \quad \forall\, v \in L\ (v^{s_i} = v^{s_i'})$$

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 $\mid$ 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 | 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:

Conditional/Loop can leak information

For instance:

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

Security type system: Implicit flow

Problem:

Conditional/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

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

Type rules

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

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

Type rules

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

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

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

Type rules

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

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

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

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

Type rules

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

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

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

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

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

Type rules

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

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

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

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

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

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

Type rules

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

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

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

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

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

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

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

Type rules

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

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

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

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

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

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

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

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

Type rules

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

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

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

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

forbid explicit flow

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

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

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

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

Type rules

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

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

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

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

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

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

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

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

forbid explicit flow

forbid implicit flow

Type rules

A program P is correctly typed if

$$[\mathrm{pc}] \vdash P$$

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

Type rules

A program P is correctly typed if

$$[\mathrm{pc}] \vdash P$$

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

Theorem

Every correctly typed program has noninterference property.

Type rules

A program P is correctly typed if

$$[\mathrm{pc}] \vdash P$$

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

Theorem

Every correctly typed program has noninterference property.

Incompleteness

There are programs which have noninterference property that cannot be typed.
For instance: $\mathtt{l}_1 := \mathtt{h}_1 - \mathtt{h}_1$

References I

[1] Martín Abadi and Phillip Rogaway. “Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption)”. In: *Journal of Cryptology* 15.2 (2002), pp. 103–127. DOI: 10.1007/S00145-001-0014-7.

[2] Mihir Bellare and Phillip Rogaway. “The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs”. In: *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings*. Ed. by Serge Vaudenay. Vol. 4004. Lecture Notes in Computer Science. Springer, 2006, pp. 409–426. DOI: 10.1007/11761679_25.

[3] Danny Dolev and Andrew Chi-Chih Yao. “On the security of public key protocols”. In: *IEEE Transactions on Information Theory* 29.2 (1983), pp. 198–207. DOI: 10.1109/TIT.1983.1056650.

References II

[4] Roger M. Needham and Michael D. Schroeder. “Using Encryption for Authentication in Large Networks of Computers”. In: *Communications of the ACM (CACM)* 21.12 (1978), pp. 993–999. DOI: 10.1145/359657.359659.

[5] Victor Shoup. “Sequences of games: a tool for taming complexity in security proofs”. In: *IACR Cryptology ePrint Archive* 2004 (2004), p. 332. URL: http://eprint.iacr.org/2004/332.