Applications of Formal Verification

Verification of Information Flow Properties

Dr. Vladimir Klebanov · Dr. Mattias Ulbrich | SS 2015
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.
OpenSSL Heartbeat Request (‘PING’, 12)

OPENSSL with 🐛

PING priv = 157
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), interprocess comm.</td>
</tr>
<tr>
<td>program using a library</td>
<td>shared memory, 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 \( P = (s, p) \)

- \( s : \text{SecretInput} \times \text{PublicInput} \rightarrow \text{SecretOutput} \)
- \( p : \text{SecretInput} \times \text{PublicInput} \rightarrow \text{PublicOutput} \)

\[ P(h, \ell) = (s(h, \ell), p(h, \ell)) \]

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

### Convention

Variables with high security status are named \( h (h_1 \text{ etc.}) \) and variables with low (public) security status are named \( \ell (\ell_1 \text{ etc.}) \).
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

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

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

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

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 (II)

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?
Quiz

```java
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
Analyze 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.

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: Atacker 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;
}
```

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
Quantitative information flow analysis

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

Examples

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

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

The self-composition

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

has two loops.

Reorder statements to reduce complexity:

\[ \text{beforeLoop}; \text{beforeLoop'}; \text{while}(\ldots) \{ \text{body'}; \text{body'} \}; \text{afterLoop}; \text{afterLoop'} \]

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.

Language Grammar:

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

Expression: \( \text{Variable} \mid \text{Expression} + \text{Expression} \)

Command: \( \text{Variable} \ := \ \text{Expression} \)  
\( \mid \text{Command} ; \text{Command} \)  
\( \mid \text{if Expression} = 0 \text{ then Command} \)  
\( \text{else Command} \) end  
\( \mid \text{while Expression} = 0 \text{ do Command} \) end
Security type system: Explicit flow

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:
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

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

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

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

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

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

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

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

\[
\frac{\text{exp} : \text{pc} \quad \text{[pc]} \vdash \text{comm}}{\quad \text{[pc]} \vdash \text{while exp = 0 do comm}}
\]
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: $l_1 := h_1 - h_1$
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!
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