Formal Methods – An Appetizer

Chapter 5: Language-Based Security

Flemming Nielson, Hanne Riis Nielson:
Formal Methods – An Appetizer.
ISBN 9783030051556, Springer 2019.

FormalMethods.dk

Three Components of Security

Confidentiality
Data is only told to those we trust: *private* data are not made *public*

Integrity
Data is only influenced by those we trust: *trusted* data are not influenced by *dubious* data

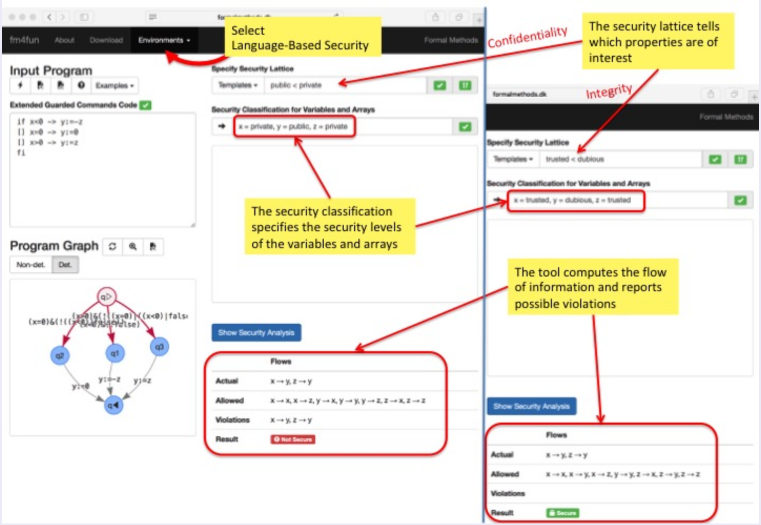
Availability
Data is not inaccessible when needed

The Role of Formal Methods
We are exploiting that program analysis techniques can be used to give *guarantees* about the behaviour of models and programs.

Here the guarantees are concerned with confidentiality and integrity properties.

The Techniques Are
- Fully automatic
- Approximative
- Efficient

FormalMethods.dk/fm4fun



Section 1

5.1 Information Flow

Flemming Nielson, Hanne Riis Nielson:
Formal Methods – An Appetizer.
ISBN 9783030051556, Springer 2019.

`FormalMethods.dk`

Ensuring Confidentiality By Information Flow

Idea: We want to limit the information flow allowed by the program.

```
if  x < 0 → y := −z
[]  x = 0 → y := 0
[]  x > 0 → y := z
fi
```

Security Classification

Assume that x and z are private but y is public.

Confidentiality (FM p 61)

We specify which data is private and which is public and want to ensure that private data do not end up in public data.

Explicit Flow

The assignment $y := z$ violates the confidentiality of z with respect to y.

Implicit Flow

The conditional $x = 0 \rightarrow y := 0$ violates the confidentiality of x with respect to y.

Ensuring Integrity By Information Flow

Idea: We want to limit the information flow allowed by the program.

$$\begin{array}{ll} \texttt{if} & \texttt{x} < 0 \rightarrow \texttt{y} := -\texttt{z} \\ [] & \texttt{x} = 0 \rightarrow \texttt{y} := 0 \\ [] & \texttt{x} > 0 \rightarrow \texttt{y} := \texttt{z} \\ \texttt{fi} & \end{array}$$

Security Classification

Assume that $\texttt{x}$ and $\texttt{z}$ are dubious but $\texttt{y}$ is trusted.

Integrity (FM p 62)

We spedify which data is trusted and which is dubious and want to ensure that trusted date is not influenced by dubious data.

Explicit Flow

The assignment $\texttt{y} := \texttt{z}$ violates the integrity of $\texttt{y}$ with respect to $\texttt{z}$.

Implicit Flow

The conditional $\texttt{x} = 0 \rightarrow \texttt{y} := 0$ violates the integrity of $\texttt{y}$ with respect to $\texttt{x}$.

Flow Relation

The Security Policy

Which flows are permissible:

- for confidentiality: $x \rightarrow y$ is allowed if x is public or y is private; otherwise $x \not\rightarrow y$
- for integrity: $x \rightarrow y$ is allowed if x is trusted or y is dubious; otherwise $x \not\rightarrow y$

The Program

Which flows happen:

- explicit flows in assignments
- implicit flows in conditionals

Is The Program Secure?

Are the possible flows of the program included in those of the security policy?

Program

```
if  x < 0 → y := −z
[]  x = 0 → y := 0
[]  x > 0 → y := z
fi
```

Program Flows

$\mathtt{x} \rightarrow \mathtt{y}$

$\mathtt{z} \rightarrow \mathtt{y}$

Security Policy

$\mathtt{x}$ is private

$\mathtt{y}$,$\mathtt{z}$ are public

Permitted by Policy

$$\{\mathtt{x}, \mathtt{y}, \mathtt{z}\} \rightrightarrows \{\mathtt{x}\}$$
$$\{\mathtt{y}, \mathtt{z}\} \rightrightarrows \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$$

$X \rightrightarrows Y$ means $\forall x \in X : \forall y \in Y : x \rightarrow y$

Hands On: Flow Relations

Assume that Alice wants to keep the contents of the array A (of size n) confidential from Bob and that Bob wants to keep the contents of the array B (of size m) confidential from Alice.

FormalMethods.dk/fm4fun

A variable (or an array) may be

- shared between Alice and Bob
- private to Alice (as for example A)
- private to Bob (as for example B)
- public to everybody

Find a security policy for the program and use the tool to check that the flows of program are admitted by the policy.

Example (FM p 63)

$$
\begin{array}{l}
\mathtt{i} := 0; \\
\mathtt{j} := 0; \\
\mathtt{do}\ (\mathtt{i} < \mathtt{n}) \land (\mathtt{j} = \mathtt{m} \lor \mathtt{i} < \mathtt{j}) \to \\
\qquad\qquad \mathtt{A}[\mathtt{i}] := \mathtt{A}[\mathtt{i}] + 27; \\
\qquad\qquad \mathtt{i} := \mathtt{i} + 1 \\
[]\ \ (\mathtt{j} < \mathtt{m}) \land (\mathtt{i} = \mathtt{n} \lor \mathtt{i} \geq \mathtt{j}) \to \\
\qquad\qquad \mathtt{B}[\mathtt{j}] := \mathtt{B}[\mathtt{j}] + 12; \\
\qquad\qquad \mathtt{j} := \mathtt{j} + 1 \\
\mathtt{od}
\end{array}
$$

In the tool you should specify

$$
\begin{array}{rl}
\text{Alice} < \text{shared} & \text{public} < \text{Alice} \\
\text{Bob} < \text{shared} & \text{public} < \text{Bob}
\end{array}
$$

in order to perform the analysis.

Section 2

5.2 Reference-Monitor Semantics

Flemming Nielson, Hanne Riis Nielson:
Formal Methods – An Appetizer.
ISBN 9783030051556, Springer 2019.

FormalMethods.dk

Using The Semantics

Idea: Given a set of flows permitted by the security policy, we can let the semantics check that the actual flows are permitted.

Checking Explicit Flows

$$\mathcal{S}[\![x := a]\!]\sigma = \begin{cases} \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma] & \text{if } \mathcal{A}[\![a]\!]\sigma \text{ is defined} \\ & \text{and } \mathsf{fv}(a) \rightrightarrows \{x\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

We check that $\mathsf{fv}(a) \rightrightarrows \{x\}$ is permitted by the policy

$\mathsf{fv}(a)$ is the set of variables occurring in a

Program

$$\begin{array}{ll} \mathtt{if} & \mathtt{x} < 0 \rightarrow \mathtt{y} := -\mathtt{z} \\ [\,] & \mathtt{x} = 0 \rightarrow \mathtt{y} := 0 \\ [\,] & \mathtt{x} > 0 \rightarrow \mathtt{y} := \mathtt{z} \\ \mathtt{fi} & \end{array}$$

Permitted by Policy

$$\{\mathtt{x},\mathtt{y},\mathtt{z}\} \rightrightarrows \{\mathtt{x}\}$$
$$\{\mathtt{y},\mathtt{z}\} \rightrightarrows \{\mathtt{x},\mathtt{y},\mathtt{z}\}$$

Insufficient to check $\mathtt{z} \rightarrow \mathtt{y}$

Checking Implicit Flows

We need to know the variables giving rise to implicit dependencies in order to make all checks.

Instrumented Program Graphs

Idea: Construct program graphs that record the implicit dependencies.

Replace actions $x := a$ with $x := a\{X\}$

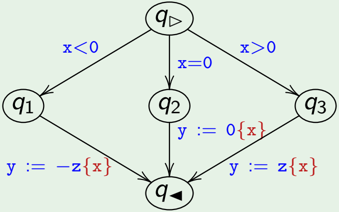
Instrumented Program Graphs (FM p 64)

The program graphs are deterministic and have actions of the form `skip`, b, $x := a\{X\}$ and $A[a_1] := a_2\{X\}$. To do so we keep track of

- the set X of implicit dependencies; it is updated when passing through tests
- the tests that previously have been passed; it is updated when inspecting the guarded commands

Program

```
if  x < 0 → y := −z
[]  x = 0 → y := 0
[]  x > 0 → y := z
fi
```



Two Semantics

Reference Monitor Semantics (FM p 65)

Checks implicit and explicit flows:

$$\mathcal{S}_{\mathrm{rm}}[\![x := a\{X\}]\!]\sigma = \begin{cases} \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma] \\ \qquad \text{if } \mathcal{A}[\![a]\!]\sigma \text{ is defined} \\ \qquad \text{and } X \cup \mathrm{fv}(a) \rightrightarrows \{x\} \\ \text{undefined otherwise} \end{cases}$$

Standard Semantics (FM p 65)

Does not check any flows:

$$\mathcal{S}[\![x := a\{X\}]\!]\sigma = \begin{cases} \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma] \\ \qquad \text{if } \mathcal{A}[\![a]\!]\sigma \text{ is defined} \\ \text{undefined otherwise} \end{cases}$$

The Relationship Between The Two Semantics (FM p 67)

- Whenever we have an execution sequence in the reference monitor semantics then we have the same execution sequence in the standard semantics
- There might be execution sequences in the standard semantics with no counterpart in the reference monitor semantics

Try It Out

Give an example of the latter

Section 3

5.3 Security Analysis

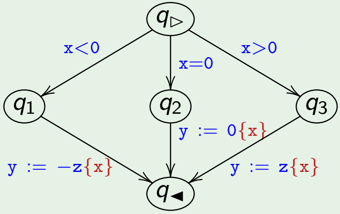
Flemming Nielson, Hanne Riis Nielson:
Formal Methods – An Appetizer.
ISBN 9783030051556, Springer 2019.

`FormalMethods.dk`

A Security Analysis

Dynamic Check of Security

The reference monitor semantics may reveal security information, if we are told where the execution of the program is stopped!



If we are stopped in q_2 because $\mathtt{x} \not\rightarrow \mathtt{y}$ then we have learned the sign of $\mathtt{x}$.

Static Checks of Security

Idea: We perform the necessary checks *before* running the program.

We prove *once for all* that if the program passes the security checks then the reference monitor will never stop the execution of the program – and hence it is no longer needed.

Security Analysis (FM p 67, 68)

sec$[\![C]\!](X)$: C is secure with respect to the set X of implicit dependencies

sec$[\![x := a]\!](X)$

if $X \cup \mathsf{fv}(a) \rightrightarrows \{x\}$

sec$[\![A[a_1] := a_2]\!](X)$

if $X \cup \mathsf{fv}(a_1) \cup \mathsf{fv}(a_2) \rightrightarrows \{A\}$

sec$[\![C_1 \,;\, \cdots \,;\, C_k]\!](X)$

$$\text{if } \begin{cases} \mathbf{sec}[\![C_1]\!](X) \wedge \\ \mathbf{sec}[\![C_2]\!](X) \wedge \\ \vdots \\ \mathbf{sec}[\![C_k]\!](X) \end{cases}$$

sec$[\![\texttt{if } b_1 \rightarrow C_1 \,[]\, \cdots \,[]\, b_k \rightarrow C_k \texttt{ fi}]\!](X)$

$$\text{if } \begin{cases} \mathbf{sec}[\![C_1]\!](X \cup \mathsf{fv}(b_1)) \wedge \\ \mathbf{sec}[\![C_2]\!](X \cup \mathsf{fv}(b_1) \cup \mathsf{fv}(b_2)) \wedge \\ \vdots \\ \mathbf{sec}[\![C_k]\!](X \cup \mathsf{fv}(b_1) \cup \cdots \cup \mathsf{fv}(b_k)) \end{cases}$$

sec$[\![\texttt{do } b_1 \rightarrow C_1 \,[]\, \cdots \,[]\, b_k \rightarrow C_k \texttt{ od}]\!](X)$

$$\text{if } \begin{cases} \mathbf{sec}[\![C_1]\!](X \cup \mathsf{fv}(b_1)) \wedge \\ \mathbf{sec}[\![C_2]\!](X \cup \mathsf{fv}(b_1) \cup \mathsf{fv}(b_2)) \wedge \\ \vdots \\ \mathbf{sec}[\![C_k]\!](X \cup \mathsf{fv}(b_1) \cup \cdots \cup \mathsf{fv}(b_k)) \end{cases}$$

Hands On: Security Analysis

Assume that the contents of the array `A` (of size `n`) is confidential whereas the contents of the array `B` (of size `m`) is public.

Confidentiality Policy

- `A`, `n`, `i` are private
- `B`, `m`, `j` are public

$x \to y$ is allowed if x is public or y is private.

Example (FM p 63)

```
i := 0;
j := 0;
do (i < n) ∧ (j = m ∨ i < j) →
        A[i] := A[i] + 27;
        i := i + 1
[]  (j < m) ∧ (i = n ∨ i ≥ j) →
        B[j] := B[j] + 12;
        j := j + 1
od
```

FormalMethods.dk/fm4fun

Use the tool to compute

- the actual flows: $\mathbf{sec}[\![C]\!](\{\,\})$
- the allowed flows of the policy

The actual flows must be allowed in order for the program to be secure.

Modify the security policy in order to obtain a secure program.

The Security Analysis Fulfills Its Promises

Instrumented Program Graphs (FM p 64)

Program graph with actions recording implicit dependencies: $x := a\{X\}$

Reference Monitor $\Longrightarrow_{\mathsf{rm}}$ (FM p 65)

Checks implicit and explicit flows:

$$\mathcal{S}_{\mathsf{rm}}[\![x := a\{X\}]\!]\sigma = \begin{cases} \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma] \\ \qquad \text{if } \mathcal{A}[\![a]\!]\sigma \text{ is defined} \\ \qquad \text{and } X \cup \mathsf{fv}(a) \rightrightarrows \{x\} \\ \text{undefined otherwise} \end{cases}$$

Standard Semantics $\Longrightarrow$ (FM p 65)

Does not check any flows:

$$\mathcal{S}[\![x := a\{X\}]\!]\sigma = \begin{cases} \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma] \\ \qquad \text{if } \mathcal{A}[\![a]\!]\sigma \text{ is defined} \\ \text{undefined otherwise} \end{cases}$$

Analysis $\mathbf{sec}[\![C]\!](X)$ (FM p 67, 68)

The analysis checks the explicit and implicit flows: $\mathbf{sec}[\![x := a]\!](X)$ if $X \cup \mathsf{fv}(a) \rightrightarrows \{x\}$

Proposition: The Reference Monitor Is Obsolete (FM p 69)

Assume $\mathbf{sec}[\![C]\!](\{\,\})$ and $\langle q_{\triangleright}; \sigma\rangle \Longrightarrow^* \langle q; \sigma'\rangle$. Then $\langle q_{\triangleright}; \sigma\rangle \Longrightarrow^*_{\mathsf{rm}} \langle q; \sigma'\rangle$.

Different Kinds Of Information Flow (FM p 69)

We may learn information about the value of a variable by observing the values of other variables or by observing the behaviour of the program.

Explicit Flows

$y := x$

Implicit Flows

$$\begin{array}{ll} \texttt{if} & x > 0 \rightarrow y := 1 \\ [] & x = 0 \rightarrow y := 0 \\ [] & x < 0 \rightarrow y := -1 \\ \texttt{fi} & \end{array}$$

Try It Out

What do we learn about x in each of the cases? – and how do we learn it?

Observing The Execution Time

$$\begin{array}{ll} y := 0; & \\ z := x; & \\ \texttt{do} & z > 0 \rightarrow z := z - 1 \\ [] & z < 0 \rightarrow z := z + 1 \\ \texttt{od} & \end{array}$$

Observing Non-determinism

$$\begin{array}{ll} y := 0; & \\ \texttt{if} & \texttt{true} \rightarrow y := 1 \\ [] & x = 0 \rightarrow \texttt{skip} \\ \texttt{fi} & \end{array}$$

Section 4

5.4 Multi-Level Security

Flemming Nielson, Hanne Riis Nielson:
Formal Methods – An Appetizer.
ISBN 9783030051556, Springer 2019.

FormalMethods.dk

Security Lattice

Idea: A security lattice specifies the security levels and an ordering between them telling how information may flow between the levels.

$(L, \sqsubseteq)$ is a partially ordered set:

- reflexive: $\forall l \in L : l \sqsubseteq l$
- transitive: $\forall l_1, l_2, l_3 \in L : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3$
- antisymmetric: $\forall l_1, l_2 \in L : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2$

$(L, \sqsubseteq)$ is a lattice if it is a partially ordered satisfying:

- the maximum (or join) of two elements written $l_1 \sqcup l_2$ exists and satisfies $\forall l \in L : (l_1 \sqsubseteq l \wedge l_2 \sqsubseteq l \Leftrightarrow l_1 \sqcup l_2 \sqsubseteq l)$
- the minimum (or meet) of two elements written $l_1 \sqcap l_2$ exists and satisfies $\forall l \in L : (l \sqsubseteq l_1 \wedge l \sqsubseteq l_2 \Leftrightarrow l \sqsubseteq l_1 \sqcap l_2)$

Example (FM p 70)

private
|
public

public $\sqsubseteq$ private

Hasse diagram

Security Lattices: Examples

Confidentiality (FM p 70)

$$
\begin{tikzcd}[arrows=-]
\text{private} \arrow[d] \\
\text{public}
\end{tikzcd}
$$

Confidentiality (FM p 71)

$$
\begin{tikzcd}[arrows=-]
\text{top secret} \arrow[d] \\
\text{secret} \arrow[d] \\
\text{classified} \arrow[d] \\
\text{unclassified}
\end{tikzcd}
$$

Confidentiality (FM p 72)

$$
\begin{tikzcd}[arrows=-]
& \{\text{financial}, \text{medical}\} \arrow[dl] \arrow[dr] & \\
\{\text{financial}\} \arrow[dr] & & \{\text{medical}\} \arrow[dl] \\
& \{\,\} &
\end{tikzcd}
$$

Integrity (FM p 70)

$$
\begin{tikzcd}[arrows=-]
\text{dubious} \arrow[d] \\
\text{trusted}
\end{tikzcd}
$$

Integrity (FM p 72)

$$
\begin{tikzcd}[arrows=-]
\text{alternative fact} \arrow[d] \\
\text{conjecture} \arrow[d] \\
\text{known fact}
\end{tikzcd}
$$

Integrity (FM p 72)

$$
\begin{tikzcd}[arrows=-]
& \{\text{host}, \text{client}\} \arrow[dl] \arrow[dr] & \\
\{\text{host}\} \arrow[dr] & & \{\text{client}\} \arrow[dl] \\
& \{\,\} &
\end{tikzcd}
$$

Security Classification

Idea: The security classification assigns security levels to all the variables and arrays. Together with the security lattice this specifies the flows permitted by the security policy.

Security Classification

An assignment of security levels to the variables and arrays:

$$\mathbf{L} : (\mathbf{Var} \cup \mathbf{Arr}) \to L$$

Flows Permitted By Policy

The security classification and the security lattice define the permitted flows:

$$x \to y \text{ if } \mathbf{L}(x) \sqsubseteq \mathbf{L}(y)$$

Security Lattice

$$\begin{array}{c} \text{private} \\ | \\ \text{public} \end{array}$$

Security Classification

$$\begin{aligned} \mathbf{L}(\mathtt{x}) &= \text{private} \\ \mathbf{L}(\mathtt{y}) &= \text{public} \\ \mathbf{L}(\mathtt{z}) &= \text{public} \end{aligned}$$

Flows Permitted By Policy

$$\begin{array}{ccc} & \mathtt{x} \to \mathtt{x} & \\ \mathtt{y} \to \mathtt{x} & \mathtt{y} \to \mathtt{y} & \mathtt{y} \to \mathtt{z} \\ \mathtt{z} \to \mathtt{x} & \mathtt{z} \to \mathtt{z} & \mathtt{z} \to \mathtt{y} \end{array}$$

Hands On: Security Policy

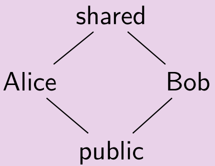
Assume that Alice wants to keep the contents of the array `A` (of size `n`) confidential from Bob and that Bob wants to keep the contents of the array `B` (of size `m`) confidential from Alice.

Example (FM p 63)

```
i := 0;
j := 0;
do (i < n) ∧ (j = m ∨ i < j) →
          A[i] := A[i] + 27;
          i := i + 1
[] (j < m) ∧ (i = n ∨ i ≥ j) →
          B[j] := B[j] + 12;
          j := j + 1
od
```

FormalMethods.dk/fm4fun

Specify the following security lattice:



Specify a security classification **L** with **L**(`A`) = Alice and **L**(`B`) = Bob.

Use the tool to compute the flows permitted by the policy and explain the outcome of the security check.

Section 5

5.5 Non-Interference

Flemming Nielson, Hanne Riis Nielson:
Formal Methods – An Appetizer.
ISBN 9783030051556, Springer 2019.

FormalMethods.dk

The Challenge: To Get It Right!

Experience shows that it is very hard to get security analysis correct.

There is a need for techniques that can scrutinize our analyses – a non-interference result is one such technique.

Example

It is easy to overlook that our analysis only works for deterministic programs!

Non-Interference Result

Assume that $\mathbf{sec}[\![C]\!](\{\,\})$ holds and that we start the execution from memories that are equal on *all* variables in $Y = \{y \mid y \to x\}$.

Then the executions will agree on the final value of x.

$$\begin{tikzcd} \langle q_{\triangleright}\,;\sigma_1\rangle \arrow[r, "\Longrightarrow^*", phantom] \arrow[d, dash, dotted, "\sigma_1 =_Y \sigma_2"'] & \langle q_{\blacktriangleleft}\,;\sigma_1'\rangle \arrow[d, dash, dotted, "\sigma_1'(x) = \sigma_2'(x)"] \\ \langle q_{\triangleright}\,;\sigma_2\rangle \arrow[r, "\Longrightarrow^*", phantom] & \langle q_{\blacktriangleleft}\,;\sigma_2'\rangle \end{tikzcd}$$

$\sigma_1 =_Y \sigma_2$ means $\forall y \in Y : \sigma_1(y) = \sigma_2(y)$