Information-Flow Security

Farzane Karami
Supervisor: Olaf Owe

Department of Informatics, University of Oslo

GEMINI IoT PhD-Seminar 15 May 2019
Introduction

- Information-flow security
- Controlling how information is propagated by a system
- Preventing dissemination of confidential information
Introduction

- Information-flow security
- Controlling how information is propagated by a system
- Preventing dissemination of confidential information
- Access control
Introduction

- Information-flow security
- Controlling how information is propagated by a system
- Preventing dissemination of confidential information
- Access control
- Making sure that the program handles information securely
Information-flow security

- A language-based technique
- Tracking flow of information during a program execution
- Preventing leakage of confidential information

- An attacker is able to observe public outputs of a program
- Public outputs must be independent of secret inputs
Information-flow security

- A language-based technique
- Tracking flow of information during a program execution
- Preventing leakage of confidential information

- An attacker is able to observe public outputs of a program
- Public outputs must be independent of secret inputs
- Noninterference semantics [1]:
  - In two executions, a program is run with different secret inputs but the same public values, the public outputs will be the same.
  - An attacker cannot see any difference between these executions
Information-flow security

- Two kinds of flow of information
  - Explicit flow: \( l := h \)
  - Implicit flow:
    \[
    l := \text{true}; \quad \text{if } h \text{ then } l := \text{false}; \quad \text{else skip};
    \]
Information-flow security

Note: Techniques for enforcing information-flow security [2]
▶ Static secure type-systems:
Information-flow security

**Note**: Techniques for enforcing information-flow security [2]

- Static secure type-systems:
  - The types of program variables and expressions are augmented with security levels
  - Typing rules:
    - $\vdash exp : high$
    - $h \notin exp$
    - $\vdash exp : low$
    - $exp : low$
    - $[low] \vdash l := exp$

---

UiO: University of Oslo
Information-flow security

Note: Techniques for enforcing information-flow security [2]

- Static secure type-systems:
  - The types of program variables and expressions are augmented with security levels
  - Typing rules:
    - \( \vdash exp : high \)
    - \( h \notin exp \)
    - \( \vdash exp : low \)
    - \( exp : low \)
    - \( [low] \vdash l := exp \)
  - Compiler
Information-flow security

Note: Techniques for enforcing information-flow security [2]

- Static secure type-systems:
  - The types of program variables and expressions are augmented with security levels
  - Typing rules:
    - $\vdash exp : high$
    - $h \notin exp$
    - $\vdash exp : low$
    - $exp : low$
    - $[low] \vdash l := exp$
- Compiler
- Dynamic analysis: security checks are performed at run-time
Static vs dynamic enforcement

Static techniques:
- Less runtime overhead
- Conservative

Dynamic techniques:
- More runtime overhead
- The exact secrecy levels are available → more precise
- More permissive

\[
\text{if } l < 0 \text{ then } l := 1; \text{ else } l := h;
\]

<table>
<thead>
<tr>
<th></th>
<th>ST</th>
<th>DT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run-time efficiency</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Exact security and permissiveness</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>
Static vs dynamic enforcement

- **Static techniques:**
  - Less runtime overhead
  - Conservative

- **Dynamic techniques:**
  - More runtime overhead
  - The exact secrecy levels are available → more precise
  - More permissive

\[
\text{if } l < 0 \text{ then } l := 1; \text{ else } l := h;
\]

<table>
<thead>
<tr>
<th></th>
<th>ST</th>
<th>DT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run-time efficiency</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>Exact security and</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>permissiveness</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Information-flow security & Active object languages

- Distributed systems
- Active object languages
  - Scala/Akka
  - ABS/Creol
  - Rebeca
  - Encore
  - ASP
Information-flow security & Active object languages

- Distributed systems
- Active object languages
  - Scala/Akka
  - ABS/Creol
  - Rebeca
  - Encore
  - ASP

- **Goal**: To enforce information-flow security in a program
- Security aspects highly depend on communication paradigms between autonomous nodes
Active object languages

What are active object languages?

- A specific category of concurrent programming languages
- Active objects are created with their own threads, behaving autonomously
- They communicate with each other through method calls
  - Asynchronous call (one-way): \( o!m(e) \)
  - Synchronous call (two-way): \( x:=o.m(e) \)
Communication paradigms

- Future mechanism: A flexible way of sharing results
Future mechanism: A flexible way of sharing results

- Futures: \( f =: o!m(e) \)
- A future is a placeholder created as a result of an asynchronous and remote method call
- Eventually contains the result of the method call
- When the caller needs the future value it requests it
First-class futures

\[ f = o! m(e) \]
First-class futures

Future is resolved
Wrappers

- Here we exploit the notion of wrapper to enforce information-flow security
- A wrapper is a kind of membrane defined around an object
- A wrapper controls security levels of communicated messages
- Preventing sending secret data to low level objects
- Confidentiality of a future
Run-time elements: objects

\[ O \]

- Code (statements)
- Fields
- Local variables
Run-time elements: objects

```
O

Code (statements)

Fields

Local variables
```

```
O

Code (statements)

f := O!m(e_H)

Fields

Local variables
```
Run-time elements: futures

\[ u \]
Run-time elements: futures

\[ u \]

\[ u \]

\[ V_L \]
Run-time elements: futures

\[ u \]

\[ V_L \]

\[ V_H \]
Invocation message / Caller side

- If at least one of the actual parameters is high
Invocation message / Caller side

- If at least one of the actual parameters is high

\[ \text{invoc} (u, m, d_H), \text{to } O \]

\[ \text{invoc} (u, m, d_H), \text{to } O \]

\[ \text{invoc} (u, m), \text{to } O \]

\[ \text{invoc} (u, m), \text{to } O \]
Invocation message / Callee side

invoc \( (u, m, d) \), to \( O \)
Method call / Callee side

\[ \text{return } e \]

Fields

Local variables
\text{callId } \rightarrow u

[u]

[\[e\] = \(V_H\)]

O

idle

Fields

Empty

O

\(V_H\)

\(u\)
Method call / Callee side

\[ \text{return } e \]

\[ \text{idle} \]

\[ [e] = V \]

Local variables
\[ \text{callId } \rightarrow u \]

\[ \text{Fields} \]

\[ \text{Empty} \]

\[ u \]

\[ V \]
Get operation

\[ x := \text{get } f \]

\[ f \rightarrow u \]

\[ x := V_L \]

\[ u \]

\[ V \]
Get operation

\[ x := \text{get } f \]

\[
\begin{array}{c}
\text{Fields} \\
\text{Local variables}
\end{array}
\]

\[ f \rightarrow u \]

\[
\begin{array}{c}
\text{Fields} \\
\text{Local variables}
\end{array}
\]

\[ \text{if } H \subseteq \text{lev}(O) \]

\[ x := V_H \]

\[
\begin{array}{c}
\text{Fields} \\
\text{Local variables}
\end{array}
\]

\[ u \]

\[ V_H \]
Get operation

\[ x := \text{get } f \]

\[ f \rightarrow u \]

\[ \text{if } H \not\in \text{lev}(O) \]

\[ \text{raise exception} \]
Conclusion

- A wrapper enforce dynamic information-flow security
- Runt-time checking for all objects in a system \(\rightarrow\) run-time overhead
- By combination of static analysis with dynamic checking to have less run-time overhead
- If statically it is shown that an object is safe \(\rightarrow\) it does not a wrapper for run-time checking
References


Thank You! :)