CFlow
A Demo with a Pinch of Theory

Cédric Fournet, Gurvan Le Guernic, Tamara Rezk

INRIA - Microsoft Research Joint Center
gleguern@gmail.com

January 29th, 2010
1. Introduction

2. Source Language

3. How it works

4. What about security?

5. Conclusion
Introduction
**Goal**

- Simplify (and not enforce) programming of *distributed* and *secured* softwares

- Source language: simple sequential language
  - globally shared memory
    - accessible from any host
  - annotations for code distribution
    - where to execute every statement
  - security level given to every global variable
    - specifies who can read and/or write

- Target language: real world language (F♯)
  - communication between hosts through TCP/IP
  - encryptions and signatures to protect globals
Security in the source

- Accessibility based on security lattices

- IF label \( \in ( \text{confidentiality lattice } \times \text{integrity lattice} ) \)
  - \( l_1 \Rightarrow l_2 \iff ( l_1 \leq C l_2 \land l_2 \leq I l_1 ) \)
  - \( x := y \iff y \Rightarrow x \)
  - A can read \( x \iff x \Rightarrow C A \)
  - A can write \( x \iff A \Rightarrow I x \)

- Security lattices are compiler plugins (2 already coded)
  - HL: \( 2 \times \) flat lattice with top and bottom
  - \( \{ L \leq C [^\text{HL}] < C H \} \times \{ L \leq I [^\text{HL}] < I H \} \)
  - ReadersWriters: 1 set of readers and 1 set of writers (\( R \), \( W \))
    - \( R_1 \leq C R_2 \iff R_2 \subseteq R_1 \)
    - \( W_1 \leq I W_2 \iff W_2 \subseteq W_1 \)
Source Language
• Define the security lattice used: \texttt{SLattice HL};
  • the compiler loads the appropriate plugin to manipulate strings corresponding to security labels

• Define the roles: \texttt{Role \#HH\# A};
  • all roles in the execution environment
    • A, B: secured line or VPN between A and B
    • A, B, others: any network with “outsiders” connected
  • compiler protects against the attacker level, either:
    • \texttt{Role \#LL\# attacker};
    • stronger weakest than all roles

• Define globals: \texttt{global string(64) \#HH\# message};
Program body

\[
\begin{align*}
e & ::= x \mid op(e_1, \ldots, e_n) \\
S & ::= \text{skip} \mid x := e \mid S ; S \\
    & \quad \mid \text{if } e \text{ then } S \text{ else } S \text{ end} \\
    & \quad \mid \text{while } e \text{ do } S \text{ done} \\
    & \quad \mid A : [S]
\end{align*}
\]

- \textit{statement localization}
- \textit{means: role A executes S}
- \textit{can be nested}
Let's write a simple chat program
How it works
A 4-steps process

- Slicing: cut into uniquely localized threads

- Control Flow Protocol: prevent thread reordering
  - check pc set by previous “visible” threads

- Variable Replication: compute with thread locals

- Encrypting & Signing: enforce security labels of globals
A 4-steps process

- Slicing: cut into uniquely localized threads
  - do: compute threads’ integrities

- Control Flow Protocol: prevent thread reordering
  - check pc set by previous “visible” threads
  - need: to have integrity assigned to threads

- Variable Replication: compute with thread locals

- Encrypting & Signing: enforce security labels of globals
A 4-steps process

- **Slicing**: cut into uniquely localized threads
  - *do*: compute threads’ integrities
  - *do*: meta-threads loop indexes instantiated

- **Control Flow Protocol**: prevent thread reordering
  - check *pc* set by previous “visible” threads
  - *need*: to have integrity assigned to threads

- **Variable Replication**: compute with thread locals
  - *do*: SSA-like: each local assigned by unique thread

- **Encrypting & Signing**: enforce security labels of globals
  - *need*: a unique tag to sign and verify
A 4-steps process

- Slicing: cut into uniquely localized threads
  - *do*: compute threads’ integrities
  - *do*: meta-threads loop indexes instantiated
  - *ensure*: static previous call graph until same host

- Control Flow Protocol: prevent thread reordering
  - check *pc* set by previous “visible” threads
  - *need*: to have integrity assigned to threads

- Variable Replication: compute with thread locals
  - *do*: SSA-like: each local assigned by unique thread
  - *need*: every thread statically knows who last wrote read variables

- Encrypting & Signing: enforce security labels of globals
  - *need*: a unique tag to sign and verify
A 4-steps process

- **Slicing**: cut into uniquely localized threads
  - \textit{do}: compute threads’ integrities
  - \textit{do}: meta-threads loop indexes instantiated
  - \textit{ensure}: static previous call graph until same host

- **Control Flow Protocol**: prevent thread reordering
  - check \textit{pc} set by previous “visible” threads
  - \textit{need}: to have integrity assigned to threads

- **Variable Replication**: compute with thread locals
  - \textit{do}: SSA-like: each local assigned by unique thread
  - \textit{need}: every thread statically knows who last wrote read variables
  - \textit{do}: assigned globals transfer at merge points

- **Encrypting & Signing**: enforce security labels of globals
  - \textit{need}: a unique tag to sign and verify
SLICING
**Control Flow Protocol**

Diagram showing control flow protocol with threads and execution flow.
Control Flow Protocol

Thread integrity

Execution

\[ t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_4 \rightarrow t_5 \rightarrow t_6 \rightarrow t_7 \]
Control Flow Protocol
goal: statically know assigning thread if remote assignment

single remote last assignment

SSA-like transformation

trick: merging threads write in merger locals

```
check (a8 i j. pc1) simeq ("a8", [i; j]) do {
   b5 i j. pc2 := ("b5", [i; j]);
   if ((a8 i j. y) mod 2) = 1
   then {b5 i j. x := (a1 i j. x) + 9}
   else {skip; b5 i j. x := a1 i j. x};
   call(a4 i j) }
```
Cryptographic Protection

- ensure IF policy
- encrypt and sign variables sent on the network
- select adequate keys
- use thread id as tag to compute MAC

\[
\text{check} \quad \text{Verify}(b.pc1_s, "a8."} ^ i ^ ." ^ j ^ .pc1", b.pc1_{mc}, K^s_{1HL}) \quad \text{do} \quad \{ \\
\text{check} \quad \text{Verify}(b.y_s, "a8."} ^ i ^ ." ^ j ^ .y", b.y_{mc}, K^s_{1HL}) \quad \text{do} \quad \{
\]
\[
b.x_{mc} := \text{Decrypt}(b.x_e, K^e_{1HL});
\]
\[
b.x := \text{Unmarshal}(b.x_{mc});
\]
\[
b.y := \text{Unmarshal}(b.y_{mc});
\]
\[
b.pc1 := \text{Unmarshal}(b.pc1_{mc});
\]
\[
\text{check} \quad b.pc1 \cong ("a8", [i; j]) \quad \text{do} \quad \{
\]
\[
b.pc2 := ("b5", [i; j]);
\]
\[
\text{if} \quad (b.y \mod 2) = 1 \\
\text{then} \quad \{b.x := b.x + 9\}
\]
\[
\text{else} \quad \{b.x := b.x\};
\]
\[
b.x_{mc} := \text{Marshal}(b.x);
\]
\[
b.pc2_{mc} := \text{Marshal}(b.pc2);
\]
\[
b.x_e := \text{Encrypt}(b.x_{mc}, [K^e_{1HL}]);
\]
What about security?
Let's put an end to a love affair!
CONFIDENTIALITY ATTACK

Let's steal some secret!
Conclusion
### Experimental Results

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>l/t</th>
<th>crypto</th>
<th>keys</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>2</td>
<td>1</td>
<td>(1+0)</td>
<td>0/0</td>
<td>0/0</td>
</tr>
<tr>
<td>running</td>
<td>18</td>
<td>3</td>
<td>(5+3)</td>
<td>2/2</td>
<td>4/4</td>
</tr>
<tr>
<td>rpc</td>
<td>11</td>
<td>2</td>
<td>(3+3)</td>
<td>2/2</td>
<td>4/4</td>
</tr>
<tr>
<td>guess</td>
<td>52</td>
<td>7</td>
<td>(13+3)</td>
<td>2/2</td>
<td>13/16</td>
</tr>
<tr>
<td>hospital</td>
<td>33</td>
<td>5</td>
<td>(9+0)</td>
<td>4/4</td>
<td>11/11</td>
</tr>
<tr>
<td>taxes</td>
<td>55</td>
<td>4</td>
<td>(7+2)</td>
<td>8/8</td>
<td>16/16</td>
</tr>
</tbody>
</table>

RPC = 6000 symmetric-key cryptographic operations

Fournet, Le Guernic, Rezk
ParSec - CFlow
## Experimental Results

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>l/t</th>
<th>crypto</th>
<th>keys</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>2</td>
<td>1 (1+0)</td>
<td>0/0</td>
<td>0/0</td>
<td>1.59</td>
</tr>
<tr>
<td>running</td>
<td>18</td>
<td>3 (5+3)</td>
<td>2/2</td>
<td>4/4</td>
<td>1.58</td>
</tr>
<tr>
<td>rpc</td>
<td>11</td>
<td>2 (3+3)</td>
<td>2/2</td>
<td>4/4</td>
<td>1.63</td>
</tr>
<tr>
<td>guess</td>
<td>52</td>
<td>7 (13+3)</td>
<td>2/2</td>
<td>13/16</td>
<td>1.69</td>
</tr>
<tr>
<td>hospital</td>
<td>33</td>
<td>5 (9+0)</td>
<td>4/4</td>
<td>11/11</td>
<td>1.70</td>
</tr>
<tr>
<td>taxes</td>
<td>55</td>
<td>4 (7+2)</td>
<td>8/8</td>
<td>16/16</td>
<td>1.71</td>
</tr>
</tbody>
</table>

RPC = 6000 symmetric-key cryptographic operations

Fournet, Le Guernic, Rezk

ParSec - CFlow
**Experimental Results**

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>l/t</th>
<th>crypto</th>
<th>keys</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>2</td>
<td>1</td>
<td>(1+0)</td>
<td>0/0</td>
<td>0/0</td>
</tr>
<tr>
<td>running</td>
<td>18</td>
<td>3</td>
<td>(5+3)</td>
<td>2/2</td>
<td>4/4</td>
</tr>
<tr>
<td>rpc</td>
<td>11</td>
<td>2</td>
<td>(3+3)</td>
<td>2/2</td>
<td>4/4</td>
</tr>
<tr>
<td>guess</td>
<td>52</td>
<td>7</td>
<td>(13+3)</td>
<td>2/2</td>
<td>13/16</td>
</tr>
<tr>
<td>hospital</td>
<td>33</td>
<td>5</td>
<td>(9+0)</td>
<td>4/4</td>
<td>11/11</td>
</tr>
<tr>
<td>taxes</td>
<td>55</td>
<td>4</td>
<td>(7+2)</td>
<td>8/8</td>
<td>16/16</td>
</tr>
</tbody>
</table>

RPC = 6000 symmetric-key cryptographic operations
CONCLUSION

- Provide programming language for secured distributed programs
  - simple memory model: universally shared globals
  - simple security mechanism: label for access to globals
  - code size efficient
  - but: not flexible enough for now
Conclusion

- Provide programming language for secured distributed programs

Theorem 1 (Main guarantee)

If an attack exists in the target semantics then it exists in the source semantics
CONCLUSION

- Provide programming language for secured distributed programs

**Theorem 1 (Main guarantee)**

*If an attack exists in the target semantics then it exists in the source semantics*

- Make security a piece of cake
CONCLUSION

- Provide programming language for secured distributed programs

**Theorem 1 (Main guarantee)**

*If an attack exists in the target semantics then it exists in the source semantics*

- Make security a piece of cake
  - ...Ok! ...a wedding cake, but ...
Provide programming language for secured distributed programs

Theorem 1 (Main guarantee)
If an attack exists in the target semantics then it exists in the source semantics

Make security a piece of cake
- ... Ok! ... a wedding cake, but ...
- ... handling security labels instead of keys, makes it easier to ...
Provide programming language for secured distributed programs

Theorem 1 (Main guarantee)

If an attack exists in the target semantics then it exists in the source semantics

Make security a piece of cake

... Ok! ... a wedding cake, but ...
... handling security labels instead of keys, makes it easier to ...

design the security policy at the source level
analyze the program security at the source level
CFlow
A Demo with a Pinch of Theory

Cédric Fournet, Gurvan Le Guernic, Tamara Rezk

INRIA - Microsoft Research Joint Center
gleguern@gmail.com

January 29th, 2010