

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



OUTLINE



- 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 (confidentiality lattice \times integrity lattice)
 - $l_1 \rightsquigarrow l_2 \Leftrightarrow (l_1 \leq_C l_2 \wedge l_2 \leq_I l_1)$
 - $x := y$ iff $y \rightsquigarrow x$
 - A can read x iff $x \rightsquigarrow_C A$
 - A can write x iff $A \rightsquigarrow_I x$
- Security lattices are compiler plugins (2 already coded)
 - HL: $2 \times$ flat lattice with top and bottom
 - $\{L <_C [\wedge HL] <_C H\} \times \{L <_I [\wedge HL] <_I H\}$
 - ReadersWriters: 1 set of readers and 1 set of writers (R, W)
 - $R_1 <_C R_2 \Leftrightarrow R_2 \subset R_1$
 - $W_1 <_I W_2 \Leftrightarrow W_2 \subset W_1$

Source Language

PROGRAM HEADER



- Define the security lattice used: `SLattice HL`;
 - the compiler loads the appropriate plugin to manipulate strings corresponding to security labels
- Define the roles: `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:
 - `Role #LL# attacker` ;
 - stronger weakest than all roles
- Define globals: `global string(64) #HH# message`;

PROGRAM BODY



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

- $A : [S]$
 - statement localization
 - means: role A executes S
 - can be nested

CODING A CHAT PROGRAM



*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
 - *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



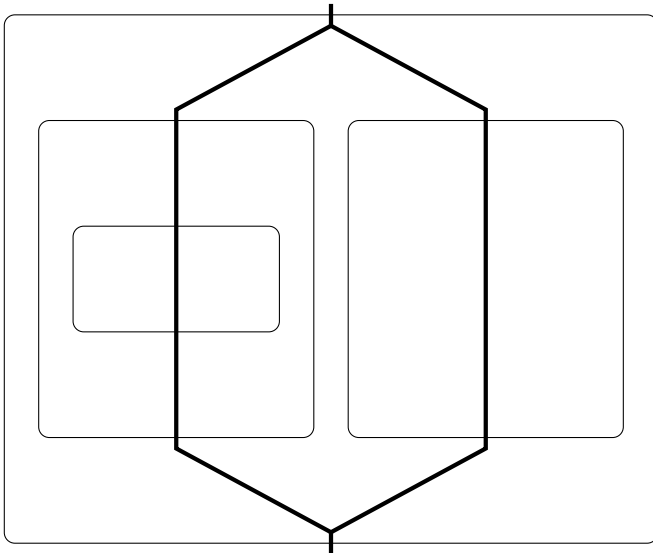
- Slicing: cut into uniquely localized threads
 - *do*: compute threads' integrities
 - *do*: meta-threads loop indexes instantiated
 - *ensure*: static previous call graph until same host
- Control Flow Protocol: prevent thread reordering
 - check *pc* set by previous "visible" threads
 - *need*: to 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

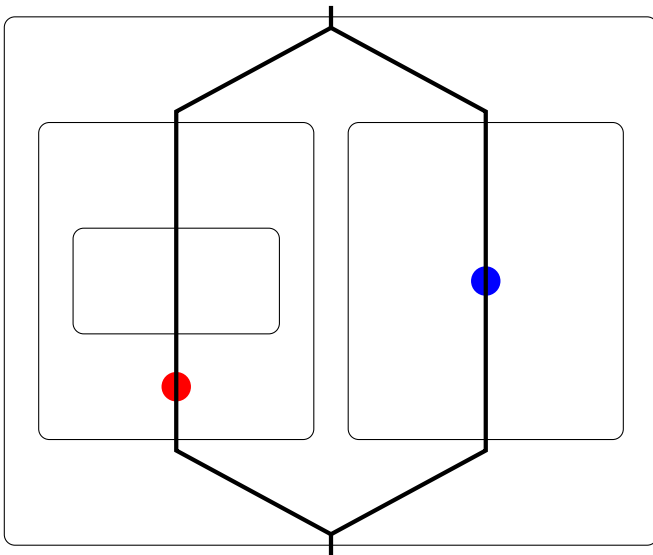


- Slicing: cut into uniquely localized threads
 - *do*: compute threads' integrities
 - *do*: meta-threads loop indexes instantiated
 - *ensure*: static previous call graph until same host
- Control Flow Protocol: prevent thread reordering
 - check *pc* set by previous "visible" threads
 - *need*: to 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
 - *do*: assigned globals transfer at merge points
- Encrypting & Signing: enforce security labels of globals
 - *need*: a unique tag to sign and verify

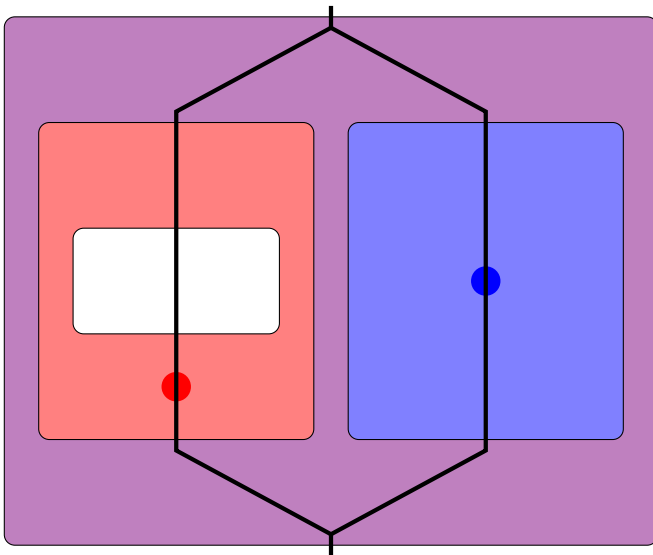
SLICING



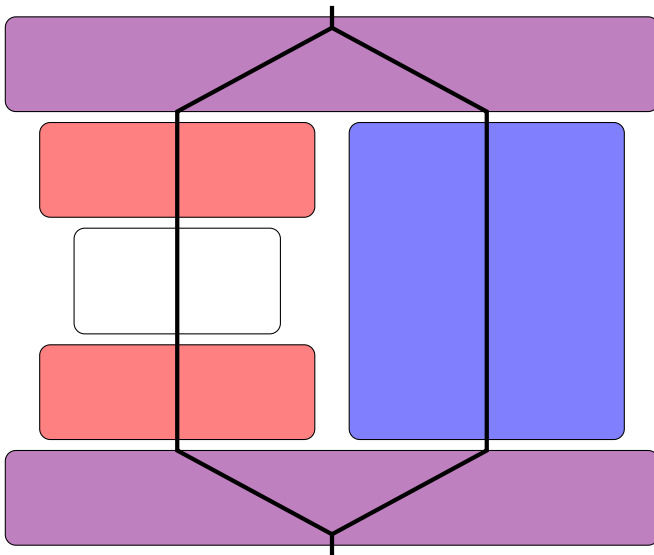
SLICING



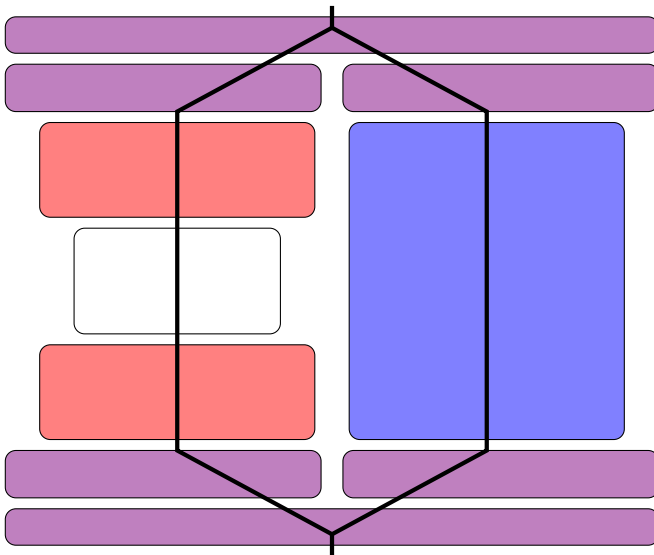
SLICING



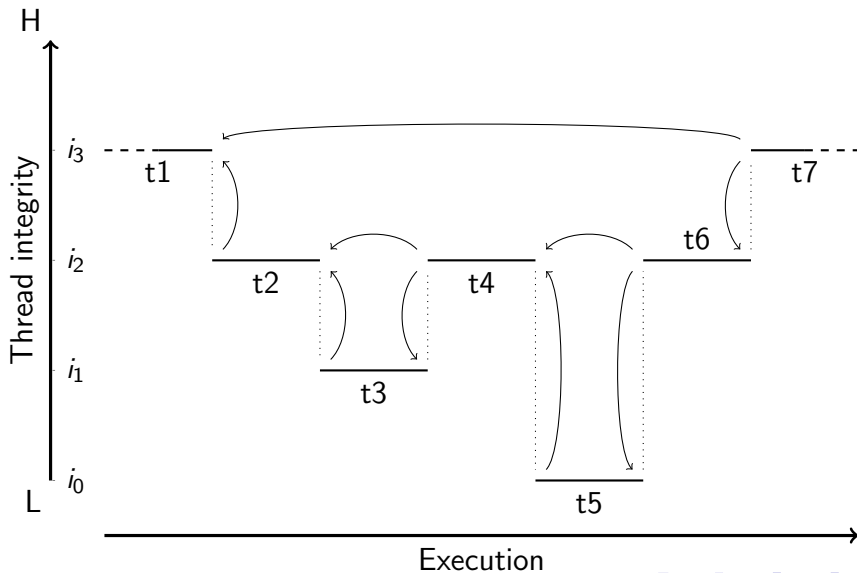
SLICING



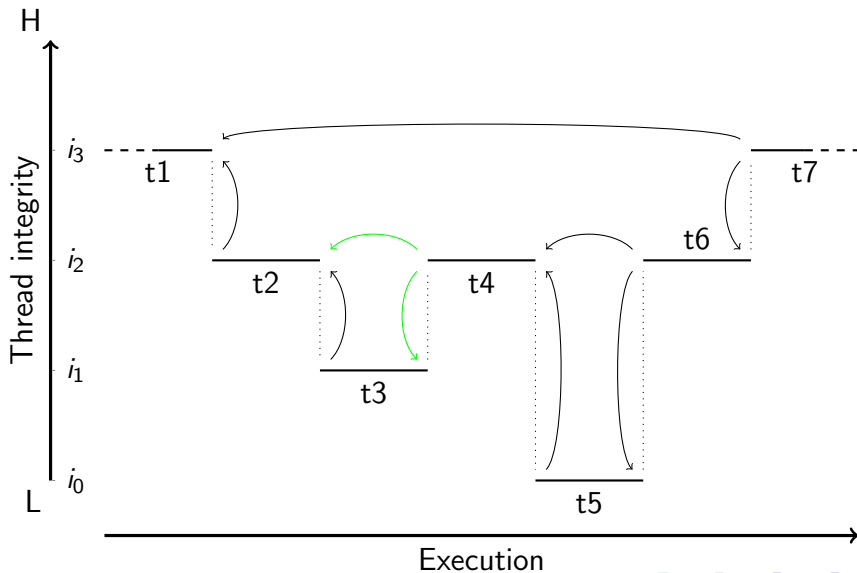
SLICING



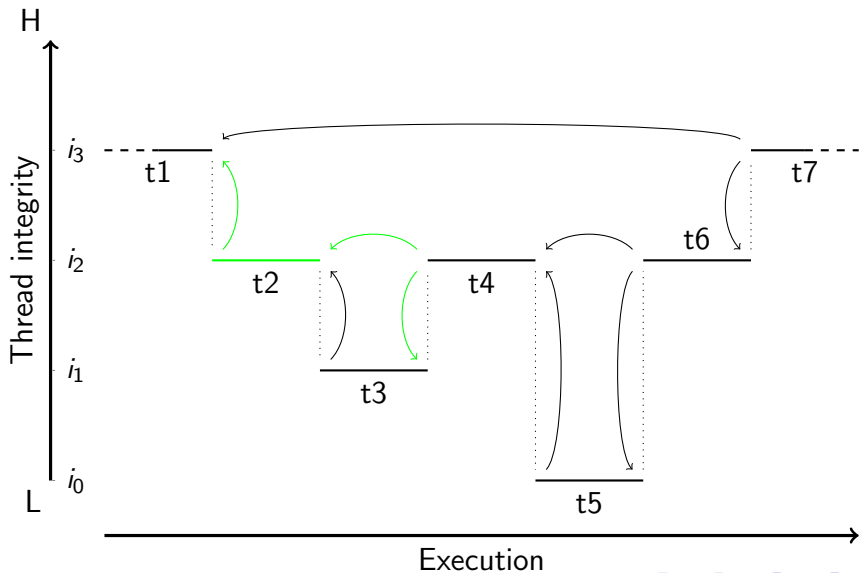
CONTROL FLOW PROTOCOL



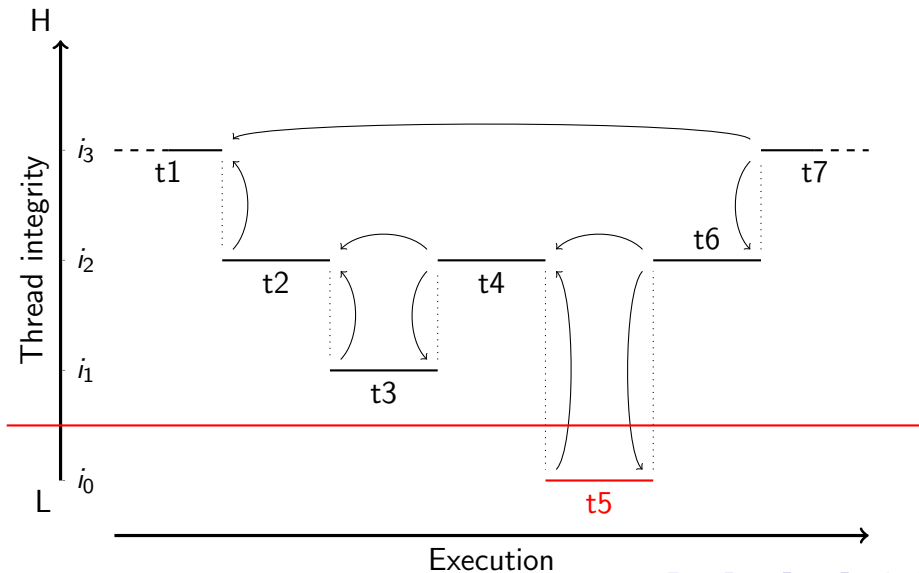
CONTROL FLOW PROTOCOL



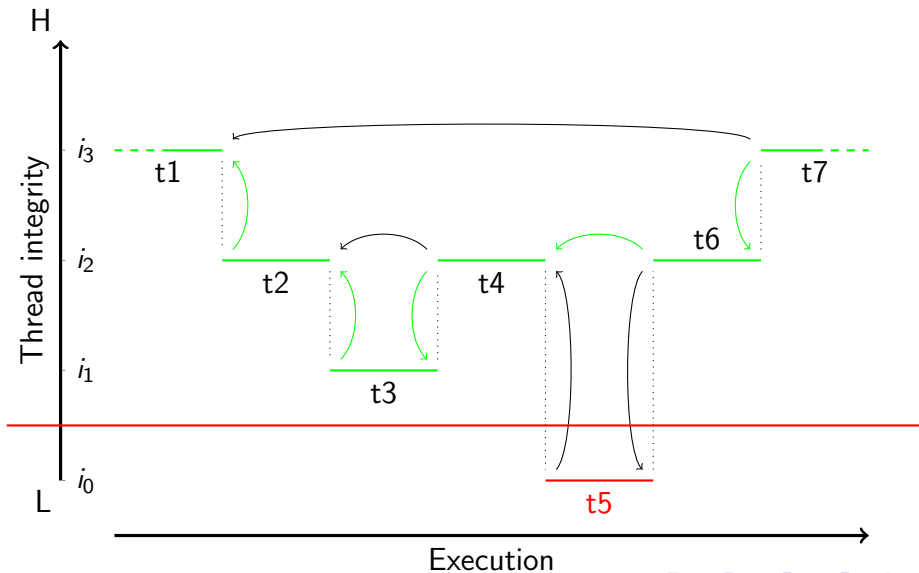
CONTROL FLOW PROTOCOL



CONTROL FLOW PROTOCOL



CONTROL FLOW PROTOCOL



STATIC SINGLE REMOTE ASSIGNER



- 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)  $\cong$  ("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

```

check Verify( $b.pc1_s$ , "a8."^i."^j".pc1",  $b.pc1_{mc}$ ,  $K_{1HL}^s$ ) do {
  check Verify( $b.y_s$ , "a8."^i."^j".y",  $b.y_{mc}$ ,  $K_{1HL}^s$ ) do {
     $b.x_{mc} := Decrypt(b.x_e, K_{1HL}^e)$ ;
     $b.x := Unmarshal(b.x_{mc})$ ;
     $b.y := Unmarshal(b.y_{mc})$ ;
     $b.pc1 := Unmarshal(b.pc1_{mc})$ ;
    check  $b.pc1 \cong ("a8", [i; j])$  do {
       $b.pc2 := ("b5", [i; j])$ ;
      if ( $b.y \bmod 2$ ) = 1
      then { $b.x := b.x + 9$ }
      else { $b.x := b.x$ };
       $b.x_{mc} := Marshal(b.x)$ ;
       $b.pc2_{mc} := Marshal(b.pc2)$ ;
       $b.x_e := Encrypt(b.x_{mc}, [K_{1HL}^e])$ ;

```

What about security?

INTEGRITY ATTACK

*Let's put an end
to a love affair!*

CONFIDENTIALITY ATTACK



*Let's steal
some secret!*

Conclusion

EXPERIMENTAL RESULTS



Program	LOC		l/t		crypto		keys	Time (s)	
empty	2	102	1	(1+0)	0/0	0/0	0/0	1.59	1.63
running	18	464	3	(5+3)	2/2	4/4	1/2	1.58	1.71
rpc	11	321	2	(3+3)	2/2	4/4	1/1	1.63	2.58
guess	52	912	7	(13+3)	2/2	13/16	2/3	1.69	1.98
hospital	33	906	5	(9+0)	4/4	11/11	4/8	1.70	1.84
taxes	55	946	4	(7+2)	8/8	16/16	4/6	1.71	1.77

EXPERIMENTAL RESULTS



Program	LOC		l/t	crypto		keys	Time (s)		
empty	2	102	1	(1+0)	0/0	0/0	0/0	1.59	1.63
running	18	464	3	(5+3)	2/2	4/4	1/2	1.58	1.71
rpc	11	321	2	(3+3)	2/2	4/4	1/1	1.63	2.58
guess	52	912	7	(13+3)	2/2	13/16	2/3	1.69	1.98
hospital	33	906	5	(9+0)	4/4	11/11	4/8	1.70	1.84
taxes	55	946	4	(7+2)	8/8	16/16	4/6	1.71	1.77

EXPERIMENTAL RESULTS



Program	LOC		l/t		crypto		keys	Time (s)	
empty	2	102	1	(1+0)	0/0	0/0	0/0	1.59	1.63
running	18	464	3	(5+3)	2/2	4/4	1/2	1.58	1.71
rpc	11	321	2	(3+3)	2/2	4/4	1/1	1.63	2.58
guess	52	912	7	(13+3)	2/2	13/16	2/3	1.69	1.98
hospital	33	906	5	(9+0)	4/4	11/11	4/8	1.70	1.84
taxes	55	946	4	(7+2)	8/8	16/16	4/6	1.71	1.77

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 ...

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 ...
 - ... handling security labels instead of keys, makes it easier to ...

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 ...
 - ... 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

