CFlow

A Demo with a Pinch of Theory

Cédric Fournet, **Gurvan Le Guernic**, Tamara Rezk

INRIA - Microsoft Research Joint Center
gleguern@gmail.com

January 29^{th}, 2010

MICROSOFT RESEARCH INRIA JOINT CENTRE

Outline

Introduction

Goal

MICROSOFT RESEARCH INRIA JOINT CENTRE

- 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
    - $\{L <_C [\hat{}HL] <_C H\} \times \{L <_I [\hat{}HL] <_I H\}$
  - 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

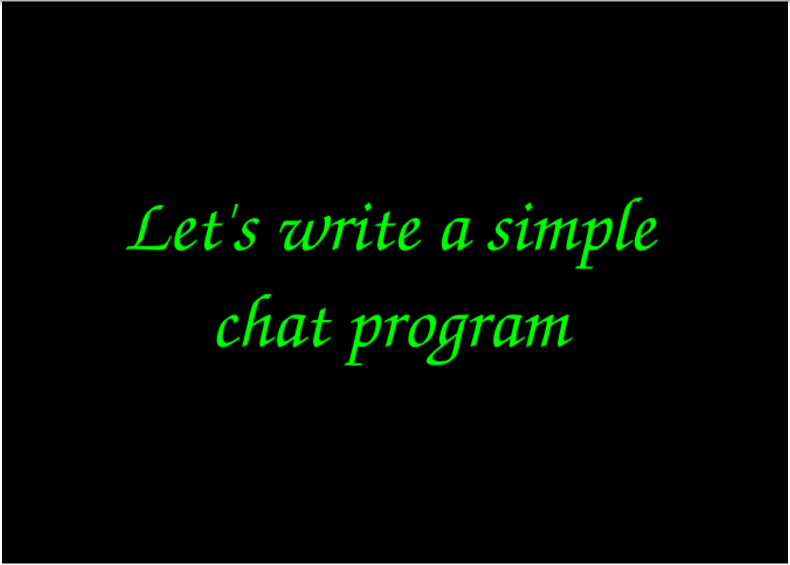
- Define the security lattice used: `SLattice` HL;
  - the compiler loads the appropriate plugin to manipulate strings corresponding to security labels
- Define the roles: `Role` #HH# *A*;
  - all roles in the execution environment
    - A, B: secured line or VPN between A and B
    - A, B, others: any 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

$$
\begin{array}{lcl}
e & ::= & x \mid op(e_1, \ldots, e_n) \\
S & ::= & \mathbf{skip} \mid x \mathrel{\mathbf{:=}} e \mid S \;;\; S \\
 & \mid & \mathbf{if}\ e\ \mathbf{then}\ S\ \mathbf{else}\ S\ \mathbf{end} \mid \mathbf{while}\ e\ \mathbf{do}\ S\ \mathbf{done} \\
 & \mid & A : [S]
\end{array}
$$

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

Coding a chat program



How it works

A 4-steps process

MICROSOFT RESEARCH INRIA JOINT CENTRE

- 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

MICROSOFT RESEARCH INRIA JOINT CENTRE

- 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

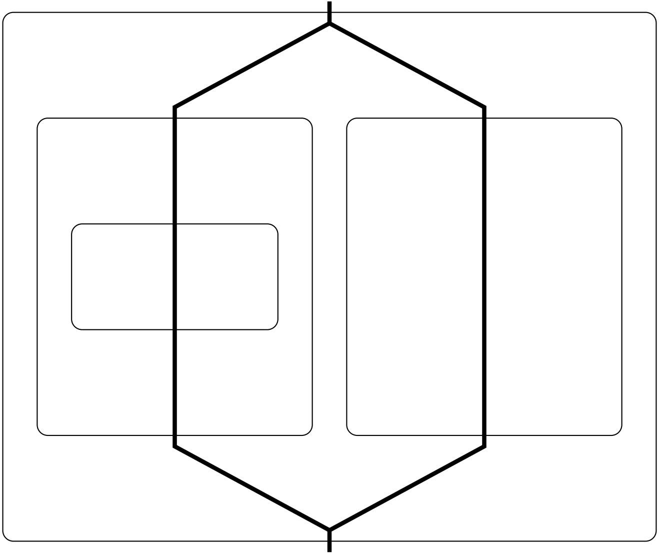
MICROSOFT RESEARCH INRIA JOINT CENTRE

- 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

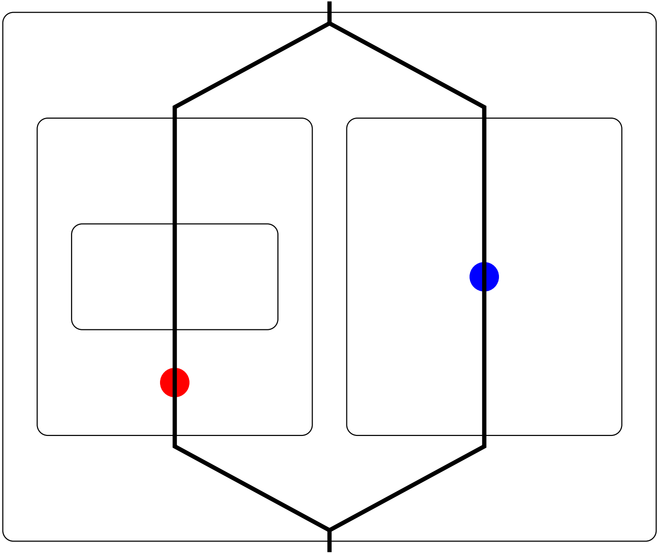
A 4-steps process

- 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

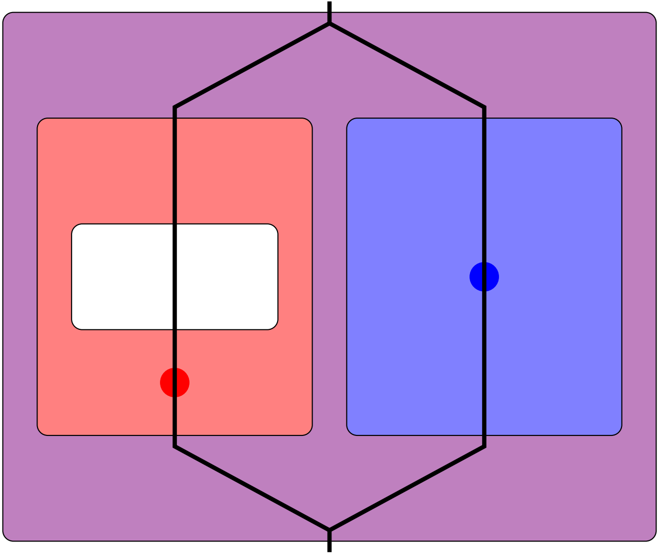
Slicing



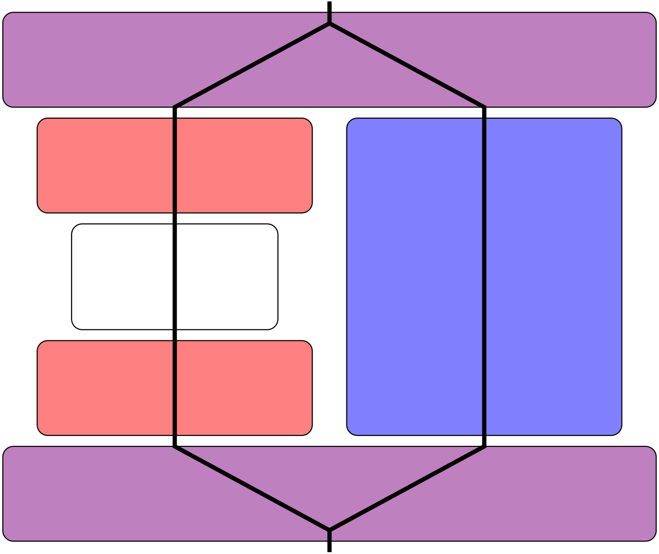
Slicing



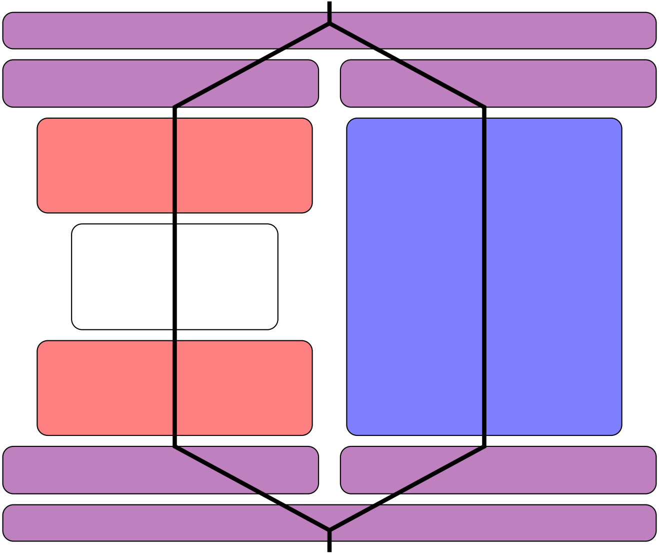
Slicing



Slicing

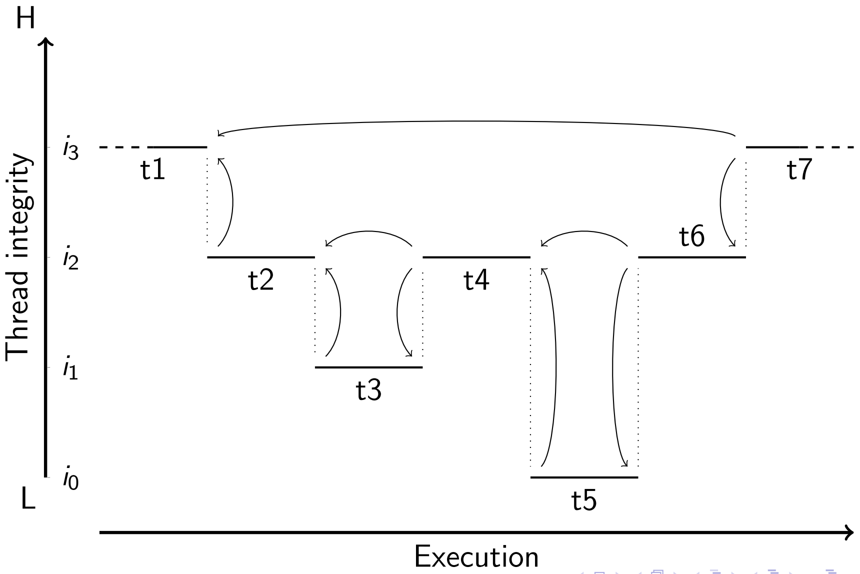


Slicing



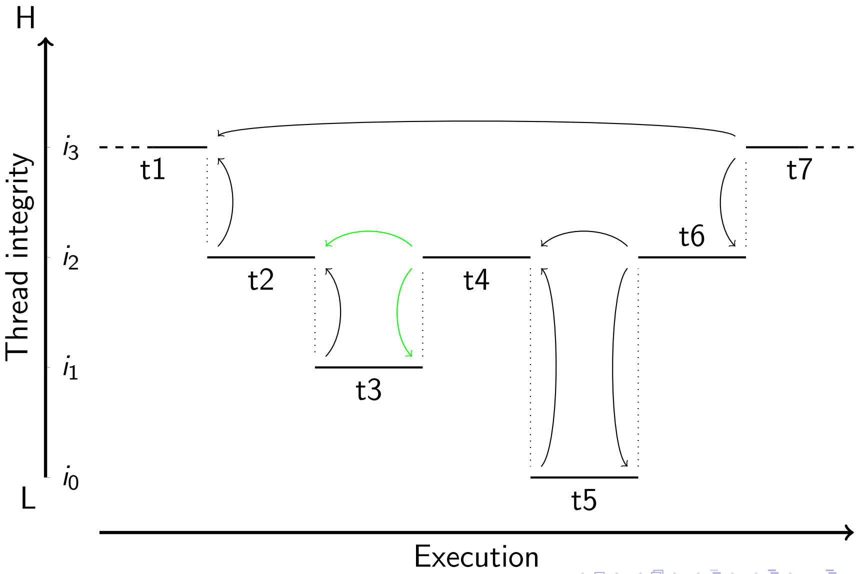
Control Flow Protocol

MICROSOFT RESEARCH INRIA JOINT CENTRE

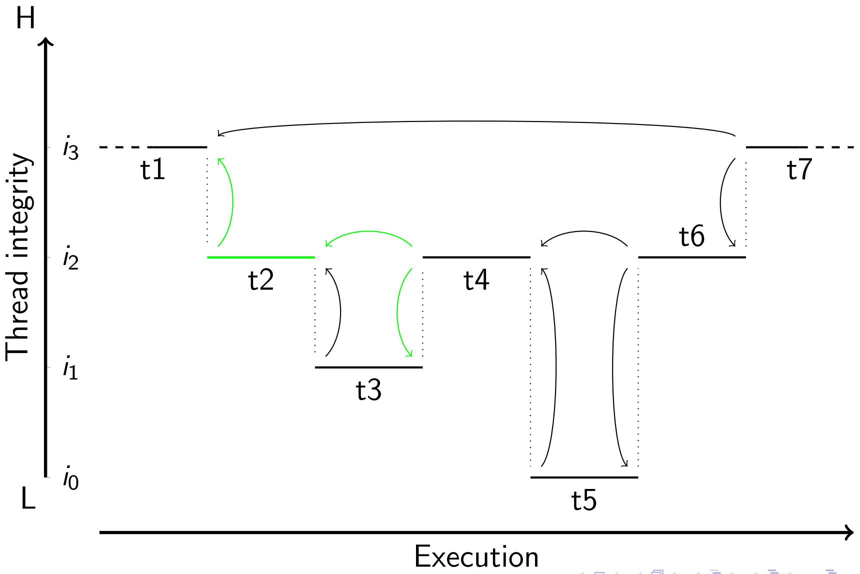


Control Flow Protocol

MICROSOFT RESEARCH INRIA JOINT CENTRE

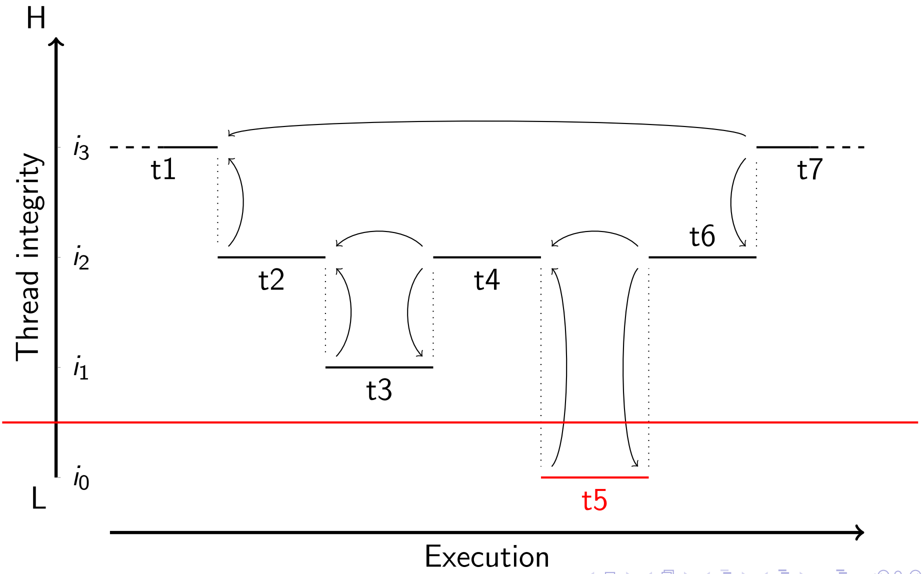


Control Flow Protocol

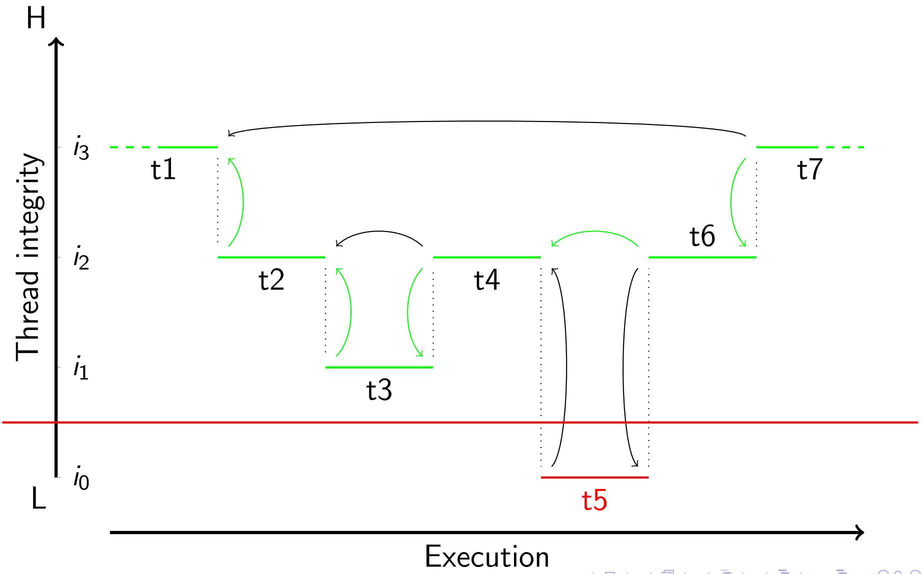


Control Flow Protocol

Microsoft Research INRIA Joint Centre



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) ≅ ("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

MICROSOFT RESEARCH INRIA JOINT CENTRE

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

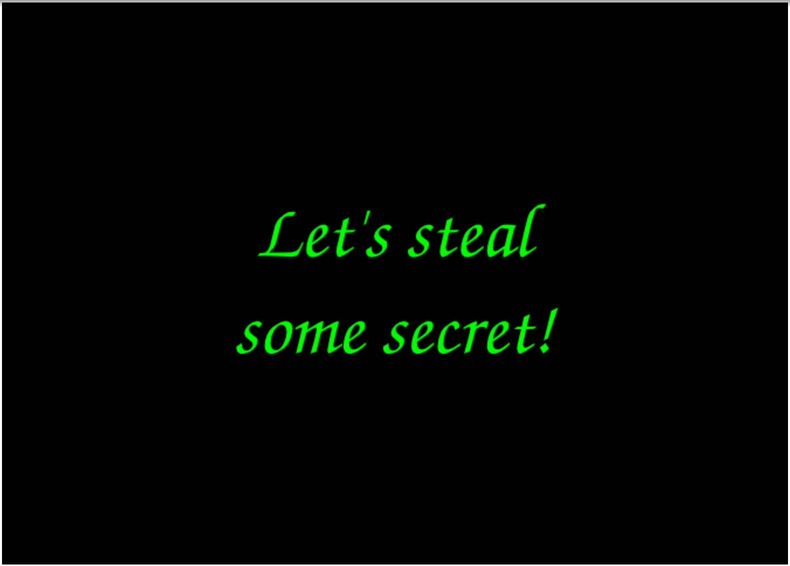
```
check Verify(b.pc1_s, "a8."^i^"."^j^".pc1", b.pc1_mc, K^s_1HL) do {
 check Verify(b.y_s, "a8."^i^"."^j^".y", b.y_mc, K^s_1HL) do {
  b.x_mc := Decrypt(b.x_e, K^e_1HL);
  b.x := Unmarshal(b.x_mc);
  b.y := Unmarshal(b.y_mc);
  b.pc1 := Unmarshal(b.pc1_mc);
  check b.pc1 ≅ ("a8", [i; j]) do {
   b.pc2 := ("b5", [i; j]);
   if (b.y mod 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^e_1HL]);
```

What about security?

Integrity attack

Confidentiality attack

MICROSOFT RESEARCH INRIA JOINT CENTRE



Conclusion

Experimental Results

Program	LOC		I/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

MICROSOFT RESEARCH INRIA JOINT CENTRE

Program	LOC		I/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

MICROSOFT RESEARCH INRIA JOINT CENTRE

Program	LOC		I/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

MICROSOFT RESEARCH INRIA JOINT CENTRE

- 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

MICROSOFT RESEARCH INRIA JOINT CENTRE

- 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

MICROSOFT RESEARCH INRIA JOINT CENTRE

- 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

MICROSOFT RESEARCH
INRIA
JOINT
CENTRE