Moscova

Jean-Jacques Lévy

INRIA Paris–Rocquencourt

March 23, 2011
Research team
Stats

- **Staff 2008-2011**
  
  Jean-Jacques Lévy, INRIA  
  Karthikeyan Bhargavan, INRIA  
  James Leifer, INRIA  
  Luc Maranget, INRIA  
  Francesco Zappa Nardelli, INRIA  
  Ricardo Corin, INRIA \(\rightarrow\) Cordoba  
  Gilles Peskine, INRIA \(\rightarrow\) Trusted logic  
  Pierre-Malo Deniélou, INRIA \(\rightarrow\) Imperial College  
  Jade Alglave, INRIA \(\rightarrow\) Oxford  
  Nataliya Guts, MSR-INRIA \(\rightarrow\) Maryland  
  Jérémy Planul, MSR-INRIA  

- INRIA Rocquencourt \(\leftrightarrow\) MSR-INRIA Saclay \(\text{(Cédric Fournet, MSRC)}\)  

- **Moscova history:**
  
  - Para (1988, Head: Lévy), Moscova (2000, Head: Gonthier \(\rightarrow\) MSRC)  
  - 18 PhDs  
  - 75% Coq proof of the 4-color thm; debugging of 3 modules of Ariane-501 PV; spinoff of Polyspace [Alain Deutsch]; etc.  
  - Polytechnique (Lévy, prof 1992-2006) \(\rightarrow\) MSR-INRIA Joint Centre (Head: Lévy)
Research themes
Research themes

- **programming languages**
  [safe marshalling, ott, like types]

- **concurrency**
  [jocaml, separation logic/c-minor/concurrency, weak memory models]

- **security compilers and verifiers**
  [secure sessions, audits, tls, information flow]
Research results
Example 1: Weak memory models

- memory models of multi-core processors
- give formal description of WMMs
- operational semantics of WMMs
- certified (back-end) compiler for some WMMs
- prove correctness of compiler optimisations in WMMs
2.1 Loads are not reordered with other loads and stores are not reordered with other stores

Intel 64 memory ordering ensures that loads are seen in program order, and that stores are seen in program order.

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov [z], 1 // M1</td>
<td>mov r1, [y] // M3</td>
</tr>
<tr>
<td>mov [z], 1 // M2</td>
<td>mov r2, [x] // M4</td>
</tr>
<tr>
<td>Initially x == y == 0</td>
<td></td>
</tr>
<tr>
<td>r1 == 1 and r2 == 0 is not allowed</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Stores are not reordered with other stores
2.3 Loads may be reordered with older stores to different locations

Intel 64 memory ordering allows load instructions to be reordered with prior stores to a different location. However, loads are not reordered with prior stores to the same location.

The first example in this section illustrates the case in which a load may be reordered with an older store – i.e. if the store and load are to different non-overlapping locations.

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov [ _x], 1 // M1</td>
<td>mov [ _y], 1 // M3</td>
</tr>
<tr>
<td>mov r1, [ _y] // M2</td>
<td>mov r2, [ _x] // M4</td>
</tr>
<tr>
<td>Initially x == y == 0</td>
<td></td>
</tr>
<tr>
<td>r1 == 0 and r2 == 0 is allowed</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.3.a: Loads may be reordered with older stores
2.5 Stores are transitively visible

Intel 64 memory ordering ensures transitive visibility of stores – i.e. stores that are causally related appear to execute in an order consistent with the causal relation.

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
<th>Processor 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov [ _x], 1 // M1</td>
<td>mov r1, [ _x] // M2</td>
<td>mov r2, [ _y] // M4</td>
</tr>
<tr>
<td>mov [ _y], 1 // M3</td>
<td></td>
<td>mov r3, [ _x] // M5</td>
</tr>
</tbody>
</table>

Initially x == y == 0

r1 == 1, r2 == 1, r3 == 0 is not allowed

Table 2.5: Stores are transitively visible
Event structures

SB with SC (sequential consistency)

- In SC, program order is strictly respected.

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov [x], 1</td>
<td>mov [y], 1</td>
</tr>
<tr>
<td>mov r1, [y]</td>
<td>mov r2, [x]</td>
</tr>
</tbody>
</table>

```
cyclic graph
```
In TSO, \( W \) followed by \( R \) can be relaxed within program order.

- For \( r_1 = 1, r_2 = 1 \):
  - \( (x, 1) \) followed by \( (y, 1) \)
  - \( (r_1, 1) \) followed by \( (r_2, 1) \)

- For \( r_1 = 1, r_2 = 0 \):
  - \( (x, 1) \) followed by \( (y, 0) \)
  - \( (r_1, 1) \) followed by \( (r_2, 0) \)

- For \( r_1 = 0, r_2 = 1 \):
  - \( (x, 0) \) followed by \( (y, 1) \)
  - \( (r_1, 0) \) followed by \( (r_2, 1) \)

- For \( r_1 = 0, r_2 = 0 \):
  - \( (x, 0) \) followed by \( (y, 0) \)
  - \( (r_1, 0) \) followed by \( (r_2, 0) \)
Event structures

MP with TSO/PSO (partial store order)

- In TSO, \( W \) followed by \( R \) relaxed

\[
\begin{array}{c|c}
\text{P0} & \text{P1} \\
\text{mov [x], 1} & \text{mov r1, [y]} \\
\text{mov [y], 1} & \text{mov r2, [x]} \\
\end{array}
\]

- In PSO, \( W \) followed by \( W \) to distinct location relaxed

\[
\begin{array}{c|c}
\text{P0} & \text{P1} \\
\text{mov [x], 1} & \text{mov r1, [y]} \\
\text{mov [y], 1} & \text{mov r2, [x]} \\
\end{array}
\]
Weak memory models

- axiomatic + operational models for Intel [Cambridge] / Power [INRIA]
- formalisation in HOL/Coq
- tests on real processor behaviour
  - http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/ppc003.html
- formal proof of simple concurrent code (eg. Linux spinlocks)
- operational reasoning: data-race freedom, separation logic
- certified compiler for concurrent languages
  - http://www.cl.cam.ac.uk/~pes20/CompCertTSO

[Zappa Nardelli, Maranget, Alglave, Braibant, Sewell et al]
[POPL 09, CACM 10; DAMP 09, CAV 10, PLDI 11; TACAS 11; POPL 11]
Weak memory models

- Proving correctness of optimisations

Fences elimination with TSO $\simeq$ 3 kloCoq
Secure sessions

- passing authenticated *(signed)* values between 2 *run-times*

- design of a mini F# + primitives for authentication
  + global contract with *sessions types*

- compiling scheme into a low-level language (≃ pi-calculus) to describe authentication protocols

- formal proof of its correctness, with *security property* induced by strong typing of F# + usage of authentication primitives

- extension to other security properties, sessions V2 (privacy of message values, integrity, dynamic number of principals, etc)

[Corin, Deniélov, Leifer, Fournet, Bhargavan]

[JCS 08, TGC 07, CSF 09, Deniélov phd 11]
Secure sessions

- passing authenticated (signed) values between 2 run-times
- design of a mini F# + primitives for authentication + global contract with sessions types
- compiling scheme into a low-level language ($\simeq$ pi-calculus) to describe authentication protocols
- formal proof of its correctness, with security property induced by strong typing of F# + usage of authentication primitives
- extension to other security properties, sessions V2 (privacy of message values, integrity, dynamic number of principals, etc)

[Corin, Deniélou, Leifer, Fournet, Bhargavan]
[JCS 08, TGC 07, CSF 09, Deniélou phd 11]
Example 2 Secure sessions

- passing authenticated (signed) values between 2 run-times
- design of a mini F# + primitives for authentication + global contract with sessions types
- compiling scheme into a low-level language (≃ pi-calculus) to describe authentication protocols
- formal proof of its correctness, with security property induced by strong typing of F# + usage of authentication primitives
- extension to other security properties, sessions V2 (privacy of message values, integrity, dynamic number of principals, etc)

[Corin, Deniélov, Leifer, Fournet, Bhargavan]
[JCS 08, TGC 07, CSF 09, Deniélov phd 11]

F# = Ocaml + modules + .NET
Simple exchange

```
session S =
  role requester : int =
    !Request:string ;
    ?(Response:int + Fault:unit)

role directory : string =
  ?Request:string;
  !(Response:int + Fault:unit)
```

```
let lookup name =
  S.requester ["client";"server"]
  (Request
    (name,
      {hResponse = (fun _ q -> q);
        hFault = (fun _ x -> failwith "Failed")
      }))
  in lookup "Ricardo"
```
Two-party negotiation

\[
\text{session } S_2 = \\
\text{role customer : string } = \\
\quad \text{!Query:int;}
\quad \text{mu start.}(\text{Accept:unit } +
\quad \quad \text{Condition:unit};! (\text{NewOffer:int;start + Reject:unit}))
\]

\[
\text{role store : string } = \\
\quad ?\text{Query:int;}
\quad \text{mu start.}(\text{Accept:unit } +
\quad \quad \text{Condition:unit};? (\text{NewOffer:int;start + Reject:unit}))
\]
Three-party session

session S3 =
  role customer :string =
    !Query:int;
    mu start.?((Accept:unit;!Confirm:unit +
      Condition:unit; !((Newoffer:int;start + Reject:unit;)))
  role store :string=
    ?Contract:int;
    mu start.!((Accept:unit +
      Condition:unit; ?(Newoffer:int;start + Reject:unit;!Abort:unit))
  role officer :string=
Visibility

- Minimal sequence of signatures that guarantee session compliance.
- Example:
No blind fork

- Some forks in protocols represent a security threat.

- Property


```
[C] -> [O]  [C] -> [S]
  Query     Contract

S -> O  C -> O
  Accept   Confirm

S -> C  C -> S
  Condition  Reject

C -> S  S -> O
  NewOffer  Abort

C -> No A, B, C
  Role Equality

[A] -> [B] -> [C]
```
Secure sessions

- passing authenticated (signed) values between 2 run-times
- design of a mini F# + primitives for authentication + global contract with sessions types
- compiling scheme into a low-level language (≃ pi-calculus) to describe authentication protocols
- formal proof of its correctness, with security property induced by strong typing of F# + usage of authentication primitives
- extension to other security properties, sessions V2 (privacy of message values, integrity, dynamic number of principals, etc)

[Corin, Deniélo, Leifer, Fournet, Bhargavan]
[JCS 08, TGC 07, CSF 09, Deniélo phd 11]
Example 3: Verified Crypto Protocol Implementations

High-Level Specification

Protocol Code, Threat Model

Security Proofs, F7, ProVerif,...

Network

Proof/Attack
Protocol Specifications in F7

\[ A \rightarrow B: m, \text{hmac } k_{AB} m \]

**F# Code**

```fsharp
let client a b k m =
    Pi.assume (ClientSent(a,b,m));
let h = hmac k m in
let msg = concat m h in
    Net.send msg

let server a b k =
    let msg = Net.recv in
    let (m,h) = concat msg in
    hmacVerify k m h;
    Pi.expect (ClientSent(a,b,m))
```

**F7 Interface**

```fsharp
val client : a:prin -> b:prin ->
    k:key{SharedKey(a,b,k)} ->
    m:bytes -> unit

val server : a:prin -> b:prin ->
    k:key{SharedKey(a,b,k)} ->
    m:bytes{ClientSent(a,b,m)}
```

*Dependent, refinement types*

*Pre-condition*  *Post-condition*
Protocol Specifications in F7

\( A \rightarrow B: m, \text{hmac } k_{AB} m \)

**F# Code**

```fsharp
let client a b k m =
    Pi.assume (ClientSent(a,b,m));
let h = hmac k m in
let msg = concat m h in
    Net.send msg

let server a b k =
    let msg = Net.recv in
    let (m,h) = iconcat msg in
    hmacVerify k m h;
    Pi.expect (ClientSent(a,b,m))
```

**Refined Crypto Interface**

```fsharp
val hmac: k: key{MKey(k)} ->
    m: bytes{MayMAC(k,m)} ->
    h: bytes

val hmacVerify: k: key{MKey(k)} ->
    m: bytes ->
    h: bytes ->
    unit{MayMAC(k,m)}
```

- **Pre-conditions**
  - Assume \( a, b, k \).
  - \( \text{SharedKey}(a,b,k) \Rightarrow MKey(k) \)
  - Assume \( a, b, k \).
  - \( \text{SharedKey}(a,b,k) \Rightarrow \) \( \neg \neg \exists m. \text{MayMAC}(k,m) \leftrightarrow \text{ClientSent}(a,b,m) \)
We implemented a subset of TLS (10 kLOC)
- Supports SSL3.0, TLS1.0, TLS1.1
  - DES, AES, RC4, SHA1, MD5
- Largest verified crypto protocol implementation till date

We used “global” cryptographic verifiers,
  ProVerif and CryptoVerif [Blanchet]

We reached the limit of this proof method:
- Whole-program analysis does not scale
- Verification takes hours on a large machine

Ongoing work: Use F7 for modular verification

[CCS 08, TOPLAS 10, APLAS 10, POPL 10, ESORICS 09, phD Guts’11]
Other works

- Acute – type safed marshalling  [Leifer, Peskine, Zappa Nardelli]
- OTT – A semantics tool  [Sewell, Zappa Nardelli]
- Scripting languages (Like types)  [Zappa Nardelli]
- Jocaml (version 3; more portable, documentation)  [Maranget, Mandel]
- Separation logic  [Appel, Zappa Nardelli]
- Security through logs  [Guts, Fournet, Zappa Nardelli]
- Information flow  [Corin, Fournet, le Guernic, Planul, Rezk]
- Pattern-matching in Ocaml  [Maranget]
Miscellaneous
- Microsoft Research Cambridge through the MSR-INRIA Joint Centre
- Sewell et al at Cambridge, Computer Lab
- **Indes**, Celtique, PPS with ANR Parsec [Zappa Nardelli]
- Gallium for general discussion about programming languages
- Andrew Appel, Princeton
- Secsi, Cascade with ERC-Crysp [Bhargavan]
Software

- diy tool suite [Alglave, Maranget]
- OTT: a semantics tool [Sewell, Zappa Nardelli]
- CompCertTSO: certified compiler for TSO [Jagannathan, Sewell, Sevcik, Vafeiadis, Zappa Nardelli]
- S2ML [Bhargavan, Corin, Deniélou]
- FS2CV [Bhargavan, Corin, Zalinescu]
- F7 [Bhargavan]
- Jocaml [Maranget, Mandel]
- 5% Ocaml (pattern matching) [Maranget]
- Hévéa: an efficient translator of Tex into Html [Maranget]
Teaching

- **MPRI (master course at Paris 7)**
  [Zappa Nardelli, Leifer]

- **École polytechnique**
  [Maranget, Bhargavan, ...Lévy (1992–2006)]
  lecture notes + web pages

- **Entrance examination at Polytechnique**
  [Maranget (4 years), Lévy (?–2009)]

- **Bertinoro, IIT-Delhi, Tsinghua, etc.**
Objectives for next years
Scientific goals

- Weak Memory Models
  - ARM multi-core + xfer to industry
  - automatic exploration of WMMs
  - automatic synchronisation of programs
  - certified compilation of C-like with C1x/C++0x WMM

- Security compilers and verifiers
  - scalable tools to verify security of programs
  - verified open source cryptographic libraries
  - web applications with formal proofs of security
Organization

INRIA
FIN

12 April 2011: http://msr-inria.inria.fr/forum2011