A BSP algorithm for the state space construction of security protocols

Frédéric Gava, Michael Guedj, Franck Pommereau

Laboratory of **A**lgorithms, **C**omplexity and **L**ogic (LACL)
University of Paris-East

Security protocols

- apparent simplicity but notoriously error-prone
- need for formal proofs
  - multiple sessions
  - in presence of an attacker
  - discover attacks (man-in-the-middle, replay, . . .)
- explicit model-checking
  - automated
  - exhaustive analysis
  - on finite scenarios
  - quick state explosion
- need for faster computation, with more memory

Protocol example: Otway-Rees

Goal: distribute a fresh shared symmetric key K_{ab} between agents A and B through trusted server S

1. $A \rightarrow B$ $\quad M, A, B, \{N_a, M, A, B\}K_{as}$
2. $B \rightarrow S$ $\quad M, A, B, \{N_a, M, A, B\}K_{as}, \{N_b, M, A, B\}K_{bs}$
3. $S \rightarrow B$ $\quad M, \{N_a, K_{ab}\}K_{as}, \{N_b, K_{ab}\}K_{bs}$
4. $B \rightarrow A$ $\quad M, \{N_a, K_{ab}\}K_{as}$

Security requirement: secrecy of K_{ab}

Dolev-Yao attacker

1. agents send messages to the network
2. spy captures messages
   - learns by recursive decomposition/decryption
   - forges new messages from learnt information
   - using only allowed operations (perfect cryptography)
3. spy delivers messages (including the original one)

Otway-Rees: known replay attack

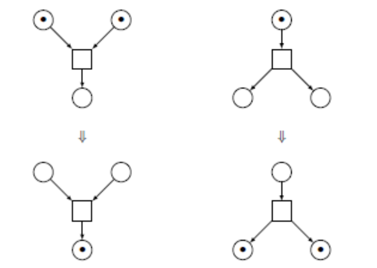
Exploits a typing error: A accepts $\{M, A, B\}$ as fresh key K_{ab}

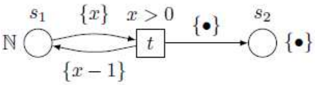
1. $A \rightarrow I(B)$ $\quad M, A, B, \{N_a, M, A, B\}K_{as}$
2. $B \rightarrow S$ $\quad M, A, B, \{N_a, M, A, B\}K_{as}, \{N_b, M, A, B\}K_{bs}$
3. $S \rightarrow B$ $\quad M, \{N_a, K_{ab}\}K_{as}, \{N_b, K_{ab}\}K_{bs}$
4. $I(B) \rightarrow A$ $\quad M, \{N_a, M, A, B\}K_{as}$

Tools: SNAKES and BSP-Python

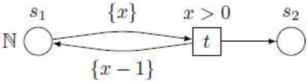
- coloured Petri nets for protocol and scenarios models
- SNAKES (LACL)
  - coloured Petri nets library
  - colour domain = Python language
  - ABCD algebra for protocols
    send/receive sequences in parallel
- BSP-Python (CNRS Physic Orleans)
  - BSP library for Python
  - used for global exchanges
- LACL cluster $\Rightarrow$ 40 CPU BSP machine
- short cycles: algorithm $\mapsto$ implementation $\mapsto$ benchmarks
- not efficient but accurate for algorithms comparison

High Level Petri Nets

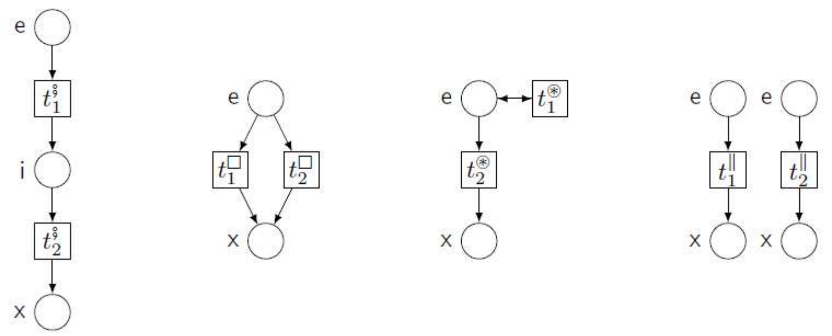




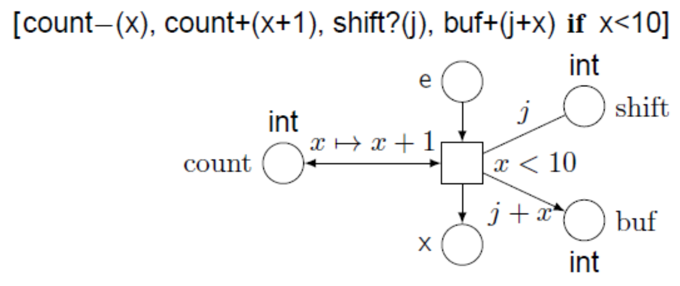
$$
\begin{aligned}
S &\stackrel{\text{df}}{=} \{s_1, s_2\}\\
T &\stackrel{\text{df}}{=} \{t\}\\
\ell &\stackrel{\text{df}}{=} \{s_1 \mapsto \mathbb{N}, s_2 \mapsto \{\bullet\}, t \mapsto x > 0, (s_1, t) \mapsto \{x\},\\
&\qquad (s_2, t) \mapsto \emptyset, (t, s_1) \mapsto \{x - 1\}, (t, s_2) \mapsto \{\bullet\}\}
\end{aligned}
$$



Algebra of High Level Petri Nets (1)

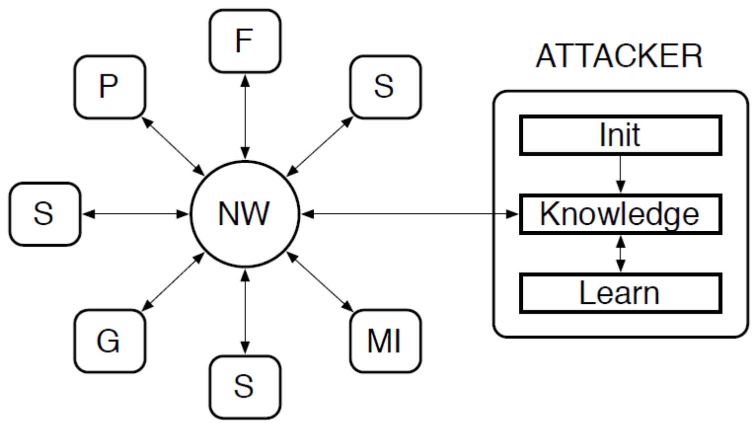


Algebra of High Level Petri Nets (2)



- communication inter-agent using a buffer (the network)
- each agent is a sequential process using its own buffers and the network

Model of Attacker



Learn is a Python code for dolev-Yao inductives rules !

Python ? Are you serious ?

- Advantages of Python:
  - untyped and interpreted
  - many efficient data-structures
  - SNAKE In Python
- Drawbacks of Python:
  - untyped and interpreted
  - non-determnism on list/set iterations
  - hashing a value is partially machine dependant

Labelled transitions systems

- initial state s_0
- successors given by $succ(s)$
- transition $s \rightarrow s'$ whenever $s' \in succ(s)$
- inductive computation of the state space
  - as a graph (explicit LTS)
  - as a set of reachable states
- we present sets for simplicity

Sequential algorithm

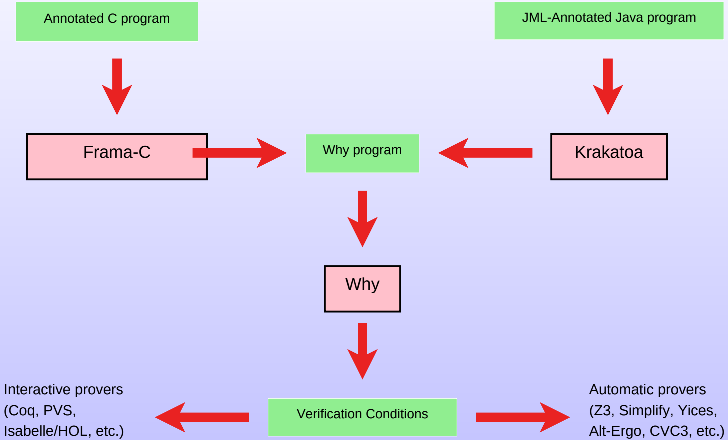
$todo \leftarrow \{s_0\}$
$known \leftarrow \emptyset$
while $todo \neq \emptyset$ **do**
    pick s **from** $todo$
    $known \leftarrow known \cup \{s\}$
    for $s' \in succ(s)$ **do**
        if $s' \notin known \cup todo$ **then**
            $todo \leftarrow todo \cup \{s'\}$
        end if
    end for
end while

Sequential algorithm (2)

$todo \leftarrow \{s_0\}$
$known \leftarrow \emptyset$
while $todo \neq \emptyset$ **do**
    pick s **from** $todo$
    $known \leftarrow known \cup \{s\}$
    $todo \leftarrow (todo \cup succ(s)) \setminus known$
end while

Do you truth me ? Is there a bug ?

The Why tools



- Annoted programs (small algorithmic/logic language)
- generated proof obligations for theorem provers
- need axiomatisation for set/list etc.

BSP model

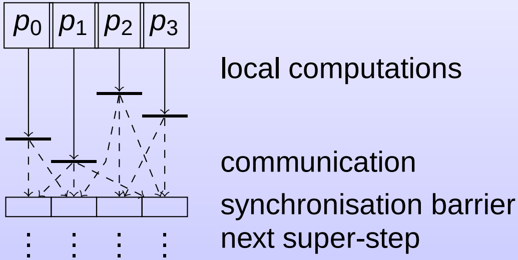


Figure: A BSP super-step

Building a BSP algorithm

Four steps:

1. start from a naive algorithm
2. improve the locality of computation
3. improve the memory consumption
4. balance computation

Naive BSP algorithm

- partition function *cpu* to place states onto processors
  - hash the state (modulo number of processors)
  - most used approach
- each processor i computes $succ(s)$ iff $cpu(s) = i$
- other states are sent to their owners
- stop when no processor has computed new states

Main loop

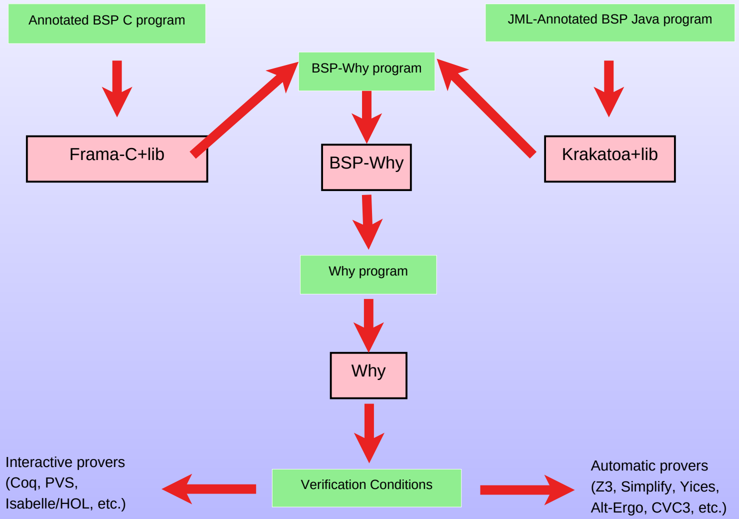
$todo \leftarrow \emptyset$
$total \leftarrow 1$
$known \leftarrow \emptyset$
if $cpu(s_0) = mypid$ **then**
    $todo \leftarrow todo \cup \{s_0\}$
end if
while $total > 0$ **do**
    $tosend \leftarrow Successor(known, todo)$
    $todo, total \leftarrow BSP\_Exchange(known, tosend)$
end while

$Successor(known, todo)$

$tosend \leftarrow \emptyset$
while $todo \neq \emptyset$ **do**
  pick s **from** $todo$
  $known \leftarrow known \cup \{s\}$
  for $s' \in succ(s)$ **do**
    if $s' \notin known \cup todo$ **then**
      if $cpu(s') = mypid$ **then**
        $todo \leftarrow todo \cup \{s'\}$
      else
        $tosend \leftarrow tosend \cup \{(cpu(s'), s')\}$
      end if
    end if
  end for
end while
return $tosend$

Trying to prove its correctness

BSP-WHY, an extension of WHY for BSP algorithms:



Problems with naive algorithm

- a new state is very likely to be sent
  - **no locality** of the computations
  - **too much communication**
- protocols yield **structured** models
- we can exploit their **properties** for more efficient algorithms

Assumptions about protocol models

1. finite and fixed number of agents
2. states are functions $\mathcal{L} \to \mathcal{D}$ (locations $\to$ data)
3. $\mathcal{L}_R \subseteq \mathcal{L}$ defines *reception locations*
4. $succ = succ_R \uplus succ_L$ such that $succ_L$ is the identity on $\mathcal{L}_R$
5. cpu_R such that $s_1|_{\mathcal{L}_R} = s_2|_{\mathcal{L}_R} \Rightarrow cpu_R(s_1) = cpu_R(s_2)$
   - use a hash on $\mathcal{L}_R$

$\Rightarrow$ cross-transitions correspond to receptions in the protocol

General assumptions, easy to implement

Improve local computation

$Successor(known, todo)$:

while $todo \neq \emptyset$ **do**
  pick s **from** $todo$
  $known \leftarrow known \cup \{s\}$
  for $s' \in succ(s)$ **do**
    if $s' \notin known \cup todo$ **then**
      if $cpu(s') = mypid$ **then**
        $todo \leftarrow todo \cup \{s'\}$
      else
        $tosend \leftarrow tosend \cup \{(cpu(s'), s')\}$
      end if
    end if
  end for
end while

$tosend \leftarrow \emptyset$
while $todo \neq \emptyset$ **do**
  pick s **from** $todo$
  $known \leftarrow known \cup \{s\}$
  for $s' \in succ_L(s) \setminus known$ **do**
    $todo \leftarrow todo \cup \{s'\}$
  end for
  for $s' \in succ_R(s) \setminus known$ **do**
    $tosend \leftarrow tosend \cup \{(cpu_R(s'), s')\}$
  end for
end while
return $tosend$

Improve memory consumption

- super-steps match protocols progression (receptions)
- known states cannot be reached in a future super-step
- at each super-step, we can **dump** memory

$todo \leftarrow \emptyset$
$total \leftarrow 1$
$known \leftarrow \emptyset$
if $cpu(s_0) = mypid$ **then**
    $todo \leftarrow todo \cup \{s_0\}$
end if
while $total > 0$ **do**
    $tosend \leftarrow Successor(known, todo)$
    dump($known$)
    $todo, total \leftarrow BSP\_Exchange($~~known,~~$tosend)$
end while

Balancing the computation

1. efficient parallel computation $\iff$ good workload balance
2. number of computed states cannot be anticipated
3. but, related to the number of received states
4. introduce a rebalancing step

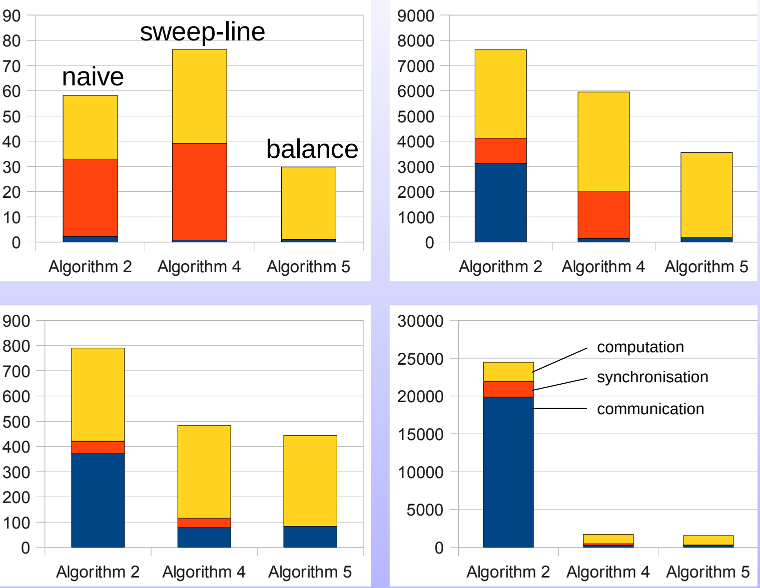
```
while total > 0 do
    tosend ← Successor(known, todo)
    dump(known)
    todo, total ← BSP_Exchange(Balance(tosend))
end while
```

- define cpu_B as cpu_R for infinitely many processors
- exchange local histograms of $cpu_B \Rightarrow$ global histogram H
- balance *tosend* wrt H (approximate bin packing)

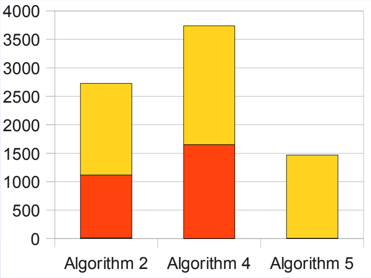
Methodology

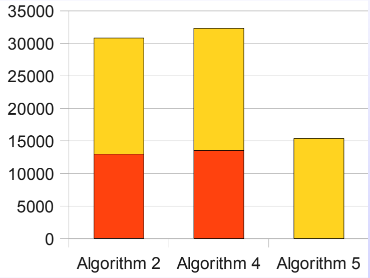
- run two scenarios (small and large) for:
  1. Needham-Schroeder mutual authentication
  2. Yahalom key sharing and mutual authentication with TTP
  3. Otway-Rees key sharing with TTP
  4. Kao-Chow key sharing and mutual authentication
- measure on each processor
  1. local computation time (yellow)
  2. synchronisation time, *i.e.*, waiting time (red)
  3. communication time (blue)

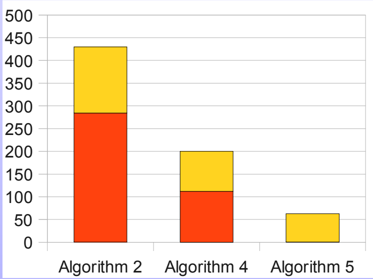
Needham-Schroeder (top) and Yahalom (bottom)

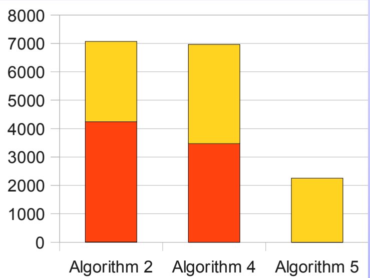


Otway-Rees (top) and Kao-Chow (bottom)









NS and Yahalom

Scenario	Naive	Balance	Nb_states
NS_1-2	0m50.222s	0m42.095s	7807
NS_1-3	115m46.867s	61m49.369s	530713
NS_2-2	112m10.206s	60m30.954s	456135

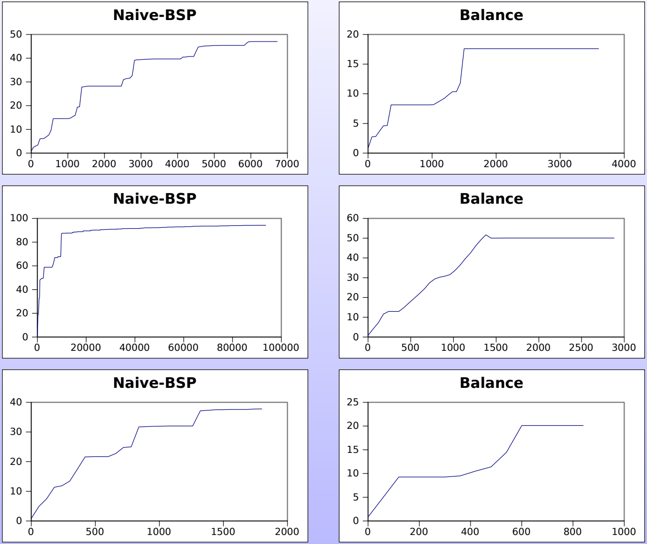
Scenario	Naive	Balance	Nb_states
Y_1-3-1	12m44.915s	7m30.977s	399758
Y_1-3-1_2	30m56.180s	14m41.756s	628670
Y_1-3-1_3	481m41.811s	25m54.742s	931598
Y_2-2-1	2m34.602s	2m25.777s	99276
Y_3-2-1	**COMM**	62m56.410s	382695
Y_2-2-2	2m1.774s	1m47.305s	67937

Otway-Rees and Woo and Lam Pi

Scenario	Naive	Balance	Nb_states
OR_1-1-2	38m32.556s	24m46.386s	12785
OR_1-1-2_2	196m31.329s	119m52.000s	17957
OR_1-1-2_3	411m49.876s	264m54.832s	22218
OR_1-2-1	21m43.700s	9m37.641s	1479

Scenario	Naive	Balance	Nb_states
WLP_1-1-1	0m12.422s	0m9.220s	4063
WLP_1-1-1_2	1m15.913s	1m1.850s	84654
WLP_1-1-1_3	**COMM**	24m7.302s	785446
WLP_1-2-1	2m38.285s	1m48.463s	95287
WLP_1-2-1_2	**SWAP**	55m1.360s	946983

Memory consumption for NS, WLP and Y



Conclusion

- BSP algorithms for state space generation
- exploit structural properties of security protocols to
  - reduce communications
  - anticipate the number of super-steps
  - decrease local storage
  - balance the workload
- properties easily computable on Petri net models
- our algorithm is
  - efficient
  - scalable
  - simple
  - provable
- benchmarks on realistic and non-trivial examples

Perspectives

- ongoing work
  - new benchmarks of other protocols and scenarios
  - correctness proofs using BSP Hoare logic (BSP-WHY)
- future work
  - more complex protocol from SPREAD project
  - LTL/CTL* model checking, exploiting state space locality
  - certified implementation ?