BSP algorithms for on-the-fly checking CTL* formulas on security protocols

Frédéric Gava, Michael Guedj and Franck Pommereau

Laboratory of Algorithms, Complexity and Logic (LACL)
University of Paris-East
Outline

1. Security Protocols, State-space and logic
2. BSP Algorithms for Model-checking Security Protocols
3. Conclusion
Outline

1. Security Protocols, State-space and logic
2. BSP Algorithms for Model-checking Security Protocols
3. Conclusion
Reminder and solutions

Problematic and related solutions

- Theorem provers
- Unbound number of agents: ProVerif, Scyther, NRL, etc.
- "Bound": AVISPA (SAT, rewrite)

Chosen solution
Problematic and related solutions

- Theorem provers
- Unbound number of agents: ProVerif, Scyther, NRL, etc.
- “Bound”: AVISPA (SAT, rewrite)
Problematic and related solutions

- Theorem provers
- Unbound number of agents: ProVerif, Scyther, NRL, etc.
- “Bound”: AVISPA (SAT, rewrite)

Chosen solution
Reminder and solutions

Problematic and related solutions

- Theorem provers
- Unbound number of agents: ProVerif, Scyther, NRL, etc.
- “Bound”: AVISPA (SAT, rewrite)

Chosen solution

The intruder induces a combinatorial explosion
⇒ BSP algorithms to check security properties
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
   - Uses only allowed operations (perfect cryptography)
3. **Spy** delivers messages (including the original one)
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
   - Uses only allowed operations (perfect cryptography)

Spy delivers messages (including the original one)
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
   - Uses only allowed operations (*perfect cryptography*)
3. **Spy** delivers messages (including the original one)
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
   - Uses only allowed operations (**perfect cryptography**)
3. **Spy** delivers messages (including the original one)
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
   - **Uses only allowed operations** *(perfect cryptography)*
3. **Spy** delivers messages *(including the original one)*
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
   - Uses only allowed operations (*perfect cryptography*)
3. **Spy** delivers messages (including the original one)
Tools: SNAKES and BSP-Python

- SNAKES (ABCD) for modelling protocols and scenarios
- BSP-Python (CNRS Physic Orleans)
  - BSP library for Python
  - Used of global exchanges
- LACL cluster → 40 CPU BSP machine
- Short cycles: algorithm → implementation → benchmarks
- Not efficient but accurate for algorithms comparison
Tools: SNAKES and BSP-Python

- SNAKES (ABCD) for modelling protocols and scenarios
- BSP-Python (CNRS Physic Orleans)
  - BSP library for Python
  - Used of global exchanges
- LACL cluster → 40 CPU BSP machine
- Short cycles: algorithm → implementation → benchmarks
- Not efficient but accurate for algorithms comparison
Tools: SNAKES and BSP-Python

- SNAKES (ABCD) for modelling protocols and scenarios
- BSP-Python (CNRS Physic Orleans)
  - BSP library for Python
  - Used of global exchanges

LACL cluster ⇒ 40 CPU BSP machine

Short cycles: algorithm → implementation → benchmarks

Not efficient but accurate for algorithms comparison
Tools: SNAKES and BSP-Python

- SNAKES (ABCD) for modelling protocols and scenarios
- BSP-Python (CNRS Physic Orleans)
  - BSP library for Python
  - Used of global exchanges

- LACL cluster $\Rightarrow$ 40 CPU BSP machine
- Short cycles: algorithm $\mapsto$ implementation $\mapsto$ benchmarks
- Not efficient but accurate for algorithms comparison
Tools: **SNAKES** and **BSP-Python**

- **SNAKES (ABCD)** for modelling protocols and scenarios
- **BSP-Python** (CNRS Physic Orleans)
  - BSP library for Python
  - Used of global exchanges
- **LACL cluster** ➔ 40 CPU BSP machine
  - Short cycles: algorithm ➔ implementation ➔ benchmarks
  - Not efficient but accurate for algorithms comparison
Tools: SNAKES and BSP-Python

- SNAKES (ABCD) for modelling protocols and scenarios
- BSP-Python (CNRS Physic Orleans)
  - BSP library for Python
  - Used of global exchanges
- LACL cluster $\Rightarrow$ 40 CPU BSP machine
- Short cycles: algorithm $\leftrightarrow$ implementation $\leftrightarrow$ benchmarks
- Not efficient but accurate for algorithms comparison
Tools: SNAKES and BSP-Python

- **SNAKES (ABCD)** for modelling protocols and scenarios
- **BSP-Python** (CNRS Physic Orleans)
  - **BSP library** for Python
  - **Used of global exchanges**
- **LACL cluster** $\Rightarrow$ 40 CPU BSP machine
- **Short cycles**: algorithm $\leftrightarrow$ implementation $\leftrightarrow$ benchmarks
- **Not efficient but accurate** for algorithms comparison
Bridging model: **Bulk Synchronous Parallelism (BSP)**

**The BSP computer**

**Defined by:**
- \( p \) pairs CPU/memory
- Communication network
- Synchronisation unit

**Properties:**
- "Confluent"
- "Deadlock-free"
- Predictable performances
- Super-steps execution
Bridging model: **Bulk Synchronous Parallelism (BSP)**

The **BSP** computer

Defined by:
- \( p \) pairs **CPU/memory**
- **Communication network**
- Synchronisation unit

Properties:
- "Confluent"
- "Deadlock-free"
- Predictable performances
- Super-steps execution

...
Bridging model: **Bulk Synchronous Parallelism (BSP)**

**The BSP computer**

Defined by:
- **p** pairs CPU/memory
- Communication network
- Synchronisation unit

Properties:
- "Confluent"
- "Deadlock-free"
- Predictable performances
- Super-step execution

---

F. Gava — Évry 2013
Bridging model: **Bulk Synchronous Parallelism (BSP)**

**The BSP computer**

Defined by:
- $p$ pairs CPU/memory
- Communication network
- Synchronisation unit

**Properties:**
- “Confluent”
- “Deadlock-free”
- Predictable performances
- Super-steps execution

Diagram illustrating local computations, communication barrier, and next super-step.
Bridging model: **Bulk Synchronous Parallelism (BSP)**

**The BSP computer**

Defined by:
- \( p \) pairs CPU/memory
- Communication network
- Synchronisation unit

**Properties:**
- "Confluent"
- "Deadlock-free"
- Predictable performances
- Super-steps execution
Bridging model: **Bulk Synchronous Parallelism (BSP)**

### The BSP computer

**Defined by:**
- \( p \) pairs CPU/memory
- Communication network
- Synchronisation unit

### Properties:
- “Confluent”
- “Deadlock-free”
- Predictable performances
- Super-steps execution
Bridging model: **Bulk Synchronous Parallelism (BSP)**

**The BSP computer**

Defined by:
- \( p \) pairs CPU/memory
- Communication network
- Synchronisation unit

**Properties:**
- “Confluent”
- “Deadlock-free”
- Predictable performances
- Super-steps execution

![Diagram of BSP computer]
State-space construction

**Definition**
- Initial state \( s_0 \) and successor \( s' \in \text{succ}(s) \equiv s \rightarrow s' \)
- Inductive definition \( \Rightarrow \) a set of reachable states (or graph)
- Most security properties (secrecy, authentication, etc.)

```plaintext
let seq_succ () =
...
while todo ≠ ∅ do
  let s = pick todo in
  known ← !known ⊕ s;
  todo ← !todo ∪ (succ(s) \ !known)
done
```

**Main idea**
- \( cpu \) assigns states to nodes
- Each processor \( i \) computes \( s' \in \text{succ}(s) \)
  iff \( cpu(s') = i \); otherwise sends \( s' \)
- Stops when no new states
State-space construction

**Definition**

- Initial state $s_0$ and successor $s' \in \text{succ}(s) \equiv s \rightarrow s'$
- Inductive definition $\Rightarrow$ a set of reachable states (or graph)
- Most security properties (secrecy, authentication, etc.)

```ocaml
let seq_succ () =
  ...
  while todo \neq \emptyset do
    let s = pick todo in
    known := !known \oplus s;
    todo := !todo \cup (succ(s) \setminus !known)
  done
```

### Parallelization

**Main idea**

- *cpu* assigns states to nodes
- Each processor $i$ computes $s' \in \text{succ}(s)$ iff $cpu(s') = i$; otherwise sends $s'$
- Stops when no new states

```ocaml
let naive_par_state_space () =
  ...
  while total \gt 0 do
    let tosend = (seq_succ known todo pastsend) in
    exchange todo total known pastsend tosend
  done
```
State-space construction

Definition

- Initial state \( s_0 \) and successor \( s' \in \text{succ}(s) \equiv s \rightarrow s' \)
- Inductive definition \( \Rightarrow \) a set of reachable states (or graph)
- Most security properties (secrecy, authentication, etc.)

```ocaml
let seq_succ () =
...
while todo \( \neq \) \( \emptyset \) do
  let s = pick todo in
  known <- !known \( \oplus \) s;
  todo <- !todo \( \cup \) \( \text{succ}(s) \setminus !\text{known} \)
done
```

Parallelization

```ocaml
let naive_par_state_space () =
...
while total > 0 do
  let tosend = (seq_succ known todo pastsend) in
  exchange todo total known pastsend tosend
done
```

Main idea

- \( \text{cpu} \) assigns states to nodes
- Each processor \( i \) computes \( s' \in \text{succ}(s) \) iff \( \text{cpu}(s') = i \); otherwise sends \( s' \)
- Stops when no new states

F. Gava — Évry 2013
State-space construction

Definition
- Initial state $s_0$ and successor $s' \in \text{succ}(s) \equiv s \rightarrow s'$
- Inductive definition $\Rightarrow$ a set of reachable states (or graph)
- Most security properties (secrecy, authentication, etc.)

```plaintext
let seq_succ () =
...
while todo \= \emptyset do
  let s = pick todo in
  known \leftarrow !known \oplus s;
  todo \leftarrow !todo \cup (\text{succ}(s) \setminus !known)
done
```

Parallelization

```plaintext
let naive_par_state_space () =
...
while total>0 do
  let tosend = (seq_succ known todo pastsend) in
  exchange todo total known pastsend tosend
done
```

Main idea
- **cpu** assigns states to nodes
- Each processor $i$ computes $s' \in \text{succ}(s)$ iff $\text{cpu}(s') = i$; otherwise sends $s'$
- Stops when no new states
Assumptions about protocol models

- \( \text{succ} = \text{succ}_L \uplus \text{succ}_R \) such that \( \text{succ}_R \) is only “reception”

- New \( \text{cpu} \), hash on these parts

\[ \Rightarrow \text{Cross-transitions correspond to receptions in the protocol} \]
Assumptions about protocol models

1. $\text{succ} = \text{succ}_L \uplus \text{succ}_R$ such that $\text{succ}_R$ is only “reception”

2. New $cpu$, hash on these parts

$\Rightarrow$ Cross-transitions correspond to receptions in the protocol
Assumptions about protocol models

1. \( succ = succ_L \cup succ_R \) such that \( succ_R \) is only “reception”

2. New \( cpu \), hash on these parts

⇒ Cross-transitions correspond to receptions in the protocol
Assumptions about protocol models

1. \( \text{succ} = \text{succ}_L \cup \text{succ}_R \) such that \( \text{succ}_R \) is only “reception”

2. New \( \text{cpu} \), hash on these parts

\[ \Rightarrow \text{Cross-transitions correspond to receptions in the protocol} \]
**Improvements**

**Local computations**
- Compute **locally** only for $succ_L$
- Send only after $succ_R$

**Memory consumption**
- Super-steps match protocols progression $\equiv$ slices

**Balance**
- Classes of states
- Histograms of classes $\rightarrow$ balanced computations

---

F. Gava — Évry 2013
**Improvements**

**Local computations**
- Compute locally only for $\text{succ}_L$
- Send only after $\text{succ}_R$

**Memory consumption**
- Super-steps match protocols progression $\equiv$ slices
- Dump memory

**Balance**
- Classes of states
- Histograms of classes $\rightarrow$ balanced computations
Improvements

Local computations
- Compute **locally** only for $succ_L$
- Send only after $succ_R$

Memory consumption
- Super-steps **match** protocols progression $\equiv$ slices
- Dump memory

Balance
- Classes of states
- Histograms of classes $\rightarrow$ balanced computations
Improvements

Local computations
- Compute **locally** only for $succ_L$
- Send only after $succ_R$

Memory consumption
- Super-steps **match** protocols progression ≡ slices
- **Dump** memory

Balance
- Classes of states
- Histograms of classes → balanced computations
Improvements

Local computations
- Compute **locally** only for $\text{succ}_L$
- **Send** only after $\text{succ}_R$

Memory consumption
- **Super-steps** match protocols
  - progression $\equiv$ slices
- **Dump** memory

Balance
- **Classes** of states
  - Histograms of classes $\rightarrow$ balanced computations
Improvements

Local computations
- Compute **locally** only for $\text{succ}_L$
- **Send** only after $\text{succ}_R$

Memory consumption
- Super-steps **match** protocols progression $\equiv$ slices
- **Dump** memory

Balance
- **Classes** of states
- **Histograms** of classes $\rightarrow$ balanced computations
Model-Checking: introduction

Temporal logic

Examples:
- **Availability** (LTL) \( AG(\text{rcv}(a, m) \Rightarrow F(\neg \text{rcv}(a, m))) \)
- **Fairness** (CTL) \( AG(\text{gift}(c, d) \Rightarrow EF(\text{money}(d, c))) \)

Verification (on-the-fly)
- LTL: unroll the formula, i.e. build a proof-structure (graph of assertions) and find an invalid SCC
- CTL*: unroll the formula and execute "LTL sessions"
Model-Checking: introduction

Temporal logic

Examples:
- **Availability (LTL)** \( AG(rcv(a, m) \Rightarrow F(\neg rcv(a, m))) \)
- **Fairness (CTL)** \( AG(gift(c, d) \Rightarrow EF(money(d, c))) \)

Verification (on-the-fly)

- **LTL**: unroll the formula, i.e. build a proof-structure (graph of assertions) and find an invalid SCC
- **CTL\(^*\)**: unroll the formula and execute “LTL sessions”
Model-Checking: introduction

Temporal logic

Examples:
- **Availability** (LTL) \( AG(rcv(a, m) \Rightarrow F(\neg rcv(a, m))) \)
- **Fairness** (CTL) \( AG(gift(c, d) \Rightarrow EF(money(d, c))) \)

Verification (on-the-fly)
- LTL: unroll the formula, i.e build a proof-structure (graph of assertions) and find an invalid SCC
- CTL*: unroll the formula and execute “LTL sessions”
Model-Checking: introduction

Temporal logic

Examples:
- **Availability** (LTL) \(\mathcal{L}(rcv(a, m) \Rightarrow F(\neg rcv(a, m)))\)
- **Fairness** (CTL) \(\mathcal{C}(gift(c, d) \Rightarrow EF(money(d, c)))\)

Verification (on-the-fly)

- **LTL**: unroll the formula, i.e. build a proof-structure (graph of assertions) and find an invalid SCC
- **CTL\(^*\)**: unroll the formula and execute “LTL sessions”
What is LTL (Linear Temporal Logic)?

Syntax

- Normal logic: \( a \land b, a \lor b, \text{ etc.} \)
- Temporal logic: \( pUq, pVq, Xp \)
What is LTL (Linear Temporal Logic)?

Syntax
- Normal logic: $a \land b$, $a \lor b$, etc.
- Temporal logic: $pUq$, $pVq$, $Xp$

Semantics
Works on all the paths (A) of the state-space (Kripke structure):

$X\phi : \bullet \rightarrow \bullet \phi \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \cdots$

$\phi_1 U \phi_2 : \bullet \phi_1 \rightarrow \bullet \phi_1 \rightarrow \bullet \phi_1 \rightarrow \bullet \phi_2 \rightarrow \bullet \rightarrow \cdots$

$\phi_1 V \phi_2 : \bullet \phi_2 \rightarrow \bullet \phi_2 \rightarrow \bullet \phi_2 \rightarrow \bullet \phi_2 \rightarrow \bullet \phi_2 \rightarrow \cdots$

or
$\bullet \phi_2 \rightarrow \bullet \phi_2 \rightarrow \bullet \phi_2 \rightarrow \bullet \phi_1 \land \phi_2 \rightarrow \bullet \rightarrow \cdots$
What is LTL (Linear Temporal Logic)?

**Syntax**
- Normal logic: \( a \land b, a \lor b, \text{etc.} \)
- Temporal logic: \( pUq, pVq, Xp \)

**Semantics**
Works on all the paths (A) of the state-space (Kripke structure):

\[
X\phi : \bullet \rightarrow \bullet \phi \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \cdots
\]

\[
\phi_1 U \phi_2 : \bullet \phi_1 \rightarrow \bullet \phi_1 \rightarrow \bullet \phi_1 \rightarrow \bullet \phi_2 \rightarrow \bullet \rightarrow \cdots
\]

\[
\phi_1 V \phi_2 : \bullet \phi_2 \rightarrow \bullet \phi_2 \rightarrow \bullet \phi_2 \rightarrow \bullet \phi_2 \rightarrow \bullet \phi_2 \rightarrow \cdots
\]

or
\[
\bullet \phi_2 \rightarrow \bullet \phi_2 \rightarrow \bullet \phi_2 \rightarrow \bullet \phi_1 \land \phi_2 \rightarrow \bullet \rightarrow \cdots
\]

Generally Buchi automata are used for verification ...
Proof-structure (graph) and SCC verification

\[ s \models A(\Phi, \phi) \quad (R1) \quad s \models A(\Phi) \quad (R2) \quad s \models A(\Phi, \phi_1 \lor \phi_2) \quad (R3) \]

\[ s \models A(\phi) \quad \text{if } s \models \phi \]

\[ s \models A(\phi_1 \land \phi_2) \quad (R4) \]

\[ s \models A(\phi_1) \quad s \models A(\phi_2) \]

\[ s \models A(\phi_1 \lor \phi_2) \quad (R6) \]

\[ s \models A(\phi_1, \ldots, \phi_n) \quad (R7) \]

\[ s_1 \models A(\phi_1, \ldots, \phi_n) \quad s_m \models A(\phi_1, \ldots, \phi_n) \quad \text{if } \text{succ}(s) = \{s_1, \ldots, s_m\} \]

Verification of a LTL formula: **SCC (Strongly Connected Components)** with only successful infinite paths: if at some point a formula of the form \( \phi_1 \lor \phi_2 \) is only repeatedly “regenerated” by application of rule R6
Example of a state-space (Kripke structure)

$s_1 \models A$

$s_2 \models A \land B$

$s_3 \models A$
Example of a proof-structure

Kripke structure:

$s_1 \models A$

$s_2 \models A \land B$

$s_3 \models A$

Proof structure:

$s_1 \models B \land V A$

$s_2 \models B \land V A$

$s_3 \models B \land V A$

$s_1 \models A$

$s_2 \models A$

$s_3 \models A$

$s_1 \models B, X(B \land V A)$

$s_2 \models B, X(B \land V A)$

$s_3 \models B, X(B \land V A)$

$s_1 \models X(B \land V A)$

$s_2 \models X(B \land V A)$

$s_3 \models X(B \land V A)$

$s_2 \models X(B \land V A)$

true

true

true
Outline

1. Security Protocols, State-space and logic

2. BSP Algorithms for Model-checking Security Protocols

3. Conclusion
BSP LTL verification for security protocols

- Rules R7 is decomposed into two rules. Example:

\[
\begin{align*}
    s &\vdash A(X\phi_1, \ldots, X\phi_n) \\
    s_1 &\vdash A(\phi_1, \ldots, \phi_n) \\
    s_m &\vdash A(\phi_1, \ldots, \phi_n)
\end{align*}
\]

\quad \text{if } \text{succ}_L(s) = \{s_1, \ldots, s_m\}

- Now SCCs can only be local (on each processor):
  - Any sequential algorithm for SCC can be used (Tarjan)
  - No extra need of communications

- Assertions (state+formula) are distributed only using the L subpart of the state

- When founding a “bad” SCC, building the trace
BSP LTL verification for security protocols

- Rules R7 is decomposed into two rules. Example:
  \[ s \vdash A(X\phi_1, \ldots, X\phi_n) \]
  \[ s_1 \vdash A(\phi_1, \ldots, \phi_n) \quad s_m \vdash A(\phi_1, \ldots, \phi_n) \] (R7')
  if \( \text{succ}_{L}(s) = \{s_1, \ldots, s_m\} \)

- Now SCCs can only be local (on each processor):
  - Any sequential algorithm for SCC can be used (Tarjan)
  - No extra need of communications

- Assertions (state+formula) are distributed only using the L subpart of the state

- When founding a “bad” SCC, building the trace
BSP LTL verification for security protocols

- Rules R7 is **decomposed** into two rules. Example:
  \[
  s \models A(X\phi_1, \ldots, X\phi_n) \\
  s_1 \models A(\phi_1, \ldots, \phi_n) \\
  s_m \models A(\phi_1, \ldots, \phi_n) \\
  \text{(R7')} \\
  \]
  if \( \text{succ}_L(s) = \{s_1, \ldots, s_m\} \)

- Now SCCs can only be **local** (on each processor):
  - Any sequential algorithm for SCC can be used (**Tarjan**)
  - No extra need of communications

- Assertions (state+formula) are distributed only using the \( L \) subpart of the state

- When founding a “bad” SCC, building the trace
Rules R7 is decomposed into two rules. Example:

\[ s \vdash A(X\phi_1, \ldots, X\phi_n) \]

\[ s_1 \vdash A(\phi_1, \ldots, \phi_n) \quad s_m \vdash A(\phi_1, \ldots, \phi_n) \quad (R7') \]

if \( \text{succ}_{L}(s) = \{s_1, \ldots, s_m\} \)

Now SCCs can only be local (on each processor):

- Any sequential algorithm for SCC can be used (Tarjan)
- No extra need of communications

Assertions (state+formula) are distributed only using the \( L \) subpart of the state

When founding a “bad” SCC, building the trace
BSP LTL verification for security protocols

- Rules R7 is decomposed into two rules. Example:
  \[ s \vdash A(X\phi_1, \ldots, X\phi_n) \]

  \[ s_1 \vdash A(\phi_1, \ldots, \phi_n) \quad s_m \vdash A(\phi_1, \ldots, \phi_n) \]  

  \[ (R7') \]

  if \( succ_L(s) = \{s_1, \ldots, s_m\} \)

- Now SCCs can only be local (on each processor):
  - Any sequential algorithm for SCC can be used (Tarjan)
  - No extra need of communications

- Assertions (state+formula) are distributed only using the \( L \) subpart of the state

- When founding a “bad” SCC, building the trace
BSP LTL verification for security protocols

- Rules R7 is decomposed into two rules. Example:
  \[ s \vdash A(X\phi_1, \ldots, X\phi_n) \]
  \[ s_1 \vdash A(\phi_1, \ldots, \phi_n) \]
  \[ s_m \vdash A(\phi_1, \ldots, \phi_n) \]
  \[ (R7') \]
  if \( \text{succ}_L(s) = \{ s_1, \ldots, s_m \} \)

- Now SCCs can only be local (on each processor):
  - Any sequential algorithm for SCC can be used (Tarjan)
  - No extra need of communications

- Assertions (state+formula) are distributed only using the \( L \) subpart of the state

- When founding a “bad” SCC, building the trace
Main loop

```python
def ParChkLTL((s ⊨ Φ) as σ) is
  Init_main()
  while flag=⊥ ∧ total>0
    send←∅
    while todo ≠ ∅ ∧ flag=⊥
      pick σ from todo
      if σ ∉ V
        flag←SeqChkLTL(σ, send, E, V)
      if flag≠ ⊥
        send←∅
        flag, todo,total←Exchange(send, flag)
    case flag
      | ⊥ → print "OK"
      | σ → Build_trace(σ)
```
Methodology

- Run on large scenario for:
  1. Needham-Schroeder (NS) mutual authentication
  2. Yahalom (YA) key sharing and mutual authentication
  3. Otway-Rees (OT) key sharing
  4. Kao-Chow (KC) key sharing and mutual authentication

- Two formulas for LTL: secrecy and aliveness
- Two formulas for CTL*: secrecy and fairness
Methodology

- Run on large scenario for:
  1. Needham-Schroeder (NS) mutual authentication
  2. Yahalom (YA) key sharing and mutual authentication
  3. Otway-Rees (OT) key sharing
  4. Kao-Chow (KC) key sharing and mutual authentication

- Two formulas for LTL: secrecy and aliveness
- Two formulas for CTL*: secrecy and fairness
Methodology

- Run on large scenario for:
  1. **Needham-Schroeder** (NS) mutual authentication
  2. **Yahalom** (YA) key sharing and mutual authentication
  3. **Otway-Rees** (OT) key sharing
  4. **Kao-Chow** (KC) key sharing and mutual authentication

- Two formulas for **LTL**: secrecy and aliveness
- Two formulas for **CTL***: secrecy and fairness
Methodology

- Run on large scenario for:
  1. Needham-Schroeder (NS) mutual authentication
  2. Yahalom (YA) key sharing and mutual authentication
  3. Otway-Rees (OT) key sharing
  4. Kao-Chow (KC) key sharing and mutual authentication

- Two formulas for LTL: secrecy and aliveness
- Two formulas for CTL*: secrecy and fairness
Methodology

Run on large scenario for:

1. Needham-Schroeder (NS) mutual authentication
2. Yahalom (YA) key sharing and mutual authentication
3. Otway-Rees (OT) key sharing
4. Kao-Chow (KC) key sharing and mutual authentication

- Two formulas for LTL: secrecy and aliveness
- Two formulas for CTL*: secrecy and fairness
Methodology

- Run on large scenario for:
  1. Needham-Schroeder (NS) mutual authentication
  2. Yahalom (YA) key sharing and mutual authentication
  3. Otway-Rees (OT) key sharing
  4. Kao-Chow (KC) key sharing and mutual authentication

- Two formulas for LTL: secrecy and aliveness

- Two formulas for CTL*: secrecy and fairness
Methodology

- Run on large scenario for:
  1. Needham-Schroeder (NS) mutual authentication
  2. Yahalom (YA) key sharing and mutual authentication
  3. Otway-Rees (OT) key sharing
  4. Kao-Chow (KC) key sharing and mutual authentication

- Two formulas for LTL: secrecy and aliveness
- Two formulas for CTL*: secrecy and fairness
Speed-ups (or time)

- Needham-Schroeder protocol
- Yahalom protocol
- Otway-Rees protocol
- Kao-Chow protocol
And now CTL* in nutshell

Syntax
LTL + (A and E) possibly everywhere

Semantics
For all paths or exists a path

Verification
Decomposition of the formula from controllability. Examples:
- Chk*(s ⊢ φ₁ ∧ φ₂) ⇒ Chk*(s ⊢ φ₁) ∧ Chk*(φ₂)
- Chk*(s ⊢ Aφ) ⇒ ChkLTL(s ⊢ Aφ) (LTL session)

F. Gava — Évry 2013
And now CTL* in nutshell

**Syntax**

LTL + \((A \text{ and } E)\) possibly everywhere

**Semantics**

For all paths or exists a path

**Verification**

- Decomposition of the formula (from CTL* to LTL). Examples:
  \[
  \text{Chk}^*(s \vdash \phi_1 \land \phi_2) \Rightarrow \text{Chk}^*(s \vdash \phi_1) \land \text{Chk}^*(\phi_2)
  \]
  \[
  \text{Chk}^*(s \vdash A\varphi) \Rightarrow \text{ChkLTL}(s \vdash A\varphi) \text{ (LTL session)}
  \]

- From LTL to CTL*:
  \[
  s \vdash A(\varphi, \varphi) \quad (R2') \quad \text{if Chk}^*(s \nvdash \varphi)
  \]
And now CTL* in nutshell

Syntax
LTL + (A and E) possibly everywhere

Semantics
For all paths or exists a path

Verification
- Decomposition of the formula (from CTL* to LTL). Examples:
  \[
  \text{Chk}^*(s \vdash \phi_1 \land \phi_2) \Rightarrow \text{Chk}^*(s \vdash \phi_1) \land \text{Chk}^*(\phi_2)
  \]
  \[
  \text{Chk}^*(s \vdash A\phi) \Rightarrow \text{ChkLTL}(s \vdash A\phi) \text{ (LTL session)}
  \]
- From LTL to CTL*:
  \[
  \frac{s \vdash A(\Phi, \phi)}{s \vdash A(\phi)} \text{ (R2')} \quad \text{if } \text{Chk}^*(s \not\vdash \phi)
  \]
And now CTL* in nutshell

Syntax
LTL + (A and E) possibly everywhere

Semantics
For all paths or exists a path

Verification
- **Decomposition** of the formula (from CTL* to LTL). Examples:
  \[ \text{Chk}^*(s \vdash \phi_1 \land \phi_2) \Rightarrow \text{Chk}^*(s \vdash \phi_1) \land \text{Chk}^*(\phi_2) \]
  \[ \text{Chk}^*(s \vdash A\phi) \Rightarrow \text{Chk}_{LTL}(s \vdash A\phi) \text{ (LTL session)} \]

- From LTL to CTL*:
  \[ s \vdash A(\phi, \phi) \]
  \[ \Rightarrow \quad \frac{s \vdash A(\phi)}{(R2')} \text{ if } \text{Chk}^*(s \not\vdash \phi) \]
And now CTL* in nutshell

**Syntax**

LTL + (A and E) possibly everywhere

**Semantics**

For all paths or exists a path

**Verification**

- **Decomposition of the formula** (from CTL* to LTL). Examples:

  \[ \text{Chk}^\ast(s \vdash \phi_1 \land \phi_2) \Rightarrow \text{Chk}^\ast(s \vdash \phi_1) \land \text{Chk}^\ast(\phi_2) \]

  \[ \text{Chk}^\ast(s \vdash A\phi) \Rightarrow \text{ChkLTL}(s \vdash A\phi) \text{ (LTL session)} \]

- From LTL to CTL*:

  \[
  \frac{s \vdash A(\phi, \phi)}{s \vdash A(\phi)} \quad (R2') \text{ if Chk}^\ast(s \not\models \phi)
  \]
And now CTL* in nutshell

Syntax

LTL + (A and E) possibly everywhere

Considered solution

Computing both \( s \vdash A(\Phi) \) and \( \text{Chk}^*(s \not\vDash \phi) \) in order to have:

- A maximal of computations over the processors
- Pure breadth computations (keeping slice progression)
- Pre-computation of some assertions
- Better balance
- But with some unnecessary computations

\[ \begin{align*}
\frac{s \vdash A(\Phi, \phi)}{s \vdash A(\Phi)} \quad (R2') \quad \text{if Chk}^*(s \not\vDash \phi)
\end{align*} \]

F. Gava — Évry 2013
And now CTL* in nutshell

Syntax
LTL + (A and E) possibly everywhere

Considered solution
Computing both $s \vdash A(\Phi)$ and $\text{Chk}^*(s \not\models \phi)$ in order to have:

- A maximal of computations over the processors
- Pure breadth computations (keeping slice progression)
- Pre-computation of some assertions
- Better balance
- But with some unnecessary computations

From LTL to CTL*:

$$
\frac{s \vdash A(\Phi, \phi)}{s \vdash A(\Phi)} \quad (R2') \quad \text{if } \text{Chk}^*(s \not\models \phi)
$$
And now CTL* in nutshell

Syntax

LTL + (A and E) possibly everywhere

Considered solution

Computing both $s \models A(\Phi)$ and $\text{Chk}^*(s \not\models \phi)$ in order to have:

- A maximal of computations over the processors
- Pure breadth computations (keeping slice progression)
- Pre-computation of some assertions
- Better balance
- But with some unnecessary computations

From LTL to CTL*:

$$\frac{s \models A(\Phi, \phi)}{s \models A(\Phi)} \quad (R2') \quad \text{if } \text{Chk}^*(s \not\models \phi)$$
And now CTL* in nutshell

**Syntax**

LTL + (A and E) possibly everywhere

**Considered solution**

Computing both $s \vdash A(\Phi)$ and $\text{Chk}^*(s \not\vdash \phi)$ in order to have:

- A maximal of computations over the processors
- Pure breadth computations (keeping slice progression)
- Pre-computation of some assertions
- Better balance
- But with some unnecessary computations

**From LTL to CTL***:

$$
\frac{s \vdash A(\Phi, \phi)}{s \vdash A(\Phi)} \quad (R2') \quad \text{if Chk}^*(s \not\vdash \phi)
$$
And now CTL* in nutshell

Syntax
LTL + (A and E) possibly everywhere

Considered solution
Computing both $s \vdash A(\Phi)$ and $\text{Chk}^*(s \not\models \phi)$ in order to have:
- A maximal of computations over the processors
- Pure breadth computations (keeping slice progression)
- Pre-computation of some assertions
- Better balance
- But with some unnecessary computations

From LTL to CTL*:
$$
\frac{\frac{s \vdash A(\Phi, \phi)}{s \vdash A(\Phi) \quad (R2')}}{\text{if } \text{Chk}^*(s \not\models \phi)}
$$
And now CTL* in nutshell

**Syntax**

LTL + (A and E) possibly everywhere

**Considered solution**

Computing both $s \vdash A(\Phi)$ and $\text{Chk}^*(s \not\models \phi)$ in order to have:

- A **maximal** of computations over the processors
- Pure **breadth** computations (keeping **slice** progression)
- **Pre-computation** of some assertions
- Better **balance**
- But with some unnecessary computations

\[ \frac{s \vdash A(\Phi, \phi)}{s \vdash A(\Phi)} (R2') \text{ if } \text{Chk}^*(s \not\models \phi) \]

From LTL to CTL*: 
Model-Checking: our BSP algorithms

Algorithm for checking LTL formulas on security protocols

- SCCs are only local
  - On nodes
  - On slices/super-steps
  \[ \Rightarrow \text{simple Tarjan algorithm} \]
Model-Checking: our BSP algorithms

Algorithm for checking LTL formulas on security protocols

- SCCs are only local
  - On nodes
  - On slices/super-steps
- ⇒ simple Tarjan algorithm
Model-Checking: our BSP algorithms

Algorithm for checking LTL formulas on security protocols

- SCCs are only local
  - On nodes
  - On slices/super-steps

⇒ simple Tarjan algorithm

Algorithms for checking CTL* formulas on security protocols

- A CTL session is executed as usual
- Two solutions:
  1. Pause for a “CTL* assertion” ⇒ new LTL session
  2. “Purely Breadth”; compute both validities of the assertion
Model-Checking: our **BSP** algorithms

Algorithm for checking **LTL** formulas on security protocols

- SCCs are only **local**
  - On **nodes**
  - On **slices/super-steps**
- $\Rightarrow$ **simple** Tarjan algorithm

Algorithms for checking **CTL**\(^*\) formulas on security protocols

1. A **LTL** session is executed as usual
2. Two solutions:
   - **Pause** for a "CTL\(^*\) assertion"
     - New **LTL** session
   - "Purely Breadth"; compute both validities of the assertion

F. Gava — Évry 2013
Model-Checking: our **BSP** algorithms

Algorithm for checking **LTL** formulas on security protocols

- **SCCs** are only **local**
  - On **nodes**
  - On **slices/super-steps**
- $\Rightarrow$ **simple Tarjan algorithm**

Algorithms for checking **CTL**\(^*\) formulas on security protocols

1. **A LTL session is executed as usual**
2. **Two solutions:**
   - Pause for a "**CTL**\(^*\) assertion" $\Rightarrow$ new LTL session
Model-Checking: our BSP algorithms

Algorithm for checking LTL formulas on security protocols

- SCCs are only local
  - On nodes
  - On slices/super-steps
- ⇒ simple Tarjan algorithm

Algorithms for checking CTL* formulas on security protocols

1. A LTL session is executed as usual
2. Two solutions:
   1. Pause for a “CTL* assertion” ⇒ new LTL session
   2. “Purely Breadth”: compute both validities of the assertion
Model-Checking: our BSP algorithms

Algorithm for checking LTL formulas on security protocols

- SCCs are only local
  - On nodes
  - On slices/super-steps
- ⇒ simple Tarjan algorithm

Algorithms for checking CTL* formulas on security protocols

1. A LTL session is executed as usual
2. Two solutions:
   1. Pause for a “CTL* assertion” ⇒ new LTL session
Model-Checking: our BSP algorithms

Algorithm for checking LTL formulas on security protocols

- SCCs are only local
  - On nodes
  - On slices/super-steps
- ⇒ simple Tarjan algorithm

Algorithms for checking CTL* formulas on security protocols

1. A LTL session is executed as usual
2. Two solutions:
   1. Pause for a “CTL* assertion” ⇒ new LTL session
Model-Checking: our BSP algorithms

Algorithm for checking LTL formulas on security protocols

- SCCs are only local
  - On nodes
  - On slices/super-steps
- ⇒ simple Tarjan algorithm

Algorithms for checking CTL* formulas on security protocols

1. A LTL session is executed as usual
2. Two solutions:
   1. Pause for a “CTL* assertion” ⇒ new LTL session
   2. “Purely Breadth”; compute both validities of the assertion
Model-Checking: our BSP algorithms

**Algorithm for checking LTL formulas on security protocols**

- SCCs are only **local**
  - On nodes
  - On slices/super-steps
- ⇒ simple Tarjan algorithm

**Algorithms for checking CTL* formulas on security protocols**

1. A LTL session is executed as usual
2. Two solutions:
   1. Pause for a “CTL* assertion” ⇒ new LTL session
   2. “Purely Breadth”; compute both validities of the assertion
Model-Checking: our BSP algorithms

Algorithm for checking LTL formulas on security protocols

- SCCs are only local
  - On nodes
  - On slices/super-steps
- ⇒ simple Tarjan algorithm

Algorithms for checking CTL* formulas on security protocols

1. A LTL session is executed as usual
2. Two solutions:
   1. Pause for a “CTL* assertion” ⇒ new LTL session
   2. “Purely Breadth”; compute both validities of the assertion
Performances: speedup

Needham-Schroeder protocol

Kao-Chow protocol

Otway-Rees protocol

- Linear
- Secrecy (Naive)
- Fairness (Naive)
- Secrecy (Breadth)
- Fairness (Breadth)
Performances: naive versus breadth

### Needham-Schroeder
- **Secrecy**
  - Naive: [Diagram]
  - Breath: [Diagram]
- **Fairness**
  - Naive: [Diagram]
  - Breath: [Diagram]

### Yahalom
- **Secrecy**
  - Naive: [Diagram]
  - Breath: [Diagram]
- **Fairness**
  - Naive: [Diagram]
  - Breath: [Diagram]

### Otway-Rees
- **Secrecy**
  - Naive: [Diagram]
  - Breath: [Diagram]
- **Fairness**
  - Naive: [Diagram]
  - Breath: [Diagram]

### Kao-Chow
- **Secrecy**
  - Naive: [Diagram]
  - Breath: [Diagram]
- **Fairness**
  - Naive: [Diagram]
  - Breath: [Diagram]
Outline

1. Security Protocols, State-space and logic
2. BSP Algorithms for Model-checking Security Protocols
3. Conclusion
Conclusion

- BSP algorithms for LTL/CTL* verification of security protocols
- 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 (ABCD)
- Our algorithm is:
  - Efficient and Scalable
  - Simple and thus Provable
- Benchmarks on realistic and non-trivial examples
**Conclusion**

- **BSP algorithms for LTL/CTL* verification of security protocols**
- **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 (ABCD)
- Our algorithm is:
  - Efficient and Scalable
  - Simple and thus Provable
- Benchmarks on realistic and non-trivial examples
Conclusion

- **BSP algorithms for LTL/CTL* verification of security protocols**
- 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 (ABCD)
- **Our algorithm is:**
  - Efficient and Scalable
  - Simple and thus Provable
- **Benchmarks on realistic and non-trivial examples**
Conclusion

- **BSP algorithms for LTL/CTL*' verification of security protocols
- 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 (ABCD)
- Our algorithm is:
  - Efficient and Scalable
  - Simple and thus Provable
- Benchmarks on realistic and non-trivial examples
Conclusion

- **BSP algorithms for LTL/CTL* verification of security protocols**
- 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 (ABCD)
- **Our algorithm is:**
  - Efficient and Scalable
  - Simple and thus Provable
- **Benchmarks on realistic and non-trivial examples**

F. Gava — Évry 2013
Conclusion

- BSP algorithms for LTL/CTL* verification of security protocols
- 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 (ABCD)
- Our algorithm is:
  - Efficient and Scalable
  - Simple and thus Provable
- Benchmarks on realistic and non-trivial examples
Conclusion

- BSP algorithms for LTL/CTL* verification of security protocols
- 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 (ABCD)
  - Our algorithm is:
    - Efficient and Scalable
    - Simple and thus Provable
- Benchmarks on realistic and non-trivial examples
Conclusion

- BSP algorithms for LTL/CTL* verification of security protocols
- 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 (ABCD)
- Our algorithm is:
  - Efficient and **Scalable**
  - Simple and thus **Provable**
- Benchmarks on realistic and non-trivial examples
BSP algorithms for LTL/CTL* verification of security protocols

- 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 (ABCD)

- Our algorithm is:
  - Efficient and Scalable
    - Simple and thus Provable

- Benchmarks on realistic and non-trivial examples
BSP algorithms for LTL/CTL\(^*\) verification of security protocols

- 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 (ABCD)

- Our algorithm is:
  - Efficient and Scalable
  - Simple and thus Provable

- Benchmarks on realistic and non-trivial examples
BSP algorithms for LTL/CTL* verification of security protocols

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 (ABCD)

Our algorithm is:
- Efficient and Scalable
- Simple and thus 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 (P2P data storing) project
- Stronger logic: ATL*, with past, etc.
- Full certified implementation?
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 (P2P data storing) project
  - Stronger logic: ATL*, with past, etc.
  - Full certified implementation?
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 (P2P data storing) project
- Stronger logic: ATL*, with past, etc.
- Full certified implementation?
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 (P2P data storing) project
  - Stronger logic: ATL*, with past, etc.
  - Full certified implementation?
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 (P2P data storing) project
  - Stronger logic: ATL*, with past, etc.
  - Full certified implementation?
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 (P2P data storing) project
  - Stronger logic: ATL*, with past, etc.
  - Full certified implementation?
Ongoing work:
- New benchmarks of other protocols and scenarios
- Correctness proofs using BSP Hoare logic (BSP-WHY)

Future work:
- More complex protocol from SPREAD (P2P data storing) project
- Stronger logic: ATL*, with past, etc.
- Full certified implementation?
Thanks and Questions?