A BSP algorithm for on-the-fly checking LTL formulas on security protocols

Frédéric Gava, Michael Guedj, Franck Pommereau

A BSP algorithm for on-the-fly checking LTL formulas on security protocols

Frédéric Gava
University of Paris-East
LACL
Créteil, France
frederic.gava@univ-paris-est.fr

Michaël Guedj
University of Paris-East
LACL
Créteil, France

Franck Pommereau
University of Évry
IBISC
Évry, France
franck.pommereau@ibisc.univ-evry.fr

***Abstract*—This paper presents a Bulk-Synchronous Parallel (BSP) algorithm to compute on-the-fly whether a structured model of a security protocol satisfies a LTL formula. Using the structured nature of the security protocols allows us to design a simple and efficient parallelisation of an algorithm which constructs the state-space under consideration in a need-driven fashion. A prototype implementation has been developed, allowing to run benchmarks.**

***Keywords*-BSP; LTL; Security Protocols;**

I. Introduction

Designing secure protocols is a challenging problem [1]. In spite of their apparent simplicity, they are notoriously error-prone. Unfortunately, checking if a cryptographic protocol is secure or not is not decidable in general and NP-complete for bounded numbers of agents and sessions [2]. Model-checking is however well-adapted to find flaws [3]. In this paper, we consider the problem of checking an LTL formula over *labelled transition systems* (LTS) that model security protocols. Checking a LTL formula over a protocol is not new [4] and has the advantage over dedicated tools for protocols to be easily extensible to non standard behaviour of honest principals (*e.g.*, contract-signing protocols: participants required to make progress) or to check some security goals that cannot be expressed as reachability properties, *e.g.*, fair exchange.

But checking an LTL formula may be expensive both in terms of memory and execution time: this is the so-called state explosion problem. This is especially true when complex data-structures are used in the model as the knowledge of an intruder in security protocols. Because this checking can cause memory crashing on single or multiple processor systems, it has led to consider exploiting the larger memory space available in distributed systems [5], which also gives the opportunity to reduce the overall execution time. One of the main technical issues is to partition the state space, *i.e.* each state is assigned to a machine. Each subset of states is thus "owned" by a single machine. While it has been shown that a pure static hashing for the partition function can effectively balance the workload [24] and achieve reasonable execution time as well, this method suffers from an obvious drawbacks: it causes too much cross transitions, *i.e.*, successor states that need to be exchanged over the network thus impairing computation locality.

Also, it is rarely necessary to compute the entire state space before finding a path that invalidates the logic formula (notably a flaw in a protocol): *on-the-fly* (local) algorithms are designed to build the state space and check the formula at the same time. Two approaches are generally used: *Nested Depth First Search* (NDFS) and *Strongly Connected Components* (SCC) algorithms for detecting on-the-fly a reachable accepting cycle in the underlying graph — mainly of a Büchi automaton. The former are known to be memory efficient and the latter to be time efficient [6] and both are hard to parallelize [7].

In this paper, we exploit the well-structured nature of security protocols and match it to a model of parallel computation called BSP [8]. This allows us to simplify the writing of an efficient BSP algorithm for checking on-the-fly an LTL formula for finite protocol sessions. It is based on the algorithm of [9] which mainly combines the construction a *proof-structure* (a graph whose nodes states of the underlying Kripke structure together with sets of logical formulas) with a Tarjan's depth-first-search based SCC algorithm. The structure of the protocols is exploited to partition the state space and reduce cross transitions while increasing computation locality. At the same time, the BSP model allows to simplify the detection of the algorithm termination and to load balance the computations.

II. Context and Definitions

A. The BSP model

A BSP computer is a set of uniform processor-memory pairs connected through a communication network allowing the inter-processor delivery of messages [8]. A BSP program is executed as a sequence of *super-steps* (see Fig. 1), each one divided into three successive disjoint phases: (1) each processor only uses its local data to perform sequential computation and to request data transfers to other nodes; (2) the network delivers the requested data; (3) a global synchronisation barrier occurs, making the transferred data available for the next super-step.

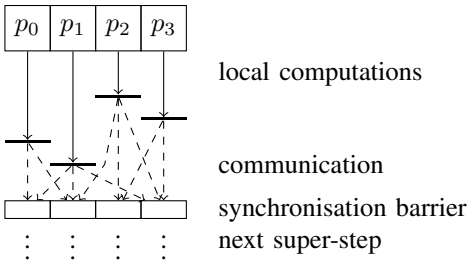

Figure 1. A BSP super-step.

Syntax of LTL:
$\phi \ ::= \ a \mid \neg a \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathbf{X}\phi \mid \phi \mathbf{U} \phi \mid \phi \mathbf{V} \phi$

Informal semantics of LTL :
$\mathbf{X}\phi : \bullet \rightarrow \bullet^{\phi} \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \cdots$

$\phi_1 \mathbf{U} \phi_2 : \bullet^{\phi_1} \rightarrow \bullet^{\phi_1} \rightarrow \bullet^{\phi_1} \rightarrow \bullet^{\phi_2} \rightarrow \bullet \rightarrow \cdots$

$\phi_1 \mathbf{V} \phi_2 : \bullet^{\phi_2} \rightarrow \bullet^{\phi_2} \rightarrow \bullet^{\phi_2} \rightarrow \bullet^{\phi_2} \rightarrow \bullet^{\phi_2} \rightarrow \cdots$
or $\bullet^{\phi_2} \rightarrow \bullet^{\phi_2} \rightarrow \bullet^{\phi_2} \rightarrow \bullet^{\phi_1 \wedge \phi_2} \rightarrow \bullet \rightarrow \cdots$

Figure 2. Syntax and informal semantics of LTL.

```
def bsp_state_space() is
  todo,known←∅,∅
  total←1
  if cpu(s_0) =mypid
    todo←todo∪{s_0}
  while total>0
    tosend←Successor(known,todo)
    todo,total←Exchange(known,tosend)
 return known

def Successor(known,todo) is
  tosend←∅
  while todo≠ ∅
    pick s from todo
    known←known∪{s}
    todo←(todo ∪ succ_L(s)) \ known
    for s' ∈ succ_R(s)
      tosend←tosend ∪{ (cpu(s'),s')}
  return tosend

def Exchange(known,tosend) is
  dump(known)
  return BSP_EXCHANGE(Balance(tosend))

def Balance(tosend) is
  histoL←{(i,♯{(i,s) ∈ tosend})}
  compute histoG from BSP_EXCHANGE(histoL)
  return BinPack(tosend,histoG)
```

Figure 3. BSP computing the state space of protocols

B. State space of security Protocols [10]

In this paper, we consider models of security protocols, involving a set of *agents*, given as a labelled transition system (LTS). We also consider a Dolev-Yao attacker that resides on the network [11]. An execution of such a model is thus a series of message exchanges as follows. (1) An agent sends a message on the network. (2) This message is captured by the attacker that tries to learn from it by recursively decomposing the message or decrypting it when the key to do so is known. Then, the attacker forges all possible messages from newly as well as previously learnt informations (*i.e.*, attacker's knowledge). Finally, these messages (including the original one) are made available on the network. (3) The agents waiting for a message reception accept some of the messages forged by the attacker, according to the protocol rules.

As a concrete formalism to model protocols, we have used an *algebra of coloured Petri nets* called ABCD [12, Sec. 3.3] allowing for easy and structured modelling. However, our approach is largely independent of the chosen formalism and it is enough to assume that the following properties hold: (P1) LTS function succ can be partitioned into two successor functions $\mathsf{succ}_{\mathcal{R}}$ and $\mathsf{succ}_{\mathcal{L}}$ that correspond respectively to transitions upon which an agent (except the intruder) receives information (and stores it), and to all the other transitions; (P2) there is an initial state s_0 and there exists a function slice from states to natural numbers (a measure) such that if $s' \in \mathsf{succ}_{\mathcal{R}}(s)$ then there is no path from s' to any state s'' such that $\mathsf{slice}(s) = \mathsf{slice}(s'')$ and $\mathsf{slice}(s') = \mathsf{slice}(s) + 1$ (it is often call a sweep-line progression [13]); (P3) there exists a function cpu from states to natural numbers (a hashing) such that for all state s if $s' \in \mathsf{succ}_{\mathcal{L}}(s)$ then $\mathsf{cpu}(s) = \mathsf{cpu}(s')$; mainly, the knowledge of the intruder is not taken into account to compute the hash of a state; (P4) if $s_1, s_2 \in \mathsf{succ}_{\mathcal{R}}(s)$ and $\mathsf{cpu}(s_1) \neq \mathsf{cpu}(s_2)$ then there is no possible path from s_1 to s_2 and *vice versa*.

On concrete models, it is generally easy to distinguish syntactically the transitions that correspond to a message reception in the protocol with information storage. Thus, is it easy to partition succ as above and, for most protocol models, it is also easy to check that the above properties are satisfied. This is the case in particular for us using the ABCD formalism.

C. BSP Computing of the state space [10]

Based on the following properties, we have designed in [10] a BSP algorithm (in a SPMD fashion) for computing the state space of security protocols as shown in Fig. 3. In this algorithm, "BSP_EXCHANGE" is a primitive that allows processors to globally exchange data: a set of pairs (pid,value) is used to define values to be sends. Mainly: (1) states are distributed across the processors using the cpu function; (2) the algorithm finishes when no states are exchanged; (3) function *Successor* is called to compute the successors of the states, then all new states from $\mathsf{succ}_{\mathcal{L}}$ are added in *todo* (states to be proceeded) and states from

$\mathsf{succ}_{\mathcal{R}}$ are sent to be treated at the next super-step, enforcing an order of exploration of the state space that match the progression of the protocol. (4) It thus becomes possible at the beginning of each super-step, to dump from the main memory all the known states because they cannot be reached anymore due to the sweep-line progression. (5) States to be sent are first balanced across the processors using an histogram histoG (which is first totally exchanged to be the same on each processor and enforce consistent decisions on all the processors: each processor send its own local histogram histoL) and according to a simple heuristic for the bin packing problem, classes of states (consistent with hash) are grouped on processors so there is no possibility of duplicated computation.

This algorithm gives better performances (less cross transitions) than a naive distributed one [10] and is able to dump all the known states at the beginning of each super-step allows to use less memory. Partial-order reductions [14] can also be introduced without really changing the algorithm.

D. Proof-structure and LTL checking [9]

Considerable attention has been devoted to the development of automatic techniques, or model-checking procedures, for verifying finite-state systems against specifications expressed using various temporal logics and notably the linear (LTL) subset. This logic permits users to characterize many properties, including safety and liveness. One may identify two basic approaches to model checking. The first uses global analysis to determine if a system satisfies a formula; the entire state space (mainly a Kripke structure) of the system is constructed and subjected to analysis. However, these algorithms may be seen to perform unnecessary work: in many cases (especially when a system does not satisfy a specification) only a subset of the states needs to be analyzed in order to determine whether or not the system satisfies a formula. On-the-fly, or local, approaches to model checking attempt to take advantage of this observation by constructing the state space in a demand-driven fashion. Due to lack of space, we do not present a formal definition of what is a Kripke structure and an LTL formula (an informal semantics is giving on Fig 2) and concentrate on the notion of proof-structure [9] for LTL checking: a collection of top-down proof rules for inferring when a state in a Kripke structure satisfies an LTL formula.

We define $M = (S, R, L)$ to be a Kripke structure where S is the set of states, $R \subset S \times S$ the relation which is assumed to be total (thus all paths in M are infinite) and $L \in S \to 2^A$ the labelling. The proof-rules appear in Fig 4 [9] and they operate on assertions of the form $s \vdash A\Phi$ where $s \in S$ and Φ is a set of path formulas. Semantically, $s \vdash A\Phi$ holds if $s \vDash A(\bigvee_{\phi \in \Phi} \phi)$. We write $A(\Phi, \phi_1, \cdots, \phi_n)$ to represent a formula of the form $A(\Phi \cup \{\phi_1, \cdots, \phi_n\})$. If σ is an assertion of the form $s \vdash A\Phi$, then we use $\phi \in \sigma$ to denote that $\phi \in \Phi$. Proof-rules are used to build proof-structures that are defined as follows:

$$\frac{s \vdash A(\Phi, \phi)}{true} \text{ if } s \vDash \phi \quad (R1) \qquad \frac{s \vdash A(\Phi, \phi)}{s \vdash A(\Phi)} \text{ if } s \nvDash \phi \quad (R2) \qquad \frac{s \vdash A(\Phi, \phi_1 \vee \phi_2)}{s \vdash A(\Phi, \phi_1, \phi_2)} \quad (R3)$$

$$\frac{s \vdash A(\Phi, \phi_1 \wedge \phi_2)}{s \vdash A(\Phi, \phi_1) \quad s \vdash A(\Phi, \phi_2)} \quad (R4)$$

$$\frac{s \vdash A(\Phi, \phi_1 \mathbf{U} \phi_2)}{s \vdash A(\Phi, \phi_1, \phi_2) \quad s \vdash A(\Phi, \phi_2, \mathbf{X}(\phi_1 \mathbf{U} \phi_2))} \quad (R5)$$

$$\frac{s \vdash A(\Phi, \phi_1 \mathbf{V} \phi_2)}{s \vdash A(\Phi, \phi_2) \quad s \vdash A(\Phi, \phi_1, \mathbf{X}(\phi_1 \mathbf{V} \phi_2))} \quad (R6)$$

$$\frac{s \vdash A(\mathbf{X}\phi_1, ..., \mathbf{X}\phi_n)}{s_1 \vdash A(\phi_1, ..., \phi_n) \quad s_m \vdash A(\phi_1, ..., \phi_n)} \text{ if } succ(s) = \{s_1, ..., s_m\} \quad (R7)$$

Figure 4. Proof rules for LTL checking [9]

Proof structures that are defined as follows:

Definition 1. *Let Σ be a set of nodes, $\Sigma' = \Sigma \cup true$, $V \subseteq \Sigma'$, $E \subseteq V \times V$ and $\sigma \in V$. Then $\langle V, E \rangle$ is a proof structure for σ if it is a maximal directed graph such that for every $\sigma' \in V$, σ' is reachable from σ, and the set $\{\sigma'' | (\sigma', \sigma'') \in E\}$ results from applying some rule to σ'.*

Intuitively, a proof structure for σ is a directed graph that is intended to represent an (attempted) "proof" of σ. In what follows, we speak of a directed graph and use traditional graph notations when speaking of proof structures. Note that in contrast with traditional definitions of proofs, proof structures may contain cycles. In order to define when a proof structure represents a valid proof of σ, we use:

Definition 2. *Let $\langle V, E \rangle$ be a proof structure. Then: (1) $\sigma \in V$ is a leaf iff there is no σ' such that $(\sigma, \sigma') \in E$. A leaf σ is successful iff $\sigma \equiv true$; (2) an infinite path $\pi = \sigma_0, \sigma_1, \cdots$ in $\langle V, E \rangle$ is successful iff for some assertion σ_i infinitely repeated on π there exists $\phi_1 \mathbf{V} \phi_2 \in \sigma_i$ such that for all $j \geq i, \phi_2 \notin \sigma_j$; (3) $\langle V, E \rangle$ is successful iff all its leaves and infinite paths are successful.*

Roughly speaking, an infinite path is successful if at some point a formula of the form $\phi_1 \mathbf{V} \phi_2$ is repeatedly "regenerated" by application of rule R6; that is, the right subgoal (and not the left one) of this rule application appears each time on the path. Note that after $\phi_1 \mathbf{V} \phi_2$ occurs on the path ϕ_2 should not, since, intuitively, if ϕ_2 would be true then the success of the path would not depend on $\phi_1 \mathbf{V} \phi_2$, while if it would be false then $\phi_1 \mathbf{V} \phi_2$ would not hold. Note also that if no rule can be applied (*i.e.*, $\Phi = \emptyset$) then the proof-structure and thus the formula is unsuccessful.

Theorem 1. *Let M be a Kripke structure with $s \in S$ and $A\phi$ an LTL formula, and let $\langle V, E \rangle$ be a proof structure for $s \vdash A\{\phi\}$. Then $s \vDash A\phi$ iff $\langle V, E \rangle$ is successful.*

One consequence of this theorem is that if σ has a successful proof structure, then all proof structures for σ are successful. Thus, in searching for a successfull proof structure for an assertion no backtracking is necessary. It also turns out that the success of a finite proof structure may be determined by looking at its strongly connected components for any accepting cycle. An obvious solution to this problem would be to construct the proof structure for the assertion and then check if the proof structure is successful. Of course, this algorithm is not on-the-fly as it does not check the success of a proof structure until after it is built. An efficient algorithm, on the other hand, combines the construction of a proof structure with the process of checking whether the structure is successful. A Tarjan's like algorithm was used in [9] but a NDFS one could also be used.

III. BSP On-the-fly LTL Checking

As explained in the previous section, we use two LTL successors functions for constructing the Kripke structure: $\mathsf{succ}_{\mathcal{R}}$ ensures a measure of progression slice that intuitivelly decomposes the Kripke structure into a sequence of slices $S_0, \ldots, S_n$ where transitions from states of S_i to states of S_{i+1} come only from $\mathsf{succ}_{\mathcal{R}}$ and there is no possible path from states of S_j to states S_i for all $i < j$. Also after $\mathsf{succ}_{\mathcal{R}}$ transitions (with different hashing), there is no possible common paths which is due to different knowledge of the agents. In this way, if we assume, as in Fig 3, a distribution of the Kripke structure across the processors using the cpu function, then the only possible accepting cycles or SCCs are locals to each processor. Thus, because proof-structures follow the Kripke structure (rule R7), accepting cycles or SCCs are also only locals. This fact ensures that any sequential algorithm to check cycles or SCCs can be used for the parallel computation.[1] Call this generic algorithm **SeqChkLTL** which takes an assertion $\sigma = s \vdash A\Phi$, a set of assertions to be sent (for the next super-step), and (V, E) the sub-part of the proof-graph (a set of assertions as vertices and a set of edges) that has been previously proceed (this sub-part can grow during this computation). Now, in the manner of [10], we can design our BSP algorithm which is mainly an iteration over the independant slices, one slice per super-step and, on each processor, working on independant sub-parts of the slice by calling **SeqChkLTL**. This algorithm is given in Fig 5.

The main function is ParChkLTL, it first calls an initialisation function in which only the one processor that owns the initial state saves it in its todo list. The variable total stores the number of states to be processed at the beginning of each super step; V and E store the proof graph; super_step stores the current super step number; dfn is used for the SCC algorithm; finally, flag is used to check whether the formula has been proved false (flag set to the violating state) or not (flag=$\bot$).

The main loop processes each σ in todo using the sequential checker SeqChkLTL, which is possible because the corresponding parts of the proof structure are independent (sec. 2.2, P4). SeqChkLTL uses subgoals to traverse the proof structure. For rules (R1) to (R6), the result remains local because the Petri net states does not change. However, for rule (R7), we compute separately the next states for $\mathsf{succ}_{\mathcal{L}}$ and $\mathsf{succ}_{\mathcal{R}}$: the former results in local states to be processed in the current step, while the latter results in states to be processed in the next step. If no local state is found but there exists remote states, we set subg$\leftarrow\{$True$\}$

[1]It is mainly admited that SCC computation gives smaller traces than NDFS. Both methods are equivalent for our purpose.

```
def Init_main() is
  super_step,dfn,V,E,todo←0,0,∅,∅,∅
  if cpu(σ_init)=mypid
    todo←todo ∪ {σ_init}
  flag, total←⊥,1

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)

def Exchange(tosend,flag) is
  dump (V,E) at super_step
  super_step←super_step+1
  tosend←tosend ∪ {(i,flag) | 0 ≤ i < p}
  rcv, total←BSP_EXCHANGE(Balance(tosend))
  flag,rcv←filter_flag(rcv)
  return flag, rcv, total

def subgoals(σ,send) is
  case σ
  | s ⊢ A(Φ, p) ⟹ subg←if s ⊢ p then {True}
       else {s ⊢ A(Φ)} (R1, R2)
  | s ⊢ A(Φ, φ1 ∨ φ2) ⟹ subg←{s ⊢ A(Φ, φ1, φ2)} (R3)
  | s ⊢ A(Φ, φ1 ∧ φ2) ⟹ subg←{s ⊢ A(Φ, φ1), s ⊢ A(Φ, φ2)} (R4)
  | s ⊢ A(Φ, φ1 U φ2) ⟹ subg←{s ⊢ A(Φ, φ1, φ2),
      s ⊢ A(Φ, φ2, X(φ1 U φ2))} (R5)
  | s ⊢ A(Φ, φ1 V φ2) ⟹ subg←{s ⊢ A(Φ, φ2),
      s ⊢ A(Φ, φ1, X(φ1 V φ2))} (R6)
  | s ⊢ A(Xφ1, ..., Xφn) ⟹
      subg←{s' ⊢ A(φ1, ...φn) | s' ∈ succ_L(s)}
      tosend←{s' ⊢ A(φ1, ...φn) | s' ∈ succ_R(s)}
      E←E ∪ {σ ↦_R σ' | σ' ∈ tosend }
      if subg=∅ ∧ tosend≠∅
        subg←{True}
      send←send ∪ tosend (R7)
  E←E ∪ {σ ↦_L σ' | σ' ∈ subg }
  return subg
```

Figure 5. A BSP algorithm for LTL checking

which indicates that the local exploration succeeded (P2) and allows to proceed to the next super step in the main loop. When all the local states have been processed, states are exchanged, which leads to the next slice (*i.e.*, the next super step). In order to terminate the algorithm as soon as one processor discovers a counterexample, each locally computed flag is sent to all the processors and the received values are then aggregated using function filter_flag that selects the non-$\perp$ flag with the lowest dfn value computed on the processor with the lowest number, which allows to ensure that every processor chooses the same flag and then computes the same trace. If no such flag is selectable, filter_flag returns $\perp$. To balance the computation, we use the number of states as well as the size of the formula (on which the number of subgoals directly depends).

Notice also that at each super step, each processor dumps V and E to its local disk, recording the super step number, in order to be able to reconstruct a trace. When a state σ that invalidates the formula is found, a trace from the initial state to σ is constructed. The data to do so is distributed among processors into local files, one per super step. We thus use exactly as many steps to rebuild the trace as we have used to reach σ. The algorithm is presented in Fig. 6: a trace π whose "oldest" state is σ is reconstructed following the proof graph backward. The processor that owns σ invokes Local_trace to find a path from a state σ', that was in todo at the beginning of the super state, to σ. Then it sends σ' to its owner to let the reconstruction continue. To simplify things, we print parts of the reconstructed trace as they are locally computed. Among the predecessors of a state, we always choose those that are not yet in the trace π (set_of_trace(π) returns the set of states in π) and selects one with the minimal dfn value (using function min_dfn), which allows to select shorter traces.

```
def Build_trace(σ) is
  end←False
  repeat
    π←ϵ
    my_round←(cpu(σ)=mypid)
    end←(σ=σ0)
    send←∅
    if my_round
      dump (V,E) at super_step
      super_step←super_step−1
      undump (V,E) at super_step
      σ,π←Local_trace(σ,π)
      F←F ∪ set_of_trace(π)
      print π
    σ←Exchange_trace(my_round,σ)
  until ¬end

def Exchange_trace(my_round,tosend,σ) is
  if my_round
    tosend←tosend ∪ {(i,σ) | 0 ≤ i < p}
  {σ},_←BSP_EXCHANGE(tosend)
  return σ

def Local_trace(σ,π) is
  if σ = σ0
    return (σ,π)
  tmp←prec(σ) \ set_of_trace(π)
  if tmp=∅
    σ'←min_dfn(prec(σ))
  else
    σ'←min_dfn(tmp)
  π←π.σ'
  if σ' ↦R σ
    return(σ',π)
  return Error_trace(σ',π)
```

Figure 6. Building the trace after an error

IV. Experimental Results

In order to evaluate our algorithm, we have used two formulas of the form ϕ U deadlock, where deadlock is an atomic proposition that holds iff state has no successor and ϕ is a formula that checks for an attack on the considered protocol: Fml1 is the classical "secrecy" and Fml2 is "aliveness" [15] – which are the most common formulas for verifying security protocols. The chosen formulas globally hold so that the whole proof graph is computed. Indeed, on several instances with counterexamples, we have observed that the sequential algorithm can be faster than the parallel version when a violating state can be found quickly: our parallel algorithm uses a global breadth-first search while the sequential exploration is depth-first, which usually succeeds earlier. But when all the exploration has to be performed, which is widely acknowledged as the hardest case, our algorithm is always much faster. Moreover, we sometimes could not compute the state space sequentially while the distributed version succeeded, thanks to the distribution of states and sweep-line strategy — which is also used for sequential computing.

We have implemented a prototype version in Python, using SNAKES [16] for the Petri net part (which also allowed for a quick modelling of the protocols, including the Dolev-Yao attacker) and a Python BSP library [17] for the BSP routines (which are close to an MPI "alltoall"). We actually used the MPI version (with MPICH) of the BSP-Python library. While largely suboptimal (Python programs are interpreted and there is no optimisation about the representation of the states in SNAKES and the implementation of the attacker is not optimal at all), this prototype nevertheless allows an accurate *comparison* for acceleration. The benchmarks presented below have been performed using a cluster with 20 PCs connected through a 1 Gigabyte Ethernet network. Each PC is equipped with a 2GHz Intel® Pentium® dual core CPU, with 2GB of physical memory. This allowed to simulate a BSP computer with 40 processors equipped with 1GB of memory each.

Our case studies involved the following four protocols: (1) Needham-Schroeder public key protocol for mutual authentication; (2) Yahalom key distribution and mutual authentication using a trusted third party; (3) Otway-Rees

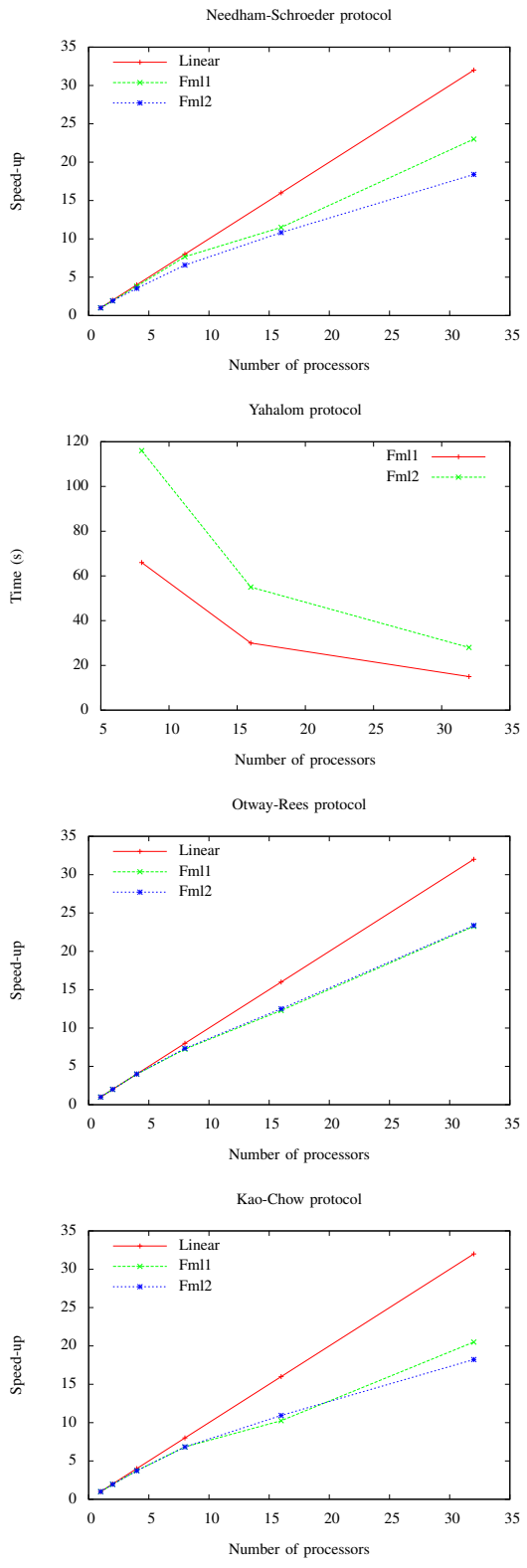

Figure 7. Benchmark results for the four protocols where Fml1 is "secrecy" and Fml2 'aliveness"

key sharing using a trusted third party; (4) Kao-Chow key distribution and authentication. These protocols and their security issues are documented at the Security Protocols Open Repository (SPORE) [18]. Fig 7 gives the speed-up for each the two formulas and two sessions of each protocol. For the Yahalom protocol, the computation fails due to a lack of main memory (swapping) if less that 4 nodes are used: we could thus not give the speedup but only times. We observed a relative speedup with respect to the number of processors. Finally, measuring the memory consumption of our algorithm, we could also confirm the benefits of our sweep-line implementation when large state spaces are computed.

V. Related works

A. *Verification of security protocols*

There are many tools dedicated to the modelling and verification of security protocols as [3], [19]. Most of them limit possible kinds of attacks or limit in their model language how addresses of agents can be manipulated in ad-hoc protocols (using arithmetic operations). Paper [20] presents different cases study of verifying security protocols with various standard tools. To summarise, there is currently no tool that provides all the expected requirements.

A distributed memory algorithm with its tool for verification of security protocols is described in [21]. The authors use buffering principle and employ a cache of recently sent states in their implementation to decrease the number of messages sent. Unfortunately, the verification of temporal properties is not supported due to the difficulties of combining the parallel checking with the symmetry reduction. [22] allows to verify some properties about some classes of protocols for an infinite number of sessions and with some possibility of replay using a process algebra. But no logic can be used here and each time a new property is needed, a new theorem needs to be proved. That can be complicated for the maintenance of the method. Also, the method cannot be applied to, *e.g.*, the Yahalom protocol. On the contrary, our approach is based on a modelling framework with explicit state space construction, that is not tied to any particular application domain and our implementation using Python allows us to manipulate any kind of data-structures are used for the modelling protocols. Using Python has been shown a good trade-off between quick modelling and performance [12] and model compilation approaches can be successfully applied to compete with state-of-the-art tools as shown in [23]. So, instead of using a dedicated framework, our approach mainly relies on the particular structure of security protocols.

B. *Distributed LTL checking*

The main idea of most known approaches to the distributed memory state space generation is similar to the naive algorithm [24]. More references can be found in [5] and in [25] for high-level Petri nets. Close to our hashing technique, [26] presents a hashing function that ensures

that most of the successors are local. For load balancing, [27] presents a new dynamic partition function scheme that builds a dynamic remapping, based on the fact that the state space is partitioned into more pieces than the number of involved machines. When load on a machine is too high, the machine releases one of the partitions it is assigned and if it is the last machine owning the partition it sends the partition to a less loaded machine.

Very close to our idea, we can cite [28] which used a partition function that enables cycles for a parallel NDFS algorithm to be local only (as for us) using SCC in the formula Büchi automata. The limits of the method are the cost of this function and furthermore the number of SCCs which is not enough to scale. [29] presents a distributed algorithm for SCC computation. In our work, all SCC are purely local, which is easier and more efficient to handle.

VI. Conclusion and future work

A. Conclusion

There are now many tools that check the security of cryptographic protocols. But none is sufficient and adaptable for complicated scenarios. Using LTL for such applications is not new but we have exploited characteristics of these models to structure the parallel computations accordingly. Our solution is to use the well-structured nature of security protocols to choose which part of the state and formulas is really needed for the partition function and to empty the data-structure at each super-step of the parallel computation. Our solution also entails automated classification of states and dynamic mapping of classes to processors. We find that our method ensures good acceleration and allows to find small counterexamples due to its breadth-first search global strategy but DFS stategy on each processor. Furthermore, we find that our method to balance states does indeed achieve better network use, memory balance and runs faster than methods based on direct states exchanges.

The common method for LTL checking is using a Büchi automaton. Using proof-structures instead theoretically has the same worst-case time but it ensures to distinguish easily local and global successors for the distribution. We have demonstrated techniques that prove the feasibility of this approach and showed its potential. Key elements to our success were (1) an automated states classification that reduces cross transitions and memory footprint, while improving the locality of computation (2) using global barriers (which is a low-overhead method) to compute a global remapping of states and thus improve balancing workload, achieving a good scalability.

B. Future work

For future work we think about extending the logic and increasing the performances. First, proof-structures were used to check CTL* formula in [9]. We are currently working to extend our algorithm for this logic and we may also consider Past operators (in the manner of [30]) that proved to be useful for protocols [4]. Second, and more pratical: we want to have a tool that could translate HSPSL models [4] into ABCD ones since HSPSL is mainly used by the community; It will also allow us to test our algorithm on different attackers than the Dolev-Yao one. To optimize the performance, using a specific library as Divine [31] and model-compilation [23] will also be considered.

We are also working on the formal proof of our algorithm. Proving a verification algorithm is highly desirable in order to certify the truth of the delivered diagnostics. Such a proof is possible because, thanks to the BSP model, our algorithm remains simple in its structure which allows us to use a specific tool for checking BSP algorithms using Hoare logic [32]. Finally, we would like to generalise our present results by extending the application domain. In the security domain, we will consider more complex protocols with branching and looping structures, as well as complex data types manipulations. In particular, we will consider protocols for secure storage distributed through peer-to-peer communication [33], [34] because it is currently modeled using ABCD and generates large state spaces.

References

[1] H. Comon-Lundh and V. Cortier, "How to prove security of communication protocols? a discussion on the soundness of formal models w.r.t. computational ones," in *STACS*, 2011, pp. 29–44.

[2] M. Rusinowith and M. Turuani, "Protocol insecurity with finite number of sessions is np-complete," in *14th Computer Security Foundations Workshop (CSFW)*. IEEE, 2001, pp. 174–190.

[3] A. Armando and *et al.*, "The AVISPA tool for the automated validation of Internet security protocols and applications," in *Proceedings of Computer Aided Verification (CAV)*, ser. LNCS, K. Etessami and S. K. Rajamani, Eds., vol. 3576. Springer, 2005, pp. 281–285.

[4] A. Armando, R. Carbone, and L. Compagna, "Ltl model checking for security protocols," *Applied Non-Classical Logics*, 2009.

[5] J. Barnat, "Distributed memory LTL model checking," Ph.D. dissertation, Faculty of Informatics Masaryk University Brno, 2004.

[6] S. Schwoon and J. Esparza, "A note on on-the-fly verification algorithms," in *TACAS*, ser. LNCS, N. Halbwachs and L. D. Zuck, Eds., vol. 3440. Springer, 2005, pp. 174–190.

[7] J. H. Reif, "Depth-first search is inherrently sequential," *Information Processing Letters*, vol. 20, no. 5, pp. 229–234, 1985.

[8] R. H. Bisseling, *Parallel Scientific Computation. A structured approach using BSP and MPI*. Oxford University Press, 2004.

[9] G. Bhat, R. Cleaveland, and O. Grumberg, "Efficient on-the-fly model checking for ctl*," in *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 1995, pp. 388–398.

[10] F. Gava, M. Guedj, and F. Pommereau, "A bsp algorithm for the state space construction of security protocols," in *PDMC*. IEEE Computer Society, 2010, pp. 37–44.

[11] D. Dolev and A. C. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–208, 1983.

[12] F. Pommereau, *Algebras of coloured Petri nets*. Lambert Academic Publisher, 2010, iSBN 978-3-8433-6113-2.

[13] S. Christensen, L. M. Kristensen, and T. Mailund, "A sweep-line method for state space exploration," in *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. LNCS, T. Margaria and W. Yi, Eds., vol. 2031. Springer, 2001, pp. 450–464.

[14] M. Torabi Dashti, A. Wijs, and B. Lisser, "Distributed partial order reduction for security protocols," *ENTCS*, vol. 198, pp. 93–99, 2008.

[15] R. Corin, "Analysis models for security protocols," Ph.D. dissertation, University of Twente, 2006.

[16] F. Pommereau, "Quickly prototyping Petri nets tools with SNAKES," in *Proc. of PNTAP'08*, ser. ACM Digital Library. ACM, 2008, pp. 1–10.

[17] K. Hinsen, "Parallel scripting with Python," *Computing in Science & Engineering*, vol. 9, no. 6, 2007.

[18] LSV, ENS Cachan, "SPORE: Security protocols open repository," http://www.lsv.ens-cachan.fr/Software/spore.

[19] C. J. F. Cremers, "Scyther - semantics and verification of security protocols," Ph.D. dissertation, Technische Universiteit Eindhoven, 2006.

[20] N. Dalal, J. Shah, K. Hisaria, and D. Jinwala, "A comparative analysis of tools for verification of security protocols," *Int. J. Communications, Network and System Sciences*, vol. 3, pp. 779–787, 2010.

[21] U. Stern and D. L. Dill, "Parallelizing the murφ verifier," in *Proceedings of Computer Aided Verification (CAV)*, ser. LNCS, O. Grumberg, Ed., vol. 1254. Springer, 1997, pp. 256–267.

[22] H. Gao, "Analysis of security protocols by annotations," Ph.D. dissertation, Technical University of Denmark, 2008.

[23] L. Fronc and F. Pommereau, "Optimising the compilation of petri net models," in *SUMO'11*, ser. CEUR, vol. 726. CEUR-WS, 2011.

[24] H. Garavel, R. Mateescu, and I. Smarandache, "Parallel state space construction for model-checking," in *Workshop on Model Checking of Software SPIN*, May 2001.

[25] C. Pajault, "Model checking parallèle et réparti de réseaux de Petri colorés de haut-niveau," Ph.D. dissertation, Conservatoire National des Arts et Métiers, 2008.

[26] D. Petcu, "Parallel explicit state reachability analysis and state space construction," in *Proceedings of ISPDC*. IEEE Computer Society, 2003, pp. 207–214.

[27] A. Lluch-Lafuente, "implified distributed model checking by localizing cycles," Institute of Computer Science at Freiburg University, Tech. Rep. 176, 2002.

[28] J. Barnat, L. Brim, and I. Cëerná, "Property driven distribution of nested dfs," in *Workshop on Verification and Computational Logic (VCL)*, M. Leuschel and U. Ultes-Nitsche, Eds., vol. DSSE-TR-2002-5. Dept. of Electronics and Computer Science, University of Southampton (DSSE), UK, Technical Report, 2002, pp. 1–10.

[29] J. Barnat, J. Chaloupka, and J. V. D. Pol, "Distributed Algorithms for SCC Decomposition," *Journal of Logic and Computation*, vol. 21, no. 1, pp. 23–44, 2011.

[30] P. Gastin and D. Oddoux, "LTL with past and two-way very-weak alternating automata," in *28th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, ser. LNCS. Springer Verlag, 2003.

[31] J. Barnat, L. Brim, M. Černá, and P. Ročkai, "DiVinE: Parallel Distributed Model Checker (Tool paper)," in *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC 2010)*. IEEE, 2010, pp. 4–7.

[32] J. Fortin and F. Gava, "BSP-Why: an intermediate language for deductive verification of BSP programs," in *4th International Workshop on High-level Parallel Programming and Applications (HLPP 2010, affiliated to conference ICFP 2010)*. ACM Press, 2010.

[33] S. Sanjabi and F. Pommereau, "Modelling, verification, and formal analysis of security properties in a P2P system," in *Workshop on Collaboration and Security (COLSEC'10)*, ser. IEEE Digital Library. IEEE, 2010, pp. 543–548.

[34] S. Chaou, G. Utard, and F. Pommereau, "Evaluating a peer-to-peer storage system in presence of malicious peers," in *SPCLOUD-11*, vol. to appear. IEEE, 2011.