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

Frédéric Gava
University of Paris-East
frederic.gava@univ-paris-est.fr

Michaël Guedj
University of Paris-East

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

***Abstract*—This paper presents a distributed algorithm to compute on-the-fly whether a structured model of a security protocol satisfies or not a CTL* formula. The design of this simple and still efficient algorithm is possible by using the structured nature of security protocols. A prototype implementation has been developed, allowing to run benchmarks.**

***Keywords*-BSP; CTL*; Security Protocols;**

I. Introduction

Security protocols are small distributed programs which aim at guaranteeing security properties such as confidentiality of data, authentication of participants, *etc*. It has long been a challenge to determine whether a given protocol is secure or not [1]. Model-checking is common solution to find flaws [2]. In this paper, we consider the problem of checking a CTL* formulae over *labelled transition systems* (LTS) that model security protocols.

The problem of checking a CTL* formula is that the generation of large discrete state spaces of security protocols is so a computationally intensive activity with extreme memory demands, highly irregular behaviour, and poor locality of references: this is a case of the so-called state explosion problem. It has thus led to consider exploiting the larger memory space available in distributed systems [3] and to reduce the overall execution time. One of the main technical issues is to partition the state space: each subset of states is thus "owned" by a single machine. Also, it is rarely necessary to compute the entire state space before finding a path that invalidates the logic formula (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. Depth First Search is the common solution but hard to parallelize [4].

In this paper, we exploit the well-structured nature of security protocols to have a specialized partition function and we used a model of parallel computation called BSP [5] to simplify the design of our algorithm which is a parallelisation of the algorithm of [6]. It 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 algorithm.

II. Context and Definitions

A. The BSP model

A BSP computer is a set of uniform processor-memory pairs connected through a communication network [5]. 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 a 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.

B. State space of security protocols [7]

We models security protocols as a labelled transition system (LTS) where *agents* send messages over a network which contains a Dolev-Yao attacker [8]. The intruder can overhear, intercept, and synthesise any message and is only limited by the constraints of the cryptographic methods used. As a concrete formalism to model protocols, we have used an *algebra of coloured Petri nets* called ABCD [9, Sec. 3.3] but our approach is largely independent of the chosen formalism. 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); (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*.

C. BSP computing of the state space [7]

Based on the following properties, we have designed in [7] a BSP algorithm (in a SPMD fashion) for computing the state space of security protocols as shown in Fig. 2. 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

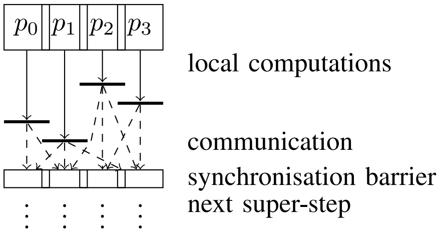

Figure 1. A BSP super-step.

```
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 2. BSP computing the state space of protocols.

Syntax of CTL* formulas:
State S ::= $a \mid \neg a \mid S \wedge S \mid S \vee S \mid \mathbf{A}P \mid \mathbf{E}P$
Path P ::= $S \mid P \wedge P \mid P \vee P \mid \mathbf{X}S \mid S\mathbf{U}S \mid S\mathbf{V}S$

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

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

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

Figure 3. Syntax and informal semantics of modal operators.

successors of the states, then all new states from succ$_{\mathcal{L}}$ are added in *todo* (states to be proceeded) and states from 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 partition function cpu) are grouped on processors so there is no possibility of duplicated computation. This algorithm gives better performances than a naive distributed one and is able to dump all the known states at the beginning of each super-step allows to use less memory. Partial-order reductions [10] can also be trivially introduced.

D. Proof-structure and LTL/CTL* checking [6]

The CTL* logic permits users to characterise many properties both linear and branching time. Syntax and informal semantics are giving on Fig. 3 where $\mathbf{A}$ and $\mathbf{E}$ are path quantifiers *i.e.* for$\mathbf{A}$ll paths in the LTS, *resp.* $\mathbf{E}$xists a path. Two important sublogics are CTL (every path modality is immediately preceded by a path quantifier) and LTL — formulas $\mathbf{A}P$ where the only state sub-formulas of P are propositions. CTL* checking can done using a collection of top-down proof rules for inferring when a state in a Kripke structure satisfies an LTL formula [6].

We define $M = (S, R, L)$ to be a Kripke structure where S is the set of states, $R \subset S \times S$ the LTS relation which is assumed to be total (thus all paths in M are infinite) and $L \in S \rightarrow 2^A$ the labelling. The (only) seven proof-rules are fully available in [6] (Fig. 4 only presents rules that are used in this work) 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:

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'.*

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 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 $s \vdash \mathbf{A}\{\phi\}$. Then $s \vDash A\phi$ iff $\langle V, E\rangle$ is successful.*

It turns out that the success of a finite proof structure

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

$$\text{if } s \models \phi \qquad\qquad \text{if } s \not\models \phi$$

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

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

$$\text{if } succ(s) = \{s_1, ..., s_m\}$$

Figure 4. Proof rules for LTL checking [6].

may be determined by looking at its strongly connected components (SCC) for any accepting cycle. A Tarjan's like algorithm is used in [6] and let us name it SeqChkLTL.

Now for CTL* checking, [6] remarks that an efficient algorithm (let name it SeqChkCTL⋆) could simplify be a recursive decomposition of the state formulas into subformulas until reaches assertions and calls SeqChkLTL appropriately when it encounters assertions of the form $s \vdash \mathbf{A}\Phi$ or $s \vdash \mathbf{E}\Phi$. SeqChkLTL is also modified as follows: each time the SCC computation of the proof-graph found an assertion of the form $s \vdash \mathbf{A}(\Phi)$ where Φ is not an LTL sub-formula then it recursively calls SeqChkCTL⋆ on Φ to determine if $s \models \Phi$ (semantically valid for s) and then decides if rule R1 or rule R2 (of Fig. 4) needs to be applied. We have thus a double-recursively of SeqChkLTL and SeqChkCTL⋆ as the syntax of CTL* suggests.

III. BSP On-the-fly CTL* Checking

For lack of space, all the algorithms are available in [11].

A. BSP on-the-fly LTL checking

As explained in the previous section, we use two LTS successors functions for constructing the Kripke structure: $\mathsf{succ}_{\mathcal{R}}$ ensures a measure of progression "slice" that intuitively decomposes the Kripke structure into a sequence of slices $S_0, \ldots, S_n$ (n is the maximal number of possible protocol sessions) 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 partition cpu), there is no possible common paths which is due to different knowledge of the agents — honest and intruder.

In [12], we have show that using the distributed state space generation of Section II-C, states and thus assertions of the proof-structures are distributed such as a SCC (if exists) can only be *local on a processor and on a slice*: 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. That is, it is sufficient to perform sequential SCC computations on each processor and for each super-step to found flaws — an unsuccessful SCC.

Between each super-step, assertions are distributed according to the balance function of states and thus our BSP algorithm for LTL checking is mainly an iteration over the independent slices, one slice per super-step and, on each processor, working on independent sub-parts of the slice by calling SeqChkLTL each time for the received assertions — furthermore, in the case of multi-core processors, these computations can also be done purely in parallel. Notice 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 by reconstructed traces as they are locally computed and by following the proof-structure backward even on distant sending— see [12] for the details.

B. A naive BSP algorithm for CTL* checking

The algorithm works as follow. As in Fig. 2, a main loop is used to compute over received assertions and for each of them, a SeqChkCTL⋆ is used to decompose the formulae and run SeqChkLTL adequately to check for an unsuccessful SCC in the proof-structure. We name this computation a "session". During the generation of the proof-structure, when a sub-formulae beginning by $\mathbf{A}$ or $\mathbf{E}$ is found (case of rules R1 and R2), the ongoing "session" is halting and is now waiting the result of a new "session" which is running for checking the validity of $s \models p$ — where p could be any CTL* formulae. The ongoing session is push on a stack of waiting sessions.

The main problems are: (1) different processors can throw sessions; (2) a session can induce several super-steps (slices) if it is a path formulae (use of modal operators); this is due to the double recursion of the CTL* checking; (3) the different sessions are not fully disjoints; states of the Kripke structures as well as assertions can be shared: this happens when the same sub-parts of the Kripke structure are generated and when sets of formulas in the assertions are not disjoints. There is thus not possibility of embarrassingly parallel computations on this set of sessions. A naive solution is to select, by all the processors, one of these generated sessions (the other remaining in a distributed global stack) and to entirely compute this session until another (child) session is throw or an answer is find — validity of $s \models p$. But this naive approach has many drawbacks.

First, each time a session is throw, this session can traverse all the state-space — in several super-steps. This can happen when the session has been throwed by a formulae which contains model operators. This can thus generated too much barriers and inducees a poor latency. Second, the sweep-line technical used in the previous section could not holds: each slice does not correspond to a super-step and thus during backtracking of the answers, the save on disks assertions must be entered in the main memory: this can be also costly. Otherwise, we can keep them all in the main memory but with a risk of swapping. Third, the balance of the assertions over the processors is done dynamically at each slice of each session: this ensures that two assertions for the same Kripke's state would be hold by the same processor. That ensures no duplication of the computations. But if two sessions are run in sequence, the first one will balance some assertions and the second session if shared the same states (but for a different set of logical formulas) must balance the assertions depending of this first partial balance: this is not optimal. A naive solution is to re-balance the assertions but it would be too costly. Fortunately, in our experience, the number of classes and the small number of assertions in each class are sufficient to not have too poor balancing even using partial informations.

C. A "purely breadth" BSP algorithm for CTL* checking

To avoid these problems we will take into account the "nature" of the proof-structures: having an explicit decomposition of the logical formulae which can help to choose where a parallel computation is needed or not. The main idea of the algorithm is based on the rules R1 and R2: computing $s \models \phi$ together with $s \vdash A(\Phi)$. In this way, we will able to choice which rule (R1 or R2) can be applied. As above, the computation of $s \models \phi$ would be performed by a LTL session while the computation of $s \vdash A(\Phi)$ would be performed by following the execution of the sequential Tarjan algorithm — SCC computation. In a sense, we expect the result of $s \models \phi$ by computing the validity of the assertion $s \vdash A(\Phi)$.

We see three main advantages. First, as we computed "simultaneously" both $s \models \phi$ and $s \vdash A(\Phi)$, we would aggregated the super-steps of the both computations and thus reduced to the maximal number of slices of the model. Second, we also aggregated the computations and the communications without unbalanced them; similarly, we would have all the assertions (and more) of each slice, which implies a better balance of the computations than the use of the partial balances of the naive algorithm. Third, the computation of the validity of $s \vdash A(\Phi)$ can be used latter in different LTL sessions. On the other side, the precomputation of $s \vdash A(\Phi)$ may generated unnecessary works, but, if we suppose a sufficient number of processors, this is not a problem for scalability: the exploration is in a breadth fashion that allows us a highest degree of parallelization.

The algorithm works as this of Fig. 2 [11]: performed until the answer of the initial assertion is computed and each super-step corresponds to a slice. The difficulty in this algorithm is the management of the answers. Indeed, we do not know, *a priori*, the answer of an assertion when we computed the validity of $s \models \phi$ or where it has been send to another processor. Thus, we need to modify the backtracking when an answer is unknown by considering a third possibility of answers: $\perp$ (and the following equation $\neg\perp = \perp$) for the case when we cannot conclude. In this manner, the "session" is halting until a theu boolean answer be computed — mainly in the next slice *i.e.* next super-step. For the management of the sending assertions, we use two distinct sets, one to store the assertions to continue the exploration of the distributed proof-structure and the second for backtracking answers. In this way, at the begin of a super-step, we first read for answers to possibly unlock halting sessions (store in a stack) which could now continue their works — SCC computations. Then, the algorithm explores the sub-parts of the proof-structures of the received assertions. All these works are done until the initial assertion (of the first session) has its answer. In the case of a flaw, we rebuild the trace as for LTL checking [12].

For the sweeping of assertions we have that states and thus assertions do not overlap between different slices. But this does not still work since some assertions do not have their answers (equal to $\perp$) during a slice. We can thus not sweep them into disks when changing of slice. To continue to sweep assertions that are no longer needed (they have their answers and are belong to a previous slice), we used a variable CACHE which contains all the assertions — the implicit graph for proof-structures and LTL sessions is memorised by additional fields in assertions, thus there is no overcost of memory. At each end of treatment of a session, we iterate on CACHE to sweep into disk unnecessary assertions making more main memory available for the next sessions.

IV. Experimental Results

We have implemented a prototype version in Python, using SNAKES [9] for the Petri net part and the BSP Python library for the BSP routines (which are close to an MPI "alltoall"). 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.

Our case studies involved the following four security protocols: Needham-Schroeder, Yahalom, Otway-Rees and Kao-Chow, all described in http://www.lsv.ens-cachan.fr/Software/spore/. In order to evaluate our two algorithms, we have used two formulas: the first is a LTL formula [2] for testing secrecy of the protocols whereas the second is CTL and is for fairness — both are common formulas for verifying security protocols. On several simple instances of the protocols 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. Thus, the chosen formulas globally hold so that the whole proof-structure is computed so that our parallel algorithms always run faster — and this is widely acknowledged as the hardest case.

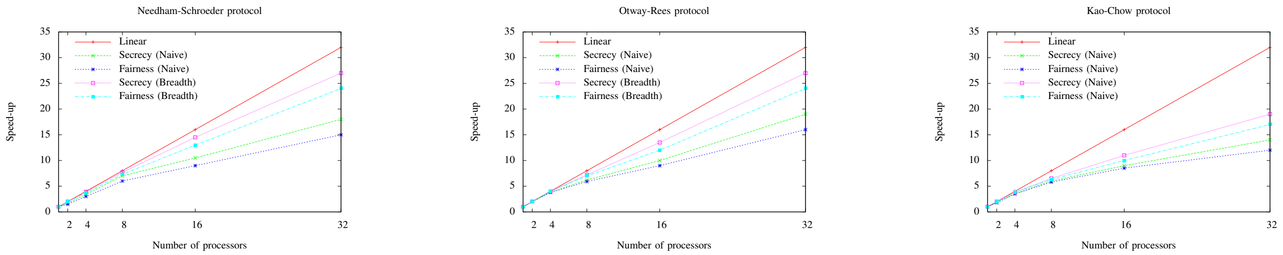

Figure 5. Speedup results for three of the protocols.

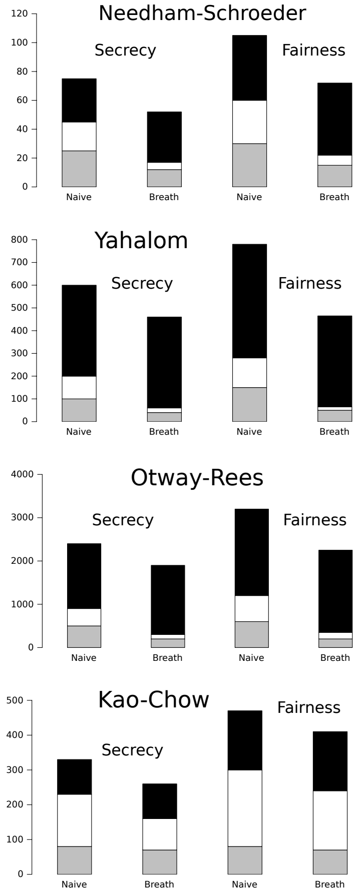

Figure 6. Timing of the two algorithms (Naive and Breadth) and formula (Secrecy and Fairness) for the protocols in seconds where times for the computations are in black, communications in gray and waiting in white.

In Fig. 5, we give the speedup of the two algorithms presented above ("Naive" and "Breadth") for three different protocols and for the two formulas — for the Yahalom protocol, the computation fails due to a lack of main memory if less that 4 nodes are used. As we might expect, the naive algorithm less scales for both formulas. Note that for Kao-Chow, both algorithms do not scale well: this is mainly due to a lack of possible attacks (even if one win); the protocol is less parallelized which implies less classes of states.

Fig. 6 shows the execution times for our two formulas for each protocol for 32 processors. In the figure, the total execution time is split into three parts: the computation time (black) that essentially corresponds to the computation of successful SCC of the proof-structures on each processor; the global and thus collective communication time (gray) that corresponds to assertions exchange; the waiting time, *i.e.* latencies (white) that occur when processors are forced to wait the others before to enter the communication phase of each super-step. Notice that because of the BSP model, these costs are obtained by considering the maximum times among the processors within each super-step, accumulated over the whole computation. We can see on these graphs that the overall performance of our "Breath" algorithm is always good compared to the naive one. As expected, the "Breath" algorithm reduce both latencies (due to less super-steps and a better balance) and communications — since they are more *en masse*. Fairness needs more computations since it is a more complicated formulae: the more the formulae and the model are bigger, the more the "Breath" algorithm is better. Finally, measuring the memory consumption of our algorithms, we could also confirm the benefits of our sweep-line strategy when large state spaces are computed.

V. Related works

There are many tools dedicated to the modelling and verification of security protocols [1]. 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 [13] 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. On the contrary, our approach is based on model-checking that is not tied to any particular application domain. Using CTL*, we can also express many complex properties that some dedicated tool cannot. But that also restrict our approach to finite scenarios and bound number of agents.

The main idea of most known approaches to the distributed memory state space generation is similar to the naive algorithm [3]. More references can be found in [14] for LTL checking. Close to our idea, we can cite [15] which used a partition function that enables cycles for a parallel NDFS algorithm to be only local. The limits of the method are the cost of this function and furthermore the number of SCCs which is not enough to scale.

A kind of tree (hesitant) Büchi automata is used in [16] where parallel SCC computations are perform. The automata is hesitant is the sense that as for rules R1 and R2, it cannot conclude and thus initiates the two possible computations. That thus generated what they call "games" (close to our "sessions") and the algorithm has to manage how to store partial results of games. A shared memory computation and heuristics are used here to simplify this management. The algorithm has also expensive management of invalid SCCs which is seems not feasible for a distributed architecture.

VI. Conclusion and future work

Designing security protocols is complex and often error prone: various attacks are reported in the literature to protocols thought to be "correct" for many years. There are now many tools that check the security of cryptographic protocols. But none is sufficient and adaptable for complicated scenarios. The use of CTL* can help to check non trivial property (*e.g.* fairness) but is computationally intensive. Distributed computation is one of the solution to reduce this overall time. 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 as much as possible the data-structures at each super-step of the parallel computation. We thus a trick to precomputed the partial validity of logical assertions to force a more breadth search which is clearly more coarse-grained even if it induces more communications. Our solution also entails automated classification of states and dynamic mapping of classes to processors which simplifies the research of logical flaws and improve balancing workload.

In future work we want to improve our implementation using compilation and not Python byte-code interpretation which is too slow for comparison with other tools as AVISPA [17]. We also want to develop tools to automatically transform standard representations of security protocols (*e.g.* the one of [17]) into ABCD. To optimise the performance, using a specific library as Divine [18] will also be considered. Finally, 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 [19] because it is currently model 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] A. Armando, R. Carbone, and L. Compagna, "LTL model checking for security protocols," *Applied Non-Classical Logics*, 2009.

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

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

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

[6] G. Bhat, R. Cleaveland, and O. Grumberg, "Efficient on-the-fly model checking for CTL*," in *Logic in Computer Science (LICS).* IEEE Computer Society, 1995, pp. 388–398.

[7] 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.

[8] 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.

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

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

[11] M. Guedj, "Bsp algorithms for LTL & CTL* model checking of security protocols," Ph.D. dissertation, University of Paris-East, 2012. [Online]. Available: \url{http://www.lacl.fr/gava/guedj_thesis.pdf}

[12] F. Gava, M. Guedj, and F. Pommereau, "A BSP algorithm for on-the-fly checking LTL formulas on security protocols," in *ISPDC.* IEEE, 2012.

[13] 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.

[14] J. Barnat, "Distributed memory LTL model checking," Ph.D. dissertation, University of Brno, 2004.

[15] J. Barnat, L. Brim, and I. Cëerná, "Property driven distribution of nested dfs," in *Workshop on Verification and Computational Logic (VCL)*, 2002, pp. 1–10.

[16] C. P. Inggs and H. Barringer, "CTL* model checking on a shared-memory architecture," *Formal Methods in System Design*, vol. 29, no. 2, pp. 135–155, 2006.

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

[18] J. Barnat, L. Brim, M. Černá, and P. Ročkai, "DiVinE: Parallel Distributed Model Checker," in *Parallel and Distributed Methods in Verification (PDMC).* IEEE, 2010, pp. 4–7.

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