Performance evaluations of a BSP algorithm for state space construction of security protocols

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

Michael Guedj
LACL
guedj@u-pec.fr

Franck Pommereau
IBISC, University of Evry, France
franck.pommereau@ibisc.fr

Abstract

This paper presents the implementation and the performance comparisons of two Bulk-Synchronous Parallel (BSP) algorithms to compute the discrete state space of models. These algorithms are simple to express and the first one is the most general one whereas the second is dedicated for structured models of security protocols. Benchmarks of security protocol scenarios has been done showing the benefits of the implementation of the dedicated algorithm.

1. 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. History has shown that even if cryptography is supposed to be perfect, the correct design of security protocols is notoriously error-prone: an attack can be conducted by exploiting weaknesses in the protocol itselve. Formally verifying security protocols is thus needed. It is a well established domain that is still actively developed. Different approaches exist as [2, 1, 6, 4]. One of them is *model-checking*, a method based on reachability analysis (*state space* exploration) and allows an automatic detection of early design errors in *finite*-state systems. But the state space construction may be very consuming both in terms of memory and execution time.

Parallels the state space construction on several machines is thus done in order to benefit from all the local memories, cpu resources and disks of each machine. This allows to reduce both the amount of memory needed on each machine and the overall execution time. One of the main technical issues in the distributed memory state space construction is to partition the state space among the participating computers: each single state is assigned to a machine and this assignment is made using a function that partitions the state space into subsets of states. Each such a subset is then "owned" by a single machine.

While it has been showed that a partition function can effectively balance the workload and achieve reasonable execution time as well [7], this method suffers from some obvious drawbacks and each author thus proposes *heuristics* [3, 10]. But these studies were done in general case and security protocols are very structured: without loss of generality each agent only performs a sequence of send/received routines. The complexity only come from the possible actions of the intruder and it is thus unknown if a general approach works fine.

In [8], we have design a BSP [13] algorithm that exploit the well-structured nature of security protocols for computing efficiently the state space of finite protocol sessions. The structure of the protocols is exploited to increasing computation locality and at the same time, the BSP paradigm allows to simplify the detection of the algorithm termination and to load balance the computations. In this work, we show that using a BSP library for Python [9] allows us to write easily both algorithms and compare the performances of the two implementations on a set of scenarios. We also present how model protocols using our methodology.

2 Modeling the protocols

In this paper, we consider models of security protocols, involving a set of *agents* where a Dolev-Yao attacker resides on the network. As a concrete formalism to model protocols, we have used an *algebra of coloured Petri nets* called ABCD [11] allowing for easy and structured modelling. However, our approach is largely independent of the chosen formalism and it is enough to assume that some properties define in [8] hold.

ABCD (Asynchronous Box Calculus with Data [11]) is a specification language that allows its users to express the behavior concurrent systems at a high level. A specification is translated into colored Petri nets. In particular, the ABCD meta syntax allows its users to define a com-

plex processes in an algebra that allows: sequential composition (P;Q); non-deterministic choice (P+Q); iteration (P*Q=Q+(P;Q)+(P;P;Q)+$\cdots$); parallel composition (P$\|$Q). Processes are built on top of atoms comprising either named sub-processes, or (atomic) actions, *i.e.* conditional accesses to typed buffers. Actions may produce to a buffer, consume from a buffer, or test for the presence of a value in a buffer, and are only executed if the given condition is met. The semantics of an action is a transition in a Petri net.

As a basic example, consider the "Woo and Lam" protocol which ensures one-way authentication of the initiator A to a responder B using symmetric-key cryptography and a trusted third-party server S with share long-term symmetric keys and a fresh and unpredictable nonce produced by B:

```
A, B, S  : principal
Nb       : nonce
Kas, Kbs : skey
1.    A  ->  B  :    A
2.    B  ->  A  :    Nb
3.    A  ->  B  :    {Nb}Kas
4.    B  ->  S  :    {A, {Nb}Kas}Kbs
5.    S  ->  B  :    {Nb}Kbs
```

which could be model using ABCD as:

```
1  net Alice (A, agents, S) :
2      buffer B_ : int = ()
3      buffer Nb_ : Nonce = ()
4      [agents?(B), B_+(B), snd+(A)]                    # 1. ->
5      ; [rcv?(Nb), Nb_+(Nb)]                           # 2. <-
6      ; [Nb_?(Nb), snd+(("crypt", ("secret", A, S), Nb))] # 3. ->
7  net Bob (B, S) :
8      buffer A_ : int = ()
9      buffer myster_ : object = ()
10     [rcv?(A), A_+(A)]                  # 1. <-
11     ; [snd+(Nonce(B))]                 # 2. ->
12     ; [rcv?(myster), myster_+(myster)] # 3. <-
13     ; [A_?(A), myster_?(myster),
14        snd+(("crypt", ("secret", B, S), A, myster))] # 4. ->
15     ; [rcv?(("crypt", ("secret", S, B), Nb))
16        if Nb == Nonce(B)]              # 5. <-
17 net Server (S) :
18     buffer B_ : int = ()
19     buffer Nb_ : Nonce = ()
20     [rcv?(("crypt", ("secret", B, S), A,
21       ("crypt", ("secret", A, S), Nb))), B_+(B), Nb_+(Nb)] # 4. <-
22     ; [B_?(B), Nb_?(Nb), snd+(("crypt", ("secret", S, B), Nb))] # 5. ->
```

The '-' operation on a buffer attempts to consume a value from it and bind it to the given variable, scoped to the current action. The language also supplies a read-only version '?', thus rcv?(Nb) will read a value from rcv into variable Nb without removing it from the buffer. Similarly, the '+' operation attempts to write a value to the buffer, and there are also flush (>>) and fill (<<) operations which perform writes into and reads from lists respectively. Note that we used two buffer called rcv and snd which model the sending and receip in a network. Encoded message are tuple with special values as "crypt" and "secret" that attacker agent could not read if he have the keys.

The attacker has three components: a buffer named knowledge which is essentially a list of the information that the attacker currently "knows", a list of initial knowledge, and a learning engine with which it uses to glean new knowledge from what it observes on the network. Intuitively, the attacker performs the following operations : (1) It intercepts each message that appears on buffer nw which represent the network and adds it to its knowledge (2) It passes each message along with its current knowledge to the learning engine (it tries to learn from it by recursively decomposing the message or decrypting it when the key to do so is known) and adds any new knowledge learned to it's current knowledge (3) It then may either do nothing, or take any message that is a valid message in the protocol that is contained in its knowledge and put it back on buffer nw (made available on the network). In ABCD, these actions are expressed by the following term:

```
1  [nw−(m), knowledge>>(k), knowledge<<(learn(m,k))];
2  [True] + [knowledge?(x), nw+(x) if message(x)]
```

Note that a branch is created in the state space for each message that can be intercepted in the first line, another for the choice in the second line, and another for each valid message in the knowledge. This is why the attacker is the most computationally intensive component of our modelling. As Python's expressions are used in this algebra, the learning engine (the Dolev-Yao inductive rules) is a Python function and could thus be extended for taking account specific properties of hashing or of crypto primitives.

3 BSP Computing of the state space

Naively, to compute the state space in parallel, once can use a partition function h that returns for each state a processor id, *i.e.*, the processor numbered $h(s)$ is the owner of s. Usually, this function is simply a hash of the considered state modulo the number of processors. The idea is that each process computes the successors for only the states it owns. However, each super-step is likely to compute few states because only few computed successors are locally owned. But we known that the learning phase of the attacker is computationally expensive, in particular when a message can be actually decomposed, which leads to recompose a lot of new messages. Among the many forged messages, only a (usually) small proportion are accepted for a reception by agents. Each such reception gives rise to a new state.

This whole process can be kept local to processors. To do so, we design a new partition function h such the data of the intruder are not taken into account which make all the attacker's computations locals. For load balancing purpose, h also works without modulo: this function defines classes of states for which h returns the same value.

In the rest of the text, the number of processors is nprocs and for each processor has its pid. Then, to enter parallel code, in a SPMD (Single Program, Multiple Data) fashion, once can make "parallel" the main function using @ParFunction. Finally, the only used BSP-Python's com-

munication routine performs also the barrier of synchronisation (end of the super-step) as the collective operator "all-to-allv" of MPI. This method called exchange() sends each data item (in the form of a list of pairs "id of destination, item to send") to the corresponding processor and returns a set object storing the received data items. The main function (for our algorithm and the naive one) is the following:

```
1  @ParFunction
2  def main (infile) :
3      initialize ( infile )
4      todo = set()
5      total  = 1
6      known = set()
7      if  h(s0) == pid :
8          todo.add(s0)
9      while total >0 :
10         tosend = states_successor(known, todo)
11         todo, total  = states_exchange(known, tosend)
```

We first read in "infile" (line 3) the Petri Net and the initial state. Sets $known$ and $todo$ are still used but become local to each processor and thus provide only a partial view on the ongoing computation. So, in order to terminate the algorithm, we use an additional variable $total$ in which we count the total number of states waiting to be proceeded throughout all the processors, *i.e.*, $total$ is the sum of the sizes of all the sets $todo$. Initially, only state $s0$ is known and only its owner puts it in its $todo$ set of processor $h(s0)$. This is performed in lines 4–8. The loop (lines 9–11) performs computations of successors and BSP exchange of states in this way. Function $state\_successor$ is called to compute the successors of the states in $todo$:

```
1  def states_successor (known, todo) :
2      tosend = collections.defaultdict(set)
3      while todo :
4          s = todo.pop()
5          known.add(state)
6          for s_ in succ_local(s) − known :
7              todo.add(s_)
8          for s_ in succ_tosend(s) :
9              tosend[h(s_)].add(s_)
10     return tosend
```

Each state s from $todo$ is processed in turn and added to a set $known$ (lines 4–5) while local successors (lines 6) are added to $todo$ and successors to be send (line 9) are added to set $tosend$. Notice that in the second loop, no state from $todo$ may be obtained through sending successors because of the progression. So we have not $-known$ in line 9. In the naive algorithm, this is not true and each state can be possibly sending or not, thus the two loops are merged and each time a test is perform on the states which is less efficient. In line 2, we define a collection of sets since h can return any natural number. The collection would be used to balancing the workload.

Then, function $state\_exchange$ is responsible for performing the actual communication between processors. It sends each state s for a pair (i, s) in $tosend$ to the processor i and returns the set of states received from the other processors, together with the total number of exchanged states. It is mainly a call to the method exchange() by also adding the exchange of "total" and dissociate this exchange to the exchange of states:

```
1  def BSP_EXCHANGE (tosend) :
2      todo = set(tosend[pid])
3      total  = sum(len(tosend[k]) for k in xrange(nprocs))
4      for (count, states) in ParMessages((i, (total, tosend[i])) for i in
5              xrange(nprocs) if i != pid).exchange().value :
6          total  += count
7          todo.update(states)
8      return total , todo
```

Then the function $state\_exchange$ returns the set of received states that are not yet known locally together with the new value of $total$. To do so, we exploit the following observation: for all the protocols we have studied so far, the number of computed states during a super-step is usually closely related to the number of states received at the beginning of the super-step. Thus, before to exchange the states themselves, we can first exchange information about how many state each processor has to send and how they will be spread onto the other processors. Using this information, we can anticipate and compensate the balancing problem.

```
1  def states_exchange (known, tosend) :
2      known.clear()
3      return BSP_EXCHANGE(balance(tosend))
```

Notice that, using the progression property, it is not possible to reach states computed in any previous super-step. They are thus dumped from the main memory (line 2). In the $balance$ function (line 3 and no presented here due to a lack of space), we compute a histogram of these classes on each processor, which summarises how h would dispatch the states. This information is then globally exchanged, yielding a global histogram that is exploited to compute on each processor a better dispatching of the states it has to send. This is made by placing the classes according to a simple heuristic for the NP-hard bin packing problem. It may be remarked that the global histogram is not fully accurate since several processors may have a same state to be sent. Notice that, by postponing communication, this algorithm allows buffered sending and forbids sending several times the same state.

4 Benchmarks

The benchmarks presented below have been performed using a cluster with 16 PCs, 2GHz Intel Pentium dual core CPU, with 2GB of physical memory, connected through a Gigabyte Ethernet network. MPICH were used as low level library for BSP-Python. Our cases study involved the following five protocols: (1) Needham-Schroeder (NS) public key protocol for mutual authentication; (2) Yahalom (Y)

For the Needham-Schroeder protocol:

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

For the Yahalom protocol:

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

For the Otway-Rees protocol:

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

For the Woo and Lam Pi protocol:

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

For the Kao-Chow protocol:

Scenario	Naive	Balance	Nb_states
KC_1-1-1	4m46.631s	1m15.332s	376
KC_1-1-2	80m57.530s	37m50.530s	1545
KC_1-1-3	716m42.037s	413m37.728s	4178
KC_1-1-1_2	225m13.406s	95m0.693s	1163
KC_1-2-1	268m36.640s	159m28.823s	4825

Figure 1. Whole performances results

key distribution and mutual authentication using a trusted third party; (3) Otway-Rees (OR) key sharing using a trusted third party; (4) Kao-Chow (KC) key distribution and authentication; (5) Woo and Lam Pi (WLP) authentification protocol with public keys and trusted server.

For each protocol, using ABCD, we have built a modular model allowing for defining various scenarios involving different numbers of each kind of agents — with only one attacker, which is always enough. We note these scenarios NS$-x-y \equiv x$ Alices, y Bobs with one unique sequential session; Y(resp. OR, KC and WLP)$-x-y-z\_n \equiv x$ Servers, y Alices, z Bobs, n sequential sequential sessions.

We give here the total time of computation. We note **SWAP** when at least one processor swaps due to a lack of main memory for storing its part of the state space. We also note **COMM** when the system is unable to received the too amount of data. We can see in the tables of Fig 1 that the overall performance of our dedicated implementation (call balance) is always very good compared to the naive and general one. This holds for large state spaces as well as for smaller ones. "Naive" can also swap which never happens for the "balance".

To see the differences in behaviour (and not only execution time), we show some graphs for several scenarios. In the Figures 2–5, we have distinguished: the computation time that essentially corresponds to the computations of successor states on each processor (in black); the communication time that corresponds to states exchange and histogram computations (in grey); the waiting times that occur when processors are forced to wait the others before to enter the communication phase of each super-step (in white). Graphs in the right are cumulative time (in percentage in ordinate) depicted for each processor point of view (abscissa) whereas graphs in the right are global points of view: cumulative times of each of the super-steps (time in ordinate). We also show the percentage (ordinate) of main memory used by the program (average of the processors) during the execution time of the program (abscissa).

We can see on these graphs that for "balance" the communications are always greatly reduced but with some time a greater waiting times. This is due to the computation of the histograms and to the fact that we perform an heuristic (of the bin packing problem) for dispatching the classes of states on the processors: some classes induce a bigger number of successors and the probability that these states are regrouped on the same classes is greater in "balance" than in the complete random distribution of "naive". Note that the hashing (random) of "naive" gives the better balancing on some scenarios. For a small OR scenario, the waiting time of "naive" is greater but more balanced. However, for a bigger scenario, "balance" outperforms "naive".

By measuring the memory consumption of our implementations, we could confirm the benefits of "balance" (emptied memory regularly) when large state spaces are computed. For instance, in the NS-2-2 scenario, we observed an improvement of the peak memory usage from 50% to 20% — maximum among all the processors. Similarly, for the WLP-1-2-1_2, the peak decreases so that the computation does not swap. For Y-3-2-1, "balance" used a little less memory but that enough to not crash the whole machine. Notice that the memory use never decrease even for "balance". This is due to the GC strategy of Python for sets which de-allocate pages of the main memory only when no enough memory is available: allocated pages are directly used for other new items.

As a last observation about our "balance" implementation, we would like to emphasise that we observed a linear speedup with respect to the number of processors.

5. Conclusion

To checked if a protocol does not contain a security trap, we have model them (plus the intruder on the network) us-

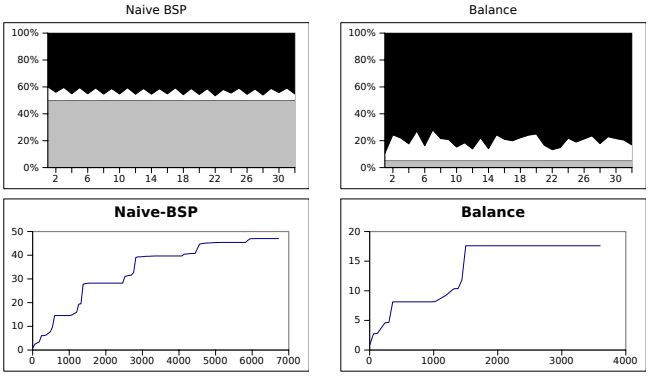

Figure 2. Performances for NS-2-2

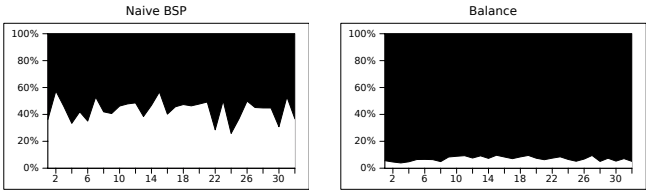

Figure 3. Performances for OR-1-2-1_2

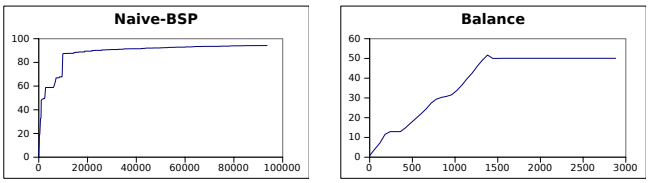

Figure 4. Performances for WLP-1-2-1_2

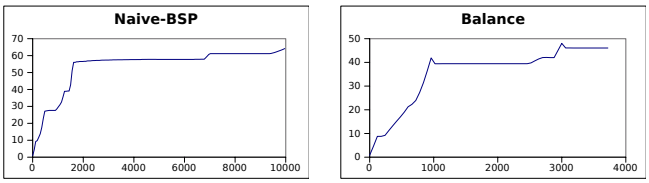

Figure 5. Performances for Y-3-2-1

ing an Algebra of Coloured Petri Nets call ABCD where Python expressions are the coloured domain of the Petri net and also the annotations. Find if there is an attack consist of generated the whole marking graph and thus all the states that are accessible from an initial one. But this activity leads to many states which is the so called state space problem. We have designed a solution which use the well-structured nature of the protocols to choose which part of the state is really needed for the partition function and to empty the data-structure in each super-step of the parallel computation. That also entails automated classification of states into classes, and dynamic mapping of classes to processors.

Using BSP-Python, we have easily implemented the solutions and compare them in several benchmarks. The general method fails in two points. First the number of cross transitions is two high and lead to a too heavy network use. Second, memorise all of them in the main memory is impossible without crashing the machine and is not clear when it is possible to put some states in disk using general heuristics [5]. We have thus empirically verify our assumption: our methods execute significantly faster and achieve better network and memory use. It is encouraging because we think we can check larger protocols [12] — secure P2P exchange of files using truth servers.

References

[1] A. Armando, R. Carbone, and L. Compagna. LTL model checking for security protocols. In *Proceedings of CSF*, pages 385–396. IEEE Computer Society, 2007.

[2] A. Armando and L. Compagna. SAT-based model-checking for security protocols analysis. *Int. J. Inf. Sec.*, 7(1):3–32, 2008.

[3] J. Barnat. *Distributed Memory LTL Model Checking*. PhD thesis, Faculty of Informatics Masaryk University Brno, 2004.

[4] C. J. F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. PhD thesis, Technische Universiteit Eindhoven, 2006.

[5] S. Evangelista and L. M. Kristensen. Dynamic State Space Partitioning for External Memory Model Checking. In *Proceedings of Formal Methods In Computer Sciences (FMICS)*, volume 5825 of *LNCS*, pages 70–85. Springer, 2009.

[6] H. Gao. *Analysis of Security Protocols by Annotations*. PhD thesis, Technical University of Denmark, 2008.

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

[8] F. Gava, M. Guedj, and F. Pommereau. A bsp algorithm for the state space construction of security protocols. In *9th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC, affiliated to conference SPIN 2010)*. IEEE Computer Society, 2010.

[9] K. Hinsen. Parallel scripting with Python. *Computing in Science & Engineering*, 9(6), 2007.

[10] C. Pajault. *Model Checking paralléle et réparti de réseaux de Petri colorés de haut-niveau*. PhD thesis, Conservatoire National des Arts et Métiers, 2008.

[11] F. Pommereau. *Algebras of coloured Petri nets*. Lambert Academic Publisher, 2010.

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

[13] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.