Université Paris-Est

Mémoire de stage

pour obtenir le diplome de
master 2 recherche de l'université de Paris-Sud 11
discipline : Informatique
présentée et soutenue publiquement par

Arthur Hidalgo

le 13 Septembre 2012

Verification of security protocols: machine-checked proof of the algos and improvement of the tools

Acknowledgements

The research thesis was done under the supervision of A/Prof Frédéric Gava and Prof. Jean Fortin, in the University Paris-Est Creteil – UPEC.

To my family.

To Frédéric and Jean, for been the most dedicated supervisors anyone could hope for. For holding my hand and standing behind me all the way through. I thank you from my bottom of my heart.

My thanks go also to all the members of our Algorithmic, Complexity and Logic Laboratory (LACL) Laboratory.

Contents

1 Introduction **2**

2 Security protocols **3**

2.1 Example . . . 3
2.2 Motivations . . . 3
2.3 Informal definition of security protocols . . . 4
2.3.1 Security Properties and possible “attacks” . . . 4
2.4 Why cryptographic protocols go wrong? . . . 7
2.5 Model checking : a technic of verification . . . 8
2.5.1 Generalities . . . 8

3 Verification **9**

3.1 Bulk-Synchronous Parallelism . . . 9
3.2 Why ABCD . . . 11
3.3 Verification of a sequential algorithm . . . 12
3.4 Verification of BSP algorithms . . . 13

4 Case study **17**

4.1 Dedicated algorithms for protocols . . . 17
4.1.1 BSP computing the state-space of security protocols [43] . . . 17
4.1.2 Verification of these dedicated parallel algorithms . . . 17
4.2 Related works . . . 19

5 Conclusion **20**

Bibliography **21**

1

Chapter 1

Introduction

Security protocols are small and standard components of systems that communicate over untrusted networks. Their relatively small size, combined with their critical role, makes them a suitable target for formal analysis [25]. Model-checking (MC) is common solution to find flaws [2]. Ideally, we would also like to have a proof of the protocolâĂŹs correctness or of the finding attack: generated a "certificate" [63] that can be checked later. But the generation of large discrete state spaces of some non traditional protocols [66] (especially when complex data-structures are used by the agents as lists of trusted servers *etc.*) is so a computationally intensive activity that the use of distributed machines is desirable and is a great challenge of research.

But MCs, especially distributed ones [**?**], like any complex softwares are subject to bugs. And generate distributed certificates to be later machine-checked using a theorem prover as Coq is currently not reasonable since provers are critical softwares that can not be altered without much attention. For this purpose, we proposed to prove the correctness of the distributed MC itself and not its results as "certifying MC" [63] generally done.

But despite [71] where authors only focus on safety properties as no overflows, no deadlocks *etc.*, we use the condition generator (VCG) Why [38] and extend the deductive verification to the correctness of the final result: has the full state-space been well computed (in parallel) without adding unknown states? We consider mechanised verifying different annotated algorithms: a sequential one as an introduction of the methodology; the traditional distributed state space algorithm and three specialised distributed algorithms for computing the state space of security protocols [43]. All distributed algorithms used a model of parallel computation called BSP [17]. The annotated source codes are available at `http://lacl.fr/gava/cert-mc.tar.gz`.

Chapter 2

Security protocols

Cryptographic protocols are communication protocols that use cryptography to achieve security goals such as secrecy, authentication, and agreement in the presence of adversaries.

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. These attacks exploit weaknesses in the protocol that are due to the complex and unexpected interleavings of different protocol sessions as well as to the possible interference of malicious participants.

Furthermore, they are not as easy that they appear [13] : the attacker is powerful enough to perform a number of potentially dangerous actions as intercepting messages flowing over the network, or replacing them by new ones using the knowledge he has previously gained; or is able to perform encryption and decryption using the keys within his knowledge [32]. Consequently the number of possible testing attacks generally growing exponentially of the size of the session.

Formal methods offer a promising approach for automated security analysis of protocols: the intuitive notions are translated into formal specifications, which is essential for a careful design and analysis, and protocol executions can be simulated, making it easier to verify certain security properties. Formally verifying security protocols is now an old subject but still relevant. Different approach exist as [3, 5, 40] and tools were dedicated for this work as [4, 27].

2.1 Example

2.2 Motivations

The possibility of violations and attacks of security protocols sometimes stems from subtle misconceptions in the design of the protocols. Typically, these attacks are simply overlooked, as it is difficult for humans, even by careful inspection of simple protocols, to determine all the complex ways that different protocol sessions could be interleaved together, with possible interference of a malicious intruder, the attacker.

The question of whether a protocol indeed achieves its security requirements or not is, in the general case, undecidable [1,34,35]. This has been proved by showing that a well-known undecidable problem (*e.g.*the Post Correspondence Problem, the halting problem for Turing machines, *etc.*) can be reduced to a protocol insecurity problem. Despite this strong undecidability result, the problem of deciding whether a protocol is correct or not it is still worthwhile to be tackled by the introduction of some restrictions can lead to identify decidable subclasses: by focusing on verification of a bounded number of sessions the problem is known to be NP-complete. This can be done by simply enumerating and exploring all traces of the protocol's state transition system looking for a violation to some of the requirements.

Although, if the general verification problem is undecidable, for many protocols, verification can be reduced to verification of a bounded number of sessions. Moreover, even for those protocols that should theorically be checked under a unbounded number of concurrent protocol

executions, violations in their security requirements often exploit only a small number of sessions. For these reasons, in many cases of interest it is sufficient to consider a finite number of sessions in which each agent performs a fixed number of steps. For instance all the attacks on the well-know SPORE and Clark-Jacob's libraries [23] can be discovered by modelling each protocol with only two protocol sessions.

However the specific nature of security protocols that make them particularly suited to be checked by specific tools. That also need how formalise those protocols to be latter checked.

2.3 Informal definition of security protocols

Communication protocols specify an exchange of messages between principals, *i.e.*the agents participating in a protocol execution (*e.g.*users, hosts, or processes). Messages are sent over open networks, such as the Internet, that cannot be considered secure. As a consequence, protocols should be designed "robust" enough to work even under worst-case assumptions, namely messages may be eavesdropped or tampered with by an intruder or dishonest or careless principals. A specific category of protocols has been devised with the purpose of securing communications over insecure networks: security (or cryptographic) protocols are communication protocols that aim at providing security guarantees such as authentication of principals or secrecy of some piece of information through the application of cryptographic primitives.

The goal of cryptographic is to convert a plain-text P into a cipher-text C (and *vice versa*) that is unintelligible to anyone (a spy) that monitoring the network. The process of converting P into C is called encryption, while the reverse procedure is called decryption. The main feature of computer's encryption is the used of an additional parameter K known as the encryption key. In order to recover the original plain-text the intended receiver should use a second key K^{-1} called the inverse key where is no way to compute easally it from K — and *vice versa*.

The best-known cryptographic algorithms for key are the well-known DES (Digital Encryption Standard) and the RSA (Rivest, Shamir, and Adleman) algorithm. The security of cryptographic algorithms relies in the difficulty of breaking them by performing a brute-force search of the key space. Hence the use of keys sufficiently long to prevent a brute force attack in a reasonable time entirely justifies the standard assumption adopted in formal analysis of security protocols and called perfect cryptography. The idea underlying such an assumption is that an encrypted message can be decrypted only by using the appropriate decryption key, *i.e.*it is possible to retrieve M from M_K only by using K^{-1} as decryption key.

Protocols are normally expressed as narrations, where some finer details are abstracted away. A protocol narration is a simple sequence of message exchanges between the different participating principals and can be interpreted as the intended trace of the ideal execution of the protocols. Informally, the scenario we are interesting in involves a set of honest agents that, according to a security protocol, exchange messages over insecure communication channels controlled by a malicious agent called intruder with the ultimate goal of achieving some security requirements. Participants (agents) perform sequence of data exchange (sending or received operators) which could be seen as "ping-pong".

2.3.1 Security Properties and possible "attacks"

What kind of attacks do there exist against security properties of protocols ? This question cannot be answered before having defined what we expect from a given security protocol. We give here an informal definition of possible and well-known "attacks" and security properties as well as some vocabulary of protocols.

Vocabulary Let us recall some elemetary vocabulary on security protocols:

- **Fresh Terms**. A protocol insecurity problem can allow for the generation of fresh terms *e.g.*Nonce. This allow to have a new value each time the protocol is used. Random numbers from the system can be used.
- **Step**. The number of steps that an honest agent can perform to execute a session of the protocol.
- **Sessions**. An agent can execute more than one time the protocol. Each use of the protocol is call a session.
- **Agents**. The participants of the protocols including intruders.

In general, the cryptographic protocol consists of agents who are willing to engage in a secure communication using an insecure network and sometime using trusted server, which generates the fresh session key used for exchanging data securely between the principals. The session key is abandoned after data exchanging is over. In fact, it is not possible to establish an authenticated session key without existing secure channels already being available [19]. Therefore it is essential that some keys are already shared between different principals, which are often referred to as master keys. Different from session keys, which expire after each session, master keys are changed less frequently, and consequently leaking master keys always causes cryptographic protocols to be broken.

Security Attacks Let us now enumerate some typical attacks. They can be categorised into the following:

- **Interruption**. The communications are destroyed or becomes unavailable or unusable. Examples include destruction of a piece of hardware, *i.e.*a hard disk, or the cutting of a physical communication line, *i.e.*a cable. An agent (as a server or else) is then unattainable.
- **Eavesdropping**. An unauthorised party gains access to the communication. The unauthorised party could be a person, a program, or a computer. Examples include wiretapping to capture data in a network, and the illegally copying of files or programs.
- **Modification**. An unauthorised party not only gains access to but tampers with the network. Examples include changing values in a data file, altering a program so that it performs differently, and modifying the content of messages being transmitted in a network.
- **Fabrication**. An unauthorised party inserts counterfeit data into the network. Examples include the inserting of spurious message in a network or the addition of records to a file.
- **Traffic analysis**. An unauthorised party intercepts and examines the messages flowing over the network in order to deduce information from the message patterns. It can be performed even when the messages are encrypted and can not be decrypted.

There are many kinds of attacking security protocol. Some well-known strategies that an intruder might employ are:

- **Man-in-the-middle** This style of attack involves the intruder imposing himself between the communications between the sender and receiver. If the protocol is purely designed he may be able to subvert it in various ways; in particular he may be able to forge as receiver to sender, for example.
- **Replay** The intruder monitors a run of the protocol and at some later time replays one or more of the messages. If the protocol does not have the mechanism to distinguish between separate runs or to detect the staleness of a message, it is possible to fool the honest agents into rerunning all or parts of the protocol. Devices like nonces, identifiers for runs and timestamps are used to try to foil such attacks.
- **Interleave** This is the most ingenious style of attack in which the intruder contrives for two or more runs of the protocol to overlap.

There are many other known styles of attack and presumably many more that have yet to be discovered. Many involve combinations of these themes. This demonstrates the difficulty in designing security protocols and emphasizes the need for a formal and rigorous analysis of these protocols.

A protocol execution is considered as involving honest (participants) principals and active attackers. The abilities of the attackers and relationship between participants and attackers together constitute a threat model and the almost exclusively used threat model is the one proposed by Dolev and Yao [32]. The Dolev-Yao threat model is a worst-case model in the sense that the network, over which the participants communicate, is thought as being totally controlled by an omnipotent attacker with all the capabilities listed above. Therefore, there is no need to assume the existence of multiple attackers, because they together do not have more abilities than the single omnipotent one. Dishonest principals do not need to be considered either: they can be viewed as attackers. Furthermore, it is generally not interesting to consider an attacker with less abilities than the omnipotent one except to verify less properties and to accelerate the formal verification of a protocol.

Security properties Each cryptographic protocol is designed to achieve one or more security-related goals after a successful execution, in other words, the principals involved may reason about certain properties; for example, only certain principals have access to particular secret information. They may then use this information to verify claims about subsequent communication, *e.g.*an encrypted message can only be decrypted by the principals who have access to the corresponding encryption key. The most commonly considered security properties include:

- **Authentication**. It is concerned with assuring that a communication is authentic. In the case of an ongoing interaction, such as the connection of a host to another host, two aspects are involved. First, at the time of connection initiation, the two entities have to be authentic, *i.e.*each is the entity that he claims to be. Second, during the connection, there is no third party who interferes in such a way that he can masquerade as one of the two legitimate parties for the purposes of unauthorized transmission or reception. For example, fabrication is an attack on authenticity.

- **Confidentiality**. It is the protection of transmitted data from attacks. With respect to the release of message contents, several levels of protection can be identified, including the protection of a single message or even specificfields within a message. For example, interception is an attack on confidentiality.

- **Integrity**. Integrity assures that messages are received as sent, with no duplication, insertion, modification, reordering, or replays. As with confidenfitiality, integrity can apply to a stream of messages, a single message, or selected fields within a message. Modification is an attack on integrity.

- **Availability**. Availability assures that a service or a piece of information is accessible to legitimate users or receivers upon request. There are two common ways to specify availability. An approach is to specify failure factors (factors that could cause the system or the communication to fail) [67], for example, the minimum number of host failures needed to bring down the system or the communication. Interruption is, for example, an attack on availability.

- **Non-repudiation**. Non-repudiation prevents either sender or receiver from denying a transmitted message. Thus, when a message is sent, the receiver can prove that the message was in fact sent by the alleged sender. Similarly, when a message is received, the sender can prove that the message was in fact received by the alleged receiver.

2.4 Why cryptographic protocols go wrong?

The first reason for the security protocols easily go wrong is that protocols were first usually expressed as narrations and most of the details of the actual deployment are ignored. And this little details and ambiguities may be the reason of an attack.

Second, as mentioned before, cryptographic protocols are mainly deployed over an open network such that everyone can join it, exceptions are where wireless or routing protocols attacker control only a subpart of the network and where agents only communicate with their neighbors [7, 15, 52, 73, 74]. One reason for security protocols easily going wrong is the existence of the attacker: he can start sending and receiving messages to and from the principals across it without the need of authorization or permission. In such an open environment, we mush anticipate that the attacker will do all sorts of actions, not just passively eavesdropping, but also actively altering, forging, duplicating, re-directing, deleting or injecting messages. These fault messages can be malicious and cause a destructive effect to the protocol. Consequently, any message received from the network is treated to have been received from the attacker after his disposal. In other words, the attacker is considered to have the complete control of the entire network and could be considered to be the network. And it is easy for humans to forget a possible combination of the attacker. Instead, automatic verification (model-checking), which is the subject of this document, would not forget one possible attack. And this number of attack growing exponentially and reduce the time of computation of generating all these attacks using parallel machine is the main goal of this document.

It is notice to say that nowadays a considerable number of cryptographic protocols have been specified, implemented and verified. Consequently analysing cryptographic protocols in order to find various kinds of attacks and to prevent them has received a lot of attention. As mentioned before, the area is remarkably subtle and a very large portion of proposed protocols have been shown to be flawed a long time after they were published. This has naturally encouraged research in this area.

Designing secure protocols is a challenging problem. In spite of their apparent simplicity, they are notoriously error-prone. In open networks, such as the Internet, protocols should work even under worst-case assumptions, namely messages may be eavesdropped or tampered with by an intruder (also called the attacker or spy) or dishonest or careless principals (where we call principals the agents participating in a protocol execution). Surprisingly, severe attacks can be conducted even without breaking cryptography, but by exploiting weaknesses in the protocols themselves, for instance by carrying out man-in-the-middle attacks, where an attacker plays off one protocol participant against another, or replay attacks, where messages from one session (*i.e.*execution of an instance of the protocol) are used in another session.

Fail in security protocols The history of security protocols is full of examples, where weaknesses of supposedly correct published protocols that have been implemented and deployed in real applications only to be found flawed years later. The most well-known case is the Needham-Schroeder authentication protocol that was found vulnerable to a man-in-the-middle attack 17 years after its publication. It has been shwo by “The Computer Security Institute” [1] that the number of vulnerabilities of protocols is highly growing up and a discovering one of them is a daily thing for compagnies and researchers. But, generally speaking, security problems are undecidable for their dynamic behaviour due to, say, mis-behaved agents and unbounded sessions of protocol executions. Therefore, verification of security properties is an important research problem. This leads to the researches in searching for a way to verify whether a system is secure or not.

[1] http://www.gocsi.com

2.5 Model checking : a technic of verification

2.5.1 Generalities

In general, one may identify two basic approaches to model-checking. The first one uses a global analysis to determine if a system satisfies a formula; the entire state space 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 system state needs to be analyzed in order to determine whether or not a system satisfies a formula. On the other hand, 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.

For example, the paper [24] presents a local algorithm for model-checking a subpart of the μ-calculus and [77] presents an algorithm for CTL — formally defined latter. [26] gives an algorithm with the same time complexity as the one of [16] for determining when a system satisfies a specification given as a Büchi automaton. In light of the correspondence between such automata and the LTL fragment of CTL* (both formally defined later), it follows that the algorithm from [26] may be used for LTL model-checking also. However, it is not clear how this approach can be extended to handle full CTL* — an exception is the work of [57], apply in [53] on security protocols, where specific game theoric automata are used for verifying on-the-fly CTL* formulas on shared-memory multi-processors but it is also not clear how adapt this method to distributed computations.

Results in an extended version of [14] suggest a model-checking algorithm for full CTL* which allows the on-the-fly construction of the state space of the system. However, this approach requires the *a priori* construction of the states of an amorphous Büchi tree automaton from the formula being checked, and the time complexity is worse than the one of [16].

Figure 3.1. The BSP model of execution

Chapter 3

Verification

This chapter extends the works of [44, 45].

3.1 Bulk-Synchronous Parallelism

Bulk-Synchronous Parallel Machines

A BSP computer has three components:

- a homogeneous set of uniform processor-memory pairs;
- a communication network allowing inter processor delivery of messages;
- a global synchronization unit which executes collective requests for a synchronization barrier.

A wide range of actual architectures can be seen as BSP computers. For example share memory machines could be used in a way such as each processor only accesses a subpart of the shared memory (which is then "private") and communications could be performed using a dedicated part of the shared memory. Moreover the synchronization unit is very rarely a hardware but rather a software ([55] presents global synchronization barrier algorithms). Supercomputers, clusters of PCs, multi-core [47] and GPUs *etc.* can be thus considered as BSP computers.

The BSP's execution model

A BSP program is executed as a sequence of *super-steps*, each one divided into (at most) three successive and logically disjointed disjoint phases (see Fig. 3.1):

1. each processor only uses its local data to perform sequential computations and to request data transfers to/from other nodes;

2. the network delivers the requested data;

3. a global (collective) synchronisation barrier occurs, making the transferred data available for the next super-step.

BSP's cost model

The performance of the BSP machine is characterised by 4 parameters:

1. the local processing speed **r**;
2. the number of processor **p**;
3. the time **L** required for a barrier;
4. and the time **g** for collectively delivering a 1-relation, a communication phase where every processor receives/sends at most one word.

The network can deliver an h-relation (every processor receives/sends at most h words) in time $\mathbf{g} \times h$. To accurately estimate the execution time of a BSP program these 4 parameters could be easily benchmarked [17].

The execution time (cost) of a super-step s is the sum of the maximal of the local processing, the data delivery and the global synchronisation times. It is expressed by the following formula:

$$\text{Cost}(s) = \max_{0 \leq i < \mathbf{p}} w_i^s + \max_{0 \leq i < \mathbf{p}} h_i^s \times \mathbf{g} + \mathbf{L}$$

where w_i^s = local processing time on processor i during superstep s and h_i^s is the maximal number of words transmitted or received by processor i during superstep s.

The total cost (execution time) of a BSP program is the sum of its S super-steps costs that is $\sum_s \text{Cost}(s)$. It is, therefore, a sum of 3 terms:

$$W + H \times \mathbf{g} + S \times \mathbf{L} \text{ where } \begin{cases} W &= \sum_s \max_i w_i^s \\ H &= \sum_s \max_i h_i^s \end{cases}$$

In general, W, H and S are functions of $\mathbf{p}$ and of the size of data n, or of more complex parameters like data skew. To minimize execution time, the BSP algorithm design must jointly minimize the number S of supersteps, the total volume h and imbalance of communication and the total volume W and imbalance of local computation.

Advantages and inconvienients

As stated in [28]: "A comparison of the proceedings of the eminent conference in the field, the ACM Symposium on Parallel Algorithms and Architectures between the late eighties and the time from the mid-nineties to today reveals a startling change in research focus. Today, the majority of research in parallel algorithms is within the coarse-grained, BSP style, domain".

This model of parallelism enforces a strict separation of communication and computation: during a super-step, no communication between the processors is allowed, only at the synchronisation barrier they are able to exchange information. This execution policy has two main advantages: first, it removes non-determinism and guarantees the absence of deadlocks; second, it allows for an accurate model of performance prediction based on the throughput and latency of the interconnection network, and on the speed of processors. This performance prediction model can even be used online to dynamically make decisions, for instance choose whether to communicate in order to re-balance data, or to continue an unbalanced computation.

However, on most of cheaper distributed architectures, barriers are often expensive when the number of processors dramatically increases — more than 10 000. But proprietary architectures and future shared memory architecture developments (such as multi-cores and GPUs) make them much faster. Furthermore, barriers have also a number of attractions: it is harder to introduce the possibility of deadlock or livelock, since barriers do not create circular data dependencies. Barriers also permit novel forms of fault tolerance [68].

The BSP model considers communication actions *en masse*. This is less flexible than asynchronous messages, but easier to debug since there are many simultaneous communication actions in a parallel program, and their interactions are typically complex. Bulk sending also provides better performances since, from an implementation point of view, grouping communication together in a seperate program phase permits a global optimization of the data exchange by the communications library.

The simplicity and yet efficiency of the BSP model makes it a good framework for teaching (few hours are needed to teach BSP programming and algorithms), low level model for multi-cores/GPUs system optimisations [47], *etc.*, since it has been conceived has a bridging model for parallel computation. The simplicity and yet expressivity of BSP programming makes it look like a good candidate for the formal proof of parallel computations. Since BSP programs are portable and cost estimate features power consumption, they can enjoy cloud-computing [6]: we can imagine a scheduler server that distributes the BSP programs depending on the cost of the BSP program to optimise power consumption and the network.

This is also merely the most visible aspects of a parallel model that shifts the responsibility for timing and synchronization issues from the applications to the communications library[1]. As with other low/high level design decisions, the applications programmer gains simplicity but gives up some flexibility and performance. In fact, the performance issue is not as simple as it seems: while a skilled programmer can in principle always produce more efficient code with a low-level tool (be it message passing or assembly language), it is not at all evident that a real-life program, produced in a finite amount of time, can actually realize that theoretical advantage, especially when the program is to be used on a wide range of machines [48, 60].

Last advantage of BSP is that it greatly facilitates debugging. The computations going on during a superstep are completely independent and can thus be debugged independently. This facility will be used here to formally prove the correctness of our algorithms. Moreover, if it is true for the correctness of the algorithm that stand true for the execution time of BSP programs: it is easy to measure during the execution of a BSP program, time speding to communicate and to synchronise by just adding chronos before and after the primitive of synchronisation. This facility will be used here to compare differents algorithms.

All this capacities are is possibles only because the runtime system knows precisely which computations are independent. In a asynchronous message-passing system as MPI, the independent sections tend to be smaller, and identifying them is much harder. But, using BSP, programers and designer have to keep in mind that some parallelism patterns are not really BSP friendly. For example, BSP does not enjoy in an optimist manner pipeline and master/slave (also knowed as farm of processes) schemes even if it is possible to have not too inefficient BSP programs from these schemes [42]. Thus, some parallel computations and optmisations would never be BSP. This is the drawback of all the restricted models of computations as well.

The BSP model has been used with success in a wide variety of problems such scientific computing [8, 17, 18, 31, 56, 72], parallel data-structure [46, 51], genetic algorithms [20] and genetic programming [33], neural network [65], parallel data-bases [9–11], constraints solver [50], graphs [22, 37, 59, 72], geometry [29], string search [36, 58], implementation of tree skeletons [61], *etc.*

3.2 Why ABCD

Why is a framework for algorithms verification. Basically, it is composed of two parts: a (polymorphic first-order) logical language called Why with an infrastructure to translate it to existing theorem provers and SMT solvers; and an intermediate verification programming language called WhyML with a VCG. The examples of the standard library propose finite sets of data and several operations with their axiomatisation — which can be proved using Coq. In the

[1]BSP libraries are generally implemented using MPI [69] or low level routines of the given specifics architectures

logical formula, x@ is the notation for the value of x in the prestate, *i*.e. at the precondition point and x@label for the value of x at a certain point (marked by a label) of the algorithm. Mutable data types can be introduced, by means of polymorphic references: a reference r to a value of type σ has type **ref** σ, is created with the function **ref**, is accessed with !r, and assigned with r $\leftarrow$ e. Algorithms are annotated using pre- and post-conditions, loop invariants, and variants to ensure termination. VCG is computed using a weakest precondition calculus and then passed to the back-end of Why to be sent to provers. Notice that in Why, sets are purely applicative and thus only a reference on a set can be modified and assigned to another set.

3.3 Verification of a sequential algorithm

Fig. 3.2 gives a common sequential algorithm in WhyML (logical assertions are in curly brackets) using an appropriate syntax for set operations. A set call known contains all the states that have been processed and would finally contain StSpace. The set todo is used to hold all the states whose successors have not been constructed yet; each state s from todo is processed in turn (lines 4 and 12) and added to known (line 13) while its successors are added to todo unless they are known already — line 14. Note that the algorithm can be made strictly depth-first by choosing the most-recently discovered state (*i.e.* todo as a heap), and breadth-first by choosing the least-recently one. This has not been studied here.

We need to prove three properties regarding this code: it does not fail, it indeed compute the state-space and it terminates. The first property is immediate since the only operation that could fail is pick (where the precondition is "not take any element from an empty set") and this is assured by the **while**'s boolean condition. Only four invariants (lines $6-9$) are needed: (1) known and todo are subsets of StSpace; at the end, todo will be empty which ensures (A); (2) these sets are disjoint which ensures that only new states are added to known; (3) and (4) known is as StSpace and when todo will be empty, then it ensures (B). Also, the termination of this algorithm is ensured by the following variants: |StSpace set_diff known| and this variant holds at every step since the algorithm only adds a new state s since (known set_inter todo)=set_empty.

All the obligations produced by the VCG of WhyML are automatically discharged by a combination of SMT provers: CVC3, Z3, Simplify, Alt-Ergo, Yices and Vampire. For each prover, we give a timeout of 10 seconds. In the following table, we give the number of generated obligations and how many are discharged by the provers:

Algo/SMT	**Total**	Alt-Ergo	Simplify	Z3	CVC3	Yices	Vampire
Seq	11	2	10	11	7	3	3

One could notice that SMT solvers Simplify and Z3 give the best results. In practice, we mostly used them: Simplify is the faster and Z3 sometime verified some obligations that had not be discharged by Simplify. We also have worked with the provers as black-boxes and we have thus no explanation for this fact. It also took a day for the first author to annotate this first algorithm.

3.4 Verification of BSP algorithms

Our tool BSPWhy extends WhyML with BSP primitives (message passing and synchronisation) and definitions of collective operations. A special constant *nprocs* (equal to **p** the number of processors) and a special variable my_pid (with range $0, \ldots, \mathbf{p}-1$) were also added to WhyML expressions. A special syntax for BSP annotations is also provided which is simple to use and is sufficient to express conditions in most practical programs: we add the construct $t<i>$ which denotes the value of a term t at processor id i, and $<x>$ denotes a **p**-values x (represented by $fparray$, purely applicative arrays of constant size **p**) that is a value on each processor by opposition to the simple notation x which means the value of x on the current processor.

We used the WhyML language as a back-end of our own BSPWhyML language. This transformation is based on the fact that, for each super-step, if we execute sequentially the code for

```
let seq_algo () =
 let known = ref set_empty ESPACE in
 let todo = ref {s0} in
  while todo ≠ ∅ do
  {
   invariant (1) (known set_union todo) Incl StSpace
         and (2) (known set_inter todo)=set_empty
         and (3) s0 In (known set_union todo)
         and (4) (forall e:state. e In known → succ(e) Incl (known set_union todo))
   variant |StSpace set_diff known|
  }
    let s = pick todo in
       known← !known set_add s;
       todo← !todo set_union (succ(s) set_diff !known)
  done;
  !known
  {result=StSpace} (* result is the value of known*)
```

Figure 3.2. Sequential annotated algorithm

each processor and then perform the simulation of the communications by copying the data, we have the same results as in really truly doing it in parallel. Also, when transforming a `if` or `while` structure, there is a risk that a global synchronous instruction (a collective operation) might be executed on a processor and not on the others. We generate an assertion to forbid this case, ensuring that the condition associated with the instruction will always be true on every processor at the same time and thus forbidding deadlocks. The details and some examples are available in [39]. This sequential algorithm can be easily parallelised in a SPMD (Single Program, Multiple Data) fashion by using a partition function cpu that returns for each state a processor id, *i.e.*, the processor numbered cpu(s) is the owner of s. The idea is that each processor computes the successors for only the states it owns. This is rendered as the BSP algorithm of Fig. 3.3. Sets known and todo are still used but become local to each processor and thus provide only a partial view on the ongoing computation. For lack of space, we only present the code of the main parallel loop: other functions are available in the source code.

Function local_successors compute the successors of the states in todo where each computed state that is not owned by the local processor is recorded in a set tosend together with its owner number. The set pastsend contains all the states that have been sent during the past super-steps — the past exchanges. This prevents returning a state already sent by the processors. Function (synchronous) exchange is responsible for performing the actual communications: it returns the set of received states that are not yet known locally together with the new value of *total*.

In order to terminate the algorithm, we use the additional variable total in which we count the total number of sent states. We have thus not used any complicated methods as the ones presented in [12, 41]. It can be noted that the value of total may be greater than the intended count of states in todo sets. Indeed, it may happen that two processors compute a same state owned by a third processor, in which case two states are exchanged but then only one is kept upon reception. In the worst case, the termination requires one more super-step during which all the processors will process an empty todo, resulting in an empty exchange and thus total=0 on every processor, yielding the termination. We use the following predicates:

- isproc(i) is defined what is a valid processor's id that is 0≤ i<**nprocs**;
- sigma_union(p_set) is the union of the sets of the **p**-value p_set that is $\bigcup_{pid=0}^{\mathbf{p}} \text{p_set}(pid)$;
- GoodPart(<p_set>) is used to indicate that each processor only contains the states it owns that is **forall** i:int. isproc(i) → **forall** s:state. s In p_set<i> → cpu(s)=i.

As above, we need to prove that the code does not fail, indeed computes the entire state-space and terminates. The first property is immediate since only pick is used as above. Absence of deadlock (the main loop contains exchange which implies a global synchronisation of all the processors) can easily be maintaining using invariant (4) (line 11): total have the same value on all the processors during the entire execution of the algorithm. Let us now focus on the two other properties.

```
let naive_state_space () =
  let known = ref set_empty ESPACE ESPACE ESPACE ESPACE ESPACE ESPACE ESPACE in
let todo = ref set_empty ESPACE in
  let pastsend = ref set_empty ESPACE ESPACE ESPACE ESPACE in let total = ref 1 in
    if cpu(s0) = bsp_pid then
     todo ← s0 set_add !todo;
    while total>0 do
     { invariant
    (1) sigma_union(<known>) set_union sigma_union(<todo>) Incl StSpace
and (2) (sigma_union(<known>) set_inter sigma_union(<todo>))=set_empty
and (3) GoodPar(<known>) and GoodPart(<todo>)
and (4) (forall i,j:int. isproc(i) → isproc(j) → total<i> = total<j>)
and (5) total<0> ≥ |sigma_union(<todo>)|
and (6) s0 In (sigma_union(<known>) set_union sigma_union(<todo>))
and (7) (forall e:state. e In sigma_union(<known>) → succ(e) Incl (sigma_union(<known>) set_union sigma_union(<todo>)))
and (8) (forall e:state. forall i:int. isproc(i) → e In known<i> → succ(e) Incl (known<i> set_union pastsend<i>))
and (9) sigma_union(<pastsend>) Incl StSpace
and (10) (forall i:int. isproc(i) → forall e:state. e In pastsend<i> → cpu(e)<>i)
and (11) sigma_union(<pastsend>) Incl (sigma_union(<known>) set_union sigma_union(<todo>))
      variant pair(total<0>,| S set_diff sigma_union(known) |) for lexico_order
     }
      let tosend=(local_successors known todo pastsend) in
       exchange todo total !known !tosend
    done;
    !known
    {sigma_union(<result>)=StSpace and GoodPart(<result>)}
```

Figure 3.3. Parallel annotated algorithm

The invariants (lines $8-18$) of the main parallel loop work as follow: (1) as in the sequential algorithm, we need to maintain that known (even distributed) is a subset of StSpace which finally ensures (A) when todo is empty; (2) as usual, the states to treated are not already known; (3) our sets are well distributed (there is no duplicate state that is, each state is only keep in a unique processor); (4) total is a global variable, we thus ensure that it is the same value on each processor; (5) ensures that no state remain in todo (to be treated) when leaving the loop since total is upper to the size of todo, total is an over-approximation of the number of sent states; (6) and (7) usually ensure property (B); (8) states in known have their successors locally present or been sent; (9) past sending states are in the state-space; (10) pastsend only contains states that are not own by the processor and (11) all these states, that were sent, are finally received and store by a processor.

In the post-condition (line 25), we can also ensures that the result is well distributed: the state-space is complete and each processor only contains the states it owns depending of the function “cpu”.

For the local computations, the termination is ensure as in the sequential algorithm. The main loop is more subtle: total is an over-approximation and thus could be greater to 0 whereas todo empty. This happens when all the received states are already in known. The termination has thus two cases: (1) in general the set known globally (that is in the point of view of all processor) grows and we have thus the cardinal of StSpace minus known which is strictly decreasing; (2) if there is no state in any todo of a processor (case of the last super-step), no new states would be computed and thus total would be equal to 0 in the last stage of the main loop. We thus used a lexicographic order (this relation is well-founded ensuring termination) on the total size of the **p** values known following with total (which is the same value on each processor) when no new states are computed and thus when no state would be send during the next super-step. At least, one processor can no received states during a super-step. We thus need an invariant in the local_successors for maintaining the fact that the set known potentially growth with at least the states of todo. We also maintain that if todo is empty then no state would be send (in local_successors) and received, making total equal to 0 after the exchange function.

With some obvious axioms on the predicates, all the produced obligations are automatically discharged by a combination of the SMT solvers. In the following table, for each part of this parallel algorithm, we give the number of obligations and how many are discharged by the provers:

part/SMT	**Total**	Alt-Ergo	Simplify	Z3	CVC3	Yices	Vampire
main	106	49	90	92	0	0	81
successor	94	45	90	88	75	0	58
exchange	90	42	80	78	74	0	75

Now the combination of all provers is needed since none of them (or at least a couple of them) is able to prove all the obligations. This is certainly due to their different heuristics. We also note that Simplify and Z3 continue to remain the most efficient. It also takes one month for the authors to annotated this parallel algorithm.

Chapter 4

Case study

4.1 Dedicated algorithms for protocols

4.1.1 BSP computing the state-space of security protocols [43]

We models security protocols as a labelled transition system (LTS) where *agents* send messages over a network which contains a Dolev-Yao attacker [32]. The intruder can overhear, intercept, and synthesise any message and is only limited by the constraints of the cryptographic methods used. 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*. Based on the following properties, we have designed in [43], in an incremental manner, three different BSP algorithms for effectively computing the state space of security protocols. Only the functions local_successors and exchange have been modified in the distributed algorithms.

In the first algorithm, called "Incr", when the function local_successors is called, 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. Another difference is the forgotten the variable "pastsend" since no state could be send twice due to this order. Fig. 4.1 schemes this idea. In the second algorithm, called "Sweep", and using the previous hypothesis, at the beginning of each super-step, we also dump from the main memory all the known states because they cannot be reached anymore due to the sweep-line progression. In the third algorithm, call "Balance", states to be sent are also first balanced across the processors. Classes of states (consistent with partition function cpu) are grouped on processors so there is no possibility of duplicated computation. Fig. 4.2 schemes this idea. These algorithms (especially the third one) give better performances than a naive distributed one for security protocols [43]. Note that partial-order reductions [75] can also be trivially introduced.

4.1.2 Verification of these dedicated parallel algorithms

For all these algorithms, the termination is proved correct as above. For lack of space, we present only the differences in the main loop of the algorithms and not in local_successor and exchange — see the source code.

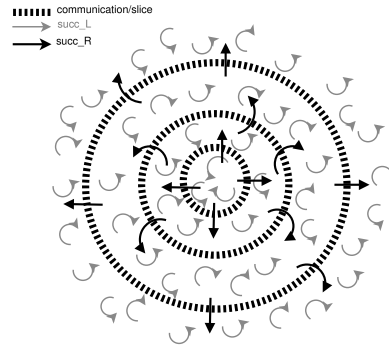

Figure 4.1. Scheme of the "Incr" distributed algorithm

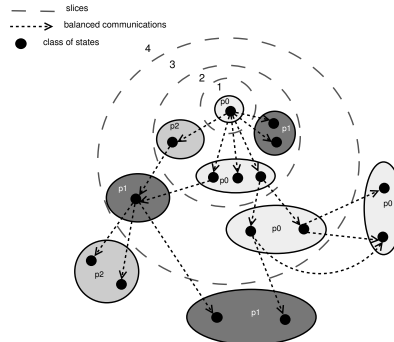

Figure 4.2. Scheme of the "Balance" distributed algorithm

Algorithm "Incr"

The invariants are the same of Fig 3.3 but with these changes. First, we need to forgotten all the behaviour about pastsend in the invariants of Fig 3.3 that is invariants (10), (15) and (18) since we no longer use this variable. Second, we introduce these two new invariants:

(12) **and** (**forall** e:state. e In sigma_union(<known>) → slice(e)<ghost_slice)
(13) **and** (**forall** e:state. e In sigma_union(<todo>) → slice(e)=ghost_slice)

We need here to introduce the ghost variable[1] ghost_slice which is incremented at each super-step and thus corresponds to the measure of progression of the protocol. Invariant (12) is need to prove that the set known contains only states of the past slices and invariant (13) proves that in todo there is only states of the current slice.

Algorithm "Sweep"

This algorithm works as "Incr" except that known is empty at the beginning of each super-step. We thus need to maintain this fact by using another invariant (14) sigma_union(known)=set_empty. Note that known only grows in function local_successor.

Also, we can thus not longer use known as the variable which contains the full state-space. We thus introduce another ghost variable called ghost_known which will grows at each super-step by recovering all the states of known. In this way, in all the previous invariants, we must replace known per ghost_known for having the correctness of this algorithm.

Algorithm "Balance"

In this algorithm, we no longer used the partition function cpu since states are distributed per class and classes are distributed across the processors using a balance. We thus need a predicate **class**(e,e') that logically define that two states belong to the same class. We also

[1] Additional codes not participating in the computation but accessing the program data and allowing the verification of the original code.

need to redefine the predicate GoodPart(<p_set>) as follow: **forall** i,j:int. isproc(i) → isproc(j) → i<>j → **forall** s,s':state. s In s' In p_set<j> → not **class**(e,e') which denotes that two states belong of two different processors are not in the same class.

We also need too assert that after the computation of the balance (currently axiomatising since a heuristic of a NP-problem [43]), sent states respect the predicate GoodPart. We also introduce this new invariant:

(14) **and** (**forall** i:int. isproc(i) → **forall** e,e':state. e In ghost_known<i> → **class**(e,e') → e' In ghost_known<i>)

which denotes that known states respect the fact that all states in a class belong to the same slice at the same processor. local_successor would verifying this fact.

"Proof obligation results"

In the following table, for each part of each parallel algorithm, we give the number of obligations and how many are discharged by the provers:

Algorithm	Part	**Total**	Alt-Ergo	Simplify	Z3	CVC3	Yices	Vampire
Incr								
	main	109	50	93	85	0	0	85
	successor	105	55	102	101	77	0	73
	exchange	32	15	28	22	19	0	27
Sweep								
	main							
	successor							
	exchange							
Balance								
	main							
	successor							
	exchange							

As above, only the combination of all provers is able to prove all the obligations. And few of them (not necessary the harder) need that provers run minutes. Simplify and Z3 still remain the most efficient. An interesting point is that the second author, as a master student (when writing this article), was able to perform the job (annotated these parallel algorithms) in three months. Based on this fact, it seems conceivable that a more seasoned team in formal methods can tackle more substantial algorithms (of model-checking) in a real programming language.

4.2 Related works

There are many tools dedicated to the verification of security protocols: see [25] for an overview. The main idea of most known approaches to the distributed memory state space generation is similar to the naive algorithm [**?**]. Some developments using theorem provers are related to model checking. In [78] and [64], authors present development of BDDs and tree automata using Coq. The verification of a μ-calculus computation has also been done in Coq in [70]. A sequential state-space algorithm (with a partial order reduction) has been checked in B in [76]. Our methodology is also based on perfect cryptography. The author of [30] annotated cryptographic algorithms to mechanized prove their correctness.

To our knowledge, there are three existing approaches for automatically generating machine-checked protocol security proofs. The first approach is in [49] where a protocol and its properties are model as a set of Horn-clauses and where the certificate is machine-checked in Coq. The second [21] used the theorem prover Isabelle and compute a fixpoint of an abstraction of the transition relation of the protocol of interest — this fixpoint over-approximates the set of reachable states of the protocol. The latter [62] also used Isabelle but two strong protocol-independent invariants have been derived from an operational semantics of the protocols. We see three main drawbacks to these approaches. First, they limits (reasonably) protocols and properties that can be checked. Second, each time the proof of the tested property of the protocol need to be machine-checked; in our approach, the results of the MC are correct by construction. Third, there is currently no possibility of distributed computations for larger protocols.

Chapter 5

Conclusion

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 and model-checking is one of the solution [25]. But model checkers use sophisticated algorithms that can miss a state which can be an unknown attack of the security protocol. Mechanized correctness is thus vital.

In this work, we focus on correctness of a well-known sequential algorithm for finite state-space construction (which is the basis for explicit model-checking) and on distributed ones where three are dedicated to security protocols. We annotated the algorithms for finite sets operations (available in Coq) and used the VCG Why (certifying in Coq [54]) to obtain goals that were entirely checked by SMT solvers. These goals ensure the termination of the algorithms as well as their correctness for any successor function — assumed correct and generating a finite state-space. We thus gained more confidence in the code. We also hope to have convinced that this approach is humanly feasible and applicable to truly (parallel or not) model-checking algorithms.

In future works, we plan to check model-checking algorithms (in the sense of determine if a logical LTL/CTL* formula holds a model) as Tarjan like algorithms. This is challenging in general but using an appropriate VCG, we believe that a team can “quickly” do it. Compressions aspects (symmetry, partial order, *etc.*) must also be studied since they can generated wrong algorithms. The work of [76] which use the B method could be a good basis. Furthermore, the transformation of BSPWhyML into WhyML is potentially not correct. The third authors is working on this. The successor function (computation of the transitions of the state-space) is currently an abstract function. A machine-checked proof of an implementation is needed. Finally, we are currently proving algorithms and not the effective code. Regarding the code structure, this is not really an issue and translating the resulting proof into a verification tool for true programs should be straightforward, mostly if high level data-structures are used: the Why framework allows a plugin of Frama-C (`http://frama-c.com/`) to generate WhyML codes from C ones — a tool for Java’s codes is also present.

Bibliography

[1] R. M. Amadio and D. Lugiez. On the reachability problem in cryptographic protocols. In C. Palamidessi, editor, *Concur*, volume 1877 of *LNCS*, pages 380–394. Springer-Verla, 2000. Page 3.

[2] A. Armando, R. Carbone, and L. Compagna. Ltl model checking for security protocols. *Applied Non-Classical Logics*, 2009. Page 2.

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

[4] A. Armando and *et* al. The AVISPA tool for the automated validation of Internet security protocols and applications. In K. Etessami and S. K. Rajamani, editors, *Proceedings of Computer Aided Verification (CAV)*, volume 3576 of *LNCS*, pages 281–285. Springer, 2005. Page 3.

[5] Alessandro Armando, Roberto Carbone, and Luca Compagna. Ltl model checking for security protocols. *Journal of Applied Non-Classical Logics*, 19(4):403–429, 2009. Page 3.

[6] M. Armbrust, A. Fox, R. Griffith, and al. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, 2009. Page 11.

[7] Mathilde Arnaud. *Formal verification of secured routing protocols.* PhD thesis, Laboratoire Spécification et Vérification, ENS Cachan, France, 2011. Page 7.

[8] A.V.Gerbessiotis. *Topics in Parallel and Distributed Computation.* PhD thesis, Harvard University, 1993. Page 11.

[9] M. Bamha and M. Exbrayat. Pipelining a Skew-Insensitive Parallel Join Algorithm. *Parallel Processing Letters*, 13(3):317–328, 2003. Page 11.

[10] M. Bamha and G. Hains. Frequency-adaptive join for Shared Nothing machines. *Parallel and Distributed Computing Practices*, 2(3):333–345, 1999. Page 11.

[11] M. Bamha and G. Hains. An Efficient equi-semi-join Algorithm for Distributed Architectures. In V. Sunderam, D. van Albada, and J. Dongarra, editors, *International Conference on Computational Science (ICCS 2005)*, LNCS. Springer, 2005. Page 11.

[12] J. Barnat. *Distributed Memory LTL Model Checking.* PhD thesis, Faculty of Informatics Masaryk University Brno, 2004. Page 13.

[13] D. Basin. How to evaluate the security of real-life cryptographic protocols? The cases of ISO/IEC 29128 and CRYPTREC. In *Workshop on Real-life Cryptographic Protocols and Standardization*, 2010. Page 3.

[14] O. Bernholtz, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. In D. L. Dill, editor, *Computer Aided Verification (CAV)*, volume 818 of *LNCS*, pages 142–155. Springer-Verlag, 1994. Page 8.

[15] Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, and James J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *omputer Security Foundations Symposium (CSF)*, pages 124–140. IEEE Computer Society, 2009. Page 7.

[16] 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)*, pages 388–398. IEEE Computer Society, 1995. Page 8.

[17] R. H. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI.* Oxford University Press, 2004. Pages 2, 10 and 11.

[18] R. H. Bisseling and W. F. McColl. Scientific computing on bulk synchronous parallel architectures. In B. Pehrson and I. Simon, editors, *Technology and Foundations: Information Processing '94, Vol. I*, volume 51 of *IFIP Transactions A*, pages 509–514. Elsevier Science Publishers, Amsterdam, 1994. Page 11.

[19] C. Boyd. Security architectures using formal methods. *IEEE journal on Selected Areas in Communications*, 11(5):684–701, 1993. Page 5.

[20] A. Braud and C. Vrain. A parallel genetic algorithm based on the BSP model. In *Evolutionary Computation and Parallel Processing GECCO & AAAI Workshop*, Orlando (Florida), USA, 1999. Page 11.

[21] A. D. Brucker and S. Mödersheim. Integrating automated and interactive protocol verification. In *Formal Aspects in Security and Trust (FAST)*, volume 5983 of *LNCS*, pages 248–262. Springer, 2009. Page 19.

[22] A. Chan, F. Dehne, and R. Taylor. Implementing and Testing CGM Graph Algorithms on PC Clusters and Shared Memory Machines. *Journal of High Performance Computing Applications*, 2005. Page 11.

[23] J. Clark and J. Jacob. A survey of authentication protocol literature : Version 1.0. Available at `http://www-users.cs.york.ac.uk/~jac/papers/drareview.ps.gz`, 1997. Page 4.

[24] R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2(121–147), 1993. Page 8.

[25] 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*, pages 29–44, 2011. Pages 2, 19 and 20.

[26] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for verication of temporal properties. *Formal Methods in System design*, 1:275–288, 1992. Page 8.

[27] C. J. F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. PhD thesis, Technische Universiteit Eindhoven, 2006. Page 3.

[28] F. Dehne. Special issue on coarse-grained parallel algorithms. *Algorithmica*, 14:173–421, 1999. Page 10.

[29] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *International Journal on Computational Geometry*, 6(3):379–400, 1996. Page 11.

[30] J. den Hartog. Towards mechanized correctness proofs for cryptographic algorithms: Axiomatization of a probabilistic hoare style logic. *Sci. Comput. Program.*, 74(1–2):52–63, 2008. Page 19.

[31] N. Deo and P. Micikevicius. Coarse-grained parallelization of distance-bound smoothing for the molecular conformation problem. In S. K. Das and S. Bhattacharya, editors, *4th International Workshop Distributed Computing, Mobile and Wireless Computing (IWDC)*, volume 2571 of *LNCS*, pages 55–66. Springer, 2002. Page 11.

[32] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983. Pages 3, 6 and 17.

[33] D. C. Dracopoulos and S. Kent. Speeding up genetic programming: A parallel BSP implementation. In *First Annual Conference on Genetic Programming*. MIT Press, July 1996. Page 11.

[34] N. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In *Workshop on Formal Methods and Security Protocols (FMSP), part of FLOC conference.*, 1999. Page 3.

[35] S. Even and O. Goldreich. On the security of multiparty ping pong protocols. In *24th IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society, 1983. Page 3.

[36] P Ferragina and F. Luccio. String search in coarse-grained parallel computers. *Algorithmica*, 24(3):177–194, 1999. Page 11.

[37] A. Ferreira, I. Guérin-Lassous, K. Marcus, and A. Rau-Chauplin. Parallel computation on interval graphs: algorithms and experiments. *Concurrency and Computation: Practice and Experience*, 14(11):885–910, 2002. Page 11.

[38] Jean-Christophe Filliâtre. Verifying two lines of C with Why3: an exercise in program verification. In *Verified Software: Theories, Tools and Experiments (VSTTE)*, 2012. Page 2.

[39] Jean Fortin and Frédéric Gava. BSP-WHY: an intermediate language for deductive verification of BSP programs. In *Proceedings of the fourth international workshop on High-Level Parallel Programming and applications (HLPP)*, pages 35–44. ACM, 2010. Page 13.

[40] H. Gao. *Analysis of Security Protocols by Annotations*. PhD thesis, Technical University of Denmark, 2008. Page 3.

[41] H. Garavel, R. Mateescu, and I. M. Smarandache. Parallel state space construction for model-checking. In M. B. Dwyer, editor, *Proceedings of SPIN*, volume 2057 of *LNCS*, pages 217–234. Springer, 2001. Page 13.

[42] I. Garnier and F. Gava. CPS Implementation of a BSP Composition Primitive with Application to the Implementation of Algorithmic Skeletons. *Parallel, Emergent and Distributed Systems*, 2011. To appear. Page 11.

[43] F. Gava, M. Guedj, and F. Pommereau. A bsp algorithm for the state space construction of security protocols. In *PDMC*, pages 37–44. IEEE Computer Society, 2010. Pages iii, 2, 17 and 19.

[44] F. Gava, M. Guedj, and F. Pommereau. A BSP algorithm for the state space construction of security protocols. In *Ninth International Workshop on Parallel and Distributed Methods in Verification (PDMC 2010)*, pages 37–44. IEEE, 2010. Page 9.

[45] F. Gava, M. Guedj, and F. Pommereau. Performance evaluations of a bsp algorithm for state space construction of security protocols. In *Euromicro Parallel and Distributed proceessing (PDP)*. IEEE, 2012. Page 9.

[46] A. V. Gerbessiotis, C. J. Siniolakis, and A. Tiskin. Parallel priority queue and list contraction: The bsp approach. *Computing and Informatics*, 21:59–90, 2002. Page 11.

[47] A. Ghuloum, E. Sprangle, J. Fang, G. Wu, and X. Zhou. Ct: A Flexible Parallel Programming Model for Tera-scale Architectures. Technical report, Intel Research, 2007. Pages 9 and 11.

[48] Sergei Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM TOPLAS*, 26(1):47–56, 2004. Page 11.

[49] J. Goubault-Larrecq. Finite models for formal security proofs. *Journal of Computer Security*, 18(6):1247–1299, 2010. Page 19.

[50] L. Granvilliers, G. Hains, Q. Miller, and N. Romero. A system for the high-level parallelization and cooperation of constraint solvers. In Y. Pan, S. G. Akl, and K. Li, editors, *Proceedings of International Conference on Parallel and Distributed Computing and Systems (PDCS)*, pages 596–601, Las Vegas, USA, 1998. IASTED/ACTA Press. Page 11.

[51] I. Gu'erin-Lassous and J. Gustedt. Portable List Ranking: an Experimental Study. *ACM Journal of Experiments Algorithms*, 7(7):1–18, 2002. Page 11.

[52] Olivier Heen, Gilles Guette, and Thomas Genet. On the unobservability of a trust relation in mobile ad hoc networks. In Olivier Markowitch, Angelos Bilas, Jaap-Henk Hoepman, Chris J. Mitchell, and Jean-Jacques Quisquater, editors, *Information Security Theory and Practice. Smart Devices, Pervasive Systems, and Ubiquitous Networks (WISTP)*, volume 5746 of *LNCS*, pages 1–11. Springer, 2009. Page 7.

[53] Nevin Heintze, J. D. Tygar, Jeannette Wing, and H. Chi Wong. Model checking electronic commerce protocols. In *Proceedings of the 2nd conference on Proceedings of the Second USENIX Workshop on Electronic Commerce*, WOEC, pages 10–10. USENIX Association, 1996. Page 8.

[54] Paolo Herms. Certification of a chain for deductive program verification. In Yves Bertot, editor, *2nd Coq Workshop, satellite of ITP'10*, 2010. Page 20.

[55] Jonathan M. D. Hill and David B. Skillicorn. Practical Barrier Synchronisation. In *6th EuroMicro Workshop on Parallel and Distributed Processing (PDP'98)*. IEEE Computer Society Press, January 1998. Page 9.

[56] Guy Horvitz and Rob H. Bisseling. Designing a BSP version of ScaLAPACK. In Bruce Hendrickson et al., editor, *Proceedings Ninth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, Philadelphia, PA, 1999. Page 11.

[57] C. P. Inggs. *Parallel Model Checking on Shared-Memory Multiprocessors*. PhD thesis, Department of Computer Science, University of Manchester, 2004. Page 8.

[58] P. Krusche and A. Tiskin. New algorithms for efficient parallel string comparison. In F. Meyer auf der Heide and C. A. Phillips, editors, *Proceedings of the 22nd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 209–216. ACM, 2010. Page 11.

[59] I. Guerin Lassous. *Algorithmes paralleles de traitement de graphes: une approche basee sur l'analyse experimentale*. PhD thesis, University de Paris VII, 1999. Page 11.

[60] E. A. Lee. The Problem with Threads. Technical Report UCB/EECS-2006-1, Electrical Engineering and Computer Sciences University of California at Berkeley, 2006. Page 11.

[61] Kiminori Matsuzaki. Efficient Implementation of Tree Accumulations on Distributed-Memroy Parallel Computers. In *Fourth International Workshop on Practical Aspects of High-Level Parallel Programming (PAPP 2007), part of The International Conference on Computational Science (ICCS 2007)*, 2007. to appear. Page 11.

[62] S. Meier, C. J. F. Cremers, and D. A. Basin. Strong invariants for the efficient construction of machine-checked protocol security proofs. In *Computer Security Foundations (CSF)*, pages 231–245. IEEE Computer Society, 2010. Page 19.

[63] Kedar S. Namjoshi. Certifying model checkers. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification (CAV)*, volume 2102 of *LNCS*, pages 2–13. Springer, 2001. Page 2.

[64] Xavier Rival and Jean Goubault-Larrecq. Experiments with finite tree automata in coq. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics (TPHOL)*, volume 2152 of *LNCS*, pages 362–377. Springer, 2001. Page 19.

[65] R. O. Rogers and D. B. Skillicorn. Using the BSP cost model to optimise parallel neural network training. *Future Generation Computer Systems*, 14(5-6):409–424, 1998. Page 11.

[66] 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)*, IEEE Digital Library, pages 543–548. IEEE, 2010. Page 2.

[67] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, 1999. Page 6.

[68] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997. Page 10.

[69] M. Snir and W. Gropp. *MPI the Complete Reference.* MIT Press, 1998. Page 11.

[70] Christopher Sprenger. A verified model checker for the modal μ-calculus in coq. In *4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1384 of *LNCS*, pages 167–183. Springer-Verlag, 1998. Page 19.

[71] Jun Sun, Yang Liu, and Bin Cheng. Model checking a model checker: A code contract combined approach. In Jin Song Dong and Huibiao Zhu, editors, *12th International Conference on Formal Engineering Methods (ICFEM)*, volume 6447 of *LNCS*, pages 518–533. Springer, 2010. Page 2.

[72] A. Tiskin. *The Design and Analysis of Bulk-Synchronous Parallel Algorithms.* PhD thesis, Oxford University Computing Laboratory, 1998. Page 11.

[73] M. Llanos Tobarra, Diego Cazorla, Fernando Cuartero, and Gregorio Díaz. Analysis of web services secure conversation with formal methods. In *International Conference on Internet and Web Applications and Services (ICIW)*, page 27. IEEE Computer Society, 2007. Page 7.

[74] M. Llanos Tobarra, Diego Cazorla, Fernando Cuartero, Gregorio Díaz, and María-Emilia Cambronero. Model checking wireless sensor network security protocols: Tinysec + leap + tinypk. *Telecommunication Systems*, 40(3-4):91–99, 2009. Page 7.

[75] M. Torabi Dashti, A. Wijs, and B. Lisser. Distributed partial order reduction for security protocols. *ENTCS*, 198:93–99, 2008. Page 17.

[76] E. Turner, M. Butler, and M. Leuschel. A refinement-based correctness proof of symmetry reduced model checking. In *Abstract State Machines, Alloy, B and Z*, LNCS, pages 231–244. Springer, 2010. Pages 19 and 20.

[77] B. Vergauwen and J. Lewi. A linear local model-checking algorithm for ctl. In E. Best, editor, *CONCUR*, volume 715 of *LNCS*, pages 447–461. Springer-Verlag, 1993. Page 8.

[78] Kumar Neeraj Verma, Jean Goubault-Larrecq, Sanjiva Prasad, and S. Arun-Kumar. Reflecting BDDs in Coq. In Jifeng He and Masahiko Sato, editors, *Advances in Computing Science - 6th Asian Computing Science Conference (ASIAN)*, volume 1961 of *LNCS*, pages 162–181. Springer, 2000. Page 19.

Softwares

[1]BIDON pour eviter endnote vide et bug latex, A supprimer pour version finale