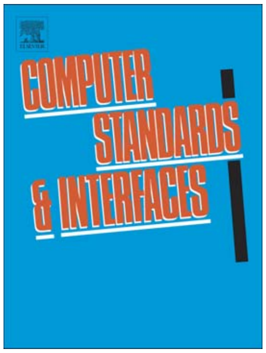

Tools for cryptographic protocols analysis: A technical and experimental comparison

Manuel Cheminod, Ivan Cibrario Bertolotti, Luca Durante, Riccardo Sisto, Adriano Valenzano

PII: S0920-5489(08)00145-1
DOI: doi: 10.1016/j.csi.2008.09.030
Reference: CSI 2639

To appear in: *Computer Standards & Interfaces*

Received date: 11 January 2008
Revised date: 17 September 2008
Accepted date: 28 September 2008

Tools for cryptographic protocols analysis: a technical and experimental comparison ⋆

Manuel Cheminod [a], Ivan Cibrario Bertolotti [a,∗], Luca Durante [a], Riccardo Sisto [b], and Adriano Valenzano [a]

[a] *IEIIT-CNR, c.so Duca degli Abruzzi 24, I-10129 Torino, Italy*

[b] *Politecnico di Torino, c.so Duca degli Abruzzi 24, I-10129 Torino, Italy*

Abstract

The tools for cryptographic protocols analysis based on state exploration are designed to be completely automatic and should carry out their job with a reasonable amount of computing and storage resources, even when run by users having a limited amount of expertise in the field. This paper compares four tools of this kind to highlight their features and ability to detect bugs under the same experimental conditions. To this purpose, the ability of each tool to detect known flaws in a uniform set of well-known cryptographic protocols has been checked. Results are also given on the relative performance of the tools when analysing several known-good protocols with an increasing number of parallel sessions.

Key words: Protocol verification, Formal methods, Network-level security and protection

⋆ This work was developed in the framework of the EU project IST-026600 DESEREC, “Dependability and Security by Enhanced Reconfigurability“, and the CNR project “Metodi e strumenti per la progettazione di sistemi software-intensive ad elevata complessità”.

∗ Corresponding author. Address: IEIIT-CNR c/o Politecnico di Torino, c.so Duca degli Abruzzi 24, I-10129 Torino, Italy.

Email addresses: manuel.cheminod@polito.it (Manuel Cheminod), ivan.cibrario@polito.it (Ivan Cibrario Bertolotti), luca.durante@polito.it (Luca Durante), riccardo.sisto@polito.it (Riccardo Sisto), adriano.valenzano@polito.it (Adriano Valenzano).

1 Introduction

In the last years, researchers devoted much effort to develop techniques to formally analyse cryptographic protocols. Their efforts have been stimulated by a widespread, increasing interest in security issues, and have led to the development of several prototypes of formal analysis tools. Such prototypes have enabled finding out new attacks, not only on new cryptographic protocols being proposed, but even on old protocols which for long had been considered secure.

Tools are usually developed with the aim of implementing and automating analysis techniques, so as to help finding out possible vulnerabilities with minimal effort. Of course, the degree up to which this objective is achieved by each tool may vary, depending on both the different analysis techniques used and the particular implementation choices made.

In this paper attention is focused on completely automatic tools, which are based on state space exploration. The main goal of the paper is to compare some features of a number of publicly-available tools in this class.

After discussing related work in Sect. 2, in this paper we compare four fully automatic state exploration tools for cryptographic protocol analysis, focusing on both qualitative and quantitative experimental issues. In particular, Sect. 3 presents the specification language and the kind of analysis performed by each tool.

Then, in Sect. 4, the results of our experimental comparison of the tools on a suitable set of protocols taken from [23], an online repository of security protocols based on the work of Clark and Jacob [11], are presented. The reported results measure both the ability of each tool to find out known vulnerabilities on the test protocols, and the time taken to completely analyse the test protocols that are considered free of bugs. Last, Sect. 5 draws some conclusions.

2 Related Work

A cryptographic protocol is usually developed in a sequence of phases, from conceptual design down to its practical implementation. Although it is possible to check a cryptographic protocol for vulnerabilities during any of its stages of development, in this paper we focus only on the high-level, conceptual design phase. Working at this level is advantageous because it is possible to keep the analysis technique independent of a number of cumbersome aspects such as, for example, message encodings, cryptographic algorithms and communication channel peculiarities.

On the other hand, any protocol flaw that depends on these aspects will not be detected, but experience shows that defining an attack-free protocol is a challenging task even if they are neglected for a moment. It may therefore be convenient to concentrate at first on creating a conceptually secure protocol built on an idealised model of the underlying cryptographic primitives, and then analyse these lower-level elements. This is also the route taken by most previous papers on the same subject. As as example, one common instance of this idealisation, namely *perfect encryption*, will be presented in Section 4.

A few main classes of formal analysis techniques have been introduced in the past:

(1) *Deductive methods* such as, for example, those described in [8,22,26] derive a formal theory that faithfully represents the protocol being analysed and prove one or more theorems — corresponding to the protocol properties of interest — in that theory. Theorem provers are used to partially automate the proof process, but a certain amount of human intervention is still needed and termination is not always guaranteed.
(2) *Static analysis methods* such as [5] apply compiler-like techniques, like type checking and data/control flow analysis in order to verify the properties of the protocol being analysed. However, they still do not solve the problem in general terms and usually deal only with authenticity properties.
(3) *State exploration methods* model the protocol being analysed as a finite state system of reasonable size and systematically explore the states of the model looking for violations of the properties of interest. For instance, the methods described in [4,6,12,16,19–21,25,27] belong to this category.

With respect to the others, state exploration methods promise complete automation and, for this reason, research is currently very active in this area. Several tools of this kind are available as well and can nowadays be applied to the analysis of protocols of practical relevance, and no longer only to synthetic case studies.

Even if they are still to be looked at as prototypes — namely, they are still far from the level of maturity required to promote their own standardisation and widespread, systematic adoption — they can nevertheless be used to validate existing standard cryptographic protocols or new protocols being considered for standardisation. Within the context of wireless network standards this has been shown, for example, in [10].

Among state exploration methods, on which this paper is focused, one of the main design trade-offs to be considered is between the sophistication and power of the properties to be checked and the complexity of the analysis.

```
A, B, S : principal
Na, Nb : number
Kab, Kbs, Kas : key

1. A -> S : A, B, Na
2. S -> B : {A, B, Na, Kab}Kas, {A, B, Na, Kab}Kbs
3. B -> A : {A, B, Na, Kab}Kas, {Na}Kab, Nb
4. A -> B : {Nb}Kab
```

Fig. 1. An Informal Specification of the *Kao Chow* Protocol

For instance, among the tools being considered in this paper, the secrecy concept adopted by [17] is stronger than the other ones, because it is based on *testing equivalence*. This feature makes the tool more capable, but often hinders its performance because the analysis technique is more complex and prevents the applications of state-space reduction techniques.

Unfortunately, to the best knowledge of the authors, not much work has been published on comparing tools for cryptographic protocol analysis. In general, each author provides experimental data about his/her own tool, but results obtained for one tool are often not directly comparable with the ones given for others. This happens mainly because each protocol comes in many different versions, which can be formalised in several slightly different ways. Thus, the sets of protocols used to test different tools are not the same. Moreover, when execution times are considered, differences in the hardware used to test the various tools are a further element that prevents comparisons.

In [18], four state exploration tools are compared, but this comparison is based only on some of the tool features, and in no way on experimental data. Another work [2] is based on a case study, and compares the use of two languages, namely Haskell and Maude, for modelling and reasoning about cryptographic protocols.

3 The Tools

In this section, several tools for cryptographic protocol analysis based on state exploration are discussed, namely Casper/FDR, STA, S^{3}A, and OFMC. As a running example, we use the first version of the *Kao Chow* protocol described in [23].

Figure 1 describes this protocol using the so-called Alice&Bob-style notation commonly used to informally specify cryptographic protocols. In the description, agents `A` and `B` are supposed to share symmetric keys `Kas` and `Kbs`, respectively, with server `S`, `Kab` is a fresh session key generated by `S` and distributed to A and B, and Na and Nb are nonces. The goals of the protocol are to ensure the secrecy of Kab and to mutually authenticate A and B.

3.1 Casper/FDR

One of the first state-exploration automatic tools used to analyse cryptographic protocols is based on the FDR model checker, a tool marketed by Formal Systems [24]. The FDR input consists of a description of the model to be analysed and a specification representing the desired properties.

Both the model description and the specification are expressed in a machine-readable dialect of the process algebra CSP. FDR can generate the state spaces of the model and of the specification and check whether the model satisfies the specification, i.e., whether the model is a refinement of the specification.

The flexibility of the CSP language makes it possible to describe cryptographic protocol models and related security specifications in an accurate way. However, since the task of writing such CSP descriptions is quite difficult and error-prone, a front-end called Casper [19] has been developed, which takes protocol models and security properties expressed in a simpler language and translates them into CSP.

Figure 2 shows a sample input to Casper, which describes the *Kao Chow* authentication protocol. The specification is divided into several sections:

- The `Free Variables` section declares the types of variables and functions used in the description (in the example, A, B, S, na, nb, kab are variables and SKey is a function). In the same section, an `InverseKeys` declaration specifies which keys are inverses of which others. Although data are typed, multiple types for variables or constants can be specified, thus enabling the search for certain type-flaw attacks. All type-flaw attacks can be searched for, in principle, but this usually leads to cumbersome descriptions. A further drawback is also that non-atomic keys are not allowed.
- The `Processes` section declares each protocol role with its formal parameters, and specifies the initial knowledge that each role has at the beginning of a protocol session. The first formal parameter identifies the agent impersonating the role.
- The `Protocol description` section describes the sequence of message exchanges a protocol session consists of. Protocol roles are identified by the first identifier of the role declaration (A, B and S in the example), and only the exchanged messages are represented, leaving unspecified how each message is processed by the receiver, because Casper assumes that all the possible decoding, decryption, and check operations are performed at message reception. This feature simplifies protocol descriptions but prevents

```
#Free variables
  A, B : Agent
  S : Server
  na, nb : Nonce
  kab : SessionKey
  SKey : Agent -> ServerKey
  InverseKeys = (kab, kab), (SKey, SKey)
#Processes
  INITIATOR(A, na, S) knows SKey(A)
  RESPONDER(B, nb) knows SKey(B)
  SERVER(S, kab) knows SKey
#Protocol description
  0. -> A : B
  [A != B]
  1. A -> S : A, B, na
  [A != B]
  2a. S -> B : {A, B, na, kab}{SKey(A)}%v
  2b. S -> B : {A, B, na, kab}{SKey(B)}
  3a. B -> A : v%{A, B, na, kab}{SKey(A)}
  3b. B -> A : {na}{kab}, nb
  4. A -> B : {nb}{kab}
#Specification
  Agreement(A, B, [kab])
#Actual variables
  Alice, Bob, Mallory : Agent
  Sam : Server
  Na, Nb, Nib, No: Nonce
  Kab, Kold : SessionKey
  Kas, Kbs, Kms : ServerKey
  InverseKeys = (Kab, Kab), (Kold, Kold)
#Functions
  SKey(Alice) = Kas
  SKey(Bob) = Kbs
  SKey(Mallory) = Kms
#System
  INITIATOR(Alice, Na, Sam)
  RESPONDER(Bob, Nb)
  SERVER(Sam, Kab)
#Intruder Information
  Intruder = Mallory
  IntruderKnowledge = {Alice, Bob, Mallory, Sam, \
    Skey(Mallory), No, Kold, {Alice,Bob,No,Kold}{SKey(Alice)}, \
    {Alice, Bob, No, Kold}{SKey(Bob)}}
```

Fig. 2. A Casper Specification of the *Kao Chow* Protocol

the analysis of scenarios where only some of the possible operations are performed at message reception. Instead, if `%v` is appended to the message, the message is simply stored into variable `v`, with no investigation of its contents.

- The `Specification` section lists the security properties that have to be checked. Casper offers a set of authentication and secrecy parametrised properties that can be specified and verified.
- The `Actual variables` and `Free variables` sections are similar, but the former specifies constants rather than variables.
- The `System` and `Intruder Information` sections describe how the protocol model is composed. In particular, the `System` section lists the instantiations of the various protocol roles, with their actual parameters, and the `Intruder Information` section specifies both the intruder identity and initial knowledge.

Casper models the intruder according to the Dolev-Yao model and in a way which is completely transparent to the user. The analysis performed by FDR is a classical explicit model checking, and it is not performed on-the-fly, i.e. the checks are executed only after the whole state space has been built. FDR2, the last revision of FDR, has some built-in reductions to limit the size of the state space. No symbolic technique is used to represent messages. In order to keep the model finite, Casper limits the length of messages built by the intruder as well as the number of agents operating in parallel.

3.2 S^{3}A

S^{3}A (Spi calculus Specifications Symbolic Analyser) [17], is a fully automatic software tool for the formal analysis of cryptographic protocols that reduces the verification of secrecy and authenticity properties to checks of testing equivalence between specifications. S^{3}A performs such checks automatically, by exhaustive state exploration.

The input language of S^{3}A is a machine-readable version of the spi calculus [1], a process algebra that derives from π calculus, with some simplifications and the addition of cryptographic primitives. The spi calculus has two basic language elements: terms, to represent data, and processes, to represent behaviours. Terms are elements of a free term algebra and are untyped, in order to provide the maximum expressive power and enable, among the other things, the specification of non-atomic keys.

The spi calculus process operators let specify the input and output operations carried out by each process, as well as the operations performed on the received data (decomposition and decryption of messages, and equality tests).

The intruder is modelled implicitly and, being represented by a spi calculus specification, has the same expressive power as the language itself. It can be shown that this includes the Dolev-Yao intruder model [13,17].

Figure 3 shows the spi calculus specification of the *Kao Chow* authentication protocol ready to be processed by S^{3}A. The specification consists of a process definition for each role (`Init`, `Resp`, and `S`), and of another process definition describing the model to be analysed (`Inst`).

Let us consider the first process definition, namely, the definition of `Init`:

- The notation `(@NI)` specifies that `NI` is a new, fresh nonce.
- `c<I, N, NI>` is an output on channel `c` of the triple `(I, N, NI)`.
- `c(x_kIS, x_NIkIR, xNR)` is an input from channel `c` of a triple. The three received messages are stored into the three specified variables.
- The next two statements are decryption operations. They are followed by equality comparisons and other output operations.

The `Inst` process in the example includes exactly one instance of each role process. Instances have their own actual parameters that substitute the formal parameters used in the process description. The `|` operator means parallel composition of instances. In the example, the `Inst` process starts with an output operation, which has been introduced to make some terms initially known to the intruder. It can be noted that the intruder behaviour is not explicitly described, but it is implicitly assumed when analysing the protocol.

Two kinds of security properties can be specified: secrecy and authenticity. To specify secrecy, it is just necessary to specify which terms are expected to be kept secret. The secrecy concept adopted by S^{3}A is stronger than the one normally adopted by other tools, because it is based on testing equivalence. If $\mathit{Inst}(M)$ is the specification of an instance of the protocol, parametrised by a data term M which should remain secret, the secrecy property can be formally expressed as:

$$\mathit{Inst}(M) \simeq \mathit{Inst}(M') \text{ if } F(M) \simeq F(M') \;\; \forall M, M' \tag{1}$$

where $\simeq$ means testing equivalence and $F(M)$ is the final action the protocol accomplishes on the secret M. It is worth noting that this way of expressing secrecy, besides capturing the fact that an intruder must not be able to acquire knowledge of M, also requires that an intruder must not be able to *infer* anything about M. In other words, this secrecy specification requires that each intruder who knows M and M' must be unable to distinguish between two sessions where M and M' are transmitted, respectively.

With respect to authenticity, S^{3}A requires that two specifications be written:

```
Init(I, R, kIS) :=
(@NI)(
 c<I, R, NI>.
 c(x_kIS, xNIkIR, xNR).
 case x_kIS of {xI, xR, xNI_1, xkIR}kIS in
   case xNIkIR of {xNI_2}xkIR in
     [xI is I] [xR is R] [xNI_1 is xNI_2] [xNI_2 is NI]
     c<{xNR}xkIR>.
     (@M)(c<{M, M}xkIR>.
          0
     )
)

Resp(R, kRS) :=
c(x_kIS, x_kRS).
case x_kRS of {xI, xR, xNI, xkIR}kRS in
 [xR is R]
 (@NR)(
   c<x_kIS, {xNI}xkIR, NR>.
   c(xNRkIR).
   case xNRkIR of {xNR}xkIR in
     [xNR is NR]
     c(xSkIR).
     case xSkIR of {xM_1, xM_2}xkIR in
       c<xM_1,xM_2>.
       0
  )

S(I, R, kIS, kRS) :=
c(xI, xR, xNI).
[xI is I] [xR is R]
(@kIR)(
 c<{xI, xR, xNI, kIR}kIS, {xI, xR, xNI, kIR}kRS>.
 0
)

Inst() :=
(@kAS)(@kBS)(
 c<A, B, Kold, Nold,
   {A, B, Nold, Kold}kAS, {A, B, Nold, Kold}kBS>.
 Init(A, B, kAS)
 | Resp(B, kBS)
 | S(A, B, kAS, kBS)
)
```

Fig. 3. A S^{3}A Specification of the *Kao Chow* Protocol

the first one is the description of the protocol while the second one is a reference specification, which is similar to the protocol specification, except for the fact

```
DeclLabel $ a1,a2,a3, b1,b2,b3, s1,s2, disclose,acceptBA $;
DeclVar   $ xKab,xNb, yToA,yNa,yKab, zNa, z             $;
DeclName  $ A,B,S,Kab,  Na,Kas, Nb,Kbs, Kold,Nold       $;

val prA = (Na new_in (a1!(A,B,Na) >>
                      a2?({A,B,Na,xKab}Kas,{Na}xKab,xNb) >>
                      a3!({xNb}xKab) >>
                      stop));

val prB = (b1?(yToA,{A,B,yNa,yKab}Kbs) >>
          (Nb new_in (b2!(yToA,{yNa}yKab,Nb) >>
                      b3?({Nb}yKab) >>
                      acceptBA!({Nb}yKab) >>
                      stop)));

val prS = (s1?(A,B,zNa) >>
          (Kab new_in (s2!({A,B,zNa,Kab}Kas,{A,B,zNa,Kab}Kbs) >>
                       stop )));

val KaoChow =
     ( [disclose!(A,B,S,Nold,Kold,
                  {A,B,Nold,Kold}Kas,
                  {A,B,Nold,Kold}Kbs)]
        @ (prA || prB || prS ) );

val Auth1  = (a3!(z) <-- acceptBA!(z));
```

Fig. 4. A STA Specification of the *Kao Chow* Protocol

that the authenticity of the messages is enforced. Testing equivalence of the two specifications implies that authenticity holds.

S^{3}A deals with the whole spi calculus, with the only exception of the replication operator, which introduces an unbounded number of processes. Leaving replication out, models are kept finite thanks to symbolic representations of messages. When checking for authenticity, S^{3}A does not work on-the-fly: it first generates the whole state space of the two specifications to be compared and then checks for equivalence. Instead, when checking for secrecy, S^{3}A can work on-the-fly. If the testing equivalence check fails, S^{3}A is capable of synthesising the spi calculus specification of an intruder that can discriminate between the checked specifications, thus possibly leading to an attack. To overcome the issue of state explosion, inherent in exhaustive state exploration methods, S^{3}A exploits state space symmetries and a limited form of partial order.

3.3 STA

STA (Symbolic Trace Analyser) [6,7] is a model checker for cryptographic protocols relying on symbolic techniques that avoid the explicit construction of the infinite messages that an attacker can send to protocol agents when they carry out an input action.

Also in this case, protocols are described by means of a dialect of the spi calculus [1] but, unlike in the language described in [1], the authors of [6,7] assume a single public channel over which data are exchanged, and no integers are allowed. In STA, according to the underlying theory, terms are untyped and their free-term algebra allows them to be arbitrarily nested, but all keys, either shared or public/private, must be atomic; the implementation of STA provides limited support for non-atomic keys.

The intruder is modelled implicitly and conforms to the well-known Dolev-Yao model [14], with the additional ability (that has to be specified explicitly) for the intruder to assume the role of a legitimate protocol participant.

STA allows to express and verify authentication properties based on correspondence assertions: in any protocol trace a certain action β must follow an action α in the same trace. Moreover, secrecy properties about certain values are verified by means of ad-hoc actions in the specification, designed so as to check that the intruder does not learn secrets at any interaction point between the intruder and the protocol.

For example, Fig. 4 shows the STA specification of the *Kao Chow* authentication protocol. The specification is made up of several sections, in which:

- The `DeclLabel` keyword introduces labels, which are useful in helping the user to read the sequence of events leading to an attack;
- `DeclVar` gives the whole set of variables which bind terms in input actions;
- `DeclName` specifies the objects' identifiers, e.g. agents' identities and keys;
- `Val <agent_name>` introduces the behaviour of each agent where:
  - `<identifier> new_in` establishes a fresh name, known in the current scope only (e.g. a nonce);
  - `a!M` is the output action, labelled as `a`, of term `M`;
  - `a?M` represents an input action where `M` is a message pattern with variables. Any (intruder-generated) message matching it is allowed as input. As an example, `b1?(yToA,{yA,yB,yNa,yKab}Kbs)` matches any pair of terms, where the first term in the pair is bound to `yToA`, while the second is a tuple encrypted with shared key `Kbs`, whose elements are bound respectively to `yA`, `yB`, `yNa` and `yKab` after the input;
  - `(M is N)` is a guard that enables the following action only if a match between terms `M` and `N` exists;

- `>>` is the sequencing operator;
- `||` is the parallel composition operator;
- `stop` labels the end of the agent's activity;
- `disclose` is the special label which introduces the intruder's initial knowledge elements. In the example they consist of the agents' names, a nonce, a session key and two protocol messages exchanged in a previous session;
- `@(...)` instantiates agents;
- `Auth1 = (a3!(z) <-- acceptBA!(z))` establishes a correspondence between the last output action of agent `A` and the corresponding message read by `B` in any protocol session: `B` must publish `z` *after* `A` has sent the *same value*. Any violation detected is an authentication flaw.

The lack of parametrisation in this language leads specification sizes to grow rapidly as more instances of a role are needed.

Roles must be finite in number and behaviour. On the other hand, the symbolic representation of terms allows to replace the infinite set of messages the intruder can send to the legitimate participants on each input action of the protocol with a finite one. Variables provide a finite representation of the infinite set in the first place, and can subsequently be constrained to either assume or not assume specific values and syntactic forms during the course of the analysis, in order to satisfy tests and requirements posed on them by the receiving agents as they check and use them.

In order to simplify the symbolic trace generation system, some symbolic traces may not have, in general, a corresponding concrete one. Thus, when a flaw is detected by the on-the-fly analysis, some kind of refinement is needed to determine if there really exists at least one concrete trace to exploit it.

The analysis stops when a violation is detected, and the protocol steps leading to it are then shown.

3.4 OFMC

OFMC (On-the-Fly Model Checker) [4] is a model-checker for cryptographic protocols relying on *lazy* techniques to reduce the computational effort required to carry out the analysis.

Protocols are described by means of HLPSL (High-Level Protocol Specification Language) that, in an untyped free-term algebra context, supports both symmetric and asymmetric non-atomic keys, one-way functions and inequalities. Moreover, some kind of support is provided for operators with algebraic properties, for example exponentiation.

```
Protocol KaoChow;
  Identifiers
    A, B, S: user;
    Na, Nb: number;
    Kas, Kbs, Kab: symmetric_key;
  Knowledge
    A: S, B, Kas;
    B: A, S, Kbs;
    S: A, B, Kas, Kbs;
  Messages
    1. A -> S: A, B, Na
    2. S -> B: {A, B, Na, Kab}Kas, {A, B, Na, Kab}Kbs
    3. B -> A: {A, B, Na, Kab}Kas, {Na}Kab, Nb
    4. A -> B: {Nb}Kab
  Session_instances
    [ A: a; B: b; S: se; Kas: kas; Kbs: kbs ];
  Intruder divert, impersonate;
  Intruder_knowledge a, b, se;
  Goal Short_Term_secret Kab;
  Goal B authenticate A on Nb;
```

Fig. 5. A HLPSL Specification of the *Kao Chow* Protocol

HLPSL specifications are then translated into an intermediate language, IF (Intermediate Format), which is used to carry out the analysis by means of a software tool written in Haskell.

The intruder is modelled implicitly and conforms to the well-known Dolev-Yao model [14], with the additional ability (that has to be specified explicitly) for the intruder to assume the role of a legitimate protocol participant.

For example, Fig. 5 gives the HLPSL specification of the *Kao Chow* authentication protocol. The specification is made up of several sections, in which:

- the `Protocol` keyword introduces the protocol name;
- `Identifiers` gives the set of identifiers used by the specification and their type, which determines their properties;
- `Knowledge` specifies what atomic messages a protocol agent must initially know in order to execute the protocol;
- `Messages` lists the sequence of exchanged messages in the commonly-used Alice&Bob-style notation;
- `Session_instances` specifies the scenario to be used in the analysis; in this case, we specify a single session where agents `a`, `b`, and `se` play the roles of `A`, `B`, and `S` respectively, with their proper keys.

The last statements of the specification define the abilities of the intruder, its initial knowledge and the security goals that the protocol should achieve, respectively.

In this case, the intruder is able to send messages under the identity of any other legitimate agent (as indicated by the `impersonate` keyword) and intercepts all messages sent by one agent to another (`divert`). The initial intruder knowledge is made up of the names of the legitimate protocol agents `a`, `b`, and `se`.

Moreover, we require that the intruder does not get hold of the session key `Kab` and that the agent playing role `B`, at the end of its session, believes that the agent playing role `A` also terminated its own session, and both of them agree on the value of `Nb`.

OFMC is based on two complementary techniques: the *lazy demand-driven search* and the *lazy intruder*. The former provides a finite representation of an infinite state space, because each portion of the state space is really computed only when it is being analysed. As a consequence, there is not any *a priori* limit set on the depth of the state space exploration, although such a limit is still needed to ensure the termination of the analysis on a protocol with no flaws.

On the other hand, the lazy intruder technique replaces the infinite set of messages the intruder can send to the legitimate participants on each input action of the protocol by symbolic *variables*. Variables provide a finite representation of the infinite set in the first place, and can subsequently be constrained to either assume or not assume specific values and syntactic forms during the course of the analysis, in order to satisfy tests and requirements posed on them by the receiving agents as they check and use them.

Recent work on OFMC has been focused on the integration of reduction techniques based on partial-order reduction [3], and the eventual introduction of heuristics is foreseen to further improve the efficiency of the analysis.

4 Experimental Results

In order to assess their error-detection capabilities, the tools described in Sect. 3 have been tested on a subset of the security protocols described in [23]. The subset has been chosen so as to specify each protocol with all the formalisms adopted by the tools under test, that is, we discarded those protocols that were based on the use of either algebraic operators that go beyond the free-term algebra assumption (for example, *exponentiation* or *exclusive or*), or cryptographic algorithms that do not satisfy the following *perfect encryption* assumptions:

Table 1
Error-Detection Capabilities of the Tools, Tested on a Subset of the Security Protocols Open Repository [23]

#	**Protocol**	**Attack Type**	**S^{3}A**	**OFMC**	**STA**	**Casper**
1	Andrew Secure RPC	Freshness	**Y**	**Y**	**Y**	**Y**
2	BAN mod. Andrew Sec. RPC	Parallel session	**Y**	**Y**	**Y**	**Y**
3	BAN concrete Andrew Sec. RPC	Parallel session	**Y**	**Y**	**Y**	**Y**
4	CCITT x509 (3)	Parallel session	**Y**	**Y**	**Y**	**Y**
5	Denning-Sacco shared key	Freshness	**Y**	**Y**	N^h	**Y**
6	Kao Chow authentication 1	Freshness	**Y**	**Y**	**Y**	**Y**
7	KSL (rep. part)	Parallel session	**Y**	**Y**	**Y**	**Y**
8		Parallel session	**Y**	**Y**	**Y**	**Y**
9	KSL	Parallel session	**Y**	N^e	**Y**	N^r
10		Parallel session	**Y**	N^e	N^f	N^r
11	Neumann Stubblebine (rep. part)	Parallel session	**Y**	**Y**	**Y**	**Y**
12	Neumann Stubblebine	Type-flaw	**Y**	**Y**	**Y**	N^h
13	Needham-Schroeder Public Key	Parallel session	**Y**	**Y**	**Y**	**Y**
14	Needham-Schroeder Symmetric Key	Freshness	**Y**	**Y**	**Y**	**Y**
15	Otway Rees	Type-flaw	N^h	**Y**	N^h	N
16		Type-flaw	N^h	**Y**	N^h	N
17		Type-flaw	N^h	N^f	N^h	N
18	SPLICE/AS	Parallel session	**Y**	**Y**	**Y**	N
19		Binding	**Y**	**Y**	**Y**	**Y**
20		Parallel session	N^r	**Y**	**Y**	N^r
21	Hwang/Chen mod. SPLICE/AS	Parallel session	**Y**	**Y**	**Y**	**Y**
22	Clark/Jacob mod. Hwang/Chen	Freshness	N^r	**Y**	N	N^r
23	TMN	Other	**Y**	**Y**	**Y**	**Y**
24		Other	**Y**	**Y**	**Y**	**Y**
25		Parallel session	N^r	N^f	N^f	N
26	Woo/Lam mutual authentication	Parallel session	**Y**	**Y**	**Y**	N
27		Type-flaw	**Y**	**Y**	**Y**	N^h
28	Woo/Lam Π	Parallel session	**Y**	N^f	N^f	N
29		Parallel session	**Y**	**Y**	**Y**	**Y**
30		Parallel session	**Y**	**Y**	**Y**	**Y**
31	Woo/Lam Π^1	Type-flaw	**Y**	**Y**	**Y**	N^h
32	Woo/Lam Π^2	Type-flaw	**Y**	**Y**	**Y**	N^h
33	Woo/Lam Π^3	Type-flaw	**Y**	**Y**	**Y**	N^h
34	Yahalom	Type-flaw	N^h	**Y**	N^h	N
35	BAN simplified Yahalom	Type-flaw	**Y**	**Y**	**Y**	N
36		Parallel session	**Y**	**Y**	**Y**	**Y**
37		Type-flaw	**Y**	**Y**	**Y**	N
	Total		**30**	**32**	**28**	**18**

- an encrypted message can be decrypted only if the right decryption key is known,
- there is enough redundancy in the cryptosystem to prevent encryption collisions and to make the decryption algorithm able to determine whether the decryption succeeded or not,
- it is impossible to guess or forge any secret data item.

Regarding timestamps, we took into account only protocols that either do not use them, or use them in a way trivial enough to handle them as if they were nonces, even if this assumption could weaken the correctness (or reliability) of the analysis to some extent.

For each protocol, the analysis has been limited to the known flaws explicitly mentioned in [4,11,23]. However, to make the results meaningful even when looking for undocumented bugs, we wrote the protocol specifications without using any "a priori" knowledge of the bugs.

We also kept the specifications as simple and close to the Alice&Bob-style notation as possible, according to the widely-accepted statement that these tools should require little expertise to be used proficiently. As a consequence, in a small number of cases where we assert that a tool does not detect a flaw, it might indeed be possible to find it by an appropriate, ad-hoc manipulation of the specification, hence we will spot this occurrence in the results.

For example, STA is unable to easily detect the replay attack in the Denning-Sacco shared key protocol, number 5 in Table 1, unless the intruder's knowledge is enlarged with messages exchanged in a previous run of the protocol, even though that run is not included in the specification to limit the complexity of the analysis itself. Similarly, S^{3}A and STA are unable to find the type-flaw attacks 15–17 and 34 in Table 1, without forcing a suitable associativity on several tuples, and Casper, being based on a typed theory, cannot detect a type-flaw attack unless the types of the data items involved in the type flaw are explicitly enumerated in the specification, as also stated in [15].

Table 1 shows the results of the tests, carried out on a total of 37 distinct flaws and fully documented in [9]. The left-hand side of the table briefly describes each flaw and the affected protocol; in analogy to the classification given in [11], we assume that:

- A *freshness* attack occurs when a message captured by the intruder in a previous protocol session is replayed, possibly as a message component, in the current protocol session; to exploit this kind of attack, the protocol sessions shall not necessarily run in parallel.
- A *type-flaw* attack involves the malicious replacement of a message component with another message of a different type by the intruder. In turn, this replacement leads the recipient and the principal, that created the compo-

nent, to different interpretations of the message itself.

- A *parallel session* attack requires the parallel execution of more than one protocol sessions to be exploited, and the intruder uses messages coming from one session to synthesise messages in the other(s).
- In a *binding* attack, the intruder exploits the protocol's failure to establish a proper binding between a public key and its owner.

When an attack may be assigned to multiple categories, the last matching category in the list prevails. It is also worth noting that several protocols listed in the table are made up of two parts: the initial part, performed once per session, is usually concerned with the generation and exchange of a session key, while the final part, that can be repeated several times, performs a mutual authentication. During the tests, the tools have been used to check both the full-fledged protocol and the mutual authentication part on its own. In the table, the latter case has been marked "rep. part".

On the right-hand side, Table 1 summarises the behaviour of the tools, namely, information is given on whether the tools were successful in discovering the flaw or not, and on any problem they encountered during the analysis.

Table 2 presents the results in aggregate form, by giving their distribution for each tool, and gives more details on the keys being used to represent the test results. In particular, the symbol N^h means that the tool was unable to find a given flaw unless it was given some help by means of a custom specification that somewhat reflected an "a priori" knowledge of the flaw itself, while N^r means that the tool was unable to complete the analysis because it ran out of system resources (more than 2 hours of CPU time, 2 GB of disk or 512 MB of RAM). Finally, the symbol N^e is used when the tool was unable to complete the analysis because either it encountered an internal error, or gave no meaningful result.

The symbol N^f is used only for those protocols affected by multiple flaws and for the tools which stop immediately after the first flaw is detected. It means that the tool was not able find out a flaw because it was "masked off" by another bug that was detected first and caused the tool to stop. In this case indeed, the tool might find the second bug if the specification were amended to fix the one detected first. However, this possibility has not been investigated further.

OFMC appears to be the best tool among those considered, both because it is able to find out the vast majority of known flaws without any help from the user (it has no failures in the N^h class) and because its resource requirements, due to the effectiveness of the lazy evaluation techniques it is based on, are quite small, as remarked by the absence of failures due to resource exhaustion (N^r). The only weakness is its impossibility to fully analyse a protocol with

Table 2
Distribution of the Test Results by Type

Key	**Meaning of the Test Result**	**S^{3}A**	**OFMC**	**STA**	**Casper**
Y	Flaw detected with a standard specification	**30**	**32**	**28**	**18**
N	Flaw undetected	—	—	1	10
N^h	Flaw detected only with a custom specification	4	—	5	5
N^r	Resources exhausted.	3	—	—	4
N^f	The tool stops when it detects the first flaw	—	3	3	—
N^e	Internal error or no result	—	2	—	—

multiple flaws in a single run, because OFMC stops immediately when it finds an attack. From this point of view, tools like S^{3}A and Casper, that perform an exhaustive enumeration of the state space, behave better and, in fact, have no failures in the N^f class. The other side of the medal is a greater consumption of CPU, memory and disk resources, which leads to several failures belonging to the N^r class. STA requirements in terms of resources are similar to those of S^{3}A, but STA also stops the analysis when it finds out a flaw.

The limited performance of S^{3}A can be partially justified if we observe that this tool carries out a testing equivalence check that is more powerful, but also more expensive, than the checks based on reachability analysis, because it limits the applicability and efficacy of the state-space reduction techniques based on symmetries or partial order reductions. An additional point in favour of S^{3}A is that it can be used to check also other kinds of security properties such as secrecy, while the other tools considered in this paper are able to verify only authentication.

With respect to type-flaw attacks, both S^{3}A and STA often need some help to explicitly describe the associativity of tuples in order to exploit the attack, when it differs from the default, while OFMC is able to find out type-flaw attacks without any intervention. On the other hand, the design of Casper makes it weak in this area; in fact, most failures in the N^h class as well as several failures in the N class can be attributed to this limitation.

Instead, other failures of Casper are probably due to its lack of symbolic data representation capabilities. In turn, this limitation forces the tool to artificially place an upper limit on the length of the messages synthesised by the intruder, thus possibly neglecting a portion of the state space containing an attack trace.

Last, it should be noted that the joint usage of OFMC and S^{3}A covers the maximum number of flaws, and that flaws detected by STA and Casper are a proper subset of them.

Table 3
Execution Times on Protocols Without Flaws

Protocol	**Scenario**	**S^3A**	**OFMC**	**STA**	**Casper**
BAN modified CCITT X.509 (3)	1A 1B	0.04 s	0.01 s	0.15 s	N^r
	2A 1B	2.11 s	0.02 s	1218.33 s	
	2A 2B	N^r	0.07 s	N^r	
	3A 2B		0.24 s		
	3A 3B		1.61 s		
	4A 3B		5.62 s		
	4A 4B		N^r		
Lowe modified Denning-Sacco shared key	1A 1B 1S	0.07 s	0.01 s	0.03 s	76.8 s
	2A 1B 1S	1.50 s	0.03 s	1.48 s	154.26 s
	2A 2B 1S	611.20 s	0.03 s	109.21 s	168.52 s
	2A 2B 2S	N^r	0.11 s	2289.59 s	N^r
	3A 3B 3S		3.93 s	N^r	
	4A 4B 4S		N^r		
Kao Chow Authentication v.2	1A 1B 1S	0.15 s	0.02 s	0.12 s	15.20 s
	2A 1B 1S	1.70 s	0.03 s	0.44 s	21.82 s
	2A 2B 1S	21.20 s	0.03 s	7.71 s	29.35 s
	2A 2B 2S	N^r	0.36 s	1162.87 s	N^r
	3A 3B 3S		N^r	N^r	
Kao Chow Authentication v.3	1A 1B 1S	0.04 s	0.03 s	0.40 s	216.10 s
	2A 1B 1S	2.09 s	0.03 s	1.33 s	260.30 s
	2A 2B 1S	26.46 s	0.04 s	14.11 s	416.49 s
	2A 2B 2S	N^r	0.45 s	N^r	N^r
	3A 2B 2S		1.42 s		
	3A 3B 2S		12.70 s		
	3A 3B 3S		N^r		
Amended Needham Schroeder Symmetric Key	1A 1B 1S	0.69 s	0.06 s	0.40 s	N^r
	2A 1B 1S	307.10 s	0.16 s	97.20 s	
	2A 2B 1S	N^r	0.47 s	N^r	
	2A 2B 2S		3.58 s		
	3A 2B 2S		N^e		
Lowe modified Yahalom	1A 1B 1S	0.07 s	0.06 s	0.03 s	11.46 s
	2A 1B 1S	4.33 s	0.20 s	0.37 s	12.76 s
	2A 2B 1S	3906.00 s	2.88 s	74.90 s	14.25 s
	2A 2B 2S	N^r	N^r	1534.90 s	20.96 s
	3A 2B 2S			N^r	385.60 s
	3A 3B 2S				N^r
Paulson's strengthened Yahalom	1A 1B 1S	0.06 s	0.02 s	0.07 s	99.19 s
	2A 1B 1S	2.95 s	0.03 s	0.62 s	124.25 s
	2A 2B 1S	1858.10 s	0.21 s	73.20 s	154.28 s
	2A 2B 2S	N^r	1.89 s	N^r	377.13 s
	3A 2B 2S		N^e		N^r

The tools have also been used to analyse several other protocols to extend the comparison to the performance point of view. The analysis has been carried out with an increasing number of instances for each role, to give an idea of how the execution time grows when the number of instances of each role increases. As mentioned before, some of the considered tools stop immediately when they detect a flaw, while others continue the analysis until they find all the flaws they are able to. For this reason tests were restricted to those protocols without known flaws; this enables a quite fair evaluation and forces all the tools to analyse protocols completely. All the tools have been run on the same machine, namely, a conventional personal computer equipped with an AMD Athlon XP 2600+ CPU and 512 MB of RAM.

Table 3 summarises the obtained results. Protocols are listed on the left, along with the scenarios being analysed, namely the number of parallel instances of the protocol roles. The right portion of the table gives the corresponding execution times for each tool. The same keys already introduced in Table 2 have been used to indicate when a tool was unable to finish the analysis and why. Clearly, when a tool failed on a scenario, it was not run on more complex scenarios of the same protocol any longer.

It is worth noting that, although [4,15] report the detection of a flaw in versions 2 and 3 of the Kao Chow protocol, they take as a reference the protocol descriptions of [11], which are not the same as those taken in [23]. When adopting the specifications in [23], all the tools considered in this paper agree on the absence of flaws and [23] does not report any known bug indeed.

In general, experiments show that the best performance levels are reached by OFMC, followed by STA, S^3A, and Casper in that order. One notable exception is the analysis of the two bug-free variations on the Yahalom protocol, where Casper behaves very well, even better than OFMC in at least one case. One explanation of this “anomaly”, can be found, bearing in mind that the Yahalom protocol involves complex messages and that Casper was unable to find some of the bugs in a buggy version of this protocol (cases 35–37 in Table 1). In fact, the reason may lie, once again, in the upper bound enforced by Casper on the length of the messages generated by the intruder; this approach shrinks the portion of the state space the tool actually explores significatively in this case. This behaviour affects the accuracy of the analysis undoubtedly, but can also make the analysis time considerably shorter.

5 Conclusions

The use of analysis and verification automatic tools based on formal methods is becoming more and more pervasive in the community of cryptographic protocols. This paper has presented some results obtained by testing some popular publicly available tools on an experimental basis. In particular, it has been aimed at evaluating and comparing the tools on a common ground from both the designer and user points of view.

To our knowledge, in fact, no work has still appeared in the literature, that offers the reader results collected by testing different tools on a (quite) large shared protocol basis, as each author usually provides different tests in different conditions for his/her own tool.

The work described in this paper should be considered as a first step to get information on the tool behaviours when they are run to analyse the same protocols in the same configurations. We have tried to set up a fair testing environment for all the tools considered in the paper. This has led to the selection of a suitable set of cryptographic protocols for the analysis and to some choices in the way specifications have been written.

With respect to the comparison, attention has been focused on two main aspects: first the ability of each tool to discover known flaws has been checked. Second, we have also obtained some performance figures by running the tools on a number of protocols that are considered bug-free, and by exploring different configurations.

Obtained results are encouraging and show that the automatic tools we considered can be of significant help in analysing protocols, even though they are still available only in a prototype version. Moreover, their performance are not totally unsatisfactory. However, in a number of situations some tools were unable to complete the analysis in a reasonable time.

Finally, we are conscious that our work cannot be considered exhaustive, but rather as a preliminary contribution to a deeper comparative evaluation of automatic tools. In fact, much work has still to be done, for instance to extend the experiments to other tools, to enlarge the protocol basis and to take into account other performance indices. In addition, the ability of the different tools to discover unknown bugs should also be tested and this will be one of the goals of our next researches in this area.

References

[1] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Inf. Comput.*, 148(1):1–70, 1999. DOI 10.1006/inco.1998.2740.

[2] D. Basin and G. Denker. Maude versus Haskell: an experimental comparison in security protocol analysis. *Electr. Notes Theor. Comput. Sci*, 36, 2000.

[3] D. Basin, S. Mödersheim, and L. Viganò. Constraint differentiation: A new reduction technique for constraint-based analysis of security protocols. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS 2003)*, pages 335–344, New York, 2003. ACM Press.

[4] D. Basin, S. Mödersheim, and L. Viganò. OFMC: A symbolic model checker for security protocols. *Int. J. Inf. Secur.*, 4(3):181–208, 2005. Special issue on ESORICS 2003.

[5] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Riis Nielson. Automatic validation of protocol narration. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW 2003)*, pages 126–140, Washington, 2003. IEEE Computer Society Press.

[6] M. Boreale and M. G. Buscemi. Experimenting with STA, a tool for automatic analysis of security protocols. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC 2002)*, pages 281–285, New York, 2002. ACM Press.

[7] M. Boreale and M. G. Buscemi. A framework for the analysis of security protocols. In *Proceedings of the 13th International Conference on Concurrency Theory (CONCUR 2002)*, volume 2421 of *Lecture Notes in Computer Science*, pages 483–498, Berlin, 2002. Springer-Verlag.

[8] M. Boreale, R. De Nicola, and R. Pugliese. Proof techniques for cryptographic processes. *SIAM J. Comput.*, 31(3):947–986, 2002. DOI 10.1137/S0097539700377864.

[9] M. Cheminod. Analisi formale di protocolli crittografici. Master's thesis, Politecnico di Torino, 2005. In italian.

[10] Ivan Cibrario Bertolotti, Luca Durante, Paolo Maggi, Riccardo Sisto, and Adriano Valenzano. Improving the security of industrial networks by means of formal verification. *Computer Standards & Interfaces*, 29(3):387–397, 2007.

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

[12] E. M. Clarke, S. Jha, and W. Marrero. Verifying security protocols with Brutus. *ACM Trans. Softw. Eng. Meth.*, 9(4):443–487, 2000. DOI 10.1145/363516.363528.

[13] R. De Nicola and M. C. B. Hennessy. Testing equivalence for processes. *Theor. Comput. Sci.*, 34(1-2):84–133, 1984.

[14] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Trans. Inf. Theory*, 29(2):198–208, 1983.

[15] B. Donovan, P. Norris, and G. Lowe. Analyzing a library of security protocols using Casper and FDR. In *Proceedings of the Workshop on Formal Methods and Security Protocols*, 1999.

[16] A. Durante, R. Focardi, and R. Gorrieri. A compiler for analyzing cryptographic protocols using noninterference. *ACM Trans. Softw. Eng. Meth.*, 9(4):488–528, 2000.

[17] L. Durante, R. Sisto, and A. Valenzano. Automatic testing equivalence verification of spi calculus specifications. *ACM Trans. Softw. Eng. Meth.*, 12(2):222–284, 2003. DOI 10.1145/941566.941570.

[18] Kieran Healy, Tom Coffey, and Reiner Dojen. A comparative analysis of state-space tools for security protocol verification. In *Proceedings of the 3rd WSEAS International Conference on E-Activities*, Crete, Greece, 2004. WSEAS.

[19] G. Lowe. Casper: a compiler for the analysis of security protocols. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop (CSFW 1997)*, pages 18–30, Washington, 1997. IEEE Computer Society Press. DOI 10.1109/CSFW.1997.596779.

[20] J. K. Millen, S. C. Clark, and S. B. Freedman. The Interrogator: Protocol security analysis. *IEEE Trans. Softw. Eng.*, 13(2):274–288, 1987.

[21] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murφ. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy (S&P 1997)*, pages 141–151, Washington, 1997. IEEE Computer Society Press.

[22] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *J. Comput. Sec.*, 6:85–128, 1998.

[23] Projet EVA (Explication et Vérification Automatique de protocoles cryptographiques). Security protocols open repository. Available online, at `http://www.lsv.ens-cachan.fr/spore/index.html`, 2003.

[24] A. W. Roscoe. *A Classical Mind, Essays in Honour of C. A. R. Hoare*, chapter Model-checking CSP. International Series in Computer Science. Prentice Hall, Hertfordshire, 1994.

[25] A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *Proceedings of the 8th IEEE Computer Security Foundations Workshop (CSFW 1995)*, pages 98–107, Washington, 1995. IEEE Computer Society Press.

[26] S. Schneider. Verifying authentication protocols in CSP. *IEEE Trans. Softw. Eng.*, 24(9):741–758, 1998. DOI 10.1109/32.713329.

[27] D. X. Song. Athena: a new efficient automatic checker for security protocol analysis. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW 1999)*, pages 192–202, Washington, 1999. IEEE Computer Society Press.