Using Interpolation for the Verification of Security Protocols[⋆]

Marco Rocchetto, Luca Viganò, Marco Volpe, and Giacomo Dalle Vedove

Dipartimento di Informatica, Università di Verona, Italy

Abstract. Interpolation has been successfully applied in formal methods for model checking and test-case generation for sequential programs. Security protocols, however, exhibit such idiosyncrasies that make them unsuitable to the direct application of such methods. In this paper, we address this problem and present an interpolation-based method for security protocol verification. Our method starts from a formal protocol specification and combines Craig interpolation, symbolic execution and the standard Dolev-Yao intruder model to search for possible attacks on the protocol. Interpolants are generated as a response to search failure in order to prune possible useless traces and speed up the exploration. We illustrate our method by means of a concrete example and discuss the results obtained by using a prototype implementation.

Keywords: Security protocols, Symbolic execution, Craig's interpolation, Formal methods, Verification.

1 Introduction

Context and Motivation. Devising security protocols that indeed guarantee the security properties that they have been conceived for is an inherently difficult problem and experience has shown that the development of such protocols is a highly error-prone activity. A number of tools have thus been developed for the analysis of security protocols at *design time*: starting from a formal specification of a protocol and of a property it should achieve, these tools typically carry out model checking or automated reasoning to either *falsify* the protocol (i.e., find an attack with respect to that property) or, when possible, *verify* it (i.e., prove that it does indeed guarantee that property, perhaps under some assumptions such as a bounded number of interleaved protocol sessions [17]). While verification is, of course, the optimal result, falsification is also extremely useful as one can often employ the discovered attack trace to directly carry out an attack on the protocol implementation (e.g., [3]) or exploit the trace to devise a suite of test cases so as to be able to analyze the implementation at *run-time* (e.g., [4,6]).

Such an endeavor has already been undertaken in the programming languages community, where, for instance, *interpolation* has been successfully applied in

⋆ Work partially supported by the FP7-ICT-2009-5 Project no. 257876, "SPaCIoS: Secure Provision and Consumption in the Internet of Services".

formal methods for model checking and test-case generation for sequential programs, e.g., [12,13], with the aim of reducing the dimensions of the search space. Since a state space explosion often occurs in security protocol verification, we expect interpolation to be useful also in this context. Security protocols, however, exhibit such idiosyncrasies that make them unsuitable to the direct application of the standard interpolation-based methods, most notably, the fact that, in the presence of a Dolev-Yao intruder [8], a security protocol is not a sequential program (since the intruder, who is in complete control of the network, can freely interleave his actions with the normal protocol execution).

Contributions. In this paper, we address this problem and present an interpolation-based method for security protocol verification. Our method starts from the formal specification of a protocol and of a security property and combines Craig interpolation [7], symbolic execution [10] and the standard Dolev-Yao intruder model [8] to search for goals (representing attacks on the protocol). Interpolation is used to prune possible useless traces and speed up the exploration.

More specifically, our method proceeds as follows: starting (Sect. 3.1) from a specification of the input system, including protocol, property to be checked and a finite number of session instances (possibly generated automatically by using a preprocessor), it first creates a corresponding sequential non-deterministic program, in the form of a *control flow graph* (Sect. 3.2), according to a procedure that we have devised, and then defines a set of goals and searches for them by symbolically executing the program (Sect. 3.3). When a goal is reached, an attack trace is extracted from the constraints that the execution of the path has produced; such constraints represent conditions over parameters that allow one to reconstruct the attack trace found. When the search fails to reach a goal, a backtrack phase starts, during which the nodes of the graph are annotated (according to an adaptation of the algorithm defined in [13] for sequential programs) with formulas obtained by using Craig interpolation. Such formulas express conditions over the program variables, which, when implied from the program state of a given execution, ensure that no goal will be reached by going forward and thus that we can discard the current branch. The output of the method is a proof of (bounded) correctness in the case when no goal location can be reached starting from a finite-state specification; otherwise one or more attack traces are produced. We illustrate our method by means of a concrete example.

In Sect. 4, we briefly report on some experiments performed by using a prototype implementation. We summarize other characteristics of our method in the concluding remarks (Sect. 5), where we also discuss future work.

2 Background

Security protocols describe how agents exchange messages, built using cryptographic primitives, in order to obtain security guarantees. The algebra of messages tells us how messages are constructed. Following [5], we consider a countable *signature* Σ and a countable set *Var* of *variable symbols* disjoint from Σ, and then write Σ^n for the symbols of Σ with arity n; thus Σ^0 is the set of *constants*, which we assume to have distinct subsets that we refer to as *agent names* (or simply just *agents*), *public keys*, *private keys* and *nonces* (we omit symmetric keys from our treatment since we do not use them in our running example, but of course our method can fully support them). The variables are, however, untyped (unless denoted otherwise) and can be instantiated with arbitrary types, yielding an *untyped* model. We will use upper-case letters to denote variables (e.g., $A, B, \ldots$ to denote agents, N for nonces, etc.) and lower-case letters to denote the corresponding constants (concrete agents names, concrete nonces, etc.) All these may be possibly annotated with subscripts and superscripts.

The symbols of Σ with arity greater than zero are partitioned into the set Σ_p of *(public) operations* and the set Σ_m of *mappings*. The public operations represent all those operations that every agent (including the intruder) can perform on messages they know. In this paper, we consider the following operations: $\{M_1\}_{M_2}$ represents the *asymmetric encryption* of M_1 with public key M_2; $\{M_1\}_{inv(M_2)}$ represents the *asymmetric encryption* of M_1 with private key $inv(M_2)$ (the mapping $inv(\cdot)$ is discussed below); $[M_1, M_2]$ represents the concatenation of M_1 and M_2. For simplicity, we will often simply write M_1, M_2 instead of $[M_1, M_2]$.

In contrast to the public operations, the mappings of Σ_m do not correspond to operations that agents can perform on messages, but are rather mappings between constants. In this paper, we use the following ones: (i) $inv(M)$ gives the private key that corresponds to public key M; (ii) for long-term key infrastructures, we assume that every agent A has a public key $pk(A)$ and corresponding private key $inv(pk(A))$; thus $pk(\cdots)$ is a mapping from agents to public keys.

Since we will below also deal with terms that contain variables, let us call *atomic* all terms that are built from constants in Σ^0, variables in Var, and the mappings of Σ_m. The set $\mathcal{T}_\Sigma(Var)$ of all *terms* is the closure of the atomic terms under the operations of Σ_p. A *ground term* is a term without variables, where we denote the set of ground terms with $\mathcal{T}_\Sigma$. It is standard in formal verification of security protocols to interpret terms in the *free algebra*, i.e., every term is interpreted by itself and thus two terms are equal iff they are syntactically equal.

Our approach is independent of the actual strength of the intruder; here we consider the Dolev and Yao [8] model of an active intruder, denoted i, who controls the network but cannot break cryptography. In particular, i can intercept messages and analyze them if he knows the proper keys for decryption, and he can generate messages from his knowledge and send them under any agent name.

3 A Security Protocol Interpolation Method

The method we propose takes as input a protocol specification, together with a finite scenario of the protocol and one or more properties to be verified in that scenario. In the following, we give a recipe for producing a sequential program for the protocol scenario that we are considering, in the form of a control flow graph. The graph is enriched with locations required for handling the goals; in particular, for each property to be verified, a *goal location* is defined, and the

$$\begin{array}{lll}
A \to B : \{N_A, A\}_{pk(B)} & A \to B : \{N_A, A\}_{pk(B)} & A \to i : \{N_A, A\}_{pk(i)} \\
B \to A : \{N_A, N_B, B\}_{pk(A)} & B \to A : \{N_A, N_B\}_{pk(A)} & i(A) \to B : \{N_A, A\}_{pk(B)} \\
A \to B : \{N_B\}_{pk(B)} & A \to B : \{N_B\}_{pk(B)} & B \to i(A) : \{N_A, N_B\}_{pk(A)} \\
 & & i \to A : \{N_A, N_B\}_{pk(A)} \\
 & & A \to i : \{N_B\}_{pk(i)} \\
 & & i(A) \to B : \{N_B\}_{pk(B)}
\end{array}$$

Fig. 1. NSL message exchange (left), NSPK message exchange (middle) and Man-in-the-middle attack on NSPK (right)

verification task consists in checking whether any execution of the protocol can reach one or more of such locations. The exploration is performed by using the algorithm of [13], which proceeds by executing symbolically the program and exploits Craig interpolation in order to prune the search over the graph. In the case when a goal location is reached, an attack trace is extracted.

3.1 Input

Given a protocol $\mathcal{P}$ involving a set $\mathcal{R}$ of *roles* (*Alice*, *Bob*, ..., a.k.a. *entities*), a *session (instance) of* $\mathcal{P}$ is a function *si* assigning an agent (honest agent or the intruder i) to each element of $\mathcal{R}$. A *scenario of a protocol* $\mathcal{P}$ is a finite number of session instances of $\mathcal{P}$. The input of our method is then: (1) a specification of a protocol $\mathcal{P}$, (2) a scenario $\mathcal{S}$ of $\mathcal{P}$, (3) a set of goals (i.e., properties to be verified) in $\mathcal{S}$. For what concerns the definition of a scenario, we remark that when a role is assigned the agent i, it is intended to be played by the intruder, either under his real name i or pretending to be some other agent.

Example 1. As a running example, we will use NSL (Fig. 1, left), the Needham-Schroeder Public Key (NSPK) protocol with Lowe's fix [11], which aims at mutual authentication between A and B. The presence of B in the second message prevents the man-in-the-middle attack that NSPK suffers from (see Fig. 1, right, where $i(A)$ denotes that the intruder is impersonating the honest agent A).

As a formal specification language, we will use a subset of ASLan++ [1, 18]. In the following extract of the specifications for NSL, the two roles are *Alice*, who is the *initiator* of the protocol, and *Bob*, the *responder*.

```
1  entity Alice(Actor, B: agent) {
2   symbols
3     Na, Nb: text;
4   body{
5    Na := fresh();
6    Actor -> B: {Na,Actor}_pk(B);
7    B -> Actor: {Na,?Nb,B}_pk(Actor);
8    Actor -> B: {Nb}_pk(B);
9   }
10 }
```

```
11 entity Bob(A, Actor: agent) {
12  symbols
13    Na, Nb: text;
14  body{
15   ? -> Actor: {?Na,?A}_pk(Actor);
16   Nb := fresh();
17   Actor -> A: {Na,Nb,Actor}_pk(A);
18   A -> Actor: {Nb}_pk(Actor);
19  }
20 }
```

The elements between parentheses in line `1` declare which variables are used to denote the agents playing the different roles along the specification of the role *Alice*: `Actor` refers to the agent playing the role of *Alice* itself, while `B` is the variable referring to the agent who plays the role of *Bob*. Similarly, the section

$Alice_1.Actor \rightarrow Alice_1.B : \{Alice_1.Na, Alice_1.Actor\}_{pk(Alice_1.B)}$	$a \rightarrow i : \{c_1, a\}_{pk(i)}$
$? \rightarrow Bob_2.Actor : \{Bob_2.Na, Bob_2.A\}_{pk(Bob_2.Actor)}$	$i(a) \rightarrow b : \{c_1, a\}_{pk(b)}$
$Bob_2.Actor \rightarrow Bob_2.A : \{Bob_2.Na, Bob_2.Nb\}_{pk(Bob_2.A)}$	$b \rightarrow i(a) : \{c_1, c_2\}_{pk(i(a))}$
$Alice_1.B \rightarrow Alice_1.Actor : \{Alice_1.Na, Alice_1.Nb\}_{pk(Alice_1.Actor)}$	$i \rightarrow a : \{c_1, c_2\}_{pk(a)}$
$Alice_1.Actor \rightarrow Alice_1.B : \{Alice_1.Nb\}_{pk(Alice_1.B)}$	$a \rightarrow i : \{c_2\}_{pk(i)}$
$Bob_2.A \rightarrow Bob_2.Actor : \{Bob_2.Nb\}_{pk(Bob_2.Actor)}$	$i(a) \rightarrow b : \{c_2\}_{pk(b)}$

Fig. 2. Symbolic attack trace of man-in-the-middle-attack on NSPK (left) and instantiated attack trace (right) obtained with our method

`symbols` declares that `Na` and `Nb` are variables of type *text*. The section `body` specifies the behavior of the role. First, the operation `fresh()` assigns to the nonce `Na` a value that is different from the value assigned to any other nonce. Then *Alice* sends the nonce, together with her name, to the agent `B`, encrypted with `B`'s public key. In line `7`, *Alice* receives her nonce back together with a further variable (expected to represent `B`'s nonce along a regular session of the protocol) and the name of `B`, all encrypted with her own public key. The "`?`" in `?Nb` is used to represent an assignment of the value received to the variable `Nb`. As a last step, *Alice* sends to `B` the nonce `Nb` encrypted with `B`'s public key.

The variable declarations and the behavior of *Bob* are specified by lines `12-21`. We omit a full description of the code and only remark that the "`?`" in the beginning of line `16` denotes the fact that the sender of such a message can be any agent, though no assignment is made for `?` in that case. □

3.2 From a Protocol Specification to a Sequential Program

The algorithm of [13] is designed for sequential programs. In order to apply it to security protocols, we define a translation from the specification of a protocol $\mathcal{P}$ for a given scenario into a corresponding sequential non-deterministic program. Such a program will be encoded in a pseudo-language admitting the standard constructs for assignments and conditional statements, as well as a type Message.

3.2.1 Translating a Session Specification into a Sequential Program

We now describe how to obtain a program for a single session instance *si*; we will then consider more session instances in Sect. 3.2.3. First of all, note that the exchange of messages in a session follows a given flow of execution that can be used to determine an order between the instructions contained in the different roles. Such a sequence of instructions will constitute the skeleton of our program. However, we will omit from the sequence those instructions contained in a role that is played by the agent i, whose behavior will be treated differently.

We use as program variables the same names used in the specification. However, in order to distinguish between variables with the same name occurring in the specification of different roles, program variables have the form `E.V` where `E` denotes the role and `V` the variable name in the specification. An additional variable `IK`, of a type *MessageSet*, is used in the program to represent the intruder knowledge. Similarly, constants of the specification become program constants.

Whenever a session is played only by honest agents, the execution of the corresponding sequential program is univocally determined. The behavior of the intruder introduces a form of non-determinism, which we capture by representing the program, in the case when the intruder plays a role, as a procedure depending on a number of parameters, denoted by variables `Y`, possibly subscripted.

3.2.1.1 Initialization of the Variables. A first section of the program initializes the variables. For each role *Alice* such that $si(Alice) \neq i$, we have an instruction `Alice.Actor := a`, where a is an agent name such that $si(Alice) = a$. Whenever *Alice* is an initiator, for each responder *Bob* with `B` being the variable referring to the role *Bob* between the agent variables of *Alice*: if $si(Bob) \neq i$, then we have the assignment `Alice.B := b`, where b is such that $si(Bob) = b$, else we have `Alice.B := Y`, for `Y` an input variable not introduced elsewhere in the program.

Finally, we need to initialize the intruder knowledge. A typical `IK` initialization has the form: `IK := {a_1,...,a_n,i,pk(a_1),...,pk(a_n),pk(i),inv(pk( i))}`. That is, i knows the agents `a_j` involved in the scenario and their public keys `pk(a_j)`, as well as his own public and private keys `pk(i)` and `inv(pk(i))`. Specific protocols might require a specific initial `IK` or the initialization of further variables, depending on the context, such as symmetric keys. In our programs, we also allow a construct of the form `IK |- M` to denote that the intruder is able to construct the message `M` from its current intruder knowledge `IK` (i.e., derive it using its inference rules for generating and analyzing messages).

3.2.1.2 Sending and Receipt of a Message. The sending of a message `Actor -> B: M` defined in a role *Alice* is translated into the instruction `IK := IK + M`, where the symbol `+` denotes the addition of the message `M` to `IK`.

In order to define the receipt of a message `R -> Actor: M` in a role *Alice* from some *Bob* we distinguish two cases. If the message is sent by the intruder, i.e., $si(Bob) = i$, then the instruction is translated into the following code:

```
If (IK |- Alice.M)
  then Alice.Q_1 := Y_1; ... ; Alice.Q_n := Y_n;
else end
```

where `Q_1, ..., Q_n` are the variables occurring preceded by `?` in `R -> Actor : M` and `Y_1, ..., Y_n` are distinct input variables not introduced elsewhere.

If $si(Bob) \neq i$, then the receipt `R -> Actor: M` corresponds to, and within the flow of execution is immediately preceded by, a sending `Actor -> R': M'` in the specification of *Bob*, which matches `R -> Actor: M`. In this case, we translate the instruction into: `Alice.Q_1 := q_1; ...; Alice.Q_n := q_n` where `Q_1, ..., Q_n` are all the variables occurring preceded by `?` in `R -> Actor: M` and `q_1, ..., q_n` the expressions matching with `Q_1, ..., Q_n`, respectively, in `Actor -> R': M'`. For instance, the receipt `? -> Actor:{?Na,?A}_pk(Actor)` at line `15` in the specification of *Bob* in Example 1 corresponds to the sending `Actor -> B: {Na,Actor}_pk(B)` at line `6` in the specification of *Alice*. We can translate such a receipt into: `Bob.Na := Alice.Na; Bob.A := Alice.Actor`.

3.2.1.3 Generation of Fresh Values. An instruction of the form `N := fresh ()` in *Alice*, which assigns a fresh value to a nonce, can be translated into the instruction `Alice.N := c_1`, where `c_1` is a constant not introduced elsewhere.

Example 2. Fig. 3 shows the programs obtained for the two session instances of the NSL scenario we are interested in: in session 1, *Alice* and *Bob* are played by a and i respectively; in session 2, they are played by i and b, respectively. □

3.2.2 Introducing Goal Locations. The next step consists in decorating the program with a goal location for each security property to be verified. As it is common when performing symbolic execution [10], we express such properties as correctness assertions, typically placed at the end of a program. Once we have represented a protocol session as a program, and defined the properties we are interested in as correctness assertions in such a program, the problem of verifying security properties over (a session of) the protocol is reduced to verifying the correctness of the program with respect to those assertions.

We consider here three common security properties (authentication, confidentiality and integrity) and show how to represent them inside the program in terms of assertions. They are expressed by means of the statement `prove`, which in symbolic execution is commonly used to represent an output assertion required to evaluate to *true* in order to have the correctness of the program. Semantically, the instruction `prove(expr)` is equivalent to `if (not(expr))then error`.

3.2.2.1 Authentication. Assume we want to verify that *Alice* authenticates *Bob* with respect to a message `M` in the specification of the protocol, in a given session instance si. We can restrict our attention to the case when $si(Bob) = i$, since if *Bob* is played by an honest agent, then the authentication property is trivially satisfied. The problem thus reduces to verifying whether the agent i is playing under his real name (in which case authentication is again trivially satisfied) or whether i is pretending to be someone else, i.e., whether the agent playing *Alice* believes she is speaking to someone who is not i. Hence, we can simply add the assertion `prove(Alice.B = i)`, where `B` is the agent variable referring to the role *Bob* inside *Alice*, immediately after the receipt of the message `M`.

Example 3. In NSL, we are interested in verifying a property of authentication in the session that assigns i to *Alice* and b to *Bob*: we want *Bob* to authenticate *Alice* with respect to the nonce `Bob.Nb` in the receipt of line `2.14` (Fig. 3). Since the statement corresponding to such a receipt is the last instruction of the program, we can just add the instruction `prove (Bob.A = i)` at the end. □

3.2.2.2 Confidentiality. Assume that we want to verify that the message corresponding to a variable `M`, in the specification of a role *Alice* of the protocol, is confidential between a given set of roles $\mathcal{R}$ in a session si. As we did for authentication, since we are in an instantiated scenario, we ignore the case when the session is played only by honest agents, in which case confidentiality is preserved. In general, we can restrict to checking whether the agent i got to know the confidential message `M` even though i is not included in $\mathcal{R}$. Inside the program, this corresponds to checking whether the message `Alice.M` can be derived from the intruder knowledge and whether any honest agent playing a role in $\mathcal{R}$ believes that at least one of the other roles in $\mathcal{R}$ is indeed played by i, which we can read as having indeed $i \in \mathcal{R}$. This corresponds to the following assertion, to be added at the end of the program:

```
prove ((not(IK |- Alice.M) or
        (Alice_1.B^1_1 = i) or ... (Alice_1.B^1_m = i) or ...
        (Alice_n.B^n_1 = i) or ... (Alice_n.B^n_m = i))
```

where $Alice_j$, for $1 \leq j \leq n$, is a role such that $Alice_j \in \mathcal{R}$ and $si(Alice_j) \neq i$, $\{Bob_1, \ldots, Bob_m\} \subseteq \mathcal{R}$ is the subset of those roles in $\mathcal{R}$ that are instantiated with i by si and `B^j_l`, for $1 \leq j \leq n$ and $1 \leq l \leq m$, is the variable referring to the role Bob_l in the specification of the role $Alice_j$.

Example 4. For NSL, assume that we want to verify the confidentiality of the variable `Nb` (contained in the specification of *Bob*) between the roles in the set $\{Alice, Bob\}$. We can express this goal by appending at the end of the program the assertion `prove ((not(IK |- Bob.Nb))or (Bob.A = i))`. □

3.2.2.3 Integrity. In this case, we assume that two variables (possibly of two different roles) are specified in input as the variables containing the value whose integrity needs to be checked. The check will consist in verifying whether the two variables, at a given point of the session execution, also given in input, evaluate to the same. Let `M` in the role *Alice* and `M'` in the role *Bob* be the two variables; then the corresponding correctness assertion will be `prove(Alice.M = Bob.M')`.

3.2.3 Combining More Sessions. Now we need to define a program that properly "combines" the programs related to all the sessions in the scenario. The idea is that such a program allows for executing, in the proper order, all the instructions of all the sessions in the scenario; the way in which instructions of different sessions are interleaved will be determined by the value of further input variables, denoted by `X`, which can be seen as choices of the intruder with respect to the flow of the execution. Namely, we start to execute each session sequentially and we get blocked when we encounter the receipt of a message sent by a role that is played by the intruder. When all the sessions are blocked on instructions of that form, the intruder chooses which session has to be reactivated.

In the following, we will see a sequential program as a graph (which can be simply obtained by representing its control flow) on which the algorithm of Sect. 3.3 will be executed. We adapt from [13] some notions concerning programs and program runs. A *program graph* is a finite, rooted, labeled graph (Λ, l_0, Δ) where Λ is a *finite set of program locations*, l_0 is the *initial location* and $\Delta \subseteq \Lambda \times \mathcal{A} \times \Lambda$ is a *set of transitions* labeled by actions from a set $\mathcal{A}$, consisting in the instructions of the program. A *program path* of length k is a sequence of the form $l_0, a_0, l_1, a_1, \ldots, l_k$, where each *step* $(l_j, a_j, l_{j+1}) \in \Delta$ for $0 \leq j < k - 1$. The set $\mathbb{D}$ of *data states* is the set of all the maps $V \to D$ from the set V of program variables to the set D of possible data values, i.e., integers (for variables of the form X_i), ground messages (for variables denoting messages) or sets of ground messages (for the special variable IK). The *semantics* $Sem(a)$ *of an action* $a \in \mathcal{A}$ is a subset of $\mathbb{D} \times \mathbb{D}$. We assume an initial data state d_0. A *program run* of length k is a pair (π, σ), where π is a program path $l_0, a_0, l_1, a_1, \ldots, l_k$ and $\sigma = d_0, \ldots, d_k$ is a sequence of data states such that $(d_j, d_{j+1}) \in Sem(a_j)$ for $0 \leq j < k$. A *state* is a pair (l, d) such that $l \in \Lambda$ and $d \in \mathbb{D}$.

We have seen in Sects. 3.2.1 and 3.2.2 how to generate the program, and thus the corresponding control flow graph of a single session. The program graph corresponding to a whole scenario can be obtained by composing the graphs of the single sessions. Given a program graph, an *intruder location* is a location of the graph corresponding to the receipt of a message sent from a role played by i. A *block* $\mathcal{B}$ *of a program graph* $\mathcal{G}'$ is a subgraph of $\mathcal{G}'$ such that its initial location is either the initial location of $\mathcal{G}'$ or an intruder location. The *exit locations of a block* $\mathcal{B}$ are the locations of $\mathcal{B}$ with no outgoing edges. Intuitively, we proceed by decomposing a session program graph $\mathcal{G}^i$ into a sequence of blocks starting at each intruder location. The idea is that each such a block will occur as a subgraph in the general scenario graph $\mathcal{G}$ (possibly with more than one occurrence). Namely, each path of the resulting graph will contain all the blocks of the scenario just once, and the set of all paths will cover all the possible sequences that respect the order of the single sessions. For instance, given the block structures $(\mathcal{B}_1^1, \mathcal{B}_2^1)$ and $(\mathcal{B}_1^2)$, the resulting graph will contain a path corresponding to the execution of $\mathcal{B}_1^1, \mathcal{B}_2^1, \mathcal{B}_1^2$ in this order, as well as a path for $\mathcal{B}_1^1, \mathcal{B}_1^2, \mathcal{B}_2^1$, as well as a path for $\mathcal{B}_1^2, \mathcal{B}_1^1, \mathcal{B}_2^1$. A simple algorithm for automatically performing this composition has been devised; we omit it due to lack of space.

Example 5. Fig. 3 shows the program graph for the scenario consisting of the session instances si_1 and si_2 such that $si_1(Alice) = a$, $si_1(Bob) = i = si_2(Alice)$ and $si_2(Bob) = b$ for NSL. Note that the set of instructions concerning a block are grouped into a single edge (and the corresponding lines of code in the programs of Example 2 are used to label the edge in the figure). For clarity, the initialization section and the goal assertions are reported on separate edges, though they belong to a larger block. Note also that, for clarity, variable names are subscripted with the number of the session where they occur, e.g., a variable `Alice.B` occurring in the program of si_2 is renamed as `Alice_2.B`. □

3.3 Algorithm for Symbolic Execution and Annotation

In this section, we recall the IntraLA algorithm of [13] and describe how we can calculate interpolants in our case. The algorithm executes symbolically a program graph searching for goal locations, which represent attacks. In the case when we fail to reach a goal, an annotation (i.e., a formula expressing a condition under which no goal can be reached) is produced by using Craig interpolation. Through a backtrack phase, such an annotation is propagated to the other nodes of the graph and can be used to block a later phase of symbolic execution along an uninteresting run, i.e., a run for which the information contained in the annotation allows one to foresee that it will not reach a goal.

```
1.1   Alice.Actor := a;
1.2   Alice.B := Y_1;
1.3   IK := {a,b,i,pk(a),pk(b),pk(i),inv(pk(i))};
1.4
1.5   Alice.Na := c_1;
1.6   IK := IK + {Alice.Na,Alice.Actor}_pk(Alice.B
           );
1.7
1.8   if (IK |- {Alice.Na,?Alice.Nb,Alice.B}_pk(
           Alice.Actor))
1.9     then
1.10      Alice.Nb = Y_2;
1.11    else
1.12      end
1.13
1.14  IK := IK + {Alice.Nb}_pk(Alice.B);
```

```
2.1   Bob.Actor := b;
2.2   IK := {a,b,i,pk(a),pk(b),pk(i),inv(pk(i))};
2.3
2.4   if (IK |- {?Bob.Na,?Bob.A}_pk(Bob.Actor))
2.5     then
2.6       Bob.Na = Y_1;
2.7       Bob.A = Y_2;
2.8     else
2.9       end
2.10
2.11  Bob.Nb := c_1;
2.12  IK := IK + {Bob.Na,Bob.Nb,Bob.Actor}_pk(Bob.
           A);
2.13
2.14  if (IK |- {Bob.Nb}_pk(Bob.Actor))
2.15    then
2.16      do nothing
2.17    else
2.18      end
```

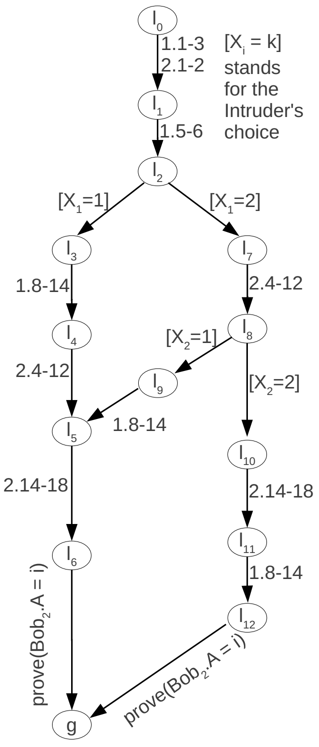

Fig. 3. NSL example: program for session si_1 (top-left), program for session si_2 (bottom-left), control flow graph for the whole scenario (right)

We will use a *two-sorted first-order language with equality*. The first sort is based on the algebra of messages, over which we allow a set of unary predicates $\mathcal{DY}^j_{IK}$ for $1 \leq j \leq n$ with a fixed $n \in \mathbb{N}$, whose meaning will be clarified below. The second sort is based on a signature containing variables (denoted in our examples by `Xi`) and uninterpreted constants (for which we use integers), and allows no functions and no predicates other than equality. We assume fixed the sets of constants and denote by $\mathcal{L}(\mathcal{V})$ the *set of well-formed formulas* of such a two-sorted first-order language defined over a (also two-sorted) set $\mathcal{V}$ of variables, to be instantiated with the variables and parameters of our programs.

First, we introduce some notions concerning symbolic execution. Let V be the set of program variables (for which, in the following, we will use standard math fonts). A *symbolic data state* is a triple (P, C, E), where P is a (two-sorted) *set of parameters*, i.e., variables not in V, $C \in \mathcal{L}(P)$ is a *constraint over the parameters*, and the *environment* E is a map from the program variables V to terms of the corresponding sort defined over P, with the only exception of the variable IK, which is mapped instead to a set of message terms. We denote by S the *set of symbolic data states*. Given its definition, a symbolic data state s can be characterized by the predicate $\chi(s) = C \wedge (\bigwedge_{v \in V \setminus \{IK\}}(v = E(v))) \wedge (\bigwedge_{M \in E(IK)} \mathcal{DY}^0_{IK}(M))$. Note that the variable IK is treated in a particular way, i.e., we translate the fact that $E(IK) = \mathcal{M}$ for some set $\mathcal{M}$ of parametric messages into a formula expressing that a predicate $\mathcal{DY}^0_{IK}$ holds for the messages in $\mathcal{M}$. A symbolic data state s can be associated to the set $\gamma(s)$ of data states produced by the map E for some valuation of the parameters satisfying the constraint C. We assume a defined initial symbolic data state $\gamma(s_0) = \{d_0\}$. A *symbolic state* is a pair $(l, s) \in \Lambda \times S$. A *symbolic interpreter* SI is a total map from the set $\mathcal{A}$ of actions to $S \times S$ such that for each symbolic data state s and action a, $\cup\gamma(SI(a)(s)) = Sem(a)(\gamma(s))$. Intuitively, SI takes a symbolic data state s and an action a and returns a non-empty set of symbolic data states, which represent the set of states obtained by executing a on s.

The *algorithm state* is a triple (Q, A, G) where Q is the set of *queries*, A is a *(program) annotation* and $G \subseteq \Lambda$ is the set of *goal locations* that have not been reached. A query is a symbolic state. During the execution of the algorithm, the set of queries is used to keep track of which symbolic states still need to be considered, i.e., of those symbolic states whose location has at least one outgoing edge that has not been symbolically executed, and the annotation is a decoration of the graph used to prune the search. Formally, a program annotation is a set of pairs in $(\Lambda \cup \Delta) \times \mathcal{L}(V)$. We will write these pairs in the form $l : \phi$ or $e : \phi$, where l is a location, e is an edge and ϕ is a formula called the *label*. When we have more than one label on a given location, we can read them as a disjunction of conditions: we define $A(l) = \bigvee\{\phi \mid l : \phi \in A\}$. For an edge $e = (l_n, a, l_{n+1})$ the label $e : \phi$ is *justified* in A if starting from the precondition formula ϕ and by executing the action a, the postcondition produced is $A(l_{n+1})$, i.e., when it implies the annotation of l_{n+1} after executing a. In that case, we write $\mathcal{J}(e : \phi, A)$. Let $Out(l)$ be the set of outgoing edges from a location l; the label $l : \phi$ is justified in A when, for all edges $e \in Out(l)$, there exists $e : \psi \in A$ such that ψ is a logical consequence of ϕ. An annotation is justified when all its elements are justified. A justified annotation is inductive and if it is initially true, then it is an inductive invariant. The algorithm maintains the invariant that A is always justified. A query $q = (l, s)$ is *blocked* by a formula ϕ when $s \models \phi$ and we then write $Bloc(q, A(\phi))$. With respect to q, the edge e is blocked when $Bloc(q, A(e))$ and the location l is blocked when $Bloc(q, A(l))$.

The rules of the algorithm IntraLA are given in Fig. 4. First, we initialize the algorithm state to $(\{(l_0, s_0)\}, \emptyset, G_0)$, i.e. the algorithm starts from the initial location, the initial symbolic data state, an empty annotation and a set G_0 of goals to search for, which is given as input.

The *Decide* rule is used to perform symbolic execution. By symbolically executing one program action, it generates a new query from an existing one. It may choose any edge that is not blocked and any symbolic successor state generated by the action a. If the generated query is itself not blocked, it is added to the query set. In the rule, SI is a symbolic interpreter, l_n and s_n are the currently considered location and symbolic data state, respectively, and l_{n+1} and s_{n+1} the location and symbolic data state obtained after executing a. The side conditions

$$\frac{Q, A, G}{Q + (l_{n+1}, s_{n+1}), A, G}\ \mathit{Decide} \qquad \frac{Q, A, G}{Q, A + e : \phi, G}\ \mathit{Learn} \qquad \frac{Q, A, G}{Q\ - q, A + l_n : \phi, G - l_n}\ \mathit{Conjoin}$$

$$\begin{array}{r} q = (l_n, s_n) \in Q \\ e = (l_n, a, l_{n+1}) \in \Delta \\ \neg Bloc(q, A(e)) \\ s_{n+1} \in SI(a)(s_n) \\ \neg Bloc((l_{n+1}, s_{n+1}), A(l_{n+1})) \end{array} \qquad \begin{array}{r} q = (l_n, s_n) \in Q \\ e = (l_n, a, l_{n+1}) \in \Delta \\ Bloc(q, \phi) \\ \mathcal{J}(e : \phi, A) \end{array} \qquad \begin{array}{r} q = (l_n, s) \in Q \\ \neg Bloc(q, A(l_n)) \\ (\,\forall e \in Out(l_n). \qquad\qquad \\ e : \phi_e \in A \wedge Bloc(q, \phi_e)\,) \\ \phi = \bigwedge\{\phi_e \mid e \in Out(l_n)\} \end{array}$$

Fig. 4. Rules of the algorithm IntraLA with corresponding side conditions

of the *Decide* rule are that moving from s_n to s_{n+1}, the first needs to be into the query set and the branch between the two nodes must exist and not be blocked.

During the backtrack phase, two rules are used: *Learn* generates annotations and *Conjoin* merges annotations coming from different branches. If some outgoing edge $e = (l_n, a, l_{n+1})$ is not blocked, but every possible symbolic step along that edge leads to a blocked state, then the rule infers a new label ϕ that blocks the edge, where the formula ϕ can be any formula ϕ that both blocks the current query and is justified. In the following, we will explain how it can be obtained by exploiting the Craig interpolation lemma [7], which states that given two first-order formulas α and β such that $\alpha \wedge \beta$ is inconsistent, there exists a formula γ (their *interpolant*) such that α implies γ, γ implies $\neg\beta$ and $\gamma \in \mathcal{L}(\alpha) \cap \mathcal{L}(\beta)$.

Let μ be a term, a formula, or a set of terms or of formulas. We write μ' for the result of adding one prime to all the non-logical symbols in μ. Intuitively, the prime is used to refer to the value of a same variable in a later step and it is used in *transition formulas*, i.e., formulas in $\mathcal{L}(V \cup V')$. Since the semantics of an action $Sem(a)$ expresses how we move from a data state to another, we can easily associate to $Sem(a)$ a transition formula. With a slight abuse of notation, in the following, we will use $Sem(a)$ to denote the corresponding transition formula.

In our context, the most interesting case is when the action a is represented by a conditional statement, with a condition of the form $IK \vdash M$ for some message M. The intuitive meaning of the statement $IK \vdash M$ is that the message M can be derived from a set of messages denoted by IK by using the standard Dolev Yao intruder inference power. In our treatment, we fix a value n as the maximum number of inference steps that the intruder can execute in order to derive M. We observe that this is not a serious limitation of our method since several results (e.g., [17]) show that, when the number of sessions is finite, as in our case, it is possible to set an upper bound on the number of inference steps needed. Such a value can be established a-priori by observing the set of messages exchanged in the protocol scenario; we assume such an n to be fixed for the whole scenario. We use formulas of the form $\mathcal{DY}^j_{IK}(M)$, for $0 \leq j \leq n$, with the intended meaning that M can be derived in j steps of inference by the intruder. In particular, the predicate $\mathcal{DY}^0_{IK}$ is used to represent the initial knowledge IK, before any inference step is performed. Under the assumption on the n mentioned above, the statement $IK \vdash M$ can be expressed as the formula $\mathcal{DY}_{IK}^{n}(M)$. The formula

$$\begin{aligned}\varphi_j = \forall M.\, (&\mathcal{DY}_{IK}^{j+1}(M) \leftrightarrow (\mathcal{DY}_{IK}^{j}(M) \vee (\exists M'.\, \mathcal{DY}_{IK}^{j}([M,M']) \vee \mathcal{DY}_{IK}^{j}([M',M]))\\ &\vee (\exists M_1,M_2.\, M=[M_1,M_2] \wedge \mathcal{DY}_{IK}^{j}(M_1) \wedge \mathcal{DY}_{IK}^{j}(M_2))\\ &\vee (\exists M_1,M_2.\, M=\{M_1\}_{M_2} \wedge \mathcal{DY}_{IK}^{j}(M_1) \wedge \mathcal{DY}_{IK}^{j}(M_2)))\\ &\vee (\exists M'.\, \mathcal{DY}_{IK}^{j}(\{M\}_{M'}) \wedge \mathcal{DY}_{IK}^{j}(inv(M'))) \vee (\exists M'.\, \mathcal{DY}_{IK}^{j}(\{M\}_{inv(M')}) \wedge \mathcal{DY}_{IK}^{j}(M'))\,,\end{aligned}$$

in which $\leftrightarrow$ denotes the double implication and each quantification has to be intended over the sort of messages, expresses (as a disjunction) all the ways in which a given message can be inferred by the intruder in one step, i.e. by an operation of analysis or construction, thus moving from a knowledge (denoted by the predicate) $\mathcal{DY}_{IK}^{j}$ to a knowledge (denoted by the predicate) $\mathcal{DY}_{IK}^{j+1}$.

A theory $\mathcal{T}_{Msg}(n)$ over the sort of messages is obtained by enriching classical first-order logic with equality with the axioms φ_j, for $1 \leq j < n$, together with additional axioms formalizing that any two distinct ground terms are not equal.

Now let $\alpha = \chi(s_n)$ and $\beta = Sem(a) \wedge \neg A(l_{n+1})'$. We can obtain the formula ϕ we are looking for, in the rule *Learn*, as an interpolant for α and β, possibly by using an interpolating theorem prover. With regard to this, we observe that, in the presence of our finite scenario assumption, when mechanizing such a search, the problem can be simplified by restricting the domain to a finite set of messages.

Finally, the rule *Conjoin* is applied when all the outgoing edges of the location in a query q are blocked. The location in q is labeled with the conjunction of the labels that block the outgoing edges. If the location is a goal, then we remove it from the set of remaining goals. Finally, the query is discarded from Q.

The algorithm terminates when no rules can be applied. In [13], the correctness of the algorithm, with respect to the goal search, is proved: the proof given there applies straightforwardly to the slightly simplified version we have given.

Theorem 1. *Let G_0 be the set of goal locations provided in input. If the algorithm terminates with the algorithm state (Q, A, G), then all the locations in $G_0 \setminus G$ are reachable and all the locations in G are unreachable.*

The output of our method can be of two types. If no goal has been reached, then we have a proof that no attack can be found, with respect to the security property of interest, in the finite scenario that we are considering. Otherwise, for each goal location that has been found, we can generate a test case, in the form of an attack trace, which can be easily inferred from the information in the symbolic data state corresponding to the last step of execution. We also note that, by a trivial modification of the rule *Conjoin*, we might easily obtain an algorithm that keeps searching for a goal that has already been reached through a different path, thus allowing to extract more attack traces for the same goal.

Example 6. Here we show the execution of the algorithm on the NSL graph of Fig. 3: Fig 5 summarizes the algorithm execution. Note that in the table, we use statements of the form $IK \vdash M$ in the constraint set as an abbreviation

N	**Rule**	**Edge**	A	C	E
0	*Init*	-	$\emptyset$	$\emptyset$	$\emptyset$
1	*Decide*	(l_0, l_1)	$\emptyset$	C_0	$E_0 \oplus \{(Alice_1.Actor, a), (Y_1.y_1), (Alice_1.Bob, y_1),$ $(Bob_2.Actor, b),$ $(IK, \{a, b, i, pk(a), pk(b), pk(i), inv(pk(i))\})\}$
2	*Decide*	(l_1, l_2)	$\emptyset$	C_1	$E_1 \oplus \{(Alice_1.Na, c_1),$ $(IK, IK_1 \cup \{c_1, a\}_{pk(y_1)})\}$
3	*Decide*	(l_2, l_3)	$\emptyset$	$C_2 \cup \{(x_1 = 1)\}$	$E_2 \oplus \{(X_1, x_1)\}$
4	*Decide*	(l_3, l_4)	$\emptyset$	$C_3 \cup \{IK_2 \vdash$ $\{c_1, y_2, y_1\}_{pk(a)}\}$	$E_3 \oplus \{(Alice_1.Nb, y_2),$ $(IK, IK_2 \cup \{y_2\}_{pk(y_1)})\}$
5	*Decide*	(l_4, l_5)	$\emptyset$	$C_4 \cup \{IK_4 \vdash$ $\{y_4, y_3\}_{pk(b)}\}$	$E_4 \oplus \{(Y_3, y_3), (Bob_2.A, y_3), (Y_4, y_4), (Bob_2.Na, y_4),$ $(Bob_2.Nb, c_2), (IK, IK_4 \cup \{y_4, c_2, b\}_{pk(y_3)})$
6	*Decide*	(l_5, l_6)	$\emptyset$	$C_5 \cup \{IK_5 \vdash \{c_2\}_{pk(b)}\}$	E_5
7	*Learn*	-	$\{(l_6, g) : Bob_2.A = i\}$	C_6	E_6
8	*Conjoin*	(l_6, g)	$A_7 \cup \{l_6 : Bob_2.A = i\}$	C_7	E_7
9	*Learn*	-	$A_8 \cup \{(l_5, l_6) : Bob_2.A = i \vee C_V\}$	C_8	E_8
10	*Conjoin*	(l_5, l_6)	$A_9 \cup \{l_5 : Bob_2.A = i \vee C_V\}$	C_9	E_9
11	*Decide*	(l_2, l_7)	A_{10}	$\{(x_1 = 2)\}$	$E_2 \oplus \{(X_1, x_1)\}$
12	*Decide*	(l_7, l_8)	A_{10}	$C_{11} \cup \{IK_2 \vdash$ $\{y_4, y_3\}_{pk(b)}\}$	$E_2 \oplus \{(Y_3, y_3), (Bob_2.A, y_3), (Y_4, y_4), (Bob_2.Na, y_4),$ $(Bob_2.Nb, c_2), (IK, IK_2 \cup \{y_4, c_2, b\}_{pk(y_3)})\}$
13	*Decide*	(l_8, l_9)	A_{10}	$C_{12} \cup \{(x_2 = 1)\}$	$E_{12} \oplus \{(X_2, x_2)\}$
14		(l_9, l_5)	A_{10}	C_{13}	E_{13}

In step 9, $C_V \in \mathcal{L}(V)$ is a constraint over V s.t. C_V entails $IK_5 \nvdash \{Bob_2.Nb\}_{pk(Bob2.Actor)}$

Fig. 5. Execution of the algorithm on the control flow graph for NSL

for the set of constraints over the parameters that make the (translation of the) statement satisfiable. Further, P_i, C_i and E_i denote, respectively, the set of parameters, the set of constraints and the environment at step i of the execution.

After the initialization, symbolic execution steps are performed from query (l_0, s_0) to (l_5, s_6) by using the rule *Decide* (steps 1–6). In step 7, we note that any symbolic execution step through the edge (l_6, g), leads to a blocked query. The algorithm thus creates interpolants and propagates them back to l_5 (steps $7-10$), where the symbolic execution restarts, via applications of *Decide*, until step 14. Again, any symbolic step on the query (l_9, s_{13}) along the edge (l_9, l_5) leads to a blocked query, i.e., it generates a symbolic state that entails the annotation $Bob_2.A = i \vee C_V$. This is a concrete example of how the annotation method can improve the search procedure: we can stop following the path of query (l_9, s_{13}) as the annotation ensures we will never reach a goal.

By applying the method to NSPK, instead, we reach the goal with an execution close to the one seen for NSL. In fact, in the corresponding of step 14, we have that the inequality $Bob_2.A \neq i$ does not make the constraint set unsatisfiable. To extract an attack trace, first we consider the values of the x_j parameters contained in the last constraint set, i.e., $\{x_1 = 2, x_2 = 1\}$, which express the order in which the two sessions are interleaved, thus obtaining a symbolic attack trace (Fig. 2, left). We can further instantiate this trace, by using parameter and constant values of the last symbolic data state, thus obtaining the instantiated attack trace (Fig. 2, right). In particular, we note that y_3 is not constrained to be equal to i; this allows the intruder to act as pretending to be the honest agent a in the second session, from which we get the man-in-the-middle-attack. $\square$

4 Experiments and Results

We have implemented a Java prototype called SPiM (Security Protocol interpolation Method) based on Z3 [16] and iZ3 [14] for satisfiability checking and interpolant generation, respectively. We use a modified version of the algorithm in [13], where we propagate annotations only if they can be effectively used to stop the execution of some other path (i.e., during the backtracking we only annotate locations and edges that can be reached by some path not visited yet).

In order to show that the method concretely speeds up the validation, we have tested SPiM with and without the interpolation part (consisting of the rules Learn and Conjoin) on NSL and NSPK. The total execution time on a general purpose computer ranges from 8s for NSPK to 83s for NSL. While for NSPK there are no pruned paths and consequently the two versions of the algorithm perform with the same time, on NSL SPiM is 1.5-3.5% (depending on the quality of the computer used) faster when using interpolation. This experiment shows that, even on examples where the annotation method does not prune the search space considerably (in NSL we only save two steps of symbolic execution), the time of validation tends to decrease when using interpolant-based annotations. This is also confirmed by the fact that, as observed during the execution on the NSL example, the average time needed to calculate and propagate an interpolant is 9.1-27.3% lower than the average time used to perform a step of symbolic execution together with the corresponding satisfiability checking.

5 Concluding Remarks

We have presented a method that starts from a formal security protocol specification and combines Craig interpolation (to prune useless traces so as to avoid a quantifier elimination phase that is usually an expensive task, cf. [13]), symbolic execution and the standard Dolev-Yao intruder model to search for goals, i.e., possible attacks on the protocol. In particular, our method adopts (almost verbatim) the IntraLA algorithm proposed by McMillan in [13]. Other approaches have similarly benefited from IntraLA, e.g., it has been integrated in the BLAST tool [9], but our results are different from theirs in terms of both the application field and the methodology we have used to perform the analysis. In fact, one of the main differences between our work and [9, 13], is the way we construct the control flow graph, in particular to accommodate the fact that security protocols are not sequential programs when we analyze them in the presence of a Dolev-Yao intruder. For this, we have taken inspiration from protocol analysis tools such as the AVANTSSAR Platform [1], from which we have lifted the input specification language and the formalization of the intruder actions.

Given its prototypical nature, some aspects of our method require further work. For instance, the full automation of the generation of control flow graphs and the handling of infinite scenarios will allow us to compare with other security protocol verification tools [1, 2], with which we also expect useful interaction.

We are currently working at extending the procedure for translating protocols into sequential programs in order to cover all the constructs of the ASLan++

language, thus enabling the application of our method to more complex security protocols, as well as at giving a formal proof of the correctness of such a translation. We also aim to extend the method with the possibility of expressing protocol goals as LTL properties (like in AVANTSSAR) as we would like to use Craig interpolation not only to prune the search space but also to check which of the possible reachable states can or can not lead to the intended goal.

References

1. Armando, A., et al.: The AVANTSSAR Platform for the Automated Validation of Trust and Security of Service-Oriented Architectures. In *TACAS*, LNCS 7214:267–282. Springer, 2012.
2. Armando, A., et al.: The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In *CAV*, LNCS 3576:281–285. Springer, 2005.
3. Armando, A., Carbone, R., Compagna, L., Cuéllar, J., Tobarra Abad, L.: Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-based Single Sign-On for Google Apps. In *FMSE*. ACM, 2008.
4. Armando, A., Pellegrino, G., Carbone, R., Merlo, Balzarotti, D.: From Model-Checking to Automated Testing of Security Protocols: Bridging the Gap. In *TAP*, LNCS 7305:3–18. Springer, 2012.
5. Basin, D., Mödersheim, S., Viganò, L.: OFMC: A symbolic model checker for security protocols. *Int. Journal of Information Security*, 4(3):181–208, 2005.
6. Büchler, M., Oudinet, J., Pretschner, A.: Security mutants for property-based testing. In *TAP*, LNCS 6706:69–77. Springer, 2011.
7. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(3):pp. 269–285, 1957.
8. Dolev, D., Yao, A.: On the Security of Public-Key Protocols. *IEEE Transactions on Information Theory*, 2(29), 1983.
9. Henzinger, T. A., Jhala, R., Majumdar, R., McMillan, K. L.: Abstractions from proofs. In *POPL*, pp. 232–244. ACM, 2004.
10. King, J. C.: Symbolic execution and program testing. *CACM*, 19(7):385–394, 1976.
11. Lowe, G.: Breaking and Fixing the Needham-Shroeder Public-Key Protocol Using FDR. In *TACAS*, LNCS 1055:147–166. Springer, 1996.
12. McMillan, K. L.: Applications of Craig Interpolants in Model Checking. In *TACAS*, LNCS 3440:1–12. Springer, 2005.
13. McMillan, K. L.: Lazy annotation for program testing and verification. In *CAV*, LNCS 6174:104–118. Springer, 2010.
14. McMillan, K. L.: Interpolants from Z3 proofs. In *FMCAD*, pp. 19–27, 2011.
15. Mitchell, J. C., Mitchell, M., Stern, U.: Automated analysis of cryptographic protocols using Murphi. In *Security and Privacy*, pp. 141–151. IEEE CS, 1997.
16. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In *TACAS*, LNCS 4963:337–340. Springer, 2008.
17. Rusinowitch, M., Turuani, M.: Protocol insecurity with a finite number of sessions and composed keys is NP-complete. *TCS*, 299:451–475, 2003.
18. von Oheimb, D., Mödersheim, S.: ASLan++ — a formal security specification language for distributed systems. In *FMCO*, LNCS 6957:1–22. Springer, 2010.