

University of Verona

DEPARTMENT OF COMPUTER SCIENCE
GRADUATE SCHOOL OF SCIENCE AND ENGINEERING
DOCTORAL PROGRAM IN COMPUTER SCIENCE
S.S.D. INF/01

DOCTORAL THESIS

**Methods and tools for design time and runtime formal analysis
of security protocols and web applications**

Doctoral Student:
Marco Rocchetto

Tutor:
Prof. Luca Viganò

Coordinator:
Prof. Paolo Fiorini

Series N°: **TD-06-15**

Università di Verona
Dipartimento di Informatica
Strada le Grazie 15, 37134 Verona
Italy

To my family

Abstract. The increasing complexity of web applications together with their underlying security protocols has rapidly increased the need of automatic security analysis methods and tools. In this thesis, I address this problem by proposing two new formal approaches for the security verification of security protocols and web applications.

The first is a design time interpolation-based method for security protocol verification. This method starts from a 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.

The second is a runtime (model-based) security analysis technique that searches for logic and implementation flaws on concrete (web-based) systems. In particular, I focused on how to use the Dolev-Yao intruder model to search for Cross-Site Request Forgery (CSRF) attacks. CSRF is listed in the top ten list of the Open Web Application Security Project (OWASP) as one of the most critical threats to web security.

Acknowledgments. I would like to thank my supervisor, Luca Viganò, for encouraging my research and for allowing me to grow as a research scientist. Your advice on both research as well as my career have been priceless.

I wish to express my special appreciation and thanks to Marco Volpe, the best help I could have hoped for.

I would especially like to thank the REGIS group as well as both the AVANTSSAR and SPaCioS members for all the inspiring conversations and suggestions.

I also really benefited from spending three very stimulating months at ETH Zurich, for which I thank Mohammad Torabi Dashti, David Basin and all the “Information Security Group”.

A special thanks to my family and my friends for your constant support.

Contents

Part I: Thesis overview

1	Introduction	3
1.1	Main Contributions	4
1.1.1	Design time verification of security protocols	4
1.1.2	Run time security analysis	5
2	About this thesis	7
2.1	Collaborations and publications related to this thesis	7
2.2	Synopsis	8

Part II: State of the art

3	Design time techniques	13
3.1	Model checking for security protocols	13
3.2	ASLan and ASLan++	14
3.3	The AVANTSSAR platform	18
3.3.1	The connectors layer	19
3.3.2	The validator	20
4	Run time techniques	23
4.1	Testing	23
4.1.1	Testing techniques	23
4.1.2	Testing tools	24
4.2	Model-based testing	26
4.2.1	Test case generation	27
4.3	The SPaCioS tool	28

Part III: Design time verification of security protocols

5	Introduction	33
----------	---------------------------	----

6	Background	35
6.1	Messages	35
6.2	The Dolev-Yao intruder model	37
6.3	Running example	38
7	An interpolation-based method for the verification of security protocols	43
7.1	Translating security protocols into sequential programs	44
7.1.1	The translation procedure	46
7.1.2	Combining more sessions	51
7.1.3	Correctness of the translation	55
7.2	Interpolation algorithm	67
8	The SPiM tool	85
8.1	The architecture of SPiM	86
8.1.1	The package VE	86
8.1.2	The ASLan++2Sil package	88
8.1.3	How to use SPiM	92
9	Experiments and results	93
10	Discussion and related work	97
<hr/>		
Part IV: Runtime security verification of web applications		
<hr/>		
11	Introduction	101
12	Dolev-Yao intruder and web applications	105
13	Cross-Site Request Forgery	111
13.1	CSRF protections	112
13.1.1	Client confirmation	112
13.1.2	Secret validation token	113
13.1.3	Combining protections	113
14	CSRF in ASLan++: modeling and detection	115
14.1	DocumentRepository description	115
14.2	ASLan++ modeling for CSRF detection	116
14.3	A more complex example	119
15	Case studies and results	123
16	Discussion and related work	127
<hr/>		
Part V: Conclusions and future work		
<hr/>		
17	Summary of contributions	131

18 Future work	133
References	i

List of Figures

3.1	The AVANTSSAR Validation Platform	19
4.1	The SPaCioS tool and its interplay with the Security Analyst and the SUV.	28
6.1	The system \mathcal{N}_{DY} of rules of the Dolev-Yao intruder.	37
6.2	Man-in-the-middle attack on the NSPK protocol.	38
6.3	ASLan++ specification of the NSL protocol	41
7.1	The SiL language.	44
7.2	A big-step semantics for SiL.	45
7.3	A derivation for an authentication goal checking by the operational semantics of SiL.	63
7.4	Rules of the algorithm IntraLA with corresponding side conditions on the left of each rule	69
7.5	NSL execution path.	82
7.6	NSL sub-graph.	83
7.7	Symbolic attack trace of the Man-in-the-middle attack on NSPK protocol at state 15 of the algorithm execution (left) and instantiated attack trace obtained with interpolation method (right).	83
8.1	The SPiM tool.	86
8.2	UML class diagram of Message	88
8.3	UML class diagram of Key	89
8.4	UML class diagram of Variable	89
8.5	UML class diagram of Constant	90
8.6	UML class diagram of Action	90
8.7	UML class diagram of Translator	91
13.1	CSRF Oracle Message Sequence Chart	112
13.2	CSRF from the intruder point of view and the barred part is not visible to the intruder	114
14.1	Application scenario - Example	119
15.1	EUBank CSRF attack. The bank transfer has been performed from a local Apache server as highlighted in the url field.	125

List of Tables

7.1	Execution of the algorithm on the control flow graph for the protocol NSL.	76
9.1	SPiA vs Full-explore.	94
9.2	SATMC, CL-AtSe and OFMC.	95
12.1	SATMC deduction system test results	107
12.2	SATMC Encryption test results	109

Part I

Thesis overview

Introduction

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 [86]). 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., [6]) 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., [8, 22]). Such an endeavor has already been undertaken in the programming languages community, where, for instance, interpolation has been successfully applied in formal methods for model checking and test-case generation for sequential programs, e.g., [60, 61], with the aim of reducing the dimensions of the search space. Since a state space explosion often occurs in security protocol verification, I expected 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 [32], 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).

On the other hand, the analysis of flaws of security or communication protocols is just a part of the security of an application built on top of these protocols. In fact, after the implementation of an application a number of logical flaws possibly linked to the architecture structure of such application can be introduced. One of the main uses of this analysis is web applications¹. In fact, due to the complexity of modern web applications, vulnerabilities might be difficult to spot. For instance, if the web

¹ A web application is a software application hosted on one or more web servers.

server uses poorly generated random values, the attacker may open simultaneous sessions with the web server, record the random values, and infer their pattern.

However, there is a wide gap between finding an abstract security vulnerability (e.g., an attack trace produced by a model checker) and reproducing it on the system implementation. This is due to the fact that formal analysis techniques work on formal specifications of the system under test and their responses abstract away many details. This has some implications like the fact that an attack reported on the formal representation of the system can be a false positive; or the abstraction level of such an attack can be too high, and then not reproducible on the system implementation. Sometimes, it can even be impossible to identify the goals/properties that one would like to test.

On the other hand, using a mathematical approach and abstracting away every “not necessary” detail, one can deeply explore the system functionalities or procedures using a strong attacker model such as the one of Dolev and Yao [32]. It is also well known [33] that state-of-the-art vulnerability scanners do not detect vulnerabilities linked to logical flaws of applications while this is usually one of the main aims of formal techniques.

The main advantage in performing a security analysis of a system at runtime (against the concrete implementation) is that when a property is tested, one considers also the behavior of the implemented system. This gives also the possibility to check implementation flaws with respect to this property. One of the most common testing techniques is penetration testing where one can perform attacks on the real system and obtain a concrete counterexample by simulating a malicious attacker.

The main disadvantage is that often little mathematical rigor is used and usually a penetration test attack is as clever as the tester is. This is because without abstracting away the implementation details there are a lot of architectural and code related technicalities that must be considered. For example, an intruder that is attacking a software has to have a concrete idea of aspects like the attack pattern [73] to be used, the languages used, the environment of the system. In general, one should proceed with care when assessing the security of productive servers for vulnerabilities with potential side-effects, e.g., such as Cross-Site Request Forgery (CSRF), since one might affect the integrity of data, making manual testing a challenging task.

1.1 Main Contributions

This thesis presents results that contribute to both design time analysis of security protocols and run time analysis of web applications.

1.1.1 Design time verification of security protocols

I present an interpolation-based method for security protocol verification. This method starts from the formal specification of a protocol and of a security property and combines Craig interpolation [28], symbolic execution [48] and the standard Dolev-Yao intruder model [32] to search for goals (representing attacks on the protocol). Interpolation is used to prune possible useless traces and speed up the exploration. More specifically, the method described in Part III proceeds as follows:

starting from a specification of the input system, including protocol, properties 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, according to a procedure that I have devised, and then defines a set of goals and searches for them by symbolically executing the program. 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 [61] 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 one 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. I illustrate this method by means of concrete examples and I also report on experiments performed by using a prototype implementation.

1.1.2 Run time security analysis

I propose a model-based technique in order to detect issues related to CSRF. The essence of the formal model is simple: the client acts as an oracle for the attacker. The attacker sends a URL link to the client and the client follows the link. The bulk of the model is therefore centered around the web server, which might have envisioned various protection mechanisms against CSRF vulnerability exploitation. The attacker, in the formal model, is allowed to interact with the web server and exhaustively search his possibilities to exploit a CSRF. The result of my technique is, when a CSRF is found, an abstract attack trace reporting a list of steps an attacker has to follow in order to exploit the vulnerability. Otherwise, the specification is safe (under a number of assumptions) with respect to CSRF. I demonstrate the effectiveness and the usefulness of my method through a made-up example and three real-world case studies: DocumentRepository and EUBank (two anonymized real-life applications) and WebAuth [87]. More specifically, I propose a model-based analysis technique that extends the usage of state of the art model checking technology for security to search for CSRF based on the ASLan++ language [101]. I investigate the usage of the intruder, à la Dolev-Yao for detecting CSRF on web applications (while it is usually used for security protocols analysis), and I also show how to concretely use the technique with real web applications. In my most extensive case study I report a CSRF on a major EU Bank web site that permits a malicious bank transfer.

About this thesis

The aim of this chapter is to list all the collaborations, projects and scientific articles related to this thesis and to give a roadmap for the thesis, highlighting the main contents of each chapter.

2.1 Collaborations and publications related to this thesis

My PhD has been supported by the FP7-ICT-2007-1 Project no. 216471, “AVANTSSAR: Automated Validation of Trust and Security of Service-Oriented Architectures” [4] and by the FP7-ICT-2009-5 Project no. 257876, “SPaCIoS: Secure Provision and Consumption in the Internet of Services” [100]. In particular, during the SPaCIoS project, I spent three months (from February to April 2013) at ETH Zurich (partner of the SPaCIoS project).

List of publications related to this thesis

Papers on design time verification for security protocols:

1. Alessandro Armando, Wihem Arsac, Tigran Avanesov, Michele Barletta, Alberto Calvi, Alessandro Cappai, Roberto Carbone, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Gabriel Erzse, Simone Frau, Marius Minea, Sebastian Mödersheim, von Oheimb, David, Giancarlo Pellegrino, Serena Elisa Ponta, Marco Rocchetto, Michaël Rusinowitch, Mohammad Torabi Dashti, Mathieu Turuani, Luca Viganò - “The AVANTSSAR Platform for the Automated Validation of Trust and Security of Service-Oriented Architectures”. In the proceedings of Tools and Algorithms for the Construction and Analysis of Systems (2012), Lecture Notes in Computer Science 7214, Springer Berlin Heidelberg, pages 267-282 [4].
2. Giacomo Dalle Vedove, Marco Rocchetto, Luca Viganò, and Marco Volpe - “Using Interpolation for the Verification of Security Protocols” (Extended Abstract). In the informal proceedings of FCS Workshop on Foundations of Computer Security (2013).
3. Marco Rocchetto, Luca Viganò, Marco Volpe, and Giacomo Dalle Vedove - “Using Interpolation for the Verification of Security Protocols”. In the proceedings

of Security and Trust Management (2013), Lecture Notes in Computer Science 8203, Springer Berlin Heidelberg, pages 99-114 [84].

4. Marco Rocchetto, Luca Viganò, and Marco Volpe - “Using Interpolation for the Verification of Security Protocols” (Extended Abstract). In the informal proceedings of Interpolation: From Proofs to Applications (2014).
5. Marco Rocchetto, Luca Viganò, and Marco Volpe - “An interpolation-based method for the verification of security protocols”. Submitted to the Special Issue of the Journal of Automated Reasoning on Interpolation Techniques for Program Verification and Synthesis (2015).

Papers on run time security verification of web applications:

1. Marco Rocchetto, Martí Ochoa, Mohammad Torabi Dashti - “Model-Based Detection of CSRF”. In the proceedings of ICT Systems Security and Privacy Protection (2014), IFIP Advances in Information and Communication Technology 428, Springer Berlin Heidelberg, pages 30-43 [83].

2.2 Synopsis

Part II - State of the art.

- In Chapter 3, I give a general overview of state-of-the-art techniques for analyzing security protocols at design time. In particular I discuss: model checking techniques, the Dolev-Yao intruder model and the AVANTSSAR platform.
- In Chapter 4, I give a general overview of state-of-the-art methods for analyzing the security of the implementations of security protocols (at run time). I then describe the general idea of model-based testing, security mutants, penetration testing techniques and the SPaCIoS tool.

Part III - Design time verification of security protocols.

- In Chapter 5 and Chapter 6, I introduce my method for design time verification of security protocols.
- In Chapter 7, I give the theoretical details of my interpolation method.
- In Chapter 8, I describe the implementation details of the interpolation method.
- In Chapter 9, I report some experiment with the SPiM tool comparing the results with state-of-the-art model checkers that supports ASLan++.
- In Chapter 10, I discuss some related work and provide some possible directions for future work.

Part IV - Runtime verification of web applications.

- In Chapter 11, I introduce my method for runtime security verification of web applications.
- In Chapter 12, I show two experiments on the usage of the Dolev-Yao intruder model for the verification of web applications. This to underline the importance of the right level of abstraction when using model-based testing techniques for web applications.

- In Chapter 13, I describe what a CSRF is and how to protect a web application against these attacks.
- In Chapter 14, I describe how to model a web application in ASLan++ to search for CSRF.
- In Chapter 15, I discuss some related work and provide some possible directions for future work.

Finally in Part V, I summarize the contents of the thesis and discuss some possible future directions.

Part II

State of the art

Design time techniques

Security protocols are designed to provide secure services. However, even small mistakes made during the design phase may produce an insecure protocol and then an insecure service. An hostile intruder could take advantage of these security related issues and compromise the services that rely on flawed protocols.

Several techniques and tools have been developed to detect security flaws in security protocol, but formal methods [102] appear to be well suited to detect these flaws. In fact, in the literature, there exist a number of flawed protocols that received an extensive manual analysis without discovering any flaw (or discovering just a subset of the flaws). The Needham-Schroeder key distribution protocol [71] is a well known example.

For the purpose of the thesis we can consider techniques based on state machines [63], usually implemented as model checker tools. The general idea of this technique is to represent the protocol, together with the parties involved and an intruder, as a state machine and check whether or not a particular state (representing a violation of a security property of the protocol) can be reached.

3.1 Model checking for security protocols

Model checking is a formal technique used in various fields of computer science, but, as already mentioned, particularly adapt to security verification of security protocols. Given a formal representation of a system (also called model), a model checker automatically checks if the model meets a given specification. In my case, I have used (symbolic) model checking for security protocols in Part III and for web applications in Part IV, and in the remainder of this section I will focus on these two applications.

Model checker tools for security protocols are based on the Dolev-Yao intruder model [32], described in Section 6.2, where the network is assumed to be under the control of an active intruder who can read and destroy all the messages passing through this network. It can also modify and create messages as long as he does not break cryptography (following the perfect cryptography assumption).

Several techniques and tools have been developed starting from the Dolev-Yao paradigm, e.g., to name a few:

- Interrogator [64] was one of the first tools to implement the Dolev-Yao model. It attempts to locate protocol security flaws by an exhaustive search of the state space. While Interrogator has been able to reproduce a number of known attacks, it has never found new (previously unknown) attacks [47].
- Longley-Rigby tool [52], written in Prolog, follows the idea of Interrogator but relies on user intervention to assist the search. This idea has permitted users to find previously unknown protocol flaws.
- NRL protocol analyzer [47] uses a search algorithm similar to Interrogator but allows an unlimited number of protocol runs and then it handles an infinite state space. Another difference with Interrogator is that the emphasis of NRL is on providing proofs that the insecure states cannot be reached (otherwise, like Interrogator, to find the insecure states).
- Athena [89] is an efficient model checking tool based on an extension of the strand space model [94]. It uses a backward reachability algorithm and implements several state space reduction techniques. Athena incorporates results from theorem proving through unreachability theorems. This theorem, permits to the tool to prune the state space at an early stage and to reduce the state space explored and increase the likelihood of termination.
- Mur ϕ [65] is an explicit state model checker with its own input language. It comes along with a wide number of different versions in which several different speedup techniques have been developed, e.g., predicate abstraction and partial order reduction. There are also different versions of the model checking algorithm it implements, e.g., precise on demand or parallel random walk.
- The AVISPA tool [5] is a push-button tool for the Automated Validation of Internet Security Protocols and Applications. It provides its own formal language for specifying protocols and their security properties. It integrates different backends that implement a variety of automatic protocol analysis techniques. It is the predecessor of the AVANTSSAR platform described below.
- The AVANTSSAR platform [4] is a toolset for the formal verification of security protocols and software oriented architectures. Due to its correlation with the thesis, it is described with more details in Section 3.3 together with the three model checkers implemented as the backend of the platform (namely, SATMC, OFMC and CL-AtSe).
- Maude-NPA [35], ProVerif [18] and Scyther [29] have been developed with the focus on performances and they also implement several speed-up techniques with the only exception of Scyther that is, however, one of the most efficient tools available.

Some of these tools are discussed afterward, in order to compare with the interpolation based method proposed in this thesis. In particular, SATMC, OFMC, CL-AtSe, Maude-NPA, ProVerif, Scyther are all discussed in Part III.

3.2 ASLan and ASLan++

Modeling and reasoning about trust and security of security protocols or web applications is a very complex task and a highly error prone activity if performed manually.

Besides the classical data security requirements including confidentiality and authentication/integrity, more elaborate goals are authorization (with respect to a policy), separation or binding of duty, and accountability or non-repudiation. Some applications may also have domain-specific goals (e.g., correct processing of orders). Finally, one may consider liveness properties under certain fairness conditions, e.g., one may require that a web service for online shopping eventually processes every order if the intruder cannot block the communication indefinitely. This diversity of goals cannot be formulated with a fixed repertoire of generic properties (like authentication); instead, it suggests the need for specification of properties in an expressive logic. The AVANTSSAR Specification Language (ASLan), describes a transition system, where states are sets of typed ground terms (facts), and transitions are specified as rewriting rules over sets of terms. A fact `iknows`, true of any message (term) known to the intruder, is used to model communication as we consider a general Dolev-Yao intruder (see Section 6.2 for more details). A key feature of ASLan is the integration of this transition system that expresses the dynamics of the model with Horn clauses, which are used to describe policies in a clear, logical way. The execution model alternates transition steps with a transitive closure of Horn clause applications. This allows one to model the effects of policies in different states: for instance, agents can become members of a group or leave it, with immediate consequences for their access rights. Moreover, to carry out the formal analysis of services, one needs to model the security goals. While this can be done by using different languages, ASLan employs a variant of linear temporal logic (LTL, e.g. [42]), with backwards operators and ASLan facts as propositions. This logic gives us the desired flexibility for the specification of complex goals, as illustrated by the problem cases that are part of the AVANTSSAR Library (that can be consulted from the website www.avantssar.eu). ASLan is a low-level formal language and is thus easily usable only by experts, so the higher-level language ASLan++ has been developed to achieve three main design goals:

- The language should be expressive enough to model a wide range of platforms while allowing for succinct specifications.
- It should facilitate the specification of services at a high level of abstraction in order to reduce model complexity as much as possible.
- It should be close to specification languages for security protocols and web services, but also to procedural and object-oriented programming languages, so that it can be employed by users who are not formal specification experts.

I give now a technical overview of the security protocol specification languages ASLan++ [101] and ASLan [11], focusing on the aspects relevant to the methods proposed in this thesis (in particular, in Section 7.1.1).

ASLan++ is a formal and typed security protocol specification language, whose semantics is defined in terms of ASLan, which I describe below. An ASLan++ specification consists in a hierarchy of *entity declarations*, which are similar to Java classes. The top-level entity is usually called *Environment* (similar to the “main” procedure of a program) and it typically contains the definition of a *Session* entity, which in turn contains a number of sub-entities (and their instantiations, i.e., `new subentity(<parameters>);`) that represent all the parties involved in a protocol. Each entity is composed by two main sections: *symbols*, in which there is the instan-

tiation of all the variables and constants used in the entity, and *body*, in which the behavior of the entity is described (e.g., message exchange). Inside the body of an entity I use three different type of statements: assignments, message send and message receive. The only type of *assignment* I use here is of the form $\text{Var} := \text{fresh}()$ that assigns to the variable Var a new constant of the proper type. A *message send* statement, $\text{Snd} \rightarrow \text{Rcv} : M$, is composed by two variables Snd and Rcv representing sender and receiver respectively and a message M exchanged between the two parties. In *message receive*, Snd and Rcv are swapped and usually, in order to assign a value to the variable M , a $?$ precedes the message M , i.e., $\text{Snd} \rightarrow \text{Rcv} : ?M$. However, in ASLan++, the *Actor* keyword refers to the entity itself (similar to “this” or “self” in object-oriented languages) and thus I actually write the send and receive statements as $\text{Actor} \rightarrow \text{Rcv} : M$ and $\text{Snd} \rightarrow \text{Actor} : ?M$ respectively.

Finally, I describe here two kinds of protocol goals in ASLan++. A *channel goal*, $\text{label}(_) : \text{Snd} <\text{chn}> \text{Rcv};$, defines a property $<\text{chn}>$ that holds on all (the “ $_$ ” is a wildcard) the exchanged messages labeled with label between the two entities Snd and Rcv . For example, I use *authentication goals* defined as $\text{auth_goal}(_) : \text{Snd} \rightarrow \text{Rcv};$ where \rightarrow defines sender authenticity. A *secrecy goal* is defined with $\text{label}(_) : \{\text{Snd}, \text{Rcv}\}$, which states that each message labeled with label can only be shared between the two entities Snd and Rcv .

An ASLan++ specification can be automatically translated (see [4]) into a more low-level ASLan specification, which ultimately defines a transition system $M = \langle S, I, \rightarrow \rangle$, where S is the set of states, $I \subseteq S$ is the set of initial states, and $\rightarrow \subseteq S \times S$ is the (reflexive) transition relation. The structure of an ASLan specification is composed by six different sections: signature of the predicates, types of variables and constants, initial state, Horn clauses¹, transition rules of \rightarrow and protocol goals. The content of the sections is intuitively described by their names. In particular, an initial state $I \in I$ is composed by the concatenation of all the predicates that hold before running any rule (e.g., the agent names and the intruder’s own keys). The transition relation \rightarrow is defined as follows. For all $S \in S$, $S \rightarrow S'$ iff there exist a rule such that $PP.NP \& PC \& NC \models [V] \Rightarrow R$ (where PP and NP are sets of positive and negative predicates, PC and NC conjunctions of positive and negative atomic conditions) and a substitution $\gamma : \{v_1, \dots, v_n\} \rightarrow T_\Sigma$ where v_1, \dots, v_n are the variables that occur in PP and PC such that: (1) $PP\gamma \subseteq [S]^H$, where $[S]^H$ is the closure of S with respect to the set of clauses H , (2) $PC\gamma$ holds, (3) $NP\gamma\gamma' \cap [S]^H = \emptyset$ for all substitutions γ' such that $NP\gamma\gamma'$ is ground, (4) $NC\gamma\gamma'$ holds for all substitutions γ' such that $NC\gamma\gamma'$ is ground and (5) $S' = (S \setminus PP\gamma) \cup R\gamma\gamma'$, where γ' is any substitution such that for all $v \in V$, $v\gamma'$ does not occur in S .

I now define the translation of the ASLan++ constructs I have considered here. Every ASLan++ entity is translated into a new *state predicate* and added to the section signature. This predicate is parametrized with respect to a *step label* (that uniquely identifies every instance) and it mainly keeps track of the local state of an instance (current values of whose variables) and expresses the control flow of the entity by means of step labels. As an example, if I have the ASLan++ entity as follows:

¹ The specifications I consider in this paper do not use Horn clauses, but a so called “prelude” file, in which all the actions of the Dolev-Yao intruder are defined as a set H of Horn clauses, is automatically imported during the translation from ASLan++ into ASLan (see [11]).

```

1 entity Snd(Actor, Rcv: agent){
2   symbols
3   Var: message;
4 }

```

the predicate `stateSnd` is added to the section signature and, supposing an instantiation of the entity `new Snd(snd, rcv)`, the new predicate below is used in transition rules to store all the informations of an entity:

```

1 state_Snd(snd, iid, sl_0, rcv, dummy_message)

```

where the ID `iid` identifies a particular instance, `sl_0` is the step label, the parameters `Actor`, `Rcv` are replaced with constants `snd` and `rcv`, respectively, and the message variable `Var` is initially instantiated with `dummy_message`.

Given that an ASLan++ is a hierarchy of entities, when an entity is translated into ASLan, this hierarchy is preserved by a `child(id_1, id_0)` predicate that states `id_0` is the parent entity of `id_1` and both `id_0` and `id_1` are entity IDs.

A variable assignment statement is translated into a transition rule inside the rules section. As an example, if in the body of the entity `Snd` defined above there is an assignment `Var := constant`; where `constant` is of the same type of `Var` I obtain the following transition rule.

```

1 state_Snd(Actor, IID, sl, Rcv, Var)
2 =>
3 state_Snd(Actor, IID, succ(sl), Rcv, constant)

```

In the case of assignments to `fresh()`, the variable `Var` is assigned to a new variable.

In the case of a message exchange (sending or receiving statements) the `iknows(message)` predicate is added to the right-hand side of the corresponding ASLan rule. This states that the message `message` has been sent over the network, where `iknows` stands for *intruder knows* and is used because, as is usual, the Dolev-Yao intruder is identified with the network itself.

The last point I describe is the translation of goals focusing on authentication and secrecy described above. The labels in send statements (e.g., `Actor -> Rcv: auth:(Na)`) generates a new predicate `witness(Actor, Rcv, label, Payload)` that is inserted into the ASLan transition rule representing the send statement. An equivalent `request(Actor, Snd, label, Payload, IID)` predicate is added for receive statements. These predicates are used in the translation of goals. In fact, an authentication goal is translated into the state (i.e., attack state) below:

```

1 not(dishonest(Snd)).
2 not(witness(Snd, Rcv, auth, Payload)).
3 request(Rcv, Snd, auth, Payload, IID)

```

where `not(dishonest(Snd))` states the sender `Snd` has not to be the intruder, `not(witness(Snd, Rcv, auth, Payload))` states the payload of the authentication message has not to be sent by the honest agent `Snd` and the last `request` predicate states the receiver `Rcv` has received the authentication message. A secrecy goal is translated into the following attack state:

```

1 iknows(Payload).
2 not(contains(i, Knowers)).

```

```
3 secret(Payload, label, Knowers)
```

where `iknows(Payload)` states that the `Payload` has to be sent over the network, that the set of knowers (`Snd` and `Rcv` in the example above) does not contain the intruder `i` and the `secret` predicate is used to check the goal only when the rule containing the secrecy goal label is fired. This is because a `secret(Payload, label, Knowers)` predicate is added to all the transition rules that are translations of statements in which the payload of the secrecy goal is used. The declaration of an attack state `AS` amounts to adding a rule `AS => AS.attack` for a nullary predicate `attack`.

3.3 The AVANTSSAR platform

The AVANTSSAR platform [4] is an integrated toolset for the formal specification and automated validation of trust and security of SOAs and, in general, of applications in the IoS and their security protocols. It has been developed in the context of the FP7 project “AVANTSSAR: Automated Validation of Trust and Security in Service-Oriented Architectures”. To handle the complexity of trust and security in service orientation, the platform integrates different technologies into a single tool, so they can interact and benefit from each other. More specifically, the platform comprises three back-ends (CL-AtSe [97], OFMC [14], and SATMC [7]), which operate on the same input specification (written in ASLan) and provide complementary automated reasoning techniques (including service orchestration and compositional reasoning, model checking, and abstraction-based validation). A connectors layer provides an interface to application-level specification languages (such as the standard BPMN, ASLan++, AnB and HLPsL++), which can be translated into the core ASLan language and vice versa.

Figure 3.1 shows the main components of the AVANTSSAR Platform, where the arrows represent the most general information flow, from input specification to validated output. In this flow, the platform takes as input specifications of the available services (including their security-relevant behavior and possibly the local policies they satisfy) together with a policy stating the functional and security requirements of the target service. In the orchestration phase, the platform applies automated reasoning techniques to build a validated orchestration of the available services that meets the security requirements. More specifically, the *Orchestrator* (short for Trust and Security Orchestrator) looks for a composition of the available services in a way that is expected but not yet guaranteed to satisfy the input policy (it may additionally receive as input a counterexample found by the Validator, if any) and outputs a specification of the target service that is guaranteed to satisfy the functional goals. Then, the *Validator* (short for Trust and Security Validator), which comprises the three back-ends CL-AtSe, OFMC and SATMC, checks whether the orchestration satisfies the security goals. If so, the orchestrated service is returned as output, otherwise, a counterexample is returned to the Orchestrator to provide a different orchestration, until it succeeds, or no suitable orchestration can be found. Instead of using the Orchestrator, a user may manually generate the target service and simply invoke the Validator, providing as input the service and its security goals. In this case, the platform outputs either validation success or the counterexample found.

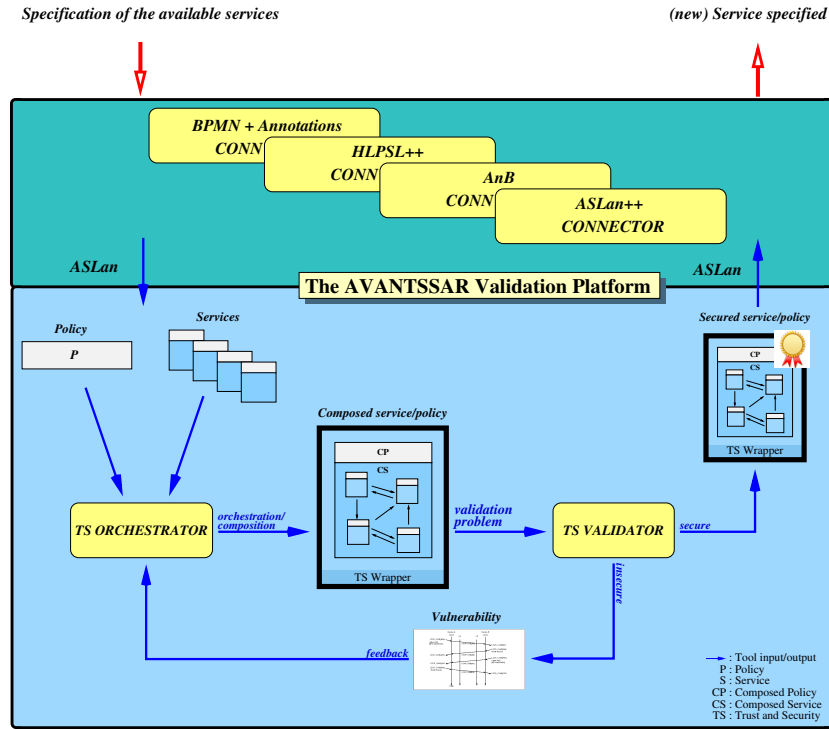


Fig. 3.1. The AVANTSSAR Validation Platform

To ease its usage, the *connectors layer* of the platform provides a set of software modules that carry out:

- The translation from application-level (e.g., ASLan++, AnB and HLPSSL++) into the low-level ASLan [9], the common input language of formal analysis by the validator back-ends.
- The reverse translation from the common output format of the validator back-ends into a higher-level MSC-like output format to ease the interpretation of the results for the user.

In the following sections, I describe the different platform components in more detail, starting with the specification languages and the connectors layer, and then considering the Orchestrator and the Validator.

3.3.1 The connectors layer

Writing formal specifications of complex systems at the low conceptual level of ASLan is not practically feasible and reasonable. The same applies to the activity of interpreting and understanding the raw output format returned by the validator back-ends. The ASLan++ connector provides translations from ASLan++ specifications to ASLan and in the reverse direction for attack traces. Security protocol/service

practitioners who are used to the more accessible but less expressive Alice-and-Bob notation or message sequence charts (MSCs) may prefer to use the AnB connector, which is based on an extended Alice-and-Bob notation [66, 67, 68], or the HLPSSL++ connector, which is based on an extension of the High-Level Protocol Specification Language HLPSSL [23], developed in the context of the AVISPA project [5, 80]. There are, also, two connectors for BPMN (see [10]): a public connector that can be used in open-source environments such as Oryx to evaluate control flow properties of a Business Process (BP) modeled in BPMN, and a proprietary SAP NetWeaver BPM connector that is a plug-in of the SAP NetWeaver Development Studio that allows BP analysts to take advantage of the AVANTSSAR Platform via a security validation service. The business analyst provides the security requirements that are critical for the compliance of the BP (e.g., need-to-know in executing a task, data confidentiality with respect to certain users or roles) through easy-to-access UIs of the security validator that returns answers in a nice graphical BPMN-like format. Connectors for other BP languages may be developed similarly.

3.3.2 The validator

The Validator takes any ASLan model of a system and its security goals and automatically checks whether the system meets its goals under the assumption that the network is controlled by a Dolev-Yao intruder. Currently, the functionality of the Validator is supported by the three different back-ends CL-AtSe, OFMC and SATMC. The user can select which back-end is used for the validation process. By default, all three are invoked in parallel on the same input specification, so that the user can compare the results of the validation carried out by the complementary automated reasoning techniques that the back-ends provide (including compositional reasoning, model checking, and abstract interpretation).

CL-AtSe

The Constraint-Logic-based Attack Searcher for security protocols and services takes as input a service specified as a set of rewriting rules, and applies rewriting and constraint solving techniques to model all states that are reachable by the participants and decides if an attack exists with respect to the Dolev-Yao intruder. The main idea in CL-AtSe consists in running the services in all possible ways by representing families of traces with positive or negative constraints on the intruder knowledge, variable values or sets, etc. Each service step execution adds new constraints on the current intruder and environment state. Constraints are kept reduced to a normal form for which satisfiability is easily checked. This allows one to decide whether some security property has been violated up to this point. CL-AtSe requires a bound on the number of service calls in case the specification allows for loops in system execution. It implements several preprocessing modules to simplify and optimize input specifications before starting a verification. If a security property is violated then CL-AtSe outputs a trace that gives a detailed account of the attack scenario.

OFMC

The Open-source Fixedpoint Model Checker (which extends the On-the-fly model checker, the previous OFMC) consists of two modules. The classical module performs verification for a bounded number of transitions of honest agents using a

constraint-based representation of the intruder behavior. The fixedpoint module allows verification without restricting the number of steps by working on an over-approximation of the search space that is specified by a set of Horn clauses using abstract interpretation techniques and counterexample-based refinement of abstractions. Running both modules in parallel, OFMC stops as soon as the classic module has found an attack or the fixedpoint module has verified the specification, so as soon as there is a definitive result. Otherwise, OFMC can just report the bounded verification results and the potential attacks that the fixedpoint module has found. In case of a positive result, one can use the computed fixedpoint to automatically generate a proof certificate for the Isabelle interactive theorem prover. The idea behind the automatic proof generator OFMC/Isabelle [20] is to gain a high reliability, since after this step the correctness of the verification result no longer depends on the correctness of OFMC and the correct use of abstractions. Rather, it only relies on: the correctness of the small Isabelle core that checks the proof generated by OFMC/Isabelle, and that the original ASLan specification (without over-approximations) indeed faithfully models the system and properties that are to be verified.

SATMC

The SAT-based Model Checker is an open, flexible platform for SAT-based bounded model checking of security services. Under the standard assumption of strong typing, SATMC performs a bounded analysis of the problem by considering scenarios with a finite number of sessions. At the core of SATMC lies a procedure that, given a security problem, automatically generates a propositional formula whose satisfying assignments (if any) correspond to counterexamples on the security problem of length bounded by some integer k . Intuitively, the formula represents all the possible evolutions, up to depth k , of the transition system described by the security problem. Finding attacks (of length k) on the service therefore reduces to solving propositional satisfiability problems. For this task, SATMC relies on state-of-the-art SAT solvers, which can handle propositional satisfiability problems with hundreds of thousands of variables and clauses or more. SATMC can also be instructed to perform an iterative deepening on the number k of steps. As soon as a satisfiable formula is found, the corresponding model is translated back into a partial-order plan (i.e., a partially ordered set of rules whose applications lead the system from the initial state to a state witnessing the violation of the expected security property).

With the AVANTSSAR platform, I conclude the discussion of tools for the analysis of security protocols at design-time. A comparison with most of them, e.g., the AVANTSSAR platform backends, will be discussed in Part III. Finally, I have also presented the ASLan++ language that is the input language for the tool presented in Part III.

Run time techniques

4.1 Testing

In this section, I give a survey about penetration and model-based testing tools, along with the techniques used during the tests and the differences between tools.

In Section 4.1.1, I present two main testing techniques used in several different tools. In Section 4.1.2, I give a first raw classification for security tools and I list some tools currently available to security analysts. Finally, in Section 4.2, I present model-based testing and in Section 4.3 I present a testing tool based on formal techniques commonly used for design time verification.

4.1.1 Testing techniques

There exist several methodologies used by different tools for testing a system. The ones I am interested in are: Black-box testing and source code analysis. In the following sections, I will give an overview of these approaches and I will categorize them based on the knowledge needed for using each tool (e.g. in black box testing one does not have the knowledge of the application source code).

4.1.1.1 Black-box testing (functional testing)

I use the term “black box testing” to describe test methods that are not based directly on application architecture source code. This term connotes a situation in which either the tester does not have access to the source code or the details of the source code are irrelevant to the properties being tested.

In this scenario the tester acquires information by testing the system using test cases; input and the expected output. All the tests are carried out from the user point of view (the external visible behavior of the software), for example, they might be based on requirements, protocol specifications, APIs, or even attempted attacks.

This kind of tests simulates the process of a real hacker but they are time-consuming and expensive.

4.1.1.2 Source code analysis and white-box testing

Source code analysis (see, e.g., [25]) is the process of checking source code for coding problems based on a fixed set of patterns or rules that might indicate possible security vulnerabilities. This process is a subset of white-box testing (see, e.g., [46]), which is a method of testing applications by checking the structure starting from the source code.

Static analysis tools are one of the most used techniques for source code analysis. These tools scan the source code and automatically detect errors that typically pass through compilers and become latent problems. The strength of static analysis depends on the fixed set of patterns or rules used by the tool; static analysis does not find all security issues.

The output of the static analysis tool still requires human evaluation. The results of the source code analysis provide a useful insight into the security posture of the application and aid in test case development.

White box testing should verify that the potential vulnerabilities uncovered by the static tool do not lead to security violations.

4.1.2 Testing tools

In the previous section I have showed which methods are used by tools in order to test applications. In this section I describe some other technique considering what kind of information can be retrieved during the testing phase.

Following [99] I use the following categorization:

- Vulnerability scanners.
- Application scanners.
- Web application assessment proxy.
- Port scanners and packet sniffer.

This classification is not exclusive and some tools could be placed in more than one category but it covers the vast majority of the security tools nowadays available.

4.1.2.1 Vulnerability scanners

A network-based vulnerability scanners attempt to test a set of known vulnerabilities on the System Under Test (SUT). Vulnerability scanners start from a database of documented network service security defects, testing each defect on each available service of the target range of hosts. Usually penetration testers run a port scanner (briefly discussed afterwards in this section) to identify the services running on the SUT and then a vulnerability scanner over a designed service previously identified.

Vulnerability scanners are generally able to scan the target operating systems and network infrastructure components, as well as any other TCP/IP device on a network to identify weaknesses of the operating system. They are usually not able to probe general purpose applications, as they lack any sort of knowledge base of how an unknown application behaves (e.g., which functionalities and how they work). Some vulnerability scanners are able to exploit network links with other sub-systems by recursively scanning the hosts on the targeted network. The tools that implement

this feature are also called host-based vulnerability scanners. They scan a host operating system for known weaknesses and un-patched software, as well as for such configuration problems as file access controls and user permissions management issues. Although they do not analyze application software directly, they are useful for finding mistakes in access control systems, configuration management, and other configuration attributes, even at an application layer.

Examples of vulnerability scanners are:

- Tenable Nessus [72].
- Core Impact [93].
- Qualys's QualysGuard [81].
- ISS's Internet Scanner [81].
- Nikto [91].
- Wikto [56].
- Maltego [58].

4.1.2.2 Application scanners

Pushing the concept of a network-based vulnerability scanner one step further, application scanners have been commonly used by penetration testers since several years ago. These tools attempt to probe web-based applications by attempting a variety of common and known attacks on each targeted application and page of the application under test. Most application scanners can observe the functional behavior of an application and then attempt a sequence of common attacks against the application. These attacks include buffer overruns, cookie manipulation, SQL injections, cross-site scripting (XSS) to name a few web attacks. It is important to note that the testing phase is black box. This implies that, although failing any of the tests is quite often linked to a security flaw, passing all of the tests cannot guarantee the security of the application tested.

Examples of application scanners are:

- SPI Dynamics's WebInspect [39].
- Rational Appscan [43].
- N-STEALTH [70].
- Metasploit [82].
- Canvas [44].
- Acunetix free ed wvs [1].
- Hailstorm [41].
- Beef [15].
- Wapiti [92].
- owasp lapse [51].

4.1.2.3 Web Application Assessment Proxies

Web application assessment proxies are perhaps the most useful of the vulnerability assessment tools for web applications. They only work on web applications and they interpose themselves between the tester's web browser and the target web server. They also allow the tester to view and modify any data content passing through the

proxy. This gives to the tester a good and flexible method for trying different approaches for testing the web application weaknesses (of both the application's user interface and associated components). This level of flexibility is why assessment proxies are generally considered essential tools for all black box testing of web applications.

Examples of web application assessment proxies are:

- Paros Proxy [26].
- OWASP's WebScarab [75].
- Burpsuite [55].
- Acunetix wvs [1].
- Grendel-Scan [40].
- PAROS pro desktop [26].
- Selenium [88].

4.1.2.4 Port scanners and packet sniffers (protocol analyzers)

Even if port scanners and packet sniffers are not tightly related with the results of this thesis, this section completes the discussion on penetration testing tools.

Port scanning tools are commonly used to gather information on the SUT. More specifically, port scanners attempt to locate all the network services available on each target host. They probe each of the designated (or default) network ports on the target system trying to identify the details of the replying service. A variety of techniques (e.g., TCP, UDP, SYN scan) is used to identify as much informations as possible on each identified service.

Examples of port scanners are:

- Nmap [57].
- Scapy [17].

Packet sniffers are commonly used to intercept and log the traffic passing over a digital network or part of a network. A sniffer captures every packet and, if needed, decodes it showing the values of various field in the packets. The captured informations are decoded from raw digital into a human-readable format that permits to the tester to easily review the exchanged information. Packet sniffers can also be hardware-based, either in probe format, or as is increasingly more common combined with a disk array. These devices record packets (or a slice of the packet) to a disk array. This allows historical forensic analysis of packets without the user having to recreate any fault.

Examples of packet sniffers are:

- Ettercap [37].
- Firesheep [38].
- Wireshark [103].

4.2 Model-based testing

Model-Based Testing (MBT) [98] consists in a variant of software testing in which a model of the SUT is used to derive test cases, namely pairs of inputs and (expected)

outputs, for its implementation. The purpose of MBT, as for software testing, is to generate a test suite (a set of test cases) accordingly to a specific criterion and to execute it on the SUT in order to acquire more confidence about the correct behavior of a system implementation, or to discover failures (i.e., unexpected behaviors).

With more details, MBT is used to extract from a formal representation of a SUT a set of pairs of inputs and outputs and test them on the implemented SUT. In other words, with MBT we check if the implementation matches the expected output over a set of inputs calculated on an abstract formal model of the system itself. This technique relies, then, on explicit behavior models of the SUT that encode the intended behavior of a system and possibly the behavior of its environment. Obviously, the more details of the system and its environment we model the more accurate set of test cases we can generate.

Advantages of adopting MBT in contrast to other testing approaches that do not rely on an abstract model of the SUT, are many-fold and mainly related to the involvement of model checkers in the testing process. The most important advantage is the possibility to generate test cases having a specific purpose in an automated way, thanks to the capability of the model checker to provide attack traces. It is indeed possible to formalize the purpose of a test suite in terms of goals and use them, with the model of the SUT, in order to cast the test case generation problem as a model checking problem. For example, in the context of coverage testing, one can generate abstract tests by using goals checking the execution of transitions. By doing so, the model checker will provide every execution trace including such transition.

4.2.1 Test case generation

A *test suite* is a finite set of test cases where a *test case* is a finite set of inputs and expected outputs: a pair of I/O in the case of deterministic systems and a tree or a graph in the case of non-deterministic reactive systems.

We can define a generic process of model-based testing as follows:

- **step 1:** Write a model of the SUT (and possibly its environment) based on the grounds of the requirements or specification documents.
- **step 2:** Choose a subset of behaviors of the model as test selection criteria.
- **step 3:** Transform the test selection criteria into test case specifications (formalization of step 2) in order to run it on the model.
- **step 4:** Generate a test suite starting from the model and the test case specification (some automatic test case generator must derive the test suite).
- **step 5:** Run the test cases. This is based on two stages:
 - **5-1:** Bridge different levels of abstraction. The concretization of the inputs and outputs of the test cases (to run them over the implementation of SUT) is performed by components that are generally called *test drivers*.
 - **5-2:** A verdict is generated whose outcomes usually are: pass, fail or inconclusive.

In general, MBT covers three majors tasks: automatic generation of abstract test cases from models, concretization of abstract tests in order to obtain executable tests, and theirs execution on the SUT by using manual or automatic means.

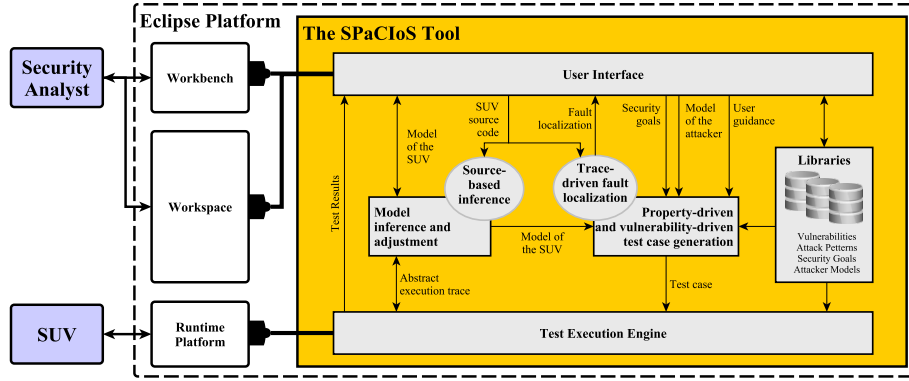


Fig. 4.1. The SPaCioS tool and its interplay with the Security Analyst and the SUV.

4.3 The SPaCioS tool

The methodologies and technologies of the SPaCioS tool cover most of the activities related to the modeling, verification and testing of web services (as well as protocols and applications). However, the SPaCioS tool has been developed to show the feasibility of using formal techniques commonly used for design time verification for runtime security verification.

In general, these steps are expected to take place during the service development process. Hence, the SPaCioS tool has been designed to be open for the integration with state-of-the-art *service development environments* (SDEs). Among the existing SDEs, SPaCioS uses by default the *Eclipse Platform* [34].

As shown in Figure 4.1, a *Security Analyst* (or *tester* for short) interacts with the *User Interface* (UI) of the SPaCioS tool via the *Eclipse workbench and workspace*. Roughly, the workbench is the collection of graphic components and controls forming the SDE interface, whereas the workspace is a portion of the file system containing the SDE resources and data. The UI of SPaCioS extends the Eclipse workbench with (i) commands, i.e., controls triggering the tool functionalities, and (ii) editors, i.e., environments supporting data insertion and visualization. For instance, commands activate model inference, test case generation and execution, whereas editors include the model editor and the message sequence chart visualizer (used to display abstract attack traces). SPaCioS relies on the Java/Eclipse runtime platform for executing test cases and interacting with the SUV.

SPaCioS allows the security analyst to execute different *workflows*, meaning that he can not only select which of the components he wishes to apply in isolation (all tool components can be executed independently) but also select different flows of application of the components.

For instance, whenever no (formal) model is initially available, the analyst can first invoke the *Model inference and adjustment* component, which has the twofold task of building a formal model of the SUV (possibly via inference from the source code hosted in the workspace) and of the environment, and to adjust the available one. Model construction is performed offline, i.e., before starting the test case generation and testing the SUV. Model adjustment is instead an online activity that is

triggered when the execution of a test reveals a discrepancy between the model and the SUV and/or the environment.

Whenever a formal model of the SUV and its environment, along with a description of the expected security property (or, dually, of a security vulnerability) is available, the analyst can invoke the *Property-driven and vulnerability-driven test case generation* component, which generates test cases starting from the property violations (and corresponding *abstract attack traces*, i.e., descriptions of exchanged messages between several components of the SUV) that have been identified by *model checking* the input model. The test case generation component also exploits a *trace-driven fault localization* based on the source code of the SUV, when available. The test cases generated are such that their execution should lead to the discovery of violation(s) of the security property, or should confirm the existence of the security vulnerability, respectively. In case a test execution fails, a discrepancy between the model and the SUV has been discovered and the abstract attack trace can be exploited for model adjustment.

The modeling language of (most of the components of) SPaCioS is ASLan++. As I already stated in Section 3.2, ASLan++ supports Horn clauses and first-order LTL formulae and thus makes SPaCioS able to support the security analysis of distributed systems that exchange messages over a wide range of secure channels, are subject to sophisticated security policies and aim at achieving a variety of security goals. ASLan++ facilitates the specification of services at a high level of abstraction in order to reduce model complexity as much as possible, and at the same time is close to specification languages for security protocols and web services, but also to procedural and object-oriented programming languages, so that it can be employed by users who are not formal specification experts.

SPaCioS has been designed for interacting with arbitrary model checkers implementing a suitable interface and, for example, the three model checkers of the AVANTSSAR Platform (CL-AtSe, OFMC and SATMC [4]) have been exploited in some experiments. The distribution package of the SPaCioS tool includes SATMC [7] as its native model checker.

In order to use models for testing purposes, the *Libraries* component of SPaCioS comprises four different categories: vulnerabilities, attack patterns (a set of rules for describing an attack [73]), security goals and attacker models. All these sets are used as input for the property-driven and vulnerability-driven test case generation component. Attack patterns are also used to guide the analyst in the iterative penetration testing phase and in the refinement of abstract traces involving respectively the Eclipse user interface and the *Test Execution Engine (TEE)*, which is in charge of executing the test cases by handling the exchange of messages with the SUV.

Concluding, in this chapter I have presented several run time techniques connected to the work I present in Part IV. In fact, in Part IV, I present a model-based testing technique to detect CSRF and the SPaCioS tool has been used to perform experiments on the importance of the Dolev-Yao intruder model to detect CSRFs. Finally, penetration testing, that is commonly used to detect CSRF, has been the basis for the modeling part and the manual concretization phase of the results in Part IV.

Design time verification of security protocols

Introduction

As described in Chapter 3, a number of tools (e.g., [4, 5, 14, 19, 35, 53, 68] just to name a few) have been developed for the analysis of security protocols at *design time*: starting from a formal specification of a protocol and of a security 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 [86]). 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., [6]) 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., [8, 22, 100] as showed in Chapter 4).

Such an endeavor has already been undertaken in the programming languages community, where, for instance, *interpolation* has been successfully applied in formal methods for model checking and test-case generation for sequential programs, e.g., [60, 61], with the aim of reducing the dimensions of the search space. Since a state space explosion often occurs in security protocol verification, I 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 (see Section 6.2) 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).

Here I address this problem and present an interpolation-based method for security protocol verification. The interpolation method starts from a formal protocol specification, along with the specification of a security property (e.g., authentication or secrecy) and a finite number of protocol sessions. It creates a corresponding sequential non-deterministic program, where parameters are introduced in order to capture the behavior of an intruder, in the form of a control flow graph, and then defines a set of goals (representing possible attacks on the protocol) and searches for them by symbolically executing the program. When a goal is reached, an attack trace is extracted from the set of constraints that the execution of the path has produced; such constraints represent conditions over parameters that allow one to reconstruct

the attack trace(s) found. When the search fails to reach a goal along a given path, a backtrack phase starts, during which the nodes of the graph are annotated (according to the IntraLA algorithm for sequential programs defined by McMillan and described in details in Section 7.2, which I have adapted and slightly modified) 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. The annotations are thus used to guide the search: I can discard a branch when it leads to a state where the annotation of the corresponding node is implied. Summing up, the output of the method is a proof of (bounded) verification in the case when no goal location can be reached starting from a finite-state specification; otherwise, one or more attack traces are produced.

Structure.

In Chapter 6, I provide some background notions on security protocol verification that will be used in Part III. In Section 6.3, I discuss the running example (the NSL protocol) considered in the rest of the chapter. In Section 7.1, I introduce SiL, the input language of the SPiM tool, which is a simple imperative programming language that I will use to define the sequential programs analyzed by the verification algorithm. I also give the details of the translation procedure from security protocols into sequential programs, for one and more protocol sessions, and prove the correctness of the translation (i.e., that it does not introduce nor delete attacks with respect to the input ASLan++ specification). In Chapter 7, I present my interpolation algorithm, which is a slightly simplified version of McMillan’s IntraLA algorithm [61]. In Chapter 8, I introduce the SPiM tool and discuss the experiments I have performed. In Chapter 10, I discuss the main results and further related work (in addition to the works already considered in the remainder of the chapter).

Background

In this chapter I describe the algebra of messages for security protocols (useful for the definition of the SiL language used by the SPiM tool), the details of the running example that will be used in the remainder of this chapter and, the the Dolev-Yao intruder model (integrated in SPiM).

6.1 Messages

Security protocols describe how agents exchange messages, built using cryptographic primitives, in order to obtain security guarantees such as confidentiality or authentication. Protocol specifications are parametric and prescribe a general recipe for communication that can be used by different agents playing in the protocol roles (sender, receiver, server, etc.). The messages transmitted are bit-strings, but, for the purposes of Part III, they can also be taken from any other appropriate set of values and the reported results of this entire chapter are independent of such details.

The *algebra of messages* tells how messages are constructed. Following [14, 67], I consider a countable *signature* Σ and a countable set Var of *variable symbols* disjoint from Σ , and write Σ^n for the symbols of Σ with arity n ; thus Σ^0 is the set of *constants*, which I assume to have disjoint subsets that I refer to as *agent names* (or just *agents*), *public keys*, *private keys*, *symmetric keys* and *nonces*. This allows to say that the intruder initially knows all agent names and public keys whenever they are created. The variables are, however, untyped (unless denoted otherwise) and can be instantiated with arbitrary types, yielding an *untyped model*. I will use upper-case letters to denote variables (e.g., A, B, \dots for 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 that have 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 thesis, I consider the following public operations: ¹

¹ I could, of course, add other operations, e.g., for hash functions but I do not do it because the expressiveness of the language already permits to show that Craig's interpolants can be effectively used as a speed up technique.

- $\{M_1\}_{M_2}$ 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 symmetric encryption of M_1 with symmetric key M_2 ; I assume that this primitive includes also some form of integrity protection (e.g., a MAC).
- $[M_1, M_2]$ (or simply M_1, M_2 when there is no risk of confusion) represents the concatenation of M_1 and M_2 . In the case of composition, I will also assume that the composition of concatenated messages is *normalized*, e.g., $[m_1, [m_2, [m_3, m_4]]]$ normalizes to $[m_1, m_2, m_3, m_4]$.

In contrast to the public operations, the mappings of Σ_m are those functions that do not correspond to operations that agents can perform on messages, but that map between constants. In Part III, I use the following two mappings:

- $inv(M)$ gives the private key that corresponds to public key M .²
- For long-term key infrastructures, I assume that every agent A has a public key $pk(A)$ and corresponding private key $inv(pk(A))$; thus $pk(\cdot)$ is a mapping from agents to public keys. In the same way, one may model further long-term key infrastructures, e.g., using $sk(A, B)$ to denote a shared key of agents A and B .

Since the mappings map from constants to constants, I consider a term like $inv(pk(a))$ as atomic since its construction does not involve any operation performed by an honest agent or the intruder, nor there is a way to “decompose” such a message into smaller parts. Since I will below also deal with terms that contain variables, I will 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, and I denote the set of ground terms with \mathcal{T}_Σ .

As it is often done in security protocol verification, I 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.³ For instance, two constant symbols n_1 and n_2 immediately represent different values. Thus, if honest agents create two random numbers (“nonces”), I abstract from the small probability that they are accidentally the same.

² Some approaches, e.g. [78], denote by K^{-1} the inverse of a symmetric key K , with $K^{-1} = K$. I cannot do this since in my models messages are untyped and hence the inverse key cannot be determined from the (type of the) key. Every message in my models has an asymmetric inverse. As I will define in the next subsection, the intruder (as well as the honest agents) can compose a message from its submessages but cannot generate M^{-1} from M . The only ways to obtain the inverse of a key are to know it initially, to receive it in a message, or when it is the private key of a self-generated asymmetric key pair.

³ Numerous algebras have been considered in security protocol verification, e.g. [27, 66], ranging from the free algebra to various formalizations of algebraic properties of the cryptographic operators employed. Here, I focus, for simplicity, on the free algebra, but I could of course also consider a more complex algebra (e.g., for protocols that make use of modular exponentiation or xor) and extend the interpolation method accordingly as, in principle, nothing in the method would prevent such an extension.

$$\begin{array}{c}
\frac{M \in IK}{M \in DY(IK)} G_{\text{axiom}} \quad \frac{M_1 \in DY(IK) \quad M_2 \in DY(IK)}{[M_1, M_2] \in DY(IK)} G_{\text{pair}} \\
\frac{M_1 \in DY(IK) \quad M_2 \in DY(IK)}{\{M_1\}_{M_2} \in DY(IK)} G_{\text{crypt}} \quad \frac{M_1 \in DY(IK) \quad M_2 \in DY(IK)}{\{M_1\}_{M_2} \in DY(IK)} G_{\text{scrypt}} \\
\frac{[M_1, M_2] \in DY(IK)}{M_i \in DY(IK)} A_{\text{pair}_i} \quad \frac{\{M_1\}_{M_2} \in DY(IK) \quad M_2 \in DY(IK)}{M_1 \in DY(IK)} A_{\text{scrypt}} \\
\frac{\{M_1\}_{M_2} \in DY(IK) \quad \text{inv}(M_2) \in DY(IK)}{M_1 \in DY(IK)} A_{\text{crypt}} \\
\frac{\{M_1\}_{\text{inv}(M_2)} \in DY(IK) \quad M_2 \in DY(IK)}{M_1 \in DY(IK)} A_{\text{crypt}}^{-1}
\end{array}$$

Fig. 6.1. The system \mathcal{N}_{DY} of rules of the Dolev-Yao intruder.

6.2 The Dolev-Yao intruder model

I consider here the standard Dolev and Yao [32] model of an active intruder, denoted by i , who controls the network but cannot break cryptography, but note that my approach is independent of the actual strength of the intruder and weaker (or stronger, e.g., being able to attack the cryptography) intruder models could be considered.

Before going into the details of this formalization it is important to recall how a public key encryption systems works. In a public key system, for every user U there exists an encryption function $E_U : \{0, 1\}^* \rightarrow \{0, 1\}^*$ and a decryption function $D_U : \{0, 1\}^* \rightarrow \{0, 1\}^*$ such that:

- (U, E_U) is publicly known.
- D_U is only known to user U .
- $E_U D_U = D_U E_U = 1$.
- Given a message M , the knowledge of $E_U(M)$ and the public directory (U, E_U) does not reveal M .

This means that everyone can send an encrypted message $E_U(M)$ to the user U and U will be able to decrypt it with $D_U(E_U(M)) = M$. As is it often done in symbolic approaches to security protocol analysis, I follow the *perfect cryptography assumption*, which postulates that the cryptographic primitives themselves cannot be attacked and hence the only way to decrypt a message is to possess the appropriate key. With other words, I assume that no users but U will be able to decrypt the message $E_U(M)$. With more details I assume:

- The encryption functions used are unbreakable.
- The public directory is secure and cannot be tampered with.
- Everyone can access to all E_U .
- Only the user U knows D_U .

i can intercept messages and analyze them if he possesses the corresponding keys for decryption, and he can generate messages from his knowledge and send them

$$\begin{aligned}
A &\rightarrow i : \{N_A, A\}_{pk(i)} \\
i(A) &\rightarrow B : \{N_A, A\}_{pk(B)} \\
B &\rightarrow i(A) : \{N_A, N_B\}_{pk(A)} \\
i &\rightarrow A : \{N_A, N_B\}_{pk(A)} \\
A &\rightarrow i : \{N_B\}_{pk(i)} \\
i(A) &\rightarrow B : \{N_B\}_{pk(B)}
\end{aligned}$$

Fig. 6.2. Man-in-the-middle attack on the NSPK protocol.

under any agent name. For a set IK of messages, I define $DY(IK)$ (for “Dolev-Yao” and “Intruder Knowledge”) to be the smallest set closed under the *generation* (G) and *analysis* (A) rules of the system \mathcal{N}_{DY} given in Figure 6.1. The G rules express that the intruder can compose messages from known messages using pairing (G_{pair}), asymmetric (G_{crypt}) and symmetric encryption (G_{scrypt}). The A rules describe how the intruder can decompose messages.

The satisfiability problem of whether a message M can be derived from $DY(IK)$ is NP-complete [86].

Note that this formalization correctly handles non-atomic keys, for instance $m \in DY(\{\{f(k_1, k_2)\}_m, k_1, k_2, f\})$. This is in contrast to other models such as [3, 54, 78, 90] that only handle atomic keys.

6.3 Running example

As a running example, I will use NSL, the Needham-Schroeder Public Key (NSPK) protocol with Lowe’s fix [53], which aims at mutual authentication between A and B :

$$\begin{aligned}
A &\rightarrow B : \{N_A, A\}_{pk(B)} \\
B &\rightarrow A : \{N_A, N_B, B\}_{pk(A)} \\
A &\rightarrow B : \{N_B\}_{pk(B)}
\end{aligned}$$

The presence of B in the second message prevents the man-in-the-middle attack that NSPK suffers from, which is shown on the left of Figure 6.2, where I write $i(A)$ to denote that the intruder is impersonating the honest agent A .

I give the overall ASLan++ specifications for the protocol NSL afterwards in Figure 6.3; here I describe only the section modeling the behavior of the two entities involved. Note that, for readability, from now on, I use math fonts instead of mixing math and typewriter fonts (e.g., I write $iknows(Payload)$ instead of `iknows(Payload)`) in the text, while I use typewriter in code listings.

```

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: {auth:(Nb)}_pk(B);
9   }
10  }
11
12  entity Bob(A, Actor: agent) {
13    symbols
14      Na, Nb: text;
15    body{
16      ? -> Actor: {?Na, ?A}_pk(Actor);
17      Nb := fresh();
18      Actor -> A: {Na, Nb, Actor}_pk(A);
19      A -> Actor: {auth:(Nb)}_pk(Actor);
20    }
21  }

```

The two roles are *Alice*, who is the *initiator* of the protocol, and *Bob*, the *responder*. 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 *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 in a regular session of the protocol) and the name of *B*, all encrypted with her own public key. 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 the listing on the right. I omit here a full description of the code (that however can be found in Figure 6.3) and only remark that the “?” in the beginning of line 5 denotes the fact that the sender of such a message can be any agent, though no assignment is made for ? in that case.

In this example, I want to verify whether the man-in-the-middle attack known for the NSPK protocol can be still applied after Lowe's fix. The scenario I am interested in can be obtained by the following ASLan++ instantiation:

```

1   body { % of Environment
2     any Session(a, i);
3     any Session(a, b);
4   }

```

In session 1, the roles of *Alice* and *Bob* are played by the agents *a* and *i*, respectively, whereas in session 2 they are played by *a* and *b*.

Finally, a set of goals needs to be specified. For simplicity, here I only require to check the authentication property with respect to the nonce of *Bob*, i.e., I will verify that the responder *Bob* authenticates the initiator *Alice*.

```

1  goals { auth:(_) A *-> B; }

```

As an example of the equivalent ASLan specification, I show the ASLan code of the translation of line 7 of the *Alice* entity

```

1  iknows(crypt(pk(E_S_A_Actor), pair(Na, pair(Nb_1,
      E_S_A_B))))).
2  state_Alice(E_S_A_Actor, E_S_A_IID, 3, E_S_A_B,
      Na, Nb)
3  =>
4  state_Alice(E_S_A_Actor, E_S_A_IID, 4, E_S_A_B,
      Na, Nb_1)

```

where, after receiving the message in the *iknows* predicate, the nonce *Nb* is updated in the state fact in the right-hand side of the transition rule (see Section 3.2 for more details).

```

1  specification NSL
2  channel_model CCM
3
4  entity Environment {
5      symbols
6      a,b:agent;
7
8      entity Session (A, B: agent) {
9
10         entity Alice (Actor, B: agent) {
11
12             symbols
13             Na, Nb: text;
14
15             body {
16                 Na := fresh();
17                 Actor -> B: {Na.Actor}_pk(B);
18                 B -> Actor: {Na.?Nb.B}_pk(Actor);
19                 Actor -> B: {Nb}_pk(B);
20             }
21         }
22
23         entity Bob (A, Actor: agent) {
24
25             symbols
26             Na, Nb: text;
27
28             body {
29                 ? -> Actor: {?Na.?A}_pk(Actor);
30                 Nb := fresh();
31                 Actor -> A: {Na.Nb.Actor}_pk(A);
32                 A -> Actor:
33                     Responder_authenticates_Initiator:(
34                         {Nb}_pk(Actor));
35             }
36         }
37
38         body { % of Session
39             new Alice(A,B);
40             new Bob(A,B);
41         }
42
43         goals
44         Responder_authenticates_Initiator:(_) A *-> B;
45     }
46
47     body { % of Environment
48         any Session(a,i);
49         any Session(a,b);
50     }
51 }

```

Fig. 6.3. ASLan++ specification of the NSL protocol

An interpolation-based method for the verification of security protocols

My method starts from the formal specification of a protocol and of a security property and combines Craig interpolation [28], symbolic execution [48] and the standard Dolev-Yao intruder model [32] to search for goals (representing attacks on the protocol). Interpolation is used to prune possible useless traces and speed up the exploration. More specifically, my method proceeds as follows: starting 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 the preprocessor ASLan++2Sil described in Chapter 8),¹ it first creates a corresponding sequential non-deterministic program, in the form of a *control flow graph* (CFG), according to a procedure that I have devised, and then defines a set of goals and searches for them by symbolically executing the program. When a goal is reached, an attack trace can be 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 back-track phase starts, during which the nodes of the graph are annotated (according to an adaptation of the algorithm defined in [61] 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 the current branch can be discarded. 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.

In order to show that my method concretely speeds up the validation, I have implemented a Java prototype called *SPiM* (*Security Protocol interpolation Method*). I report here also on some experiments performed: I considered 7 case studies and compared the analysis of SPiM with and without interpolation, thereby showing that interpolation does indeed speed up security protocol verification by reducing the search space and the execution time. I also compare the SPiM tool with the three state-of-the-art model checkers for security protocols that are part of the AVANTSSAR platform [4], namely, CL-AtSe [97], OFMC [14] and SATMC [7]. This comparison shows, as I expected, that SPiM is not yet as efficient as these ma-

¹ Similar to other methods, e.g., [4, 5, 14].

ture tools but that there is considerable room for improvement, e.g., by enhancing my interpolation-based method with some of the optimization techniques that are integrated in the other tools.

7.1 Translating security protocols into sequential programs

I introduce a simple imperative programming language, SiL (SPiM input Language), which will be used to define the sequential programs to be analyzed by the verification algorithm. SiL provides standard constructs, like assignments and conditionals, together with mechanisms used to handle specific aspects of security protocols, like the possibility of generating messages. I have three special identifiers: IK , referring to the intruder knowledge, $attack$, which takes boolean values and is set to *true* when an attack is found, and $witness$, a predicate with three arguments (a sender, a receiver, and a message) and is used in order to verify a goal of authentication.² Furthermore, SiL comprises a construct of the form $IK \vdash M$, which is used to state that the message M can be generated by the intruder. Figure 7.1 shows the full grammar of SiL, where X ranges over a set of *variable locations* Loc and c ranges over the set $\Sigma^0 \cup \mathbb{N}$.

I denote with $V = Loc \cup \{IK, attack, witness\}$ the set of *program variables* and with $D = \Sigma^0 \cup \mathbb{N} \cup \mathcal{P}(\mathcal{T}_\Sigma) \cup \{true, false\} \cup \mathcal{P}(\Sigma^0 \times \Sigma^0 \times \mathcal{T}_\Sigma)$ the set of *possible data values*, i.e., natural numbers, ground messages, sets of ground messages, boolean values and sets of triples (agent, agent, message) for the *witness* predicate. A (*SiL data*) *state* is a partial function $\sigma : V \rightarrow D$ and I denote with \mathbb{D} the set of all such functions. Figure 7.2 (discussed afterwards in this section) presents a big-step operational semantics for SiL, where I also use the following meta-variables: m ranges over \mathcal{T}_Σ , l ranges over lists of elements of \mathcal{T}_Σ , p ranges over $\mathcal{P}(\mathcal{T}_\Sigma)$, and $b \in \{true, false\}$. I denote with $\sigma[m/X]$ the state obtained from σ by replacing the content of X by m , i.e., $\sigma[m/X](Y) = m$ if $Y = X$ and $\sigma[m/X](Y) = \sigma(Y)$ otherwise.

$$\begin{aligned}
B &::= true \mid false \mid IK \vdash M \mid E = E \mid witness(E, E, M) \mid not(B) \mid B \text{ or } B \mid B \text{ and } B \\
M &::= E \mid [M, M] \mid \{M\}_M \mid \{M\}_{inv(M)} \mid \{M\}_M \\
E &::= X \mid c \\
C &::= X := E \mid IK := S \mid attack := B \mid witness(E, E, M) := true \mid C; C \mid \\
&\quad \mid \text{if } B \text{ then } C \text{ else } C \mid skip \mid end \\
L &::= M \mid L, M \\
S &::= \{L\} \mid IK \mid S + S
\end{aligned}$$

Fig. 7.1. The SiL language.

² Two remarks are in order. First, for simplicity, I give the syntax in the case of a single goal to be considered; in case of more goals, a distinct *attack* variable can be added for each goal. Second, note that, by the definition of the translation procedure into a SiL program, an authentication goal is verified immediately after the receipt of the message on which the authentication is based. Thus I do not need in SiL an equivalent of the ASLan predicate *request*.

$$\begin{array}{c}
\langle c, \sigma \rangle \Downarrow c \quad \langle X, \sigma \rangle \Downarrow \sigma(X) \quad \langle \text{skip}, \sigma \rangle \Downarrow \sigma \quad \langle \text{end}, \sigma \rangle \Downarrow \sigma \\
\\
\frac{\langle C_0, \sigma \rangle \Downarrow \sigma'' \quad \langle C_1, \sigma'' \rangle \Downarrow \sigma'}{\langle C_0; C_1, \sigma \rangle \Downarrow \sigma'} \quad \frac{\langle \text{end}, \sigma \rangle \Downarrow \sigma \quad \langle C_1, \sigma' \rangle \Downarrow \sigma'}{\langle \text{end}; C_1, \sigma \rangle \Downarrow \sigma'} \\
\\
\frac{\langle E, \sigma \rangle \Downarrow c}{\langle X := E, \sigma \rangle \Downarrow \sigma[c/X]} \quad \frac{\langle S, \sigma \rangle \Downarrow p}{\langle IK := S, \sigma \rangle \Downarrow \sigma[p/IK]} \\
\\
\frac{\langle B, \sigma \rangle \Downarrow \text{true} \quad \langle C_0, \sigma \rangle \Downarrow \sigma'}{\langle \text{if } B \text{ then } C_0 \text{ else } C_1, \sigma \rangle \Downarrow \sigma'} \quad \frac{\langle B, \sigma \rangle \Downarrow \text{false} \quad \langle C_1, \sigma \rangle \Downarrow \sigma'}{\langle \text{if } B \text{ then } C_0 \text{ else } C_1, \sigma \rangle \Downarrow \sigma'} \\
\\
\frac{\langle M_1, \sigma \rangle \Downarrow m_1 \quad \langle M_2, \sigma \rangle \Downarrow m_2}{\langle [M_1, M_2], \sigma \rangle \Downarrow [m_1, m_2]} \quad \frac{\langle M_1, \sigma \rangle \Downarrow m_1 \quad \langle M_2, \sigma \rangle \Downarrow m_2}{\langle \{M_1\}_{M_2}, \sigma \rangle \Downarrow \{m_1\}_{m_2}} \\
\\
\frac{\langle M_1, \sigma \rangle \Downarrow m_1 \quad \langle M_2, \sigma \rangle \Downarrow m_2}{\langle \{M_1\}_{\text{inv}(M_2)}, \sigma \rangle \Downarrow \{m_1\}_{\text{inv}(m_2)}} \quad \frac{\langle M_1, \sigma \rangle \Downarrow m_1 \quad \langle M_2, \sigma \rangle \Downarrow m_2}{\langle \{M_1\}_{M_2}, \sigma \rangle \Downarrow \{m_1\}_{m_2}} \\
\\
\frac{\langle L, \sigma \rangle \Downarrow l}{\langle \{L\}, \sigma \rangle \Downarrow \{l\}} \quad \frac{\langle L, \sigma \rangle \Downarrow l \quad \langle M, \sigma \rangle \Downarrow m}{\langle L, M, \sigma \rangle \Downarrow l, m} \quad \frac{\langle S_1, \sigma \rangle \Downarrow p_1 \quad \langle S_2, \sigma \rangle \Downarrow p_2}{\langle S_1 + S_2, \sigma \rangle \Downarrow p_1 \cup p_2} \\
\\
\langle IK, \sigma \rangle \Downarrow \sigma(IK) \quad \langle \text{true}, \sigma \rangle \Downarrow \text{true} \quad \langle \text{false}, \sigma \rangle \Downarrow \text{false} \\
\\
\frac{\langle E_1, \sigma \rangle \Downarrow c_1 \quad \langle E_2, \sigma \rangle \Downarrow c_2 \quad \langle M, \sigma \rangle \Downarrow m}{\langle \text{witness}(E_1, E_2, M), \sigma \rangle \Downarrow \text{true}} (c_1, c_2, m) \in \sigma(\text{witness}) \\
\\
\frac{\langle E_1, \sigma \rangle \Downarrow c_1 \quad \langle E_2, \sigma \rangle \Downarrow c_2 \quad \langle M, \sigma \rangle \Downarrow m}{\langle \text{witness}(E_1, E_2, M), \sigma \rangle \Downarrow \text{false}} (c_1, c_2, m) \notin \sigma(\text{witness}) \\
\\
\frac{\langle IK, \sigma \rangle \Downarrow \sigma(IK) \quad \langle M, \sigma \rangle \Downarrow m}{\langle IK \vdash M, \sigma \rangle \Downarrow \text{true}} m \in DY(\sigma(IK)) \\
\\
\frac{\langle IK, \sigma \rangle \Downarrow \sigma(IK) \quad \langle M, \sigma \rangle \Downarrow m}{\langle IK \vdash M, \sigma \rangle \Downarrow \text{false}} m \notin DY(\sigma(IK)) \\
\\
\frac{\langle B, \sigma \rangle \Downarrow b}{\langle \text{not}(B), \sigma \rangle \Downarrow \neg b} \quad \frac{\langle B_1, \sigma \rangle \Downarrow b_1 \quad \langle B_2, \sigma \rangle \Downarrow b_2}{\langle B_1 \text{ or } B_2, \sigma \rangle \Downarrow b_1 \vee b_2} \\
\\
\frac{\langle B_1, \sigma \rangle \Downarrow b_1 \quad \langle B_2, \sigma \rangle \Downarrow b_2}{\langle B_1 \text{ and } B_2, \sigma \rangle \Downarrow b_1 \wedge b_2} \quad \frac{\langle B, \sigma \rangle \Downarrow b}{\langle \text{attack} := B, \sigma \rangle \Downarrow \sigma[b/\text{attack}]} \\
\\
\frac{\langle E_1, \sigma \rangle \Downarrow c_1 \quad \langle E_2, \sigma \rangle \Downarrow c_2 \quad \langle M, \sigma \rangle \Downarrow m}{\langle \text{witness}(E_1, E_2, M) := \text{true}, \sigma \rangle \Downarrow \sigma[\text{witness} \cup \{(c_1, c_2, m)\} / \text{witness}]} \\
\\
\frac{\langle E_1, \sigma \rangle \Downarrow c_1 \quad \langle E_2, \sigma \rangle \Downarrow c_2 \quad c_1 = c_2}{\langle E_1 = E_2, \sigma \rangle \Downarrow \text{true}} \quad \frac{\langle E_1, \sigma \rangle \Downarrow c_1 \quad \langle E_2, \sigma \rangle \Downarrow c_2 \quad c_1 \neq c_2}{\langle E_1 = E_2, \sigma \rangle \Downarrow \text{false}}
\end{array}$$

Fig. 7.2. A big-step semantics for SiL.

The rules in Figure 7.2 are semantics rules commonly used for any imperative programming language. For example, the *skip* rule does not change the state (i.e., σ is not changed by applying the rule) as in any imperative programming language.

I now describe some of them to improve their readability.

- $\langle \text{skip}, \sigma \rangle \Downarrow \sigma$: *skip* (like *end*) is a very simple command and it is the empty instruction where the state σ does not change its value.
- $\frac{\langle S, \sigma \rangle \Downarrow p}{\langle IK := S, \sigma \rangle \Downarrow \sigma[p/IK]}$: an assignment of a value S to IK ($IK := S$) requires that the evaluation p of S in σ will be the evaluation of IK after the assignment.
- $\frac{\langle M_1, \sigma \rangle \Downarrow m_1 \quad \langle M_2, \sigma \rangle \Downarrow m_2}{\langle \{M_1\}_{M_2}, \sigma \rangle \Downarrow \{m_1\}_{m_2}}$: the evaluation of an encryption of message M_1 with M_2 results in $\{m_1\}_{m_2}$ if their evaluation in σ is m_1 and m_2 respectively.

7.1.1 The translation procedure

Given a protocol \mathcal{P} involving a set \mathcal{R} of roles (*Alice, Bob, ...*, a.k.a. *entities*), a *session instance* (or *session*, for short) 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 my method is then: (1) an ASLan++ 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} .

I now describe how to obtain a program for a single session. First of all, in my translation, and according to the ASLan++/ASLan instantiation mechanism, a session instance between two honest agents is represented as the composition of two sessions, where each of the honest agents communicates with the intruder. I will refer to the session instances obtained after such a translation as *program instances*.

Example 7.1. For instance, the second session of my example (between a and b) is obtained by the composition of two sessions, played by a and $i(b)$, and $i(a)$ and b , respectively, thus giving rise to the following three program instances:

Program	Alice	Bob
1	a	i
2	a	$i(b)$
3	$i(a)$	b

where $i(x)$ denotes the intruder playing the role of x , for x an agent name. I also remark that an intruder is not obliged to play dishonestly; thus, e.g., a program instance between a and $i(b)$ does also capture the case of an honest session between a and b .

□

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 the program. After a first section that concerns the initialization of the variables, the program will indeed contain a proper translation, based on the semantics of ASLan++, of the instructions in such a sequence. For each program instance, I will follow the

flow of execution of the honest agents, as I can think of the intruder actions as not being driven by any protocol, and model the intruder interaction with the honest agents by means of $IK \vdash M$ statements and updates of IK .

For simplicity, for the variables and constants of the resulting program I will use the same names as the ones used in the ASLan++ 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. In the case when more than one session are considered, I also prefix an index denoting the session to the program variable name, e.g., as in $SI.E.V$.

The behavior of the intruder introduces a form of non-determinism, which I capture by representing the program as a procedure depending on a number of input values, one for each choice of the intruder. Along a single program, input variables are denoted by the symbol Y , possibly subscripted with an index. Finally, symbols of the form $c.i$, for i an integer, are used to denote constants to be assigned to nonces.

In what follows, I describe how variables are initialized and statements translated.

7.1.1.1 Initialization of the variables

A first section of the program consists in the initialization of the variables. Let pi be the program instance of the program I am considering. For each role *Alice* such that $pi(Alice) = a$, for some agent name $a \neq i$, I have an initialization instruction $Alice.Actor := a$. Furthermore, for the same *Alice*, and for each other role *Bob*, with B being the variable referring to the role *Bob* amongst the agent variables of *Alice*: if $si(Bob) = b$, then I have the assignment $Alice.B := b$. Finally, it is necessary to initialize the intruder knowledge. A typical initialization instruction for IK has the form:

$$IK := a.I, \dots, a.n, i, pk(a.I), \dots, pk(a.n), pk(i), inv(pk(i))$$

That is, i knows each agent $a.j$ involved in the scenario and his 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 intruder knowledge or the initialization of further variables, depending on the context, such as symmetric keys or hash functions, which are possibly defined in the Prelude section of the ASLan++ specification.

7.1.1.2 Sending of a message

The sending of a message $Actor \rightarrow B : M$ defined in a role *Alice* is translated into the instruction $IK := IK + \{M\}$, where the symbol $+$ denotes set union (corresponding to \cup) so that the intruder knowledge is increased with the message M .

7.1.1.3 Receipt of a message

Consider the receipt of a message $R \rightarrow Actor : M$ in a role *Alice*. Assume the message is sent from a role *Bob*. Then the instruction is translated into the following code, where $Q.I, \dots, Q.n$ are all the variables occurring preceded by $?$ in M and $Y.I, \dots, Y.n$ are distinct input variables not introduced elsewhere:

```

1  If (IK |- M)
2      then Alice.Q_1 := Y_1;
3          ...
4          Alice.Q_n := Y_n;
5      else end;

```

7.1.1.4 Generation of fresh values

Finally, an instruction of the form $N := \text{fresh}()$ in *Alice*, which assigns a fresh value to a nonce, can be translated into the instruction $\text{Alice}.N := c_I$, where c_I is a constant not introduced elsewhere.

7.1.1.5 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 [48], I express such properties as correctness assertions, typically placed at the end of a program. Once I have represented a protocol session as a program (or more programs in the case when a session instance is split into more program instances), and defined the properties I am 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.

I 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 a statement of the form *if* ($\text{not}(\text{expr})$) *then* $\text{attack} := \text{true}$, where expr is an expression referring to the goal considered, as described below.

7.1.1.6 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 program instance by the ASLan++ statement: $B \rightarrow \text{Actor} : \text{auth} : (M)$ where auth is the label of the goal and a corresponding sending statement is included in the specification. We can restrict the attention to the case when according to the program instance under consideration *Bob* is played by i , since otherwise 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, one of the conditions required in order to reach the goal is $\text{not}(\text{Alice}.B = i)$, where B is the agent variable referring to the role *Bob* inside *Alice*. A second condition is necessary and concerns the fact that the message M has not been sent by $\text{Alice}.B$ to $\text{Alice}.\text{Actor}$. This can be verified by using the witness predicate, which is set to true when the message is sent and whose state is checked when a goal is searched for, i.e., immediately after the receipt of the message M .

Example 7.2. In NSL, we are interested in verifying a property of authentication in the session that assigns i to *Alice* and b to *Bob*: namely, we want *Bob* to authenticate *Alice* with respect to the nonce $Bob.Nb$ contained in the reception in line 8 on the right of the NSL example (Section 6.3). Such a receipt corresponds to the sending of line 8 on the left. Thus we can add a witness assignment of the form $witness(Alice.Actor, Alice.B, [Alice.Nb, pk(Alice.B)]) := true$ after the sending, and the instruction

```

1  if (not(Bob.A = i) and not(witness(Bob.A,
    Bob.Actor, {Bob.Nb}_pk(Bob.Actor))))
2    then attack := true;
3    else skip;

```

after the receipt of the message. \square

7.1.1.7 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 $\mathcal{R} = \{Alice_1, \dots, Alice_n\}$ in a session si , i.e., I have a sending statement $Actor \rightarrow B : \{secret : (M)\}$, where *secret* is the goal label, for a confidentiality goal expressed as $secret : (-) \{Alice_1, \dots, Alice_n\}$. This amounts 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 verifying 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 I can read as having indeed $i \in \mathcal{R}$. The following assertion is added at the end of the SiL program:

```

1  if ((IK |- Alice.M) and (not((Alice_1.B^1_1 = i)
    or
2    ... (Alice_1.B^1_m = i) or ...
3    (Alice_n.B^n_1 = i) or ... (Alice_n.B^n_m =
    i))))
4    then attack := true;
5    else skip;

```

where $Alice_j$, for $1 \leq j \leq n$, is a role such that $Alice_j \in \mathcal{R}$ and $si(Alice_j) \neq i$, $\{Bob_1, \dots, Bob_m\} \subseteq \mathcal{R}$ is the subset of those roles in \mathcal{R} that are instantiated with i by si and B_l^j , 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 7.3. For NSL, assume that I want to verify the confidentiality of the variable Nb (contained in the specification of *Bob*) between the roles in the set $\{Alice, Bob\}$. I can express this goal by appending at the end of the program the assertion

```

1  if ((IK |- Bob.Nb) and (not(Bob.A = i)))
2    then attack := true;
3    else skip;

```

\square

7.1.1.8 Integrity

In this case, I 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:

```

1  if (not(Alice.M = Bob.M'))
2    then attack := true;

```

Example 7.4. The program instances described in example 7.1 give rise to the following three *SiL* programs, for which a single *IK* initialization instruction holds:

$IK := \{a, b, i, pk(a), pk(b), pk(i), inv(pk(i))\}$

Program 1

```

1  S1_Alice.Actor := a;
2  S1_Alice.B := i;
3  S1_Alice.Na := c_1;
4
5  IK := IK + {{{S1_Alice.Na,
                  S1_Alice.Actor}}_pk(S1_Alice.B)}};
6
7  if (IK |- {{{S1_Alice.Na, {S1_Alice.Y_1,
                  S1_Alice.B}}}_pk(S1_Alice.Actor)})
8    then S1_Alice.Nb := S1_Alice.Y_1;
9    else end;
10
11 IK := {{{S1_Alice.Nb}_pk(S1_Alice.B)}};
12 witness(S1_Alice.Actor, S1_Alice.B,
          {{{S1_Alice.Nb}_pk(S1_Alice.B)}})

```

Program 2

```

1  S2_Alice.Actor := a
2  S2_Alice.B := b
3
4  S2_Alice.Na := c_1
5
6  IK := IK + {{{S2_Alice.Na,
                  S2_Alice.Actor}}_pk(S2_Alice.B)}}
7
8  if (IK |- {{{S2_Alice.Na, {S2_Alice.Y_1,
                  S2_Alice.B}}}_pk(S2_Alice.Actor)})
9    then S2_Alice.Nb := S2_Alice.Y_1;
10   else end;
11
12 IK := IK + {{{S2_Alice.Nb}_pk(S2_Alice.B)}}

```

```

13   witness(S2_Alice.Actor, S2_Alice.B,
          {S2_Alice.Nb}_pk(S2_Alice.B))

```

Program 3

```

1   S2_Bob.A := a
2   S2_Bob.Actor := b
3
4   if (IK |- {{S2_Bob.Y_1,
              S2_Bob.Y_2}}_pk(S2_Bob.Actor))
5   then S2_Bob.Na := S2_Bob.Y_1;
6        S2_Bob.A := S2_Bob.Y_2;
7   else end;
8
9   S2_Bob.Nb := c_2
10
11  IK := IK + {{S2_Bob.Na, {S2_Bob.Nb,
                          S2_Bob.Actor}}}_pk(S2_Bob.A)}
12
13  if (IK |- {S2_Bob.Nb}_pk(S2_Bob.Actor))
14  then
15      if (witness(S2_Bob.A, S2_Bob.Actor,
                  {S2_Bob.Nb}_pk(S2_Bob.Actor))
16          and
17          (not(S2_Bob.A = i)))
18      then attack := true;
19  else end;

```

□

7.1.2 Combining more sessions

Now I need to define a global 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 (possibly subscripted), which can be seen as choices of the intruder with respect to the flow of the execution.³ 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.

For what follows, it is convenient to see a sequential program as a graph (which can be simply obtained by representing its control flow) on which the algorithm of Section 7.2 for symbolic execution and annotation will be executed. I recall here some notions concerning programs and program runs.

³ In other words, like in standard security protocol analysis, we have an underlying interleaved trace semantics, where the Dolev-Yao intruder controls the network and can do everything but break cryptography.

Definition 7.5. A (SiL) 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} , containing the assignments and conditional statements provided by the language SiL. A (SiL) program path of length k is a sequence of the form $l_0, a_0, l_1, a_1, \dots, l_k$, where each step $(l_j, a_j, l_{j+1}) \in \Delta$ for $0 \leq j < k-1$.

Definition 7.6. Let σ_0 be the initial data state. A (SiL) program run of length k is a pair (π, ω) , where π is a program path $l_0, a_0, l_1, a_1, \dots, l_k$ and $\omega = \sigma_0, \dots, \sigma_{k+1}$ is a sequence of data states such that $\langle a_j, \sigma_j \rangle \Downarrow \sigma_{j+1}$ for $0 \leq j \leq k$.

Let \mathcal{S} be a scenario of a protocol \mathcal{P} with m program instances pi_1, \dots, pi_m . We can associate to each program instance pi_j , for $1 \leq j \leq m$, a sequential program by following the procedure described in Section 7.1.1.

For each $1 \leq j \leq m$, we have a program graph $\mathcal{G}^j = (\Lambda^j, l_0^j, \Delta^j)$ corresponding to the program of pi_j . The program graph corresponding to a given scenario is obtained by composing the graphs of the single program instances. Below I describe an algorithm for concretely obtaining such a graph for \mathcal{S} . For simplicity, I will assume that the specification of \mathcal{P} is such that no receipts of messages are contained inside a loop statement or an if-statement.

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 .

Definition 7.7. A block of a program graph \mathcal{G}^j is a subgraph of \mathcal{G}^j such that its initial location is either the initial location of \mathcal{G}^j or an intruder location. The exit locations of a block \mathcal{B} are the locations of \mathcal{B} with no outgoing edges.

A program graph can be simply seen as a sequence of blocks. Namely, I can associate to the program graph \mathcal{G}^j , for each $1 \leq j \leq m$, its *block structure*, i.e., a sequence $\mathcal{B}_1^j, \dots, \mathcal{B}_n^j$ of blocks of \mathcal{G}^j , such that:

- (i) The initial location of \mathcal{B}_1^j is the initial location of \mathcal{G}^j .
- (ii) Each intruder location of \mathcal{G}^j is the initial location of \mathcal{B}_k^j for some $1 \leq k \leq m$.
- (iii) For $1 \leq k < n$, the initial location of \mathcal{B}_{k+1}^j coincides, in \mathcal{G}^j , with an exit location of \mathcal{B}_k^j .
- (iv) The graph obtained by composing $\mathcal{B}_1^j, \dots, \mathcal{B}_n^j$, i.e., by letting the initial location of \mathcal{B}_{k+1}^j coincide with the corresponding exit location of \mathcal{B}_k^j , is \mathcal{G}^j itself.

Intuitively, we decompose a session program graph \mathcal{G}^i into sequential 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, the procedure for generating the scenario graph will create a graph that allows one to execute all the blocks of the scenario just once, in any possible sequence that respects 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 $\mathcal{B}_1^2, \mathcal{B}_1^1, \mathcal{B}_2^1$. Note also that, under the restriction on \mathcal{P} introduced above, each block has at most one “interesting” exit location, in the sense that at most one

of its exit locations does not correspond to a location with no outgoing edges even in the original session graph. In the algorithm of Algorithm 1, I will refer to such an exit location as the *main exit location*.

In Algorithm 1, I give an algorithm that I have devised to incrementally build the graph $\mathcal{G} = (\Lambda, l_0, \Delta)$ starting from the root and adding blocks step by step. I assume the number of program instances m given. In the algorithm I use a procedure *attach*, which given a block \mathcal{B} and a location l , adds the subgraph \mathcal{B} to \mathcal{G} (by letting the initial location of \mathcal{B} coincide with l) and updates the sets Λ and Δ accordingly. During the construction, the set $T \subseteq \Lambda$ contains the locations of the graph to be still expanded. Two functions $pc : \Lambda \times \{1, \dots, m\} \rightarrow \mathbb{N}$ and $ic : \Lambda \rightarrow \mathbb{N}$ are used to keep track of the status of the construction. Their intended meaning is the following: assume that the location l in the graph is still to be expanded; then for each $1 \leq j \leq m$, $\mathcal{B}_{pc(l,j)}^j$ is the next block to be added for what concerns the program instance pi_j (i.e., each path going from the root to l has already executed \mathcal{B}_h^j for $1 \leq h < pc(l, j)$) and the next input variable to be used is $X_{ic(l)}$.

Algorithm 1: An algorithm for building the graph $\mathcal{G} = (\Lambda, l_0, \Delta)$.

```

create a location  $l$ ;
 $\Lambda := \{l\}$ ;  $l_0 := l$ ;  $\Delta := \emptyset$ ;  $pc(l, j) := 1$  for  $1 \leq j \leq m$ ;  $ic(l) := 1$ ;
for  $h = 1$  to  $m$  do
    if (initial location of  $\mathcal{B}_1^h$  is not intruder location) then
        attach  $\mathcal{B}_1^h$  to  $l$ ;
        let  $l'$  be the main exit location of  $\mathcal{B}_1^h$ ;
         $pc(l', j) := pc(l, j)$  for all  $j \neq h$ ;
         $pc(l', h) := pc(l, h) + 1$ ;
         $ic(l') := 1$ ;
         $l := l'$ ;
    end
end
 $T := \{l\}$ ;
repeat
    pick a location  $l \in T$ ;
    for  $h = 1$  to  $m$  do
        if ( $\mathcal{B}_{pc(l, h)}^h$  does exist) then
            create a location  $l^*$ ;
             $\Lambda := \Lambda \cup \{l^*\}$ ;
             $\Delta := \Delta \cup \{(l, \text{if } X_k = i, l^*)\}$ , where  $k = ic(l)$ ;
            attach  $\mathcal{B}_{pc(l, h)}^h$  to  $l^*$ ;
            let  $l'$  be the main exit location of  $\mathcal{B}_{pc(l, h)}^h$ ;
             $pc(l', j) := pc(l, j)$  for all  $h \neq j$ ;
             $pc(l', h) := pc(l, h) + 1$ ;
             $ic(l') := ic(l) + 1$ ;
             $T := T \cup \{l'\}$ ;
        end
    end
     $T := T \setminus \{l\}$ ;
until ( $T \neq \emptyset$ );

```

The first **for** loop in the pseudo-code of the algorithm composes, in a sequence, the first blocks of each session program graph. Then the **while** loop expands the graph by adding a fork at each intruder choice.

The resulting graph $\mathcal{G} = (\Lambda, l_0, \Delta)$ can be finally simplified by making indistinguishable nodes collapse into one, according to standard graph and transition system optimization techniques.

Example 7.8. Figure 7.5 shows a path of the program graph for NSL in the scenario described in the previous examples. The entire graph is obtained by unifying some equivalent nodes in the graph produced by the algorithm of Algorithm 1. \square

7.1.3 Correctness of the translation

I now show that the translation into SiL preserves important properties of the original specification. In particular, I show that given an ASLan++ specification, an attack state can be reached by analyzing its ASLan translation if and only if an attack state can be found by executing its SiL translation.

7.1.3.1 Equivalence of single steps

Definition 7.9. An ASLan term M' and a SiL term M'' are equivalent, $M' \sim M''$, iff one of the following conditions holds:

- $M' \equiv c', M'' \equiv c''$ and $c' = c''$.
- $M' \equiv \text{pair}(M'_1, M'_2), M'' \equiv [M''_1, M''_2]$ and $M'_1 \sim M''_1, M'_2 \sim M''_2$.
- $M' \equiv \text{crypt}(M'_1, M'_2), M'' \equiv \{M''_2\}_{M''_1}$ and $M'_1 \sim M''_1, M'_2 \sim M''_2$.
- $M' \equiv \text{sCrypt}(M'_1, M'_2), M'' \equiv \{|M''_2|\}_{M''_1}$ and $M'_1 \sim M''_1, M'_2 \sim M''_2$.
- $M' \equiv \text{inv}(M'_1), M'' \equiv \text{inv}(M''_1)$ and $M'_1 \sim M''_1$. □

In the following, I consider an ASLan++ program and the corresponding ASLan translation. As described in Section 3.2, for each signature in the *SignatureSection* I will have a corresponding state fact.

Definition 7.10. A variable mapping is a function $f(E, A)$ that given an entity name E and a variable name A returns the value i corresponding to the index of the position of variable A in the state fact state_E . □

Note that such a function always exists and it is implicitly created at translation time from the translation procedure from ASLan++ into ASLan described in Section 3.2.

In order to handle multiple sessions, let pi_1, \dots, pi_n be the program instances of the considered protocol scenario; I can assume to have a function g such that $g(j) = \text{SID}$ where SID is the identifier of the state fact $\text{state_Session.j}(\dots, \text{SID}, \dots) \subseteq S$ representing the symbolic session corresponding to the program instance pi_j ; note that such a function is implicitly created when a symbolic session is instantiated (Section 3.2) and is bijective.

Then I will write $S(E_j, i)$ to indicate the value v_i of the state predicate $\text{state}_E(v_1, \text{SID}, \dots, v_n)$ such that $\text{child}(g(j), \text{SID}) \subseteq S$.

Definition 7.11. An ASLan state S and a SiL state σ are equivalent, $S \sim \sigma$, iff:

- For each SiL ground term M' and ASLan ground term M'' such that $M' \sim M''$, $M' \in \text{DY}(\sigma(\text{IK})) \Leftrightarrow \text{iknows}(M'') \subseteq \lceil S \rceil^H$.
- $\sigma(\text{Sj}_E.A) = S(E_j, f(E, A))$ for each E representing an entity name involved in the protocol, for each A representing an ASLan++ variable name or parameter name of entity E , for each session instance si_j .
- $\sigma(\text{attack}) = \text{true} \Leftrightarrow \text{attack} \subseteq \lceil S \rceil^H$.
- $(M, M_1, M_2) \in \sigma(\text{witness}) \Leftrightarrow \text{witness}(M', M'_1, M'_2, \dots) \subseteq \lceil S \rceil^H$, where M, M_1 and M_2 are SiL ground terms and M', M'_1 and M'_2 are ASLan ground terms such that $M \sim M', M_1 \sim M'_1$ and $M_2 \sim M'_2$.

Notice that while an ASLan transition occurs when there exists a substitution (of values for variables) that makes a rule applicable, in SiL I simulate, and in a sense make more explicit, such a substitution by using the Y input variables. This establishes a correspondence between ASLan substitutions and assignments of values to SiL input variables, which will be important in the following proofs, and that I will handle by means of the following notion of *extension* of a SiL state.

Definition 7.12. *Given a SiL state σ and a set of input variables Y_1, \dots, Y_n such that $\sigma(Y_i)$ is undefined, an extension $\bar{\sigma}$ of σ is defined as a SiL state where $\bar{\sigma}$ is defined for Y_1, \dots, Y_n and for each other variable A , $\bar{\sigma}(A) = \sigma(A)$.*

Since the input variables of the form Y_i are not involved in the definition of equivalence, if an ASLan state S and a SiL state σ are equivalent, that is $S \sim \sigma$, and $\bar{\sigma}$ is an extension of σ , then also S and $\bar{\sigma}$ are equivalent, that is $S \sim \bar{\sigma}$.

Let r be an ASLan rule; I will write $S \xrightarrow{r} S'$ iff there exists a transition from an ASLan state S to an ASLan state S' obtained by applying the rule r .

Lemma 7.13. *Let I be an ASLan++ statement, r the corresponding ASLan rule and w the corresponding SiL code, as defined in Section 3.2 and Section 7.1.1, respectively. Given an ASLan state S and a SiL state σ such that $S \sim \sigma$:*

1. *If $S \xrightarrow{r} S'$ then there exists an extension $\bar{\sigma}$ of σ such that $\langle w, \bar{\sigma} \rangle \Downarrow \sigma'$ and $S' \sim \sigma'$.*
2. *If there exists an extension $\bar{\sigma}$ of σ such that $\langle w, \bar{\sigma} \rangle \Downarrow \sigma'$, then either there exists an S' such that $S \xrightarrow{r} S'$ and $S' \sim \sigma'$ or $S = S'$.*

Proof. The proof has been done by exhaustion on all the four possible cases.

(i) **Generation of a nonce**

Let the statement I considered be the *generation of a nonce* having the form:

```

1  entity Environment {
2    ...
3    entity Session (A, B: agent) {
4      ...
5      entity Alice(Actor, B: agent) {
6        ...
7        body {
8          ...
9          N := fresh();
10         ...
11      }

```

The corresponding ASLan rule r has the form:

```

1  step step_...(...) :=
2    PF'.
3    state_Alice(B_1, ..., B_m).
4    NF
5    =[exists N_n]=>
6    R'.
7    state_Alice(B'_1, ..., B'_m)

```

where N_n is the ASLan translation of N and $\forall j. 1 \leq j \leq m$ if $j = f(\text{Alice}, N)$ then $B'_j f(\text{Alice}, N) = N_n$ otherwise $B'_j = B_j$.

For simplicity, I ignore in the variable names the prefixes referring to the session instance. w has the form:

```
1 Alice.N := c_k;
```

where N of $\text{Alice}.N$ is the translation of the ASLan N_n .

(\Rightarrow) Let S' be such that $S \xrightarrow{r} S'$. By the semantics of ASLan there must exist a substitution γ such that:

$$\text{state_Alice}(B_1, \dots, B_m)\gamma \subseteq \lceil S \rceil^H$$

Furthermore, there exists a substitution γ'' such that:

$$\text{state_Alice}(B_1, \dots, B_m)\gamma\gamma'' \subseteq \lceil S' \rceil^H$$

It is not restrictive to suppose that $\gamma''(N_n) = c_k$ since both N_n and c_k are fresh values. For any $\bar{\sigma}$ of σ (note that there are no input variable, i.e. Y_i , in this case so I can choose any extension of σ , for example, $\bar{\sigma} = \sigma$) we have the following derivation:

$$\frac{\frac{\langle c_k, \bar{\sigma} \rangle \Downarrow c_k}{\langle \text{Alice}.N := c_k; , \bar{\sigma} \rangle \Downarrow \bar{\sigma}[\text{Alice}.N \leftarrow c_k]} \quad \langle \text{skip}, \bar{\sigma}[\text{Alice}.N \leftarrow c_k] \rangle \Downarrow \bar{\sigma}[\text{Alice}.N \leftarrow c_k]}{\langle \text{Alice}.N := c_k; , \bar{\sigma} \rangle \Downarrow \bar{\sigma}[\text{Alice}.N \leftarrow c_k] \equiv \sigma'}$$

So, I can conclude that:

$$\gamma''(B'_j f(\text{Alice}, N)) = S'(Alice, f(\text{Alice}, N)) = \sigma'(Alice.N)$$

and then $S' \sim \sigma'$.

(\Leftarrow) Given that there are no Y_i (input variable) in the above derivation, I can define γ and γ'' as above. This implies that, for each assignment to Y_i , there exist a substitution γ, γ'' such that $S \xrightarrow{r} S'$ and $S' \sim \sigma'$.

(ii) Sending of a message

Let the statement I considered be the *sending of a message* having the form:

```
1 entity Environment {
2   ...
3   entity Session (A, B: agent) {
4     ...
5     entity Alice(Actor, B: agent) {
6       ...
7       body {
8         ...
9         Actor -> B: M(A_1, ..., A_n);
10        ...
11   }
```

The corresponding ASLan rule r has the form:

```

1  step step_... (...) :=
2    PF '.
3    state_Alice(B_1, ..., B_m) .
4    NF
5    = [...] =>
6    R '.
7    state_Alice(B_1, ..., B_m)
8    knows(M'(N_1, ..., N_n))

```

where M' is the ASLan translation of M , $n \leq m$ and $\forall j. 1 \leq j \leq m$ if $j = f(\text{Alice}, A_i)$ for some $1 \leq i \leq n$, then $B'_j = N_i$, otherwise $B'_j = B_j$.

For simplicity, I ignore in the variable names the prefixes referring to the session instance. w has the form:

```

1  IK := IK + { M''(Alice.A_1, ..., Alice.A_n) };

```

where M'' is the SiL translation of M .

(\Rightarrow) Let S' be such that $S \xrightarrow{r} S'$. By the semantics of ASLan, there must exist a substitution γ such that:

$$\text{state_Alice}(B_1, \dots, B_m)\gamma \subseteq [S]^H$$

Furthermore, there exists a substitution γ'' such that:

$$\text{state_Alice}(B_1, \dots, B_m).\text{knows}(M'(N_1, \dots, N_n))\gamma\gamma'' \subseteq [S']^H$$

Then, for any extension $\bar{\sigma}$ of σ (as in the previous case, nonce statement, no input variable Y_i will be involved in the derivation), e.g. $\bar{\sigma} = \sigma, M''(\text{Alice}.A_1, \dots, \text{Alice}.A_n)\bar{\sigma} \equiv M''(\text{Alice}.A_1, \dots, \text{Alice}.A_n)\sigma$ where M'' is the SiL translation of M . By the semantics of SiL I can construct the following derivation:

$$\begin{array}{c}
\vdots \\
\frac{\langle IK, \bar{\sigma} \rangle \Downarrow \bar{\sigma}(IK) \quad \frac{\langle \{M''\}, \bar{\sigma} \rangle \Downarrow \{M''\sigma\}}{\langle IK + \{M''\}, \bar{\sigma} \rangle \Downarrow \sigma(IK) \cup \{M''\sigma\}}} \\
\frac{\langle IK := IK + \{M''\}, \bar{\sigma} \rangle \Downarrow \bar{\sigma}[IK \leftarrow \bar{\sigma}(IK) \cup \{M''\sigma\}]}{\langle IK := IK + \{M''\};, \bar{\sigma} \rangle \Downarrow \bar{\sigma}[IK \leftarrow \bar{\sigma}(IK) \cup \{M''\sigma\}] \equiv \sigma'} \Psi
\end{array}$$

I get that $\langle w, \bar{\sigma} \rangle \Downarrow \sigma' \equiv \sigma_n$, where I have used the abbreviation:

$$\Psi \equiv \langle \text{skip}, \bar{\sigma}[IK \leftarrow \bar{\sigma}(IK) \cup \{M''\sigma\}] \rangle \Downarrow \bar{\sigma}[IK \leftarrow \bar{\sigma}(IK) \cup \{M''\sigma\}]$$

Hence, $\text{knows}(M'\gamma) \subseteq [S']^H$ and $\sigma'(IK) \vdash \{M''\sigma\}$ since $\{M''\sigma\} \subseteq \sigma'(IK)$ and then $S' \sim \sigma'$.

(\Leftarrow) Given that there are no Y_i (input variable) in the above derivation, I can define γ and γ'' as above. This implies that, for each assignment to Y_i , there exist a substitution γ, γ'' such that $S \xrightarrow{r} S'$ and $S' \sim \sigma'$.

(iii) **Receipt of a message**

Let the statement I considered be the *receipt of a message* having the form:

```

1  entity Environment {
2    ...
3    entity Session (A, B: agent) {
4      ...
5      entity Alice(Actor, B: agent) {
6        ...
7        body {
8          ...
9          B -> Actor: M(?A_1, ..., ?A_n)
10         ...
11      }

```

The corresponding ASLan rule r has the form:

```

1  step ... (...) :=
2    PF'.
3    iknows(M'(N_1, ..., N_n)).
4    state_Alice(B_1, ..., B_m)
5    =>
6    R'.
7    state_Alice(B'_1, ..., B'_m)

```

where M' is the ASLan translation of M , $n \leq m$ and $\forall j. 1 \leq j \leq m$ if $j = f(\text{Alice}, A_i)$ for some $1 \leq i \leq n$, then $B'_j = N_i$, otherwise $B'_j = B_j$.

For simplicity, I ignore in the variable names the prefixes referring to the session instance. w has the form:

```

1  if (IK |- M''(Y_1, ..., Y_n))
2    then
3      Alice.A_1 = Y_1;
4      ...
5      Alice.A_N = Y_N;
6    else
7      end

```

where M'' is the SiL translation of M where I have replaced $?A_1, \dots, ?A_n$ with Y_1, \dots, Y_n .

(\Rightarrow) Let S' be such that $S \xrightarrow{r} S'$. By the semantics of ASLan, there must exist a substitution γ such that:

$$\text{iknows}(M'(N_1, \dots, N_n)).\text{state_Alice}(B_1, \dots, B_m)\gamma \subseteq \lceil S \rceil^H$$

Furthermore, there exists a substitution γ'' such that:

$$\text{state_Alice}(B'_1, \dots, B'_m)\gamma\gamma'' \subseteq \lceil S' \rceil^H$$

Then I can build an extension $\bar{\sigma}$ of σ such that:

- $\bar{\sigma}(Y_i) = \gamma(N_i)$ for $1 \leq i \leq n$.
- $\bar{\sigma}(A) = \gamma(A)$ for any other variable A .

It follows that $M'(N_1, \dots, N_n)\gamma \sim M''(Y_1, \dots, Y_n)\bar{\sigma}$ and since $iknows(M'(N_1, \dots, N_n))\gamma \subseteq \lceil S \rceil^H$ then, by hypothesis, $M''(Y_1, \dots, Y_n) \in DY(\bar{\sigma}(IK))$ which implies $< IK \vdash M''(Y_1, \dots, Y_n), \bar{\sigma} > \Downarrow true$. By using this fact in the following derivation:

$$\begin{array}{c}
 \frac{< Y_1, \bar{\sigma} > \Downarrow \bar{\sigma}(Y)}{< \Phi_1, \bar{\sigma} > \Downarrow \sigma_1} \quad \frac{< Alice.A.2 := Y_2, \sigma_1 > \Downarrow \sigma_2}{< \Phi_2, \bar{\sigma} > \Downarrow \sigma_2} \quad \dots \\
 \vdots \\
 \frac{< Y_n, \sigma_{n-1} > \Downarrow \bar{\sigma}(Y_n)}{< \Psi_n, \sigma_{n-1} > \Downarrow \sigma_n \equiv \sigma'} \\
 \frac{< \Phi_{n-1}, \bar{\sigma} > \Downarrow \Sigma_{n-1} \quad < \Psi_n, \sigma_{n-1} > \Downarrow \sigma_n \equiv \sigma'}{< \Phi_n, \bar{\sigma} > \Downarrow \sigma'} \\
 \frac{< IK \vdash M''(Y_1, \dots, Y_n), \bar{\sigma} > \Downarrow true \quad < \Phi_n, \bar{\sigma} > \Downarrow \sigma'}{< if (IK \vdash M''(Y_1, \dots, Y_n)) then \Phi_n else end, \bar{\sigma} > \Downarrow \sigma'}
 \end{array}$$

I get that $< w, \bar{\sigma} > \Downarrow \sigma' \equiv \sigma_n$, where I have used the abbreviations:

$$\Phi_i \equiv Alice.A.1 := Y_1; \dots; Alice.A.i := Y_i;$$

$$\Psi_i \equiv Alice.A.1 := Y_i; \dots; Alice.A.i := Y_n;$$

$$\sigma_i \equiv \bar{\sigma}[Alice.A.1 \leftarrow \bar{\sigma}(Y_1), \dots, Alice.A.i \leftarrow \bar{\sigma}(Y_i)].$$

I have that $S'(Alice, f(Alice, A_i)) = \gamma(B'f(Alice, A_i)) = \gamma(N_i) = \sigma'(Alice.A_i)$, for $1 \leq i \leq n$. Since S' and σ' coincide with S and σ , respectively, for what concerns the other variables, I can conclude $S' \sim \sigma'$.

(\Leftarrow) Assume there exists an extension $\bar{\sigma}$ of σ such that $< w, \bar{\sigma} > \Downarrow \sigma'$. The case when $< IK \vdash M''(Y_1, \dots, Y_n), \bar{\sigma} > \Downarrow false$ is trivial, since $\sigma' \equiv \bar{\sigma}$ and I can easily take $S' \equiv S$. Let $< IK \vdash M''(Y_1, \dots, Y_n), \bar{\sigma} > \Downarrow true$. Then $M''(Y_1, \dots, Y_n)\bar{\sigma} \in DY(\sigma(IK))$. It follows that I can choose a substitution γ such that $\gamma(N_i) = \bar{\sigma}(Y_i)$, for $1 \leq i \leq n$, and thus $iknows(M'(N_1, \dots, N_n))\gamma \subseteq \lceil S \rceil^H$. By applying the rules of SiL semantics as above and the rule r , I get an S' such that $S \xrightarrow{r} S'$ and $S' \sim \sigma'$.

(iii) Authentication and confidentiality goals

Authentication: Assume that Alice wants to authenticate Bob and consider, without loss of generality, a program instance pi where $pi(Alice) = a$ and $pi(Bob) = i$, since if Bob is played by an honest agent, then the authentication property is trivially satisfied. I has the form:

```

1  entity Environment {
2    ...
3  entity Session (A, B: agent) {
4    ...
5    entity Alice(Actor, B: agent) {
6      ...
7      body {
8        ...
9        B -> Actor: auth:(M);
10       ...
11     }

```

```

12     ...
13   }
14   ...
15   goals
16     auth:(_) B *-> A;
17     ....
18   }

```

and is a particular case of a receipt. As such, it is translated as a common receipt, treated in case (iii), plus special constructs/rules aimed at handling the goal conditions, which will be treated here. The corresponding ASLan attack state is described by:

```

1  attack_state auth(M', Actor, B,...) :=
2    not(dishonest(B)).
3    not(witness(B, Actor, M', auth)).
4    request(Actor, B, M', auth, ...)

```

where M' is the ASLan translation of M (for simplicity, I assume here that the payload on which authentication is based is the whole message). I also add a corresponding ASLan rule r of the form:

```

1  AS => AS.attack

```

which simply adds the 0-ary predicate *attack* to an attack state AS containing the predicates described above.

The corresponding SiL statement w has the form:

```

1  if(not(Alice.B = i) and not(witness(Alice.B,
2    Alice.Actor, M'')))
3    then
4      attack := true;
5    else
6      skip;

```

where M'' is the SiL translation of M . First, notice that while the rule r can be applied at any step in an ASLan run, the corresponding SiL statement w is placed, by the translation procedure, immediately after the receipt instruction. For simplicity, I will restrict to consider those ASLan runs where attack rules concerning authentication goals, like r above, are only applied immediately after the receipt of the corresponding message. This can be done without loss of generality (and is also the reason why I do not need a *request* predicate in SiL).

(\Rightarrow) In order to apply the rule r , by the semantics of ASLan, there must exist a substitution γ such that:

$$request(Actor, B, M', auth, \dots).state_Alice(\dots, B, \dots)\gamma \subseteq [S]^H$$

where, in particular, $\gamma(B) = S(Alice, f(Alice, B))$. At the same time: $dishonest(B)\gamma \not\subseteq [S]^H$ and $witness(B, Actor, M', auth)\gamma \not\subseteq [S]^H$.

Since, as for every ASLan state, $dishonest(i) \subseteq [S]^H$, I get $\gamma(B) \neq i$. Let $\bar{\sigma}$ be an extension of σ . By hypothesis, $S \sim \sigma$, from which I infer $\gamma(B) = S(Alice,$

$f(Alice, B) = \sigma(Alice.B) = \bar{\sigma}(Alice.B) \neq i$. With analogous arguments, I infer $(\bar{\sigma}(Alice.B), \bar{\sigma}(Alice.Actor), \bar{\sigma}(M'')) \notin \bar{\sigma}(witness)$. By using these facts, I obtain the derivation in Figure 7.3.

I have that S' and σ' differ from S and $\bar{\sigma}$, respectively, only for the value of the predicate *attack*. By observing that $attack \subseteq [S']^H$ and $\sigma'(attack) = true$, I conclude $S' \sim \sigma'$.

(\Leftarrow) Let $\bar{\sigma}$ be an extension of σ such that $\langle w, \sigma \rangle \Downarrow \sigma'$. The case when $\langle \Psi, \bar{\sigma} \rangle \Downarrow false$, where Ψ is defined as in (\Rightarrow) above, is trivial. Consider $\langle \Psi, \bar{\sigma} \rangle \Downarrow true$. By hypothesis, $S \sim \sigma$ and thus the preconditions of r concerning the predicates *dishonest* and *witness* are enabled in S . As for the condition on the *request*, it is enabled by the fact that the corresponding receipt has just been encountered, by construction of a SiL graph. It follows that r can be applied and I get an ASLan state S' , which differs from S only in the fact that $attack \subseteq [S']^H$. Moreover, by applying the same derivation as in case (\Rightarrow) above, I have $\sigma'(attack) = true$, from which I conclude $S' \sim \sigma'$. \square

Confidentiality: Assume that a message, corresponding to the variable M , has to be confidential between Alice and Bob, i.e. $knowers\{Alice, Bob\}$, for a particular session instance si . Suppose the message variable M is defined in the scope of Alice entity, then I has the form:

```

1  entity Environment {
2    ...
3    entity Session (A, B: agent) {
4      ...
5      entity Alice(Actor, B: agent) {
6        ...
7        body {
8          ...
9          secret_M:(M);
10         ...
11        }
12       ...
13      }
14     ...
15     goals
16       secret_M:(_) {A,B};
17     ....
18   }

```

The corresponding ASLan rule r has the form:

```

1  attack_state secret_...(knowers, M') :=
2    iknows(M').
3    not(contains(i, knowers)).
4    secret(M', ..., knowers)

```

where M' is the ASLan translation of M . I also add a corresponding ASLan rule r of the form:

```

1  AS => AS.attack

```

$$\begin{array}{c}
\frac{
\frac{
\frac{
\langle Alice.B, \bar{\sigma} \rangle \Downarrow \bar{\sigma}(Alice.B) \quad \langle i, \bar{\sigma} \rangle \Downarrow i
}{\langle Alice.B = i \rangle \bar{\sigma} \Downarrow false}
\quad
\frac{
\langle Alice.B = i \rangle \bar{\sigma} \Downarrow true
}{\langle not(Alice.B = i) \rangle \bar{\sigma} \Downarrow true}
}{\langle Alice.B, \bar{\sigma} \rangle \Downarrow \bar{\sigma}(Alice.B) \quad \langle i, \bar{\sigma} \rangle \Downarrow i}
\quad
\frac{
\frac{
\langle Alice.B, \bar{\sigma} \rangle \Downarrow \bar{\sigma}(Alice.B) \quad \langle Alice.Actor, \bar{\sigma} \rangle \Downarrow \bar{\sigma}(Alice.Actor)
}{\langle \Phi, \bar{\sigma} \rangle \Downarrow false}
\quad
\frac{
\langle not(\Phi), \bar{\sigma} \rangle \Downarrow true
}{\langle not(\Phi), \bar{\sigma} \rangle \Downarrow true}
}{\langle \Psi, \bar{\sigma} \rangle \Downarrow true}
\quad
\frac{
\langle true, \bar{\sigma} \rangle \Downarrow true
}{\langle attack := true, \bar{\sigma} \rangle \Downarrow \bar{\sigma}[true/attack]}
}{
\langle if \Psi \text{ then } attack := true \text{ else skip, } \bar{\sigma} \rangle \Downarrow \bar{\sigma}[true/attack] \equiv \sigma'
}$$

In the derivation, I used $\Phi \equiv witness(Alice.B, Alice.Actor, M'')$ and $\Psi \equiv not(Alice.B = i)$ and $(not(\Phi))$ as abbreviations.

Fig. 7.3. A derivation for an authentication goal checking by the operational semantics of SiL.

which simply adds the 0-ary predicate *attack* to an attack state *AS* containing the predicates described above.

The corresponding SiL code *w* is of the form:

```

1  if (IK ⊢ M'' and not (B = i))
2    then
3      attack := true;
4    else
5      skip;

```

where *M''* is the SiL translation of *M*.

(\Rightarrow) In order to apply the rule *r*, by the semantics of ASLan, there must exist a substitution γ such that:

$$iknows(M').secret(M', ..., knowers).contains(B, knowers) \\ .state_Alice(..., B, ...) \gamma \subseteq [S]^H$$

where, in particular, $\gamma(B) = S(Alice, f(Alice, B))$, $dishonest(B) \gamma \not\subseteq [S]^H$ and $contains(i, knowers) \gamma \gamma' \cap [S]^H = \emptyset$. Since, as for every ASLan state, $dishonest(i) \subseteq [S]^H$, I get $\gamma(B) \neq i$. Let $\bar{\sigma}$ be an extension of σ , By hypothesis, $S \sim \sigma$, from which I infer:

$$\gamma(B) = S(Alice, f(Alice, B)) = \sigma(Alice.B) = \bar{\sigma}(Alice.B) \neq i$$

Since $iknows(M') \subseteq [S]^H$ I also infer that $\sigma(IK) \vdash \{M'' \bar{\sigma}\}$ which implies $\langle IK \vdash M'', \bar{\sigma} \rangle \Downarrow true$. Since $S \xrightarrow{r} S'$, by the semantics of ASLan I obtain: $attack \subseteq [S']^H$. Given that $\langle w, \bar{\sigma} \rangle \Downarrow \sigma'$ by the semantics of SiL I get:

- $\Psi \equiv \bar{\sigma}[attack \leftarrow true]$.
- $\Phi \equiv \frac{\langle attack := true, \bar{\sigma} \rangle \Downarrow \Psi \quad \langle skip, \Psi \rangle \Downarrow \Psi}{\langle attack := true; \bar{\sigma} \rangle \Downarrow \Psi}$.

And then I obtain the following:

$$\frac{\frac{\frac{\langle IK \vdash M'', \bar{\sigma} \rangle \Downarrow true}{\langle IK \vdash M'' \text{ and } not(Alice.B = i), \bar{\sigma} \rangle \Downarrow true} \quad \frac{\frac{\langle Alice.B, \bar{\sigma} \rangle \Downarrow \bar{\sigma}(Alice.B) \quad \langle i, \bar{\sigma} \rangle \Downarrow i}{\langle Alice.B = i, \bar{\sigma} \rangle \Downarrow false} \quad \bar{\sigma}(Alice.B) \neq i}{\langle not(Alice.B = i), \bar{\sigma} \rangle \Downarrow true}}{\langle if(IK \vdash M'' \text{ and } not(Alice.B = i)) \text{ then } attack := true; \text{ else, } \bar{\sigma} \rangle \Downarrow \Psi} \Phi$$

Finally, I have that $attack \subseteq [S']^H$ and $\sigma'(attack) = true$ and then $S' \sim \sigma'$.

(\Leftarrow) I can here distinguish between two cases:

(1) $\langle IK \vdash M'' \text{ and } not(Alice.B = i), \bar{\sigma} \rangle \Downarrow true$, so, by hypothesis, there must exist a substitution γ such that

$$iknows(M').secret(M', ..., knowers) \\ .contains(B, knowers).state_Alice(..., B, ...) \gamma \subseteq [S]^H$$

where $\gamma(B) = S(Alice, f(Alice, B)) = \sigma(Alice.B) = \bar{\sigma}(Alice.B) \neq i$. Hence, for each substitution γ' I get: $contains(i, knowers) \gamma \gamma' \cap [S]^H = \emptyset$.

Now by the semantics of ASLan and SiL, as I have already done for (\Rightarrow) , I again obtain $attack \subseteq \lceil S' \rceil^H$ and $\sigma'(attack) = true$, and then $S' \sim \sigma'$.

(2) $\langle IK \vdash M'' \text{ and } not(Alice.B = i), \bar{\sigma} \rangle \Downarrow false$. This means that at least one of the two predicates does not hold. If $\langle IK \vdash M'', \bar{\sigma} \rangle \Downarrow false$, it must be that for each substitution γ : $iknows(M')\gamma \not\subseteq \lceil S \rceil^H$, so, by the semantics of ASLan, I have that $S \xrightarrow{r} S \equiv S'$ and by the semantics of SiL:

$$\frac{\frac{\langle IK \vdash M'', \bar{\sigma} \rangle \Downarrow false \quad \vdots}{\langle IK \vdash M'' \text{ and } not(Alice.B = i), \bar{\sigma} \rangle \Downarrow false} \quad \langle skip, \bar{\sigma} \rangle \Downarrow \bar{\sigma}}{\langle if(IK \vdash M'' \text{ and } not(Alice.B = i)) \text{ then } attack := true; \text{ else, } \bar{\sigma} \rangle \Downarrow \bar{\sigma} \equiv \sigma'}$$

and then $S' \sim \sigma'$.

Otherwise, if $\langle not(Alice.B = i), \bar{\sigma} \rangle \Downarrow false$, by the semantics of SiL I get:

$$\frac{\frac{\frac{\langle Alice.B, \bar{\sigma} \rangle \Downarrow \bar{\sigma}(Alice.B) \quad \langle i, \bar{\sigma} \rangle \Downarrow i}{\langle Alice.B = i, \bar{\sigma} \rangle \Downarrow true} \quad \bar{\sigma}(Alice.B) = i}{\vdots \quad \langle not(Alice.B = i), \bar{\sigma} \rangle \Downarrow false} \quad \frac{\langle IK \vdash M \text{ and } not(Alice.B = i), \bar{\sigma} \rangle \Downarrow false \quad \langle skip, \bar{\sigma} \rangle \Downarrow \bar{\sigma}}{\langle if(IK \vdash M \text{ and } not(Alice.B = i)) \text{ then } attack := true; \text{ else, } \bar{\sigma} \rangle \Downarrow \bar{\sigma} \equiv \sigma'}$$

By hypothesis there must exist a substitution γ such that:

$$iknows(M').secret(M', ..., knowers)$$

$$.contains(B, knowers).state_Alice(..., B, ...) \gamma \subseteq \lceil S \rceil^H$$

where $\gamma(B) = S(Alice, f(Alice, B)) = \sigma(Alice.B) = \bar{\sigma}(Alice.B) = i$. This implies that for each substitution γ' I have:

$$contains(i, knowers) \gamma \gamma' \cap \lceil S \rceil^H \neq \emptyset$$

That is, by the semantics of ASLan I have that $S \xrightarrow{r} S \equiv S'$ and then $S' \sim \sigma'$.

7.1.3.2 Equivalence of runs

I have showed that, starting from equivalent states, the application of ASLan rules and SiL code fragments that have been generated from the same ASLan++ statements brings to states that are still equivalent. Now I will show that given an ASLan++ specification, for each run in the SiL translation, there exists a sequence of corresponding ASLan rules in the ASLan translation.

Consider a protocol \mathcal{P} and let E_1, \dots, E_n be the entity names involved in \mathcal{P} . I denote with $I_e \equiv I_{e,1}, \dots, I_{e,m_e}$ the sequence of ASLan++ statements corresponding to the entity E_e . Given a scenario \mathcal{S} , for each program instance $pi(j)$, I denote with $r_{e,1}^j, \dots, r_{e,m_e}^j$ the sequence of ASLan rules and with $w_{e,1}^j, \dots, w_{e,m_e}^j$ the sequence of SiL actions corresponding to I_e . I note that, strictly speaking, the translation of an ASLan++ statement into SiL is not always an atomic action, e.g., in the case of a receipt, the corresponding SiL action comprises both a conditional and some assignments. For simplicity, in the remainder of this section, I will use the term actions

also to refer to such *compound actions*, i.e., small sequences of atomic actions arising from the translation of a single ASLan++ statement. Furthermore, I notice that in a sequence like $w_{e,1}^j, \dots, w_{e,m_e}^j$ as defined above, I ignore the conditionals with respect to X_i variables, i.e., those used in SiL to handle the interleaving between sessions. I will refer to such sequences of (possibly compound) actions as *SiL action paths* and define a *SiL action run* as a pair (π, ω) where $\pi = w_0, \dots, w_k$ is a SiL action path and $\omega = \sigma_0, \dots, \sigma_{k+1}$ is a sequence of data states such that $\langle a_j, \sigma_j \rangle \Downarrow \sigma_{j+1}$ for $0 \leq j \leq k$. It is easy to see that given a program graph, each SiL path corresponds to a SiL action path (obtained by ignoring the locations in the SiL path, removing the X_i -conditionals and possibly grouping some consecutive atomic actions).

Definition 7.14. An ASLan path (for a protocol scenario \mathcal{S}) is a sequence r_0, \dots, r_k of ASLan rules such that:

- For each entity E_e , program instance $pi(j)$ and $1 \leq l \leq m_e$, there is one and only one $0 \leq i \leq k$ such that $r_i \equiv r_{e,l}^j$.
- For $0 \leq i \leq k$, $r_i \equiv r_{e,l}^j$ for some e, l and j .
- For $0 \leq i \leq k$, if state $E(\dots, sl, \dots)$, where sl is the index referring to the step label, is in the left-hand side of $r_i \equiv r_{e,l}^j$ then either $sl = 1$ or there exists $h < i$ such that state $E(\dots, sl, \dots)$ is in the right-hand side of r_h and $r_h \equiv r_{e,l-1}^j$.

The intuition behind this definition is that, given an ASLan transition system, the set of ASLan paths collects all the “potential” sequences of applications of ASLan rules, i.e., those admissible by only taking care of respecting the order given by the step labels inside the rules, no matter how the rest of the state evolves.

Definition 7.15. An ASLan run (for a protocol scenario \mathcal{S}) is a pair (τ, ρ) where τ is an ASLan path r_0, \dots, r_k and $\rho = S_0, \dots, S_{k+1}$ is a sequence of ASLan states such that $S_i \xrightarrow{r_i} S_{i+1}$ for $0 \leq i \leq k$.

Definition 7.16. An ASLan path r_0, \dots, r_k and a SiL action path w_0, \dots, w_k are equivalent iff for each $0 \leq i \leq k$, r_i and w_i can be obtained as the translation of the same ASLan++ statement.

Lemma 7.17. Let \mathcal{S} be a protocol scenario and \mathcal{G} the corresponding program graph. Then there exists a SiL action path w_0, \dots, w_k for \mathcal{G} iff there exists an ASLan path r_0, \dots, r_k for \mathcal{S} and the paths are equivalent.

Proof. It is enough to observe that SiL action paths and ASLan paths follow, for a given program instance, the order in which the actions are executed in the protocol: this is obtained by the definition of the graph construction in the case of SiL, and by using step labels inside the rules in the case of ASLan. Furthermore, in both cases, each possible interleaving between sessions is admitted, i.e., whenever in a SiL path an action of the program instance $pi(i)$ is followed by an action of the program instance $pi(j)$, there is a corresponding possible choice for a next rule r to be applied in ASLan such that $r = r_{e,l}^j$ for some e and l ; vice versa, for each ASLan rule in an ASLan path letting one switch from a session i to a session j , there is a corresponding branch where $X_h = j$ giving rise to a corresponding SiL path. \square

Theorem 7.18. *There exists a SiL action run (π, ω) of graph \mathcal{G} corresponding to the protocol scenario \mathcal{S} , where $\omega = \sigma_0, \dots, \sigma_{k+1}$, iff there exists an ASLan run (τ, ρ) for \mathcal{S} , where $\rho = S_0, \dots, S_{k+1}$, and $\sigma_i \sim S_i$ for $0 \leq i \leq k+1$.*

Proof. Let σ_0 be the data state obtained after the initialization block of the SiL program graph and S_0 the ASLan initial state, as defined in Chapter 6. It is easy to check that $\sigma_0 \sim S_0$. Then, the thesis follows by using Lemma 7.17 (for each SiL action path, there is an equivalent ASLan path) and Lemma 7.13 (equivalent steps preserve equivalence of states). \square

Finally, I can use the previous theorem to show that an attack state can be found in an ASLan path iff a goal location can be reached in the corresponding SiL path.

Corollary 7.19. *Let \mathcal{S} be a protocol scenario and \mathcal{G} the corresponding program graph. An attack state can be found in an ASLan path for \mathcal{S} iff a goal location can be reached in a SiL path for \mathcal{G} .*

Proof. Let S be an ASLan attack state, i.e., $\text{attack} \subseteq [S]^H$. By Theorem 7.18, S is in an ASLan run for \mathcal{S} iff there exists $\sigma \sim S$ in a SiL run for \mathcal{G} . By Definition 7.11, $\sigma(\text{attack}) = \text{true}$, i.e., a goal location referring to the given attack has been reached. \square

7.2 Interpolation algorithm

In this section, I give a slightly simplified version of the IntraLA algorithm of [61] (obtained by removing some fields only used there to deal with program procedures) and describe how interpolants can be calculated in the specific case we are interested in. The algorithm executes symbolically a program graph searching for goal locations, which in my case represent attacks found on the given scenario of the protocol. In the case when the algorithm fails 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.

I will use a *two-sorted first-order language with equality*. The first sort is based on the algebra of messages defined in Chapter 6, over which I also allow a set of unary predicates DY_{IK}^j for $1 \leq j \leq n$ with a fixed $n \in \mathbb{N}$, whose meaning will be clarified below, and a ternary predicate *witness*. The second sort is based on a signature containing a set of variables (denoted in my examples by X possibly subscripted) and uninterpreted constants (for which I use integers as labels), and allows no functions and no predicates other than equality. I 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, which I will instantiate with the concrete program variables of my SiL programs.

Before presenting the algorithm, I introduce some notions concerning symbolic execution.

Definition 7.20. Let V be the set of program variables. A symbolic data state is a triple (P, C, E) , where P is a (again, 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 . Note, in particular, that IK is mapped to a set of message terms and witness to a set of triples of message terms. I denote by Ξ the set of symbolic data states.

Given its definition, a symbolic data state ξ can be characterized by the predicate

$$\chi(\xi) = C \wedge (\bigwedge_{v \in V \setminus \{IK\}} (v = E(v))) \wedge (\bigwedge_{m \in E(IK)} DY_{IK}^0(m)) (\bigwedge_{(m_1, m_2, m_3) \in E(\text{witness})} \text{witness}(m_1, m_2, m_3))$$

Note that the variable IK is treated in a particular way, i.e., I translate the fact that $E(IK) = M$ for some set M of parametric messages into a formula expressing that a predicate DY_{IK}^0 holds for all the messages in M .

A symbolic data state ξ can be associated to the set $\varepsilon(\xi)$ of data states produced by the map E for some valuation of the parameters satisfying the constraint C . I assume a defined initial symbolic data state $\varepsilon(\xi_0) = \{d_0\}$.

Definition 7.21. A symbolic state is a pair $(l, \xi) \in \Lambda \times \Xi$.

Definition 7.22. A symbolic interpreter SI is a total map from the set \mathcal{A} of SiL actions to $\Xi \times \Xi$ such that for each symbolic data state ξ and action a , $\cup \varepsilon(SI(a)(\xi)) = \text{Sem}(a)(\gamma(\xi))$.

Intuitively, SI takes a symbolic data state ξ and an action a and returns a non-empty set of symbolic data states, which represent the set of states obtained by executing the action a on ξ .

Definition 7.23. 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 $(\Lambda \cup \Delta) \times \mathcal{L}(V)$. I 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 there are more than one label on a given location, one can read them as a disjunction of conditions: $A(l) = \bigvee \{\phi \mid l : \phi \in A\}$.

Definition 7.24. For an edge $e = (l_n, a, l_{n+1})$, the label $e : \phi$ is justified in A . Let $\text{Out}(l)$ be the set of outgoing edges from a location l ; the label $l : \phi$ is justified in A when, for all edges $e \in \text{Out}(l)$, 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.

Definition 7.25. A query $q = (l, \xi)$ is blocked by a formula ϕ when $\xi \models \phi$ and I then write $\text{Bloc}(q, A(\phi))$. With respect to q , the edge e is blocked when $\text{Bloc}(q, A(e))$ and the location l is blocked when $\text{Bloc}(q, A(l))$.

$$\begin{array}{c}
\overline{\{(l_0, \xi_0)\}, \emptyset, G_0} \text{ Init} \\
\\
\begin{array}{l}
q = (l_n, \xi_n) \in Q \\
e = (l_n, a, l_{n+1}) \in \Delta \\
\neg \text{Bloc}(q, A(e)) \\
\xi_{n+1} \in SI(a)(\xi_n) \\
\neg \text{Bloc}((l_{n+1}, \xi_{n+1}), A(l_{n+1}))
\end{array}
\frac{Q, A, G}{Q + (l_{n+1}, \xi_{n+1}), A, G} \text{ Decide} \\
\\
\begin{array}{l}
q = (l_n, \xi_n) \in Q \\
e = (l_n, a, l_{n+1}) \in \Delta \\
\text{Bloc}(q, \phi) \\
\mathcal{J}(e : \phi, A)
\end{array}
\frac{Q, A, G}{Q, A + e : \phi, G} \text{ Learn} \\
\\
\begin{array}{l}
q = (l_n, \xi) \in Q \\
\neg \text{Bloc}(q, A(l_n)) \\
(\forall e \in \text{Out}(l_n). \\
e : \phi_e \in A \wedge \text{Bloc}(q, \phi_e)) \\
\phi = \bigwedge \{\phi_e \mid e \in \text{Out}(l_n)\}
\end{array}
\frac{Q, A, G}{Q - q, A + l_n : \phi, G - l_n} \text{ Conjoin}
\end{array}$$

Fig. 7.4. Rules of the algorithm IntraLA with corresponding side conditions on the left of each rule

I now describe in details all the rules of the algorithm. Note that some of the side conditions are labelled with (r_i) where r is the initial letter of the name of the rule and i an identification number. The first rule applied is always *Init*, which initializes the algorithm state:

$$\overline{\{(l_0, \xi_0)\}, \emptyset, G_0} \text{ Init}$$

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 together with the graph.

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. The rule is the following:

$$\begin{array}{ll}
q = (l_n, \xi_n) \in Q & (d_1) \\
e = (l_n, a, l_{n+1}) \in \Delta & (d_2) \\
\neg \text{Bloc}(q, A(e)) & (d_3) \\
\xi_{n+1} \in SI(a)(\xi_n) & (d_4) \\
\neg \text{Bloc}((l_{n+1}, \xi_{n+1}), A(l_{n+1})) & (d_5)
\end{array}
\quad \frac{Q, A, G}{Q + (l_{n+1}, \xi_{n+1}), A, G} \text{Decide}$$

where SI is a symbolic interpreter, l_n and ξ_n denote the currently considered location and symbolic data state, respectively, and l_{n+1} and ξ_{n+1} the location and symbolic data state obtained after executing the action a (d_4). The side conditions of the *Decide* rule are that moving from ξ_n to ξ_{n+1} , the first needs to be into the query set (d_1) and the branch between the two nodes must exist (d_2) and not be blocked (neither the edge (d_3) nor the location (d_5)).

During the backtrack phase, two rules are used: *Learn* generates annotations and *Conjoin* merges annotations coming from distinct 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 *Learn* infers a new label ϕ that blocks the edge:

$$\begin{array}{ll}
q = (l_n, \xi_n) \in Q \\
e = (l_n, a, l_{n+1}) \in \Delta \\
\text{Bloc}(q, \phi) & (l_1) \\
\mathcal{J}(e : \phi, A) & (l_2)
\end{array}
\quad \frac{Q, A, G}{Q, A + e : \phi, G} \text{Learn}$$

where the formula ϕ can be any formula ϕ that both blocks the current query (l_1) and is justified (l_2). In the following, I will explain how it can be obtained by exploiting the Craig's interpolation lemma [28], which states that given two first-order formulas α and β such that $\alpha \wedge \beta$ is inconsistent, there exists a formula ϕ (their *interpolant*) such that α implies ϕ , ϕ implies $\neg\beta$ and $\phi \in \mathcal{L}(\alpha) \cap \mathcal{L}(\beta)$.

Let μ be a term, a formula, or a set of terms or of formulas. I 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 a SiL action (see Section 7.1) expresses how to move from a data state to another, I can easily associate to it a transition formula. In the following, I will write $Sem(a)$ to denote the transition formula corresponding to the action a .

In the context of the graph we are interested in, 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 , which intuitively means that the message M can be derived from a set of messages IK by using the rules of \mathcal{N}_{DY} of Figure 6.1. In my treatment, I fix a value n as the maximum number of inference steps that the intruder can execute in order to derive M . I observe that this is not a serious limitation of my method since several results (e.g., [86]) show that, when considering a finite number of sessions, as in my case, it is indeed 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 along the protocol scenario; I assume such an n to be fixed for the whole scenario.⁴

I use formulas of the form $DY_{IK}^j(M)$, for $0 \leq j \leq n$, with the intended meaning that M can be derived in n steps of inference by using the rules of \mathcal{N}_{DY} . In particular, the predicate DY_{IK}^0 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 in my language as the formula $DY_{IK}^n(M)$.

The formula

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

in which \leftrightarrow denotes the double implication and every quantification has to be intended over the sort of messages, expresses (as a disjunction) all the ways in which a given message can be obtained by the intruder in one inference step, i.e., by a single application of one of the rules in the system \mathcal{N}_{DY} , thus moving from a knowledge (denoted by the predicate) DY_{IK}^j to a knowledge (denoted by the predicate) 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 an additional set of axioms that formalize that in the free algebra of messages any two distinct ground terms are not equal, e.g., $\forall M_1. M_2. M_3. M_4. ([M_1, M_2] = [M_3, M_4]) \supset (M_1 = M_3 \wedge M_2 = M_4)$.

The translation of the program statement $IK \vdash M$ into the formula $DY_{IK}^n(M)$ is justified by the following result.

Theorem 7.26. Let M be a ground message, $n \in \mathbb{N}$, IK a set of ground messages and \mathcal{I} an interpretation of $\mathcal{T}_{Msg}(n)$ such that $IK = \mathcal{I}(DY_{IK}^0)$. Then \mathcal{I} satisfies the formula $DY_{IK}^n(M)$ iff there exists a derivation of $M \in DY(IK)$ of height⁵ at most $n+1$ in the system \mathcal{N}_{DY} .

Proof. (\Rightarrow) Assume that the interpretation \mathcal{I} satisfies the formula $DY_{IK}^n(M)$, denoted $\mathcal{I} \models DY_{IK}^n(M)$. I proceed by induction on n . If $n = 0$, then I have $\mathcal{I} \models DY_{IK}^0(M)$, i.e., $M \in \mathcal{I}(DY_{IK}^0)$ which by hypothesis gives $M \in IK$. But then there exists a derivation in \mathcal{N}_{DY} of $M \in DY(IK)$, obtained by a single application of the rule G_{axiom} . Now assume proved the assert for $n = j$ and consider $n = j + 1$. Since \mathcal{I} satisfies

⁴ The ability of the intruder of generating new messages can be simulated by enriching his initial knowledge with a set of constants not occurring elsewhere in the protocol specification. Since I consider finite scenarios, the size of such a set can also be bounded a-priori.

⁵ By *height of a derivation* Π I mean the maximum number of rule applications along one of the branches of Π .

the premise of the left-to-right implication in ϕ_j , i.e., $DY_{IK}^{j+1}(M)$, then it must also satisfy one of the disjuncts in the conclusion. I have a case for each disjunct.

- (i) Let $\mathcal{S} \models DY_{IK}^j(M)$. By induction hypothesis, there exists a derivation of $M \in DY(IK)$ in \mathcal{N}_{DY} of height at most j , which is the derivation I was looking for.
- (ii) Let $\mathcal{S} \models \exists M'. DY_{IK}^j([M, M']) \vee DY_{IK}^j([M', M])$. I can assume there exists a message M' such that $\mathcal{S} \models DY_{IK}^j([M, M'])$ (the other case is symmetrical). By induction hypothesis, there exists a derivation of $[M, M'] \in DY(IK)$ in \mathcal{N}_{DY} of height at most j . A further application of A_{pair_i} gives a derivation of $M \in DY(IK)$ of height at most $j+1$.
- (iii) Let $\mathcal{S} \models \exists M_1, M_2. M = [M_1, M_2] \wedge DY_{IK}^j(M_1) \wedge DY_{IK}^j(M_2)$. I can assume there exist two messages M_1 and M_2 such that $\mathcal{S} \models DY_{IK}^j(M_1) \wedge DY_{IK}^j(M_2)$. By induction hypothesis, there exists a derivation of $M_1 \in DY(IK)$ and a derivation of $M_2 \in DY(IK)$ in \mathcal{N}_{DY} of height at most j . A further application of G_{pair} gives a derivation of $M \in DY(IK)$ of height at most $j+1$.
- (iv) Let $\mathcal{S} \models \exists M_1, M_2. M = \{M_1\}_{M_2} \wedge DY_{IK}^j(M_1) \wedge DY_{IK}^j(M_2)$. I can assume there exist two messages M_1 and M_2 such that $\mathcal{S} \models DY_{IK}^j(M_1) \wedge DY_{IK}^j(M_2)$. By induction hypothesis, there exists a derivation of $M_1 \in DY(IK)$ and a derivation of $M_2 \in DY(IK)$ in \mathcal{N}_{DY} of height at most j . A further application of G_{crypt} gives a derivation of $M \in DY(IK)$ of height at most $j+1$.
- (v) Let $\mathcal{S} \models \exists M'. DY_{IK}^j(\{M\}_{M'}) \wedge DY_{IK}^j(\text{inv}(M'))$. I can assume there exists a message M' such that $\mathcal{S} \models DY_{IK}^j(\{M\}_{M'}) \wedge DY_{IK}^j(\text{inv}(M'))$. By induction hypothesis, there exist a derivation of $\{M\}_{M'} \in DY(IK)$ and a derivation of $\text{inv}(M') \in DY(IK)$ in \mathcal{N}_{DY} of height at most j . A further application of A_{crypt} gives a derivation of $M \in DY(IK)$ of height at most $j+1$.
- (vi) Let $\mathcal{S} \models \exists M'. DY_{IK}^j(\{M\}_{\text{inv}(M')}) \wedge DY_{IK}^j(M')$. I can assume there exists a message M' such that $\mathcal{S} \models DY_{IK}^j(\{M\}_{\text{inv}(M')}) \wedge DY_{IK}^j(M')$. By induction hypothesis, there exist a derivation of $\{M\}_{\text{inv}(M')} \in DY(IK)$ and a derivation of $M' \in DY(IK)$ in \mathcal{N}_{DY} of height at most j . A further application of A_{crypt}^{-1} gives a derivation of $M \in DY(IK)$ of height at most $j+1$.
- (vii) Let $\mathcal{S} \models \exists M_1, M_2. M = \{M_1\}_{M_2} \wedge DY_{IK}^j(M_1) \wedge DY_{IK}^j(M_2)$. I can assume there exist two messages M_1 and M_2 such that $\mathcal{S} \models DY_{IK}^j(M_1) \wedge DY_{IK}^j(M_2)$. By induction hypothesis, there exist a derivation of $M_1 \in DY(IK)$ and a derivation of $M_2 \in DY(IK)$ in \mathcal{N}_{DY} of height at most j . A further application of G_{crypt} gives a derivation of $M \in DY(IK)$ of height at most $j+1$.
- (viii) Let $\mathcal{S} \models \exists M'. DY_{IK}^j(\{M\}_{M'}) \wedge DY_{IK}^j(M')$. I can assume there exists a message M' such that $\mathcal{S} \models DY_{IK}^j(\{M\}_{M'}) \wedge DY_{IK}^j(M')$. By induction hypothesis, there exist a derivation of $\{M\}_{M'} \in DY(IK)$ and a derivation of $M' \in DY(IK)$ in \mathcal{N}_{DY} of height at most j . A further application of A_{crypt} gives a derivation of $M \in DY(IK)$ of height at most $j+1$.

(\Leftarrow) Again, I proceed by induction on n . If $n = 0$, the only admissible derivation of $M \in DY(IK)$ is the one given by an application of G_{axiom} . It follows that $M \in IK$. Then $IK = \mathcal{S}(DY_{IK}^0)$ implies $\mathcal{S} \models DY_{IK}^0(M)$. Now let $n = j$ and assume I have a derivation of $M \in DY(IK)$ of length at most $j+1$. Let r be the last rule applied. I have one case for each rule in \mathcal{N}_{DY} .

- (i) Let r be G_{pair} . It follows that I have two derivations, of length at most j , of $M_1 \in DY(IK)$ and $M_2 \in DY(IK)$, respectively, where $M = [M_1, M_2]$. By induction hypothesis, I have $\mathcal{S} \models DY_{IK}^j(M_1)$ and $\mathcal{S} \models DY_{IK}^j(M_2)$, which implies that \mathcal{S} satisfies one of the disjuncts in the premise of the right-to-left implication of φ_j , i.e., $\exists M_1, M_2. M = [M_1, M_2] \wedge DY_{IK}^j(M_1) \wedge DY_{IK}^j(M_2)$. It follows that $\mathcal{S} \models DY_{IK}^{j+1}(M)$.
- (ii) Let r be G_{crypt} . It follows that I have two derivations, of length at most j , of $M_1 \in DY(IK)$ and $M_2 \in DY(IK)$, respectively, where $M = \{M_1\}_{M_2}$. By induction hypothesis, I have $\mathcal{S} \models DY_{IK}^j(M_1)$ and $\mathcal{S} \models DY_{IK}^j(M_2)$, which implies that \mathcal{S} satisfies one of the disjuncts in the premise of the right-to-left implication of φ_j , i.e., $\exists M_1, M_2. M = \{M_1\}_{M_2} \wedge DY_{IK}^j(M_1) \wedge DY_{IK}^j(M_2)$. It follows that $\mathcal{S} \models DY_{IK}^{j+1}(M)$.
- (iii) Let r be G_{script} . It follows that I have two derivations, of length at most j , of $M_1 \in DY(IK)$ and $M_2 \in DY(IK)$, respectively, where $M = \{|M_1|\}_{M_2}$. By induction hypothesis, I have $\mathcal{S} \models DY_{IK}^j(M_1)$ and $\mathcal{S} \models DY_{IK}^j(M_2)$, which implies that \mathcal{S} satisfies one of the disjuncts in the premise of the right-to-left implication of φ_j , i.e., $\exists M_1, M_2. M = \{|M_1|\}_{M_2} \wedge DY_{IK}^j(M_1) \wedge DY_{IK}^j(M_2)$. It follows that $\mathcal{S} \models DY_{IK}^{j+1}(M)$.
- (iv) Let r be A_{pair_i} . It follows that I have a derivation, of length at most j , of $[M, M'] \in DY(IK)$ (or $[M', M] \in DY(IK)$). By induction hypothesis, I have $\mathcal{S} \models DY_{IK}^j([M, M'])$ (or $\mathcal{S} \models DY_{IK}^j([M', M])$), which implies that \mathcal{S} satisfies one of the disjuncts in the premise of the right-to-left implication of φ_j , i.e., $\exists M'. DY_{IK}^j([M, M']) \vee DY_{IK}^j([M', M])$. It follows that $\mathcal{S} \models DY_{IK}^{j+1}(M)$.
- (v) Let r be A_{script} . It follows that I have two derivations, of length at most j , of $\{M\}_{M'} \in DY(IK)$ and $M' \in DY(IK)$, respectively. By induction hypothesis, I have $\mathcal{S} \models DY_{IK}^j(\{M\}_{M'})$ and $\mathcal{S} \models DY_{IK}^j(M')$, which implies that \mathcal{S} satisfies one of the disjuncts in the premise of the right-to-left implication of φ_j , i.e., $\exists M'. DY_{IK}^j(\{M\}_{M'}) \wedge DY_{IK}^j(M')$. It follows that $\mathcal{S} \models DY_{IK}^{j+1}(M)$.
- (vi) Let r be A_{crypt} . It follows that I have two derivations, of length at most j , of $\{M\}_{M'} \in DY(IK)$ and $\text{inv}(M') \in DY(IK)$, respectively. By induction hypothesis, I have $\mathcal{S} \models DY_{IK}^j(\{M\}_{M'})$ and $\mathcal{S} \models DY_{IK}^j(\text{inv}(M'))$, which implies that \mathcal{S} satisfies one of the disjuncts in the premise of the right-to-left implication of φ_j , i.e., $\exists M'. DY_{IK}^j(\{M\}_{M'}) \wedge DY_{IK}^j(\text{inv}(M'))$. It follows that $\mathcal{S} \models DY_{IK}^{j+1}(M)$.
- (vii) Let r be A_{crypt}^{-1} . It follows that I have two derivations, of length at most j , of $\{M\}_{\text{inv}(M')} \in DY(IK)$ and $\text{inv}(M') \in DY(IK)$, respectively. By induction hypothesis, I have $\mathcal{S} \models DY_{IK}^j(\{M\}_{\text{inv}(M')})$ and $\mathcal{S} \models DY_{IK}^j(\text{inv}(M'))$, which implies that \mathcal{S} satisfies one of the disjuncts in the premise of the right-to-left implication of φ_j , i.e., $\exists M'. DY_{IK}^j(\{M\}_{\text{inv}(M')}) \wedge DY_{IK}^j(\text{inv}(M'))$. It follows that $\mathcal{S} \models DY_{IK}^{j+1}(M)$.

□

Now let $\alpha = \chi(\xi_n)$ and $\beta = \text{Sem}(a) \wedge \neg A(l_{n+1})'$. Then I 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, I observe that, in the presence

of my finite scenario assumption, when mechanizing such a search, the problem can be simplified by restricting the domain to a finite set of messages.

The last rule of the algorithm is *Conjoin* which is used when all the outgoing edges of the location in a query q are blocked.

$$\begin{array}{c}
 q = (l_n, \xi) \in Q \\
 \neg \text{Bloc}(q, A(l_n)) \\
 (\forall e \in \text{Out}(l_n). e : \phi_e \in A \wedge \text{Bloc}(q, \phi_e)) \quad (c_1) \quad \frac{Q, A, G}{Q - q, A + l_n : \phi, G - l_n} \text{Conjoin} \\
 \phi = \bigwedge \{ \phi_e \mid e \in \text{Out}(l_n) \} \quad (c_2)
 \end{array}$$

The rule blocks (c_1) the query q by labeling its location with the conjunction of the labels that block the outgoing edges (c_2) . If the location is a goal, then it can be removed from the set of remaining goals. Finally, the query is discarded from the set Q .

The algorithm terminates when no rules can be applied, which implies that the query set is empty. In [61], the correctness of the algorithm, with respect to the goal search, is proved: the proof given there applies straightforwardly for the slightly simplified version I have given here.

Theorem 7.27. 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.

Proof. All the rules preserve the invariant that A is justified (therefore inductive), that all the locations in G are unlabeled (meaning their annotation is equivalent to F) and that no queries are blocked. Now suppose the algorithm is in a state where no rules can be applied and consider some $q \in Q$. Since *Decide* does not apply, all possible successor queries are blocked. Thus, since *Learn* does not apply, all outgoing edges are blocked. Thus, since *Conjoin* does not apply, q is blocked, a contradiction. Since Q is empty, it follows that the initial query is blocked, meaning that $d_0 \models A(l_0)$. Therefore, A is an inductive invariant. Since all remaining goals are annotated F , it follows that they are unreachable. Moreover, since goals are only removed from G when reached, all locations in $G_0 \setminus G$ are reachable.

The output of the method can be of two types. If no goal has been reached, then this yields a proof of the fact that no attack can be found, with respect to the security property of interest, in the finite scenario that I am considering. Otherwise, for each goal location that has been found, I can generate an abstract attack trace. I also note that, by a trivial modification of the rule *Conjoin*, I can easily obtain an algorithm that keeps searching for a given goal even when this has already been reached through a different path, thus allowing for extracting more attack traces for the same goal on a given scenario.

Such traces can be inferred from the information deducible from the symbolic data state (P, C, E) corresponding to the last step of execution. I proceed as follows. First of all, I can reconstruct the order in which sessions have been interleaved. Such an information is completely contained in the value of the parameters corresponding to the variables X_j , for j an integer. The value of such parameters is specified in C .

This allows for obtaining the sequence of messages exchanged, expressed in terms of program variables. Then, by using the maps in E , each such a variable can be associated to a function over the set of parameters P , and possibly further specified by the constraints over the parameters in C . It follows that the final result will be a sequence of messages where all the variables have been replaced by (functions over) parameters. Such a sequence constitutes an attack trace. In the case when the value of some parameter is not fully specified by the conditions in C , I have a parametrical attack trace, which can be instantiated in more than one way. A concrete example of this can be found in Example 7.28.

Example 7.28. I continue the running example by showing the execution of the algorithm on some interesting paths of the graph defined in Section 7.1.1 for the protocol NSL: Table 7.1 summarizes the algorithm execution.

For readability, I have not reported the entire goal definition in Table 7.1. I remark that the goals set is initialized with the goal locations corresponding to the translation of the authentication goal *auth* (see Section 7.1.1 for details) but, given that no goal is reached, the goals set does not change during the execution of the algorithm. Note that in Table 7.1 I use statements of the form $IK \vdash M$ in the constraint set as an abbreviation for the set of constraints over the parameters that make the (translation of the) statement satisfiable, according to the definition above. Q_i , C_i and E_i denote, respectively, the set of queries, the set of constraints and the environment at step i of the execution. I have also used *Step* to indicate the step number and *Rule* to indicate which rule is applied using the first letter of the rule.

The first path I show (summarized by the Message Sequence Chart (MSC) in Figure 7.5) reaches a goal location with an unsatisfiable state and then annotates it with an interpolant, while the other ones reach the previously annotated path and then block their executions (thus saving some execution steps). The algorithm starts, as described in Table 7.1), by using the *Init* rule to initialize the algorithm state and then it symbolically executes the CFG from query (l_0, ξ_0) to (l_{18}, ξ_{18}) using the *Decide* rule (steps 0–19). In step 20, the algorithm blocks its symbolic execution because the edge (l_{19}, l_{20}) is labeled with the goal action for an *authentication goal* and any possible symbolic execution step leads to a blocked symbolic data state. The backtrack phase starts and, until step 33, the algorithm creates interpolants to annotate the control flow graph and then it propagates annotations up to the location l_{14} (where the symbolic execution restarts with the *Decide* rule but I have not reported it in Table 7.1 for readability).

As shown in Figure 7.6, there are other two paths that reach location l_{18} . Each path that reaches this location has already executed an action of the form $IK \vdash \{N_A, N_B, B\}_{pk(A)}$. As described in [53], it is impossible for the Dolev-Yao intruder to create a message of the form $\{N_A, N_B, B\}_{pk(A)}$ from its knowledge (IK) if the intruder is not explicitly playing the role of the sender, i.e., A . This means that each symbolic state that reaches location l_{18} implies the interpolant $S2_Bob.A = i$. This is a concrete example of how the annotation method can help (and improve) the search procedure: in NSL I can stop following every path that reaches location l_{18} as the annotation method ensures that we will never reach a goal location.

I now provide detailed explanations for each step of the execution of the interpolation algorithm on the graph (reported as a MSC for readability) in Figure 7.5 for

Step	Rule	Query	Edge	Q	A	G	P	C	E
0	I	(l_0, s_0)	-	l_0, \tilde{s}_0	\emptyset	$\{auth\}$	\emptyset	\emptyset	\emptyset
1	D	(l_0, \tilde{s}_0)	(l_0, l_1)	$(l_0, \tilde{s}_0), (l_1, \tilde{s}_1)$	\emptyset	$\{auth\}$	$P_0 \cup \{y_0\}$	C_0	$E_0 \oplus \{(S2_Alice.B, y_0), (S2_Alice.Na, c_1), (S2_Alice.Actor, a), (S2_Alice.Y_0, y_0), (S2_Bob.Actor, b), (S1_Alice.Na, c_0), (S1_Alice.Actor, a), (S1_Alice.B, i), (IK, \{a, b, i, pk(a), pk(b), pk(i), inv(pk(i))\})\}$
2	D	(l_1, \tilde{s}_1)	(l_1, l_2)	$Q_1 \cup \{(l_2, \tilde{s}_2)\}$	\emptyset	$\{auth\}$	P_1	C_1	$E_1 \oplus \{(Alice.Na, c_1), (IK, IK_1 \cup \{c_1, d, pk(s_0)\})\}$
3	D	(l_2, \tilde{s}_2)	(l_2, l_3)	$Q_2 \cup \{(l_3, \tilde{s}_3)\}$	\emptyset	$\{auth\}$	$P_2 \cup \{x_1\}$	$C_2 \cup \{(x_1 = 3)\}$	$E_2 \oplus \{(X_1, x_1)\}$
4	D	(l_3, \tilde{s}_3)	(l_3, l_4)	$Q_3 \cup \{(l_4, \tilde{s}_4)\}$	\emptyset	$\{auth\}$	$P_3 \cup \{y_1, y_2\}$	$C_3 \cup \{(IK_2 \vdash \{(y_1, y_2), pk(b)\})\}$	$E_3 \oplus \{(S2_Bob.Y_1, y_1), (S2_Bob.Y_2, y_2)\}$
5	D	(l_4, \tilde{s}_4)	(l_4, l_5)	$Q_4 \cup \{(l_5, \tilde{s}_5)\}$	\emptyset	$\{auth\}$	P_4	C_4	$E_4 \oplus \{(S2_Bob.Na, y_1)\}$
6	D	(l_5, \tilde{s}_5)	(l_5, l_6)	$Q_5 \cup \{(l_6, \tilde{s}_6)\}$	\emptyset	$\{auth\}$	P_5	C_5	$E_5 \oplus \{(S2_Bob.A, y_2)\}$
7	D	(l_6, \tilde{s}_6)	(l_6, l_7)	$Q_6 \cup \{(l_7, \tilde{s}_7)\}$	\emptyset	$\{auth\}$	P_6	C_6	$E_6 \oplus \{(S2_Bob.Nb, c_2)\}$
8	D	(l_7, \tilde{s}_7)	(l_7, l_8)	$Q_7 \cup \{(l_8, \tilde{s}_8)\}$	\emptyset	$\{auth\}$	P_7	C_7	$E_7 \oplus \{(IK, IK_7 \cup \{(y_1, c_2, b), pk(y_2)\})\}$
9	D	(l_8, \tilde{s}_8)	(l_8, l_9)	$Q_8 \cup \{(l_9, \tilde{s}_9)\}$	\emptyset	$\{auth\}$	P_8	$C_8 \cup \{(x_{11} = 2)\}$	$E_8 \oplus \{(X_{11}, x_{11})\}$
10	D	(l_9, \tilde{s}_9)	(l_9, l_{10})	$Q_9 \cup \{(l_{10}, \tilde{s}_{10})\}$	\emptyset	$\{auth\}$	P_9	$C_9 \cup \{(IK_8 \vdash \{(c_1, y_1, y_0), pk(a)\})\}$	E_9
11	D	(l_{10}, \tilde{s}_{10})	(l_{10}, l_{11})	$Q_{10} \cup \{(l_{11}, \tilde{s}_{11})\}$	\emptyset	$\{auth\}$	$P_{10} \cup \{y_3\}$	C_{10}	$E_{10} \oplus \{(S2_Alice.Nb, y_3), (S2_Alice.Y_3, y_3)\}$
12	D	(l_{11}, \tilde{s}_{11})	(l_{11}, l_{12})	$Q_{11} \cup \{(l_{12}, \tilde{s}_{12})\}$	\emptyset	$\{auth\}$	P_{11}	C_{11}	$E_{11} \oplus \{(IK, IK_{11} \cup \{(y_3), pk(s_0)\})\}$
13	D	(l_{12}, \tilde{s}_{12})	(l_{12}, l_{13})	$Q_{12} \cup \{(l_{13}, \tilde{s}_{13})\}$	\emptyset	$\{auth\}$	P_{12}	$C_{12} \cup \{(witness(a, y_0), \{y_3\}, pk(y_2))\}$	E_{12}
14	D	(l_{13}, \tilde{s}_{13})	(l_{13}, l_{14})	$Q_{13} \cup \{(l_{14}, \tilde{s}_{14})\}$	\emptyset	$\{auth\}$	P_{13}	$C_{13} \cup \{(w_0 = 1)\}$	$E_{13} \oplus \{(X_0, x_0)\}$
15	D	(l_{14}, \tilde{s}_{14})	(l_{14}, l_{15})	$Q_{14} \cup \{(l_{15}, \tilde{s}_{15})\}$	\emptyset	$\{auth\}$	$P_{14} \cup \{y_4\}$	$C_{14} \cup \{(c_0, y_4, i), pk(a)\}$	$E_{14} \oplus \{(S1_Alice.Y_1, y_4)\}$
16	D	(l_{15}, \tilde{s}_{15})	(l_{15}, l_{16})	$Q_{15} \cup \{(l_{16}, \tilde{s}_{16})\}$	\emptyset	$\{auth\}$	P_{15}	C_{15}	$E_{15} \oplus \{(S1_Alice.Nb, y_4)\}$
17	D	(l_{16}, \tilde{s}_{16})	(l_{16}, l_{17})	$Q_{16} \cup \{(l_{17}, \tilde{s}_{17})\}$	\emptyset	$\{auth\}$	P_{16}	C_{16}	$E_{16} \oplus \{(IK, IK_{16} \cup \{(y_4), pk(i)\})\}$
18	D	(l_{17}, \tilde{s}_{17})	(l_{17}, l_{18})	$Q_{17} \cup \{(l_{18}, \tilde{s}_{18})\}$	\emptyset	$\{auth\}$	P_{17}	$C_{17} \cup \{(witness(a, i, \{y_4\}), pk(i))\}$	E_{17}
19	D	(l_{18}, \tilde{s}_{18})	(l_{18}, l_{19})	$Q_{18} \cup \{(l_{19}, \tilde{s}_{19})\}$	\emptyset	$\{auth\}$	P_{18}	$C_{18} \cup \{(IK \vdash \{c_2\}, pk(b))\}$	E_{18}
20	L	(l_{19}, \tilde{s}_{19})	-	Q_{19}	$(l_{19}, l_{20}) : S2_Bob.A = i$	$\{auth\}$	P_{19}	C_{19}	E_{19}
21	C	(l_{19}, \tilde{s}_{19})	(l_{19}, l_{20})	Q_{18}	$A_{20} \cup \{l_{19} : S2_Bob.A = i\}$	$\{auth\}$	P_{20}	C_{20}	E_{20}
22	L	(l_{18}, \tilde{s}_{18})	-	Q_{18}	$A_{21} \cup \{(l_{18}, l_{19}) : S2_Bob.A = i\}$	$\{auth\}$	P_{21}	C_{21}	E_{21}
23	C	(l_{18}, \tilde{s}_{18})	(l_{18}, l_{19})	Q_{17}	$A_{22} \cup \{l_{18} : S2_Bob.A = i\}$	$\{auth\}$	P_{22}	C_{22}	E_{22}
...									
33	C	(l_{14}, \tilde{s}_{14})	(l_{14}, l_{15})	Q_{27}	$A_{32} \cup \{l_{14} : S2_Bob.A = i\}$	$\{auth\}$	P_{32}	C_{32}	E_{32}

Table 7.1. Execution of the algorithm on the control flow graph for the protocol NSL.

the SiL programs defining the protocol NSL (see example 7.4), summarized in Table 7.1.

Step 0: The first rule applied is *Init*, which initializes the symbolic data state to the triple $(\{(l_0, \xi_0 = (P_0, C_0, E_0))\}, \emptyset, \{auth\})$ where $P_0 = \emptyset$, $C_0 = \emptyset$ and $E_0 = \emptyset$. The initial set of queries contains only one symbolic state, consisting in the initial location of the program, an empty set of constraints and an empty environment. The set of annotations is initially empty, while *auth* represents all the authentication goal locations in the graph of the running example NSL.

Step 1: After the initialization phase, to start the (symbolic) execution of the specification, the *Decide* rule is applied to the query (l_0, ξ_0) with respect to the edge (l_0, l_1) and I obtain: $(Q_1 = \{(l_0, \xi_0), (l_1, \xi_1 = (P_1, C_1, E_1))\}, \emptyset, \{auth\})$, where:

$$\begin{aligned} P_1 &= P_0 \cup \{y_1\} \\ C_1 &= C_0 \\ E_1 &= E_0 \oplus \{(S2_Alice.B, y_0), (S2_Alice.Na, c_1), \\ &\quad (S2_Alice.Actor, a), (S2_Alice.Y_0, y_0), \\ &\quad (S2_Bob.Actor, b), (S1_Alice.Na, c_0), \\ &\quad (S1_Alice.Actor, a), (S1_Alice.B, i), \\ (IK, IK_1 &= \{a, b, i, pk(a), pk(b), pk(i), inv(pk(i))\})\} \end{aligned}$$

This corresponds to a symbolic data state where only the variables defined by the instantiation of the scenario are initialized.

Step 2: I apply *Decide* on the query (l_1, ξ_1) with respect to the edge (l_1, l_2) and I obtain $(Q_2 = Q_1 \cup \{(l_2, \xi_2 = (P_2, C_2, E_2))\}, \emptyset, \{auth\})$, where:

$$\begin{aligned} P_2 &= P_1 \\ C_2 &= C_1 \\ E_2 &= E_1 \oplus \{(Alice_1.Na, c_1), \\ (IK, IK_2 &= IK_1 \cup \{c_1, a\}_{pk(y_1)}\}) \end{aligned}$$

Here, I have used the notation $E \oplus E'$ to denote the set $E' \cup \{(A, v) \mid (A, v) \in E' \text{ and there is no } (A, v') \in E' \text{ for any } v'\}$. In other words, the operator \oplus updates, and possibly extends, the environment E with the information contained in E' .

The algorithm keeps performing a symbolic execution, by applications of the rule *Decide*, as long as there are outgoing edges that are not blocked. In this case, I arbitrarily choose to follow the branch towards l_3 because other three non-blocked edges, representing other session instances, could be followed.

Step 3: I apply *Decide* on the query (l_2, ξ_2) with respect to the edge (l_2, l_3) obtaining $(Q_3 = Q_2 \cup \{(l_3, \xi_3 = (P_3, C_3, E_3))\}, \emptyset, \{auth\})$, where:

$$\begin{aligned} P_3 &= P_2 \cup \{x_1\} \\ C_3 &= C_2 \cup \{x_1 = 3\} \\ E_3 &= E_2 \oplus \{(X_1, x_1), \\ (IK, IK_3 &= IK_2)\} \end{aligned}$$

The constraint $x_1 = 3$ is used to keep track of the session chosen by the algorithm during the execution of the path. Here the session is between two honest agents a and b for Alice and Bob respectively.

Step 4: I apply *Decide* on the query (l_3, ξ_3) with respect to the edge (l_3, l_4) and I obtain $(Q_4 = Q_3 \cup \{(l_4, \xi_4 = (P_4, C_4, E_4))\}, \emptyset, \{auth\})$, where:

$$\begin{aligned} P_4 &= P_3 \cup \{y_1, y_2\} \\ C_4 &= C_3 \cup \{IK_2 \vdash \{y_1, y_2\}_{pk(b)}\} \\ E_4 &= E_3 \oplus \{(S2_Bob.Y_1, y_1), (S2_Bob.Y_2, y_2), \\ &\quad (IK, IK_4 = IK_3)\} \end{aligned}$$

Note that here, in order to be able to derive the encrypted message $\{y_1, y_2\}_{pk(b)}$ (corresponding to the SiL message $\{S2_Bob.Y_1, S2_Bob.Y_2\}_{pk(S2_Bob.Actor)}$ of Example 7.4) from the intruder knowledge, it is necessary that either the constant b (constant that identifies the agent Bob) or i (agent constant for the intruder) is assigned to the agent parameter y_0 . In the first case the intruder can directly use the message $\{c_1, a\}_{pk(y_0)}$ of IK_2 while in the latter it can decrypt the message in IK_2 and re-encrypt it with the public key of Bob ($pk(b)$). It follows that, in terms of the parameters that we are using, the symbolic data state s_4 implies $(y_0 = b \wedge y_1 = c_1 \wedge y_2 = a) \vee y_0 = i$.

Step 5: I apply *Decide* on the query (l_4, ξ_4) with respect to the edge (l_4, l_5) and I get $(Q_5 = Q_4 \cup \{(l_5, \xi_5 = (P_5, C_5, E_5))\}, \emptyset, \{auth\})$, where:

$$\begin{aligned} P_5 &= P_4 \\ C_5 &= C_4 \\ E_5 &= E_4 \oplus \{(S2_Bob.Na, y_1), \\ &\quad (IK, IK_5 = IK_4)\} \end{aligned}$$

Step 6: I apply *Decide* on the query (l_5, ξ_5) with respect to the edge (l_5, l_6) and I obtain $(Q_6 = Q_5 \cup \{(l_6, \xi_6 = (P_6, C_6, E_6))\}, \emptyset, \{auth\})$, where:

$$\begin{aligned} P_6 &= P_5 \\ C_6 &= C_5 \\ E_6 &= E_5 \oplus \{(S2_Bob.A, y_2), \\ &\quad (IK, IK_6 = IK_5)\} \end{aligned}$$

Step 7: I apply *Decide* on the query (l_6, ξ_6) with respect to the edge (l_6, l_7) and I obtain $(Q_7 = Q_6 \cup \{(l_7, \xi_7 = (P_7, C_7, E_7))\}, \emptyset, \{auth\})$, where:

$$\begin{aligned} P_7 &= P_6 \\ C_7 &= C_6 \\ E_7 &= E_6 \oplus \{(S2_Bob.Nb, c_2), \\ &\quad (IK, IK_7 = IK_6)\} \end{aligned}$$

Step 8: I apply *Decide* on the query (l_7, ξ_7) with respect to the edge (l_7, l_8) and I obtain $(Q_8 = Q_7 \cup \{(l_8, \xi_8 = (P_8, C_8, E_8))\}, \emptyset, \{auth\})$, where:

$$\begin{aligned} P_8 &= P_7 \\ C_8 &= C_7 \\ E_8 &= E_7 \oplus \{(IK, IK_8 = IK_7 \cup \{\{y_1, c_2, b\}_{pk(y_2)}\})\} \end{aligned}$$

Step 9: I apply *Decide* on the query (l_8, ξ_8) with respect to the edge (l_8, l_9) and I obtain $(Q_9 = Q_8 \cup \{(l_9, \xi_9 = (P_9, C_9, E_9))\}, \emptyset, \{auth\})$, where:

$$\begin{aligned} P_9 &= P_8 \\ C_9 &= C_8 \cup \{x_{11} = 2\} \\ E_9 &= E_8 \oplus \{(X_{11}, x_{11}), \\ &\quad (IK, IK_9 = IK_8)\} \end{aligned}$$

The interpolation algorithm keeps track of the choosen session.

Step 10: I apply *Decide* on the query (l_9, ξ_9) with respect to the edge (l_9, l_{10}) and I obtain $(Q_{10} = Q_9 \cup \{(l_{10}, \xi_{10} = (P_{10}, C_{10}, E_{10}))\}, \emptyset, \{auth\})$, where:

$$\begin{aligned} P_{10} &= P_9 \\ C_{10} &= C_9 \cup \{IK_8 \vdash \{c_1, y_1, y_0\}_{pk(a)}\} \\ E_{10} &= E_9 \oplus \{(IK, IK_{10} = IK_9)\} \end{aligned}$$

Note that here, in order to derive the encrypted message $\{c_1, y_1, y_0\}_{pk(a)}$, the intruder cannot directly forward a message of his knowledge IK . This because the only message he could use is $\{y_1, c_2, b\}_{pk(y_2)}$ obtained in step 8. However, the constraint $c_1 = y_1 \wedge y_1 = c_2$ is unsatisfiable because $c_1 \neq c_2$. This means the intruder is not able to merely forward messages he has received.

Step 11: I apply *Decide* on the query (l_{10}, ξ_{10}) with respect to the edge (l_{10}, l_{11}) and I obtain $(Q_{11} = Q_{10} \cup \{(l_{11}, \xi_{11} = (P_{11}, C_{11}, E_{11}))\}, \emptyset, \{auth\})$, where:

$$\begin{aligned} P_{11} &= P_{10} \cup \{y_3\} \\ C_{11} &= C_{10} \\ E_{11} &= E_{10} \oplus \{(S2_Alice.Nb, y_3), (S2_Alice.Y_3, y_3), \\ &\quad (IK, IK_{11} = IK_{10})\} \end{aligned}$$

Step 12: I apply *Decide* on the query (l_{11}, ξ_{11}) with respect to the edge (l_{11}, l_{12}) and I obtain $(Q_{12} = Q_{11} \cup \{(l_{12}, \xi_{12} = (P_{12}, C_{12}, E_{12}))\}, \emptyset, \{auth\})$, where:

$$\begin{aligned} P_{12} &= P_{11} \\ C_{12} &= C_{11} \\ E_{12} &= E_{11} \oplus \{(IK, IK_{12} = IK_{11} \cup \{\{y_3\}_{pk(y_0)}\})\} \end{aligned}$$

Step 13: I apply *Decide* on the query (l_{12}, ξ_{12}) with respect to the edge (l_{12}, l_{13}) and I obtain $(Q_{13} = Q_{12} \cup \{(l_{13}, \xi_{13} = (P_{13}, C_{13}, E_{13}))\}, \emptyset, \{auth\})$, where:

$$\begin{aligned} P_{13} &= P_{12} \\ C_{13} &= C_{12} \cup \{\text{witness}(a, y_0, \{y_3\}_{pk(y_2)})\} \\ E_{13} &= E_{12} \oplus \{(IK, IK_{13} = IK_{12})\} \end{aligned}$$

The added constaraint states that the honest agent a has sent to i the message $\{y_4\}_{pk(y_2)}$. This will be used by the algorithm for checking the goal location as explained in Section 7.1.

Step 14: I apply *Decide* on the query (l_{13}, ξ_{13}) with respect to the edge (l_{13}, l_{14}) and I obtain $(Q_{14} = Q_{13} \cup \{(l_{14}, \xi_{14} = (P_{14}, C_{14}, E_{14}))\}, \emptyset, \{auth\})$, where:

$$\begin{aligned} P_{14} &= P_{13} \\ C_{14} &= C_{13} \cup \{(x_9 = 1)\} \\ E_{14} &= E_{13} \oplus \{(X_9, x_9), \\ &\quad (IK, IK_{14} = IK_{13})\} \end{aligned}$$

Step 15: I apply *Decide* on the query (l_{14}, ξ_{14}) with respect to the edge (l_{14}, l_{15}) and I obtain $(Q_{15} = Q_{14} \cup \{(l_{15}, \xi_{15} = (P_{15}, C_{15}, E_{15}))\}, \emptyset, \{auth\})$, where:

$$\begin{aligned} P_{15} &= P_{14} \cup \{y_4\} \\ C_{15} &= C_{14} \cup \{IK \vdash \{\{c_0, y_4, i\}_{pk(a)}\}\} \\ E_{15} &= E_{14} \oplus \{(S1_Alice.Y1, y_4), \\ &\quad (IK, IK_{15} = IK_{14})\} \end{aligned}$$

Step 16: I apply *Decide* on the query (l_{15}, ξ_{15}) with respect to the edge (l_{15}, l_{16}) and I obtain $(Q_{16} = Q_{15} \cup \{(l_{16}, \xi_{16} = (P_{16}, C_{16}, E_{16}))\}, \emptyset, \{auth\})$, where:

$$\begin{aligned} P_{16} &= P_{15} \\ C_{16} &= C_{15} \\ E_{16} &= E_{15} \oplus \{(S1_Alice.Nb, y_4), \\ &\quad (IK, IK_{16} = IK_{15})\} \end{aligned}$$

Step 17: I apply *Decide* on the query (l_{16}, ξ_{16}) with respect to the edge (l_{16}, l_{17}) and I obtain $(Q_{17} = Q_{16} \cup \{(l_{17}, \xi_{17} = (P_{17}, C_{17}, E_{17}))\}, \emptyset, \{auth\})$, where:

$$\begin{aligned} P_{17} &= P_{16} \\ C_{17} &= C_{16} \\ E_{17} &= E_{16} \oplus \{(IK, IK_{17} = IK_{16} \cup \{\{y_4\}_{pk(i)}\})\} \end{aligned}$$

Step 18: I apply *Decide* on the query (l_{17}, ξ_{17}) with respect to the edge (l_{17}, l_{18}) and I obtain $(Q_{18} = Q_{17} \cup \{(l_{18}, \xi_{18} = (P_{18}, C_{18}, E_{18}))\}, \emptyset, \{auth\})$, where:

$$\begin{aligned} P_{18} &= P_{17} \\ C_{18} &= C_{17} \cup \{\text{witness}(a, i, \{y_4\}_{pk(i)})\} \\ E_{18} &= E_{17} \oplus \{(IK, IK_{18} = IK_{17})\} \end{aligned}$$

Step 19: I apply *Decide* on the query (l_{18}, ξ_{18}) with respect to the edge (l_{18}, l_{19}) and I obtain $(Q_{19} = Q_{18} \cup \{(l_{19}, \xi_{19} = (P_{19}, C_{19}, E_{19}))\}, \emptyset, \{auth\})$, where:

$$\begin{aligned} P_{19} &= P_{18} \\ C_{19} &= C_{18} \cup \{IK \vdash \{c_2\}_{pk(b)}\} \\ E_{19} &= E_{18} \oplus \{(IK, IK_{19} = IK_{18})\} \end{aligned}$$

Due to the constraint introduced in this step ($IK \vdash \{c_2\}_{pk(b)}$), the constraint set, in order to be satisfiable, must permit the intruder to produce the ASLan++ message $\{S2_Bob.Nb\}_{pk(S2_Bob.A)}$ instantiated with $S2_Bob.Nb = c_2$. However, the intruder can produce such a message only from the message introduced in step 8: $\{y_1, c_2, b\}_{pk(y_2)}$, because is the only message from which he can extract c_2 . By doing so, the constraint $y_2 = i$ has to be added, otherwise the intruder cannot decrypt the aforementioned message. y_2 corresponds to the ASLan++ variable $S2_Bob.A$ and then the last constraint means that the intruder must play the role of Alice ($S2_Bob.A$) in session 2. This is obviously against the authentication goal that states that both roles Alice and Bob must be played by honest agents.

Steps 20–21: The edge (l_{19}, l_{20}) is labelled with the goal action that impose $S2_Bob.A \neq i$. Since any possible symbolic execution step on the query (l_{19}, s_{19}) with respect to the edge (l_{19}, l_{20}) leads to a blocked symbolic data state ($S2_Bob.a$ is actually i), I apply now the rule *Learn* to obtain $(Q_{19}, \varphi_0 := (l_{19}, l_{20}) : S2_Bob.A = i, \{auth\})$.

Steps 22–33: The interpolation algorithm continues to propagate back the annotation of the previous step by using *Learn* and *Conjoin* until all the edges are blocked to an annotation. To conclude, it is important to highlight that any symbolic step leads to a blocked symbolic data state, i.e., a symbolic data state where the annotation $S2_Bob.A = i$ is entailed. This is a concrete example of how the annotation method can help (and improve) the search procedure.

While with NSL the algorithm concludes with no attacks found, if we consider the original protocol NSPK (i.e., remove Lowe’s addition of “ B ” in the second message of the protocol), then my method reaches the goal location with an execution close to the one I have just provided. In fact, in NSPK, when I compute the step after the 19th, the intruder rules lead to the goal with the inequality $S2_Bob.A \neq i$. This is because the intruder i can perform a man-in-the-middle attack using the initiator entity of the first session in order to decrypt the messages that the receiver sends to i in the second one [53]. To show the attack trace, I first check the path that is used during the algorithm execution to reach the goal location and that is represented by the values of x_j parameters contained in the C_{19} set. In this case, $\{x_{11} = 2, x_9 = 1\} \subseteq C_{19}$, which produces the symbolic attack trace (at state 19 of the algorithm execution) shown in the middle of Figure 7.7.

Now, by using the information in Ξ_{19} , I can instantiate this trace using parameter and constant values, and thus obtain the instantiated attack trace shown on the right of Figure 7.7. I can note from IK_{19} that y_2 has no constraints on the fact that it has to be i , i.e., the intruder acts as if he were an honest agent (under his real name) in the first session, and then I write the concretization as $i(a)$ to show that the intruder is acting as the honest agent a in the second session and this makes possible the man-in-the-middle-attack.

It is not difficult to extract from this instantiated attack trace a test case, which can then be applied to test the actual protocol implementation. In fact, the constraint set contains a sequence of equalities $x_i = n$ that express which is the session to follow at each branch of the executed path. \square

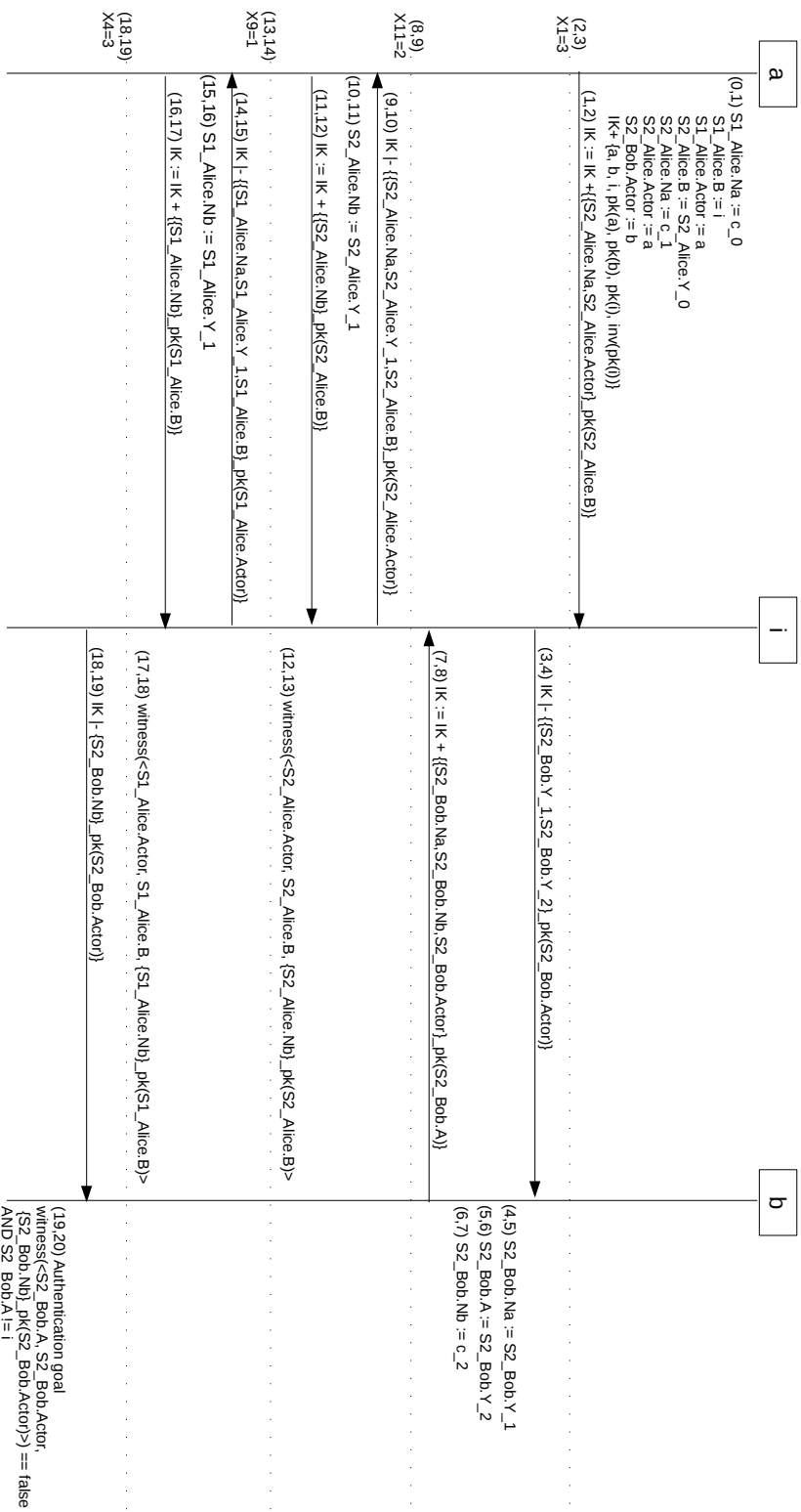


Fig. 7.5. NSL execution path.

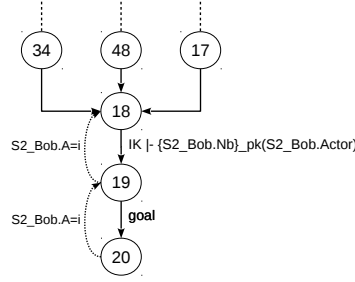


Fig. 7.6. NSL sub-graph.

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

Fig. 7.7. Symbolic attack trace of the Man-in-the-middle attack on NSPK protocol at state 15 of the algorithm execution (left) and instantiated attack trace obtained with interpolation method (right).

The SPiM tool

In order to show that the interpolation method concretely speeds up the validation, I have implemented a Java prototype called *SPiM* (*Security Protocol interpolation Method*), which is available at <http://regis.di.univr.it/spim.php>. As shown in Figure 8.1, SPiM takes an ASLan++ specification as input that is automatically translated into a *CFG* in *SiL* by the translator *ASLan++2SiL*. The *CFG* is then given as input to the *Verification Engine* (*VE*), which verifies the protocol by searching for goal locations that represent attacks on the protocol. *VE* is composed by three main components: (i) *quantifier elimination*, (ii) *Dolev-Yao intruder* and *EUF* (*Equalities and Uninterpreted Functions*) theories, used by (iii) *Z3* [30] and *iZ3* [62] for SAT solving and interpolant generation, respectively.¹ Both *Z3* and *iZ3* are invoked by *SPiA* (*Security Protocol interpolation Algorithm*), which is my implementation of the algorithm in Section 7.2.

More specifically, the *VE* symbolically executes a *CFG* producing a formula, after the execution of each action branching from a node to the following, that represents the symbolic state reached. *Z3* is then used for a satisfiability check on the newly produced formula. When the symbolic execution of a given path fails to reach a goal, the *VE* calls *iZ3* that produces an annotation (i.e., a formula expressing a condition under which no goal can be reached) by using Craig interpolation. By a backtracking phase, *SPiA* propagates the annotation through the *CFG*. Such an annotation is possibly 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 no goal will ever be reached. *SPiM* concludes reporting either all the different reachable attack states (from which abstract attack traces can be extracted) or that no attack has been found for the given specification

¹ The first two components are related to the usage of *Z3* and *iZ3*. In fact, as shown in Section 7.2, my algorithm needs to handle many quantifications and, for performance issues, I have developed a module that unfolds each quantifier over the finite set of possible messages. Moreover, the Dolev-Yao theory has been properly axiomatized (with respect to each formula produced by *SPiA*) in *Z3* and *iZ3*, which do not support it by default.

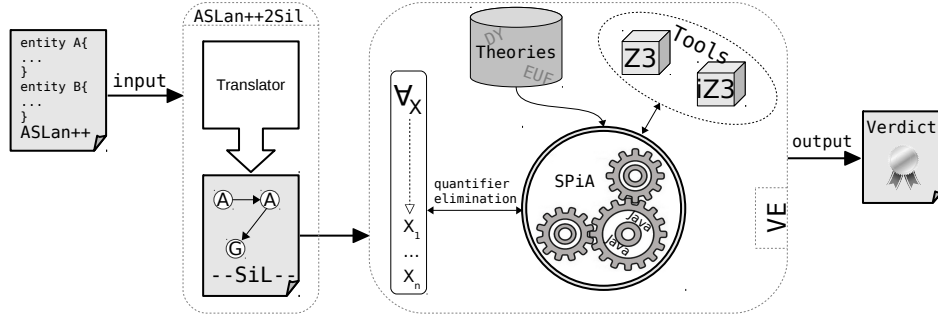


Fig. 8.1. The SPiM tool.

8.1 The architecture of SPiM

I now describe the entire architecture of the SPiM tool, focusing on two package of the tool: translation (ASLan++2SiL, 12 Java classes) and verification (VE, 149 Java classes). As reported in Figure 8.1, the input specification (written in ASLan++) is translated into a SiL specification by the ASLan++2SiL. This translation package implements several features:

- While reading and parsing the ASLan++ specification file it checks if it is well formed with respect to the ASLan++ guidelines.
- For each ASLan++ construct, the ASLan++2SiL constructs an equivalent SiL action (see Section 7.1.3).
- The ASLan++2SiL organizes the SiL code in blocks (as described in Section 7.1.2) and then it builds the CFG.
- The ASLan++2SiL can generate a graphical representation of the CFG in DOT [50] language.

The VE package (namely package “se” in the Java code) implements and runs the modified version of the IntraLA algorithm (SPiA) that performs the model checking and outputs the verdict and the main features are:

- While parsing the SiL specification it creates a domain of messages used during the quantifier elimination phase.
- A quantifier elimination phase where the VE unfolds all the quantifiers over a finite domain of messages.
- Supports two different algorithm for model checking the specification: SPiA (the interpolation algorithm) and Full-Explore (mainly used for comparisons and debug, see Chapter 9).
- It takes advantage of Z3 and iZ3 for satisfiability checking and interpolant generation.

8.1.1 The package VE

In the following I give a description of the classes and interfaces of the VE package, together with UML (Uniform Modeling Language [85]) diagrams of these classes.

Specification

A class which provides all the necessary methods/constructors used by the VE to build the program graph. It also runs the SPiA (or Full-Explore) algorithm using the *run()* method.

Message

is an abstract class which represents the messages exchange in a (security) protocol. The class, as showed in Figure 8.2, is extended by:

- **GenericMessage**, an abstract class which represents generic constants, variables and input variables. It is extended by the classes **GenericConstant**, **GenericVariable** and **InputGenericVariable**.
- **ConcatenatedMessage**, a class which models the concatenation of two objects of type **Message**.
- **Key**, an abstract class (see Figure 8.3) which is used for asymmetric keys, extended by the classes **PrivateKey** and **PublicKey**. Those classes are, in turn, extended respectively by **PrivateKeyConstant**, which represents the constant associated to a private key, and **PublicKeyConstant**, which represents the constant associated to a public key.
- **SymKey**, an abstract class which represents symmetric keys, extended by the classes **SymKeyConstant**, **SymKeyParameter** and **SymKeyVariable**, which represents respectively the constant associated to a symmetric key, a parameter associated to a symmetric key and a variable associated to a symmetric key.
- **Agent**, a class which models agents. It is extended by the classes **AgentConstant**, **AgentVariable** and **InputAgentVariable** and characterized by a **PublicKey** and a **PrivateKey**.
- **EncryptedMessage**, a class which models a message encrypted with a **PrivateKey** or a **PublicKey**.
- **SymEncryptedMessage**, a class which is used for message encrypted with a **SymmetricKey**.
- **Nonce**, an abstract class which represents nonces. It is extended by the classes **InputNonceVariable**, **NonceConstant** and **NonceVariable**.

Evaluable

An interface implemented by the class **Message** and **MessageSet**. The latter class represents a set of messages, used to update the knowledge of the intruder. **Evaluable** is also extended by the interfaces:

- **Variable**, which is used to represent variables such as nonces, symmetric keys, agents and so on. Among the classes which implements this interface, we are interested in the following: **IKVariable**, **AgentVariable**, **SymKeyVariable**, **NonceVariable** and **GenericVariable** (see Figure 8.4).
- **Constant**, which is implemented by the classes **GenericConstant**, **SwitchIntruderConstant**, **BooleanConstant**, **NonceConstant**, **SymKeyConstant** and **AgentConstant** (see Figure 8.5).

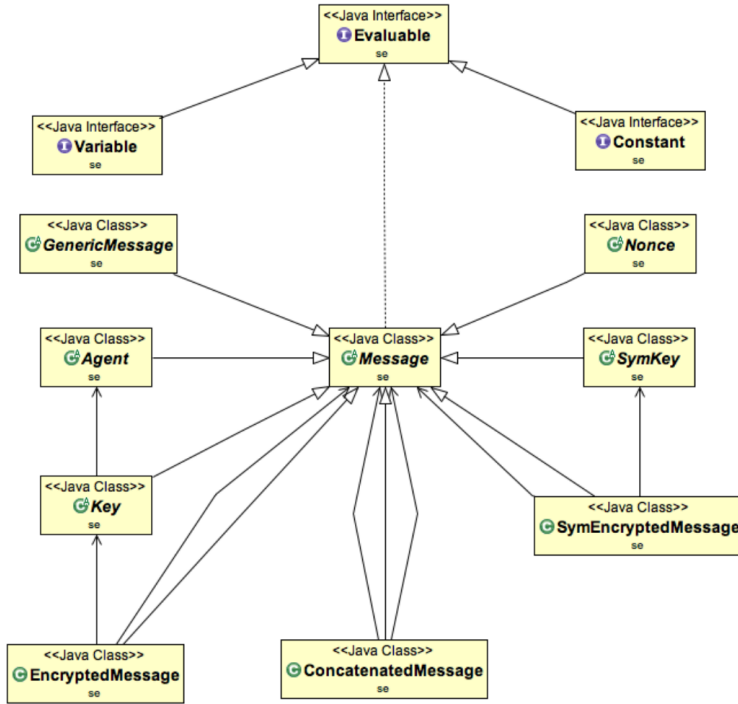


Fig. 8.2. UML class diagram of Message

Action

Action is an abstract class which models the instructions of a SiL program. In Figure 8.6 I present the UML class diagram containing the hierarchy and the relations between the class and interfaces previously presented. In order to model different kind of messages the class is extended by:

- Assignment, a class which is used to represent an assignment of an object of type Evaluable to an object of type Variable.
- Conditional, an abstract class which models a conditional statement. The class is extended by IKConditional, which represents the ability of the intruder to derive a message, and StdConditional, which models disjunctive and conjunctive boolean operators (through the classes ConjunctiveConditional and DisjunctiveConditional), as well as equalities and inequalities (through the classes EqOPConditional and InEqOPConditional).
- IKUpdate, a class which models an update of the intruder knowledge.

8.1.2 The ASLan++2Sil package

The ASLan++2Sil package was created after the core of SPiM (the *se* package). As the source code of SPiM is written in Java, the ASLan++2Sil package was developed as a Java package, named *translator*, into the SPiM project. The package

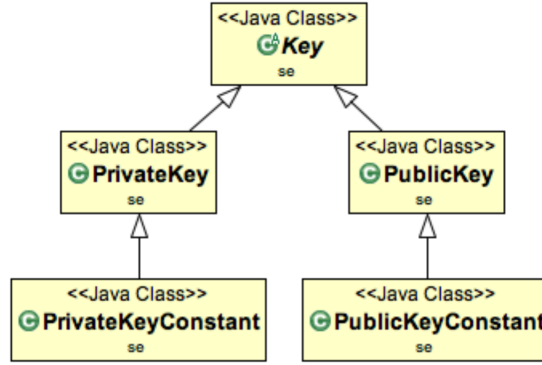


Fig. 8.3. UML class diagram of Key

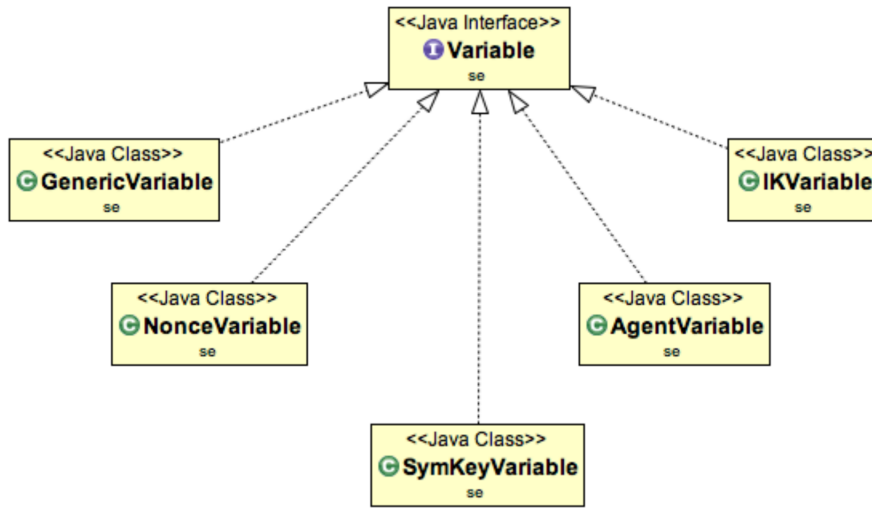


Fig. 8.4. UML class diagram of Variable

contains all necessary classes written for the parsing and for the communication with the VE (ASLanppFilter, Entity, Environment, Main, Session) along with the classes automatically generated by Java Compiler Compiler (JavaCC) [49] (ParseException, Parser, ParserConstants, ParserTokenManager, Token, SimpleCharStream, TokenMgrError). Figure 8.7 outlines the hierarchy and the relations between the classes in the translator package.

Entity class

The class represents a SiL entity, i.e. the translation of an ASLan++ entity. The main fields declared inside are the following:

- **parent**: it points to the parent entity, if present. It is used to represent the nesting of entities and thus the scope of their variables.
- **symbols**: it maps symbol names to the instance of Message (possibly belonging to an ancestor Entity) representing it.

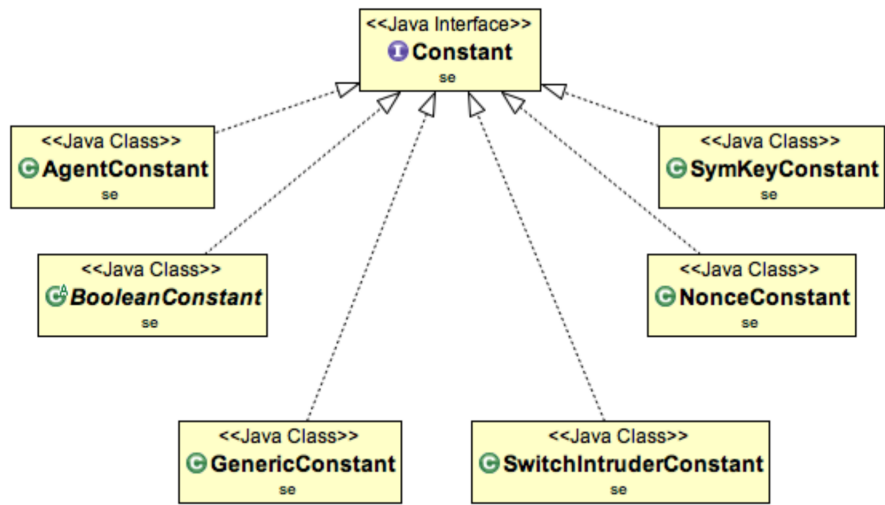


Fig. 8.5. UML class diagram of Constant

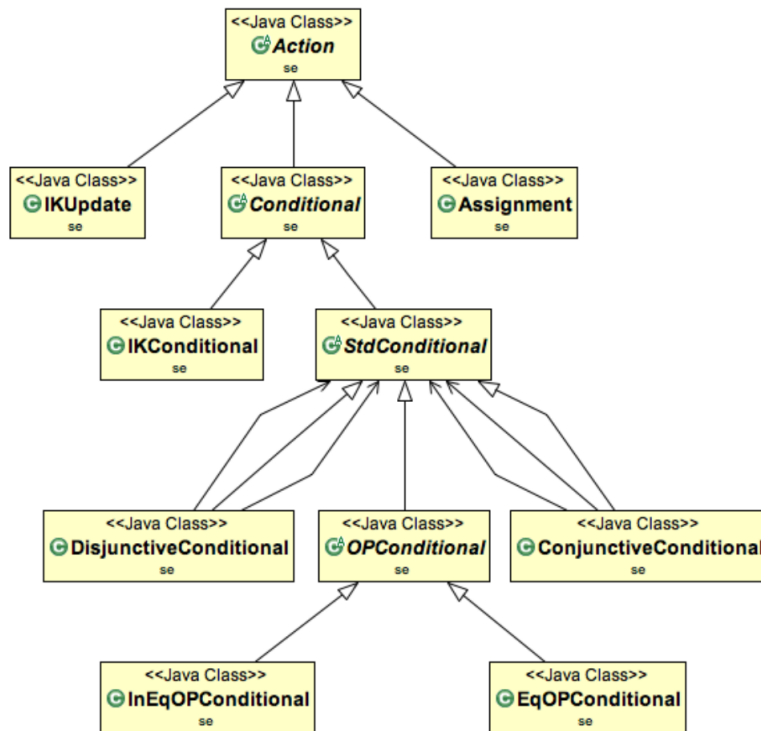


Fig. 8.6. UML class diagram of Action

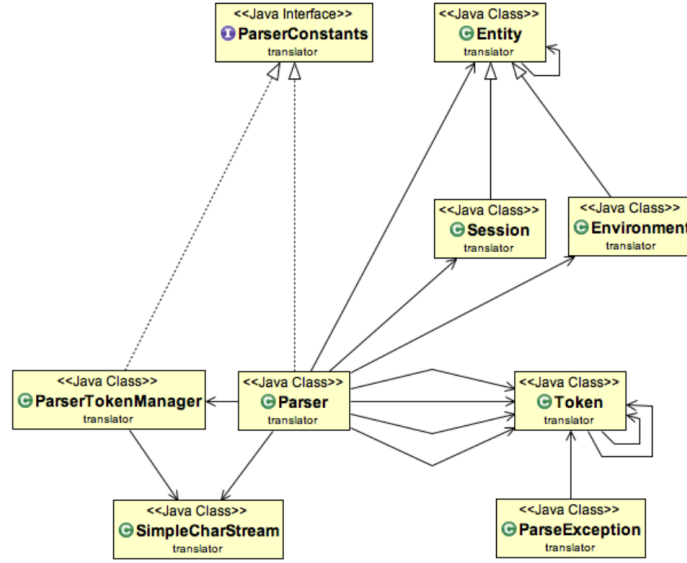


Fig. 8.7. UML class diagram of Translator

- actions: it contains a list with all the SiL actions associated with the body of the entity.
- goalTempActions: temporary data structures needed to add the actions in the proper order when a goal label occurs in a statement.
- tempActions: temporary data structures needed to add the actions in the proper order when the reception of a message occurs.
- name: represents the name of the entity.
- formalParams: it contains a list of entity formal parameters i.e., the parameters contained in the signature of the entity.
- actualParams: it contains a list of actual parameters i.e., session variables instantiated which will be bounded during the entity instantiation to the formal parameters of the entity. They are represented as instances of Message.
- entityGoals: it contains a list of instances of Action and represents the goals of the entity.

Following the information hiding principle [16], the access to all the data fields is restricted through the use of the private keyword. Thus in order to read or write them we must use the corresponding public methods. As shown in Figure 8.7 the classes Session and Environment extend the class Entity. The Session class represents the translation of an ASLan++ symbolic session entity. The main field encapsulated in it is goalMap, which maps a goal name to an array containing the type of the goal (auth or secr) and a list of goal parameters, with the agents involved in the goal. The Environment class represents the root entity, thus it does not have a parent.

Parser

In order to implement the parser there was the possibility to choose whether implementing a new parser from scratch or to use a parser generator tool. The main motivations that has lead to the latter options are:

- *Scalability*, this option keeps the code simpler to understand and hence allows easier future extensions.
- *Performance*, many parser generators have good performance because their overhead (mainly caused by the dirty code added) can be considered as negligible.
- *Complexity*, the grammar of an ASLan++ protocol specification is quite complex, so the resulting parser would have required a lot more time and effort. By adopting JavaCC [49], a popular parser and lexical analyzer generator for use with Java applications the result has been clear (only 13 classes) and easy to implement (JavaCC reads a grammar specification and converts it to a Java program that can recognize matches to the grammar). In addition to the parser generator itself, JavaCC provides a useful debugging tool. Moreover it natively supports EBNF grammars, blending them and Java in a neat and elegant way.

8.1.3 How to use SPiM

The Main class is used to launch SPiM. The launch configuration is:

```
translator.Main [fileName] [-gv (--graphviz) [dir]]
```

If the parameter `fileName` is present it will be interpreted as the ASLan++ specification input file. Otherwise a graphic user interface (GUI) will allow the user to choose the ASLan++ specification input file. In the latter case, I use a particular class (`ASLanppFilter` which in turn extends the class `FileFilter`) in order to filter files with the right extension (`aslan++`). The option `-gv` (or `--graphviz`) permits the generation of a graphical representation of the program graph. In this case, it is possible to specify the “.dot” and “.png” output directory from command line; otherwise a GUI will allow to choose the output directory. Finally, the `Main` method receives the program graph specification by invoking the `translate` method in the `Parser` class and from that specification launches SPiM.

Experiments and results

I considered 7 case studies and compared the analyzes performed by using interpolation-driven exploration (SPiA) and full exploration (*Full-explore*) of the CFG. Full-explore explores the entire graph checking, at each step, if the state is satisfiable or not. If there is an inconsistency, SPiM blocks the execution of the path resuming from the first unexplored path, until it has explored all paths. I could have modified the Full-explore algorithm and checked for inconsistencies at the end of the path instead of at any step but this would have lead to an unfair comparison. In fact, a similar improvement could have been implemented also for SPiM, but then it would have been difficult to distinguish between the steps pruned by interpolation and those pruned by these improvements.

Table 9.1 shows the results obtained (with a general purpose computer), by making explicit the time required for symbolic execution steps (applications of *Decide*) and for interpolant generation (applications of *Learn*). The usage of SPiA has allowed a speed up of the validation (in the context of security protocols, i.e, using the Dolev-Yao intruder) by (i) reducing the number of states to explore and then (ii) lowering the execution time. The relation between (i) and (ii) is due to the fact that the time needed to perform a *Decide* is comparable to the one required to perform a *Learn*, and the time used to propagate the annotations (*Conjoin* rule) is negligible. For example, the time needed to symbolically execute a (sub-)path twice, using Full-explore, is comparable to the time used to execute and annotate the same (sub-)path. But from that point on, if the annotation blocks the execution, only the Full-explore will execute that (sub-)graph again. I have observed that, in the case studies analyzed, the annotations block the executions of all those (sub-)paths that do not reach a goal location, thus ensuring a clear improvement of the performances. In particular, when applying a *Decide* moving from a node l_1 to a node l_2 , I generate a formula, which describes the state of the execution at node l_2 and the axiomatization of the Dolev-Yao theory; this formula is then given to Z3 that “decides” whether it is satisfiable or not. On the other hand, in order to execute a *Learn* between the same S_1 and S_2 , I translate the state S_1 with the axiomatized Dolev-Yao theory into a formula α and the semantics of the action a together with all previous annotations into a formula β . In order to find an interpolant I use iZ3 that performs a satisfiability check on the formula $\alpha \wedge \beta$ (very similar to what a *Decide* would do) and from the refutation by resolution steps an interpolant can be calculated in linear time [59, 60]. Finally,

Specification (sessions)	SPiA: Decide+Learn (time)	Full-explore: Decide (time)	Result
ISO6 (ab,ab)	311+274 (205m6s)	467* (278m12s)	no attack found
NSL (ab,ab)	257+234 (57m37s)	631 (173m7s)	no attack found
NSL (ai,ab)	89+22 (1m30s)	119 (1m49)	no attack found
NSPK (ab,ab)	257+234 (26m5s)	631 (76m20s)	no attack found
NSPK (ai,ab)	101+22 (0m56s)	123 (0m51s)	attack found
Helsinki (ab,ab)	311+274 (112m7s)	660* (261m47s)	no attack found
Helsinki (ai,ab)	167+88 (13m41)	407 (46m44s)	attack found

Table 9.1. SPiA vs Full-explore.

the *Conjoin* rule propagates these interpolants without performing other satisfiability checks.

Empirically, the more the CFG grows the more the annotations prune the search space. This is due to the fact that the number of states pruned by interpolation is usually related to the size of a CFG; this is confirmed by the results in Table 9.1 and, in particular, by the case studies for which Full-explore has not concluded the execution (marked with an asterisk).

I have also compared the SPiM tool with the three state-of-the-art model checkers for security protocols that are part of the AVANTSSAR platform [4]: CL-AtSe [97], OFMC [14] and SATMC [7].¹ Not surprisingly, Table 9.2 shows that their average computational times of execution are in general better than mine. This is thanks to several speed-up techniques implemented by these model checkers and to empirical conditions that can stop the execution (both not implemented yet in SPiM). In Table 9.2, I have also reported the number of transitions and/or nodes reached during the validations with the exception of SATMC, which does not report them as output. However, for each safe specification (in which no attacks are found), SATMC reached the maximum number of steps (80) permitted as default and the reported timings are comparable to SPiM's for some specifications; in the case when they are not comparable, it is interesting to observe that SPiM executes a number of rules much higher than 80. For both CL-AtSe and OFMC, on safe specifications, the number of transitions and nodes explored is, in most cases, higher than the number of rules (transitions) of SPiM (Table 9.1). On unsafe specifications (where an attack is found), these numbers seem to be in disfavor of SPiM but this is because SATMC, OFMC and CL-AtSe stop their executions once a goal is found, while SPiM searches for every possible attack trace in the CFG (i.e., SPiM features a multi-attack-trace support).

I remark that the aim of SPiM is mainly to show that Craig interpolation can be used as a speed-up technique also in the context of security protocols and not (yet)

¹ For this comparison, given that all these tools support ASLan++, I have used the same input files and I have also used the same general purpose computer used to generate the results in Table 9.1. I have considered OFMC v2012c, which is the last version that supports ASLan++ although it only supports untyped analysis, while for SATMC and CL-AtSe I have considered versions 3.4 and 2.5-21, respectively, which support typed analysis as SPiM does. Note that the times reported in Table 9.1 also consider the translation from ASLan++ to CFG (usually several seconds), while in Table 9.2 I do not report the translation time from ASLan++ to ASLan (the input supported by the three tools), usually less than one second.

Specification (sessions)	SATMC (v.3.4)	CL-AtSe (v.2.5-21)			OFMC (v.2012c)		Result
	time	transitions	states	time	nodes	time	
ISO6 (ab,ab)	6.318s	452	236	0.034s	8432	3.804s	no attack found
NSL (ab,ab)	14m28s	794	534	0.052s	3236	3.295s	no attack found
NSL (ai,ab)	6m51s	93	69	0.015s	575	0.327s	no attack found
NSPK (ab,ab)	14m10s	794	534	0.053s	8180	3.208s	no attack found
NSPK (ai,ab)	1m56s	14	10	0.014s	96	0.134s	attack found
Helsinki (ab,ab)	7.01s	794	534	0.061s	8180	3.795s	no attack found
Helsinki (ai,ab)	50.8s	14	10	0.017s	96	0.121s	attack found

Table 9.2. SATMC, CL-AtSe and OFMC.

to propose an *efficient* implementation of a model checker for security protocol verification. In fact, I do not see my approach as an alternative to such more mature and widespread tools, but I actually expect some interesting and useful interaction. For example, CL-AtSe implements many optimizations, like simplification and rewriting of input specifications, and OFMC implements a specific technique, called *constraint differentiation*, which considerably prunes the state space as well as some optimizations at the intruder level. I do not see any incompatibility in using interpolation together with such optimization techniques, as interpolation works on reducing the search space by excluding some paths during the analysis (while, e.g., constraint differentiation prunes the state space by not considering the same state twice). The only possible side effect that I foresee is that the number of paths pruned by interpolation could decrease when I use it in combination with other optimization techniques. In general, however, although I do not have experimental evidence yet, I expect that if enhanced with such techniques, SPiM could then reach even higher speed-up rates. I am currently working in this direction.

Discussion and related work

I believe that my interpolation-based method, together with its prototype implementation in the SPiM tool and my experimental evaluation, shows that I can indeed use interpolation to reduce the search space and speed up the execution also in the case of security protocol verification. In particular, as I have shown, I can use a standard security protocol specification language (ASLan++, but, I believe with little effort, also other languages that specify the different protocol roles as interacting processes) and translate automatically into SPiM's input language SiL with the guarantee that in doing so I will not introduce nor lose any attack. The tool then proceeds automatically and concludes reporting either all the different reachable states (from which one or more abstract attack traces can be extracted) or that no attack has been found for the given specification.

In [61], McMillan presented the IntraLA algorithm that I have used as a basis for this work. However, my application field is network security while IntraLA has been developed for software verification and this has led to a number of substantial differences between the two works. My case studies are security protocols, and thus parallel programs, while IntraLA works on sequential ones. I have then defined a language to create CFGs and provided a translation procedure from protocol specifications, e.g., expressed in ASLan++, into my CFG language (proving the correctness of the translation with respect to the semantics of ASLan++). In particular, I have defined, for my CFGs, statements to handle the actions of the Dolev-Yao intruder and I have then used the Dolev-Yao theory in both the symbolic execution of the CFG (Decide rule, Section 7.2) and for interpolants generation (Learn rule, Section 7.2). My goals also differ from the ones in [61]; in fact, I define security goals like authentication and integrity. The same differences can be found between SPiM and IMPACT II (the implementation of [61]). IMPACT II takes as input CFGs from C program and has been tested on drivers source codes. The algorithm implementations do also have some differences. In particular, in SPiA, I have implemented an optimization, according to which an interpolant is calculated, at a given node or edge, only when the graph presents an unexplored path that can be blocked by such an interpolant.

Besides McMillan's works on interpolation applied to model checking, there are a wide number of model checkers that implements different techniques to speed-up the search of goal locations. In particular, for the purpose of the comparison with SPiM, I consider here a list of model checkers that implements the Dolev-

Yao intruder theory (and not already considered in Chapter 9): Maude-NPA [36], ProVerif [18] and Scyther [29].

Maude-NPA is a analysis tool for cryptographic protocols that supports a wide range of theories (besides the Dolev-Yao one) such as associative commutative plus identity. It has been implemented with particular focus on performances and in fact, during the analysis, it takes advantage of various state space reduction techniques. They ranges from a modified version of the lazy intruder (called, super lazy intruder) to a partial order reduction technique. We can notice that the idea behind the speed-up techniques of Maude-NPA are very similar to the one of SPiM: reducing the number of state to explore and try to not explore a state after having the evidence that from this state the model checker will never reach the goal location (i.e., the initial state given that Maude-NPA performs backward reachability search). As for all the AVANTSSAR's model checker (as already reported in Chapter 9), I do not see any incompatibility in implementing the interpolation based technique I have proposed in this thesis as a speed up technique in Maude-NPA along side the ones it already implements. However, Maude-NPA performs backward reachability analysis while my technique has been defined for forward reachability based approaches. This does not prevent possible useful interaction between the two approaches but that this would require further work and possibly a modification of the interpolation algorithm.

ProVerif is a model checker used in the analysis of cryptographic protocols that implements the Dolev-Yao intruder model that has been implemented to be a very efficient model checker. It proposes an abstract representation of the protocol by using Prolog rules in order to handle multiple executions of a protocol. It also implements an efficient algorithm that, combined with an unification technique along with rules optimization procedures, avoids the problem of state space explosion. It is not clear if ProVerif could benefit from my interpolation based technique given that the author in [18] states that the state space explosion has been avoided in Proverif.

Scyther is an efficient tool for the automatic verification of security protocols. As stated in [29] it is based on pattern refinement algorithm, providing concise representations of (infinite) sets of traces. It does not use approximation methods nor abstraction techniques and then it could benefits from implementing my interpolation technique, in particular, when unbounded verification is performed. However, as for Maude-NPA, due to Scyther backward searching algorithm, this integration would require further study.

I am not aware of any other tool for security protocol verification that uses a Craig interpolation-based speed-up technique, and as I reported above I believe that actually interpolation might be proficiently used in addition (and not in alternative) to other optimization techniques for security protocol verification. I am thus currently investigating possible useful interactions between interpolation and such optimization techniques, given that there are no theoretical or technical incompatibilities between them. Symmetrically, it would be interesting to investigate also whether more mature tools might benefit from the integration of interpolation-based techniques such as mine to provide an additional boost to their performance. This will of course be a much more challenging endeavor to undertake, as it will possibly require some internal changes to already deployed tools.

Runtime security verification of web applications

Introduction

It is of utmost importance to help web applications (or systems in general) producers in avoiding vulnerabilities in the production phase by using design time security verification techniques, e.g. as for security protocols in Part III. This is, however, not sufficient to guarantee the security over the entire life cycle of a system (e.g., web applications). There are, in fact, a lot of implementation flaws (e.g., SQL-injections, buffer overflow, Cross-site Request Forgery and such) that can be introduced by mistake during the implementation phase. Therefore, security validation should be applied not only at production time (as we have already seen in Part III) but also at provision and consumption time when a system is used by administrators and users in general, as we have already discussed in Part II. In particular, I will focus the analysis to web applications since web applications can be big structured systems with many different protocols and servers underlying the web application itself. In this scenario, manual testing is a highly error prone activity and relays on the ability and expertise of the analyst, i.e., the more experienced the analyst is the more likely he will find vulnerabilities if any are present.

HTTP and HTTPS, the dominant web access protocols, are stateless and web servers therefore use *cookies*, among other means, to keep track of their sessions with web clients. Cookies are stored by the client's browser; whenever the client sends an HTTP(S) request to the web server, the browser automatically attaches to the request the cookie that is originated from the web server. This mechanism allows the clients to experience a seamless stateful web browsing, while in fact using an inherently *stateless protocol* such as HTTP.

Cross-site request forgery attacks (CSRF) exploit the aforementioned mechanism of automatically-attached cookies. A typical CSRF occurs as explained in the following. The attacker tricks the client into accessing a sensitive web server by making a *rogue URL link* available to the client: the link instructs the web server to perform a transaction on behalf of the client (e.g. to transfer money). If the client accesses the web server through the rogue link, then in effect the client requests the web server to perform the transaction. The only missing part of the puzzle is a valid cookie that needs to be attached to the request, so that the web server authenticates the client.

Now, if it happens that the client accesses, via the rogue link, the web server *while* a session between the client and the web server is active, then the client's browser automatically attaches the proper cookie to the request. The web server would then

accept the attacker-generated request as one genuinely sent by the client, and the attack is successful. The web server can deter the attack by checking that critical requests are in fact generated by the client: the requests may have to include an extra *random value* that is only known to the client and the web server passed as a POST parameter, the web server might prompt the client to solve a CAPTCHA to demonstrate that he is aware of the transaction taking place, etc.

However, a number of related vulnerabilities and design flaws might render such countermeasures against CSRF useless. Due to the complexity of modern web applications¹, those vulnerabilities might be difficult to spot. For instance, if the web server uses poorly generated random values, the attacker may open simultaneous sessions with the web server, record the random values, and infer their pattern. It is also well known [33] that state-of-the-art vulnerability scanners do not detect vulnerabilities linked to logical flaws of applications. In general, one should proceed with care when assessing the security of productive servers for vulnerabilities with potential side-effects such as CSRF, since one might affect the integrity of data, making manual testing a challenging task.

To address these problems, I propose a *model-based technique* in order to detect issues related to CSRF during the design-phase. The essence of the formal model is simple: the client acts as an *oracle* for the attacker. The attacker sends a URL link to the client and the client follows the link. The bulk of the model is therefore centered around the web server, which might have envisioned various protection mechanisms against CSRF vulnerability exploitation. The attacker, in my formal model, is allowed to interact with the web server and exhaustively search his possibilities to exploit a CSRF. The expected result of this technique is, when a CSRF is found, an abstract attack trace reporting a list of steps an attacker has to follow in order to exploit the vulnerability. Otherwise, the specification is safe (under a number of assumptions, as described in Section 14.2) with respect to CSRF. I demonstrate the effectiveness and the usefulness of my method through a made-up example (Section 14.3) and three real-world case studies (Chapter 15): DocumentRepository and EUBank (two anonymized real-life applications) and WebAuth [87].

More specifically, I propose a model-based analysis technique that (i) extends the usage of state of the art model checking technology for security to search for CSRF based on the ASLan++ language [101]. I also (ii) investigate the usage of the intruder, à la Dolev-Yao [32] for detecting CSRF on web applications (while it is usually used for security protocols analysis) and, finally, I (iii) show how to concretely use the technique with real web applications, reporting two CSRF attacks. In my most extensive case study I report a CSRF on a major EU Bank web site that permits a malicious bank transfer.

Structure.

In Chapter 12, I give an introduction on model-based testing applied to web applications where I describe the role of the Dolev-Yao intruder in the formal verification of web applications. In particular, I show the results of two experiments on 37 ASLan++ specifications. In Chapter 13, I give a general overview of CSRF. In Chapter 14, I

¹ A web application is a software application hosted on one or more web servers that run in a web browser.

describe how to model a web application in order to search for CSRF and in Chapter 15 I present three case studies, and discuss my findings. In Chapter 16, I discuss related work and some concluding remarks proposing future research directions.

Dolev-Yao intruder and web applications

In the context of the SPaCIoS project I have used model checkers (e.g., SATMC [7]) that implement the Dolev-Yao [32] intruder for generating test cases Section 4.2. While this attacker has been widely used for validating security cryptographic protocol specifications, there are indications that Dolev-Yao is not suited for web applications [2, 12] that are in contrast with the work I will present afterward in this chapter (Part IV). This is because the Dolev-Yao intruder is not well suited for searching some web attacks (e.g., SQL injection) and I will show when and how to use the Dolev-Yao intruder for searching for attacks on web applications.

Many threats on the web are not linked to the system specification (i.e., are not logical flaws) but to implementation issues. However, a web application is not isolated from communication/network protocols but constantly interacts with them. To determine whether or not a Dolev-Yao intruder can be useful in this kind of analysis, before defining the entire process, I have investigated all the variations/mutations of a SPaCIoS case study (Webgoat [76]), and tested if the specification goals can be reached with and/or without the Dolev-Yao intruder model.

I have tested 37 ASLan++ specifications and mutations of them in two different ways. The first experiment has been performed using SATMC with and without the entire intruder deduction system and the second one just excluding encryption/decryption from the set of rules of the intruder in SATMC.

Experiment I — Excluding the deduction system of Dolev-Yao intruder.

I describe here the result of testing every specification of the Webgoat case study in SPaCIoS using SATMC with and without the intruder deduction system. This experiment has been done using a beta version of SATMC 3.5 that is not publicly available yet but the developers have provided me for this specific purpose. In this beta version there is the option `--intruder_rules` that if set to `false` excludes the entire deduction system of the intruder. Unfortunately this option gives conflicts with the step compression procedure of SATMC, which I have excluded with `--sc=false`. The only side effect of this conflict is at computational level, i.e. SATMC's performance decreases, but it has not afflicted the results. Another option I have set is a threshold to decide the maximum amount of time after which a validation can be considered as inconclusive. In this experiment the time limit for inconclusive results has been

(empirically) set to 300 seconds, i.e. after 300 seconds a SIGTERM signal is sent to the process.

The results of this experiment is summarized in Table 12.1 which reports either the SATMC output or an empty field for timeout for each specifications for the two considered cases: with and without intruder deduction system. With respect to the experiments, let us observe that:

- For 6 *specifications* SATMC correctly finds the same attack even without the attacker deduction system.
- For 7 *specifications* SATMC correctly reports no attack found with and without the attacker.
- 2 *specifications* are reporting an attack only with the intruder.
- The other 22 either do not conclude in 300 seconds (timelimit threshold) or do not report the expected attack, but it is important to highlight that the result is the same with and without the intruder.

I have checked why for two specifications the attack was not found without the intruder. For `rbac_1_http_mutated.aslan++` the intruder `i` was explicitly used in the requirements. Then I have renamed it with another honest agent and the attack has been found using SATMC also without intruder.

For `xss_stored_goal2_nonSan.alan++` the attack should be also found without the intruder given that the attack trace reported by SATMC with intruder (Listing 12.1) does not use cryptography or pair rules.

Listing 12.1. `xss_stored_goal2_nonSan.alan++` attack trace

```

1  <tom>*->*webServer:login(tom,password(tom,
    webServer))
2  webServer*->*<tom>:listStaffOf(tom)
3  <tom>*->*webServer:viewProfileOf(tom)
4  webServer*->*<tom>:profileOf(tom)
5  jerry*->*<webServer>:login(jerry,
    password(jerry,webServer))
6  <tom>*->*webServer:editProfileOf(tom)
7  webServer*->*<tom>:profileOf(tom)
8  <jerry>*->*webServer:login(jerry
    password(jerry,webServer))
9  webServer*->*<jerry>:listStaffOf(jerry)
10 <webServer>*->*jerry:listStaffOf(jerry)
11 jerry*->*<webServer>:viewProfileOf(tom)
12 <jerry>*->*webServer:viewProfileOf(tom)
13 webServer*->*<jerry>:profileOf(tom)
14 <webServer>*->*jerry:profileOf(tom)

```

Even after adding all the needed sessions in order to let the model checker find the attack I have not managed to find this attack and it is still not clear to me if there is a bug in this beta version of SATMC with this particular set up but it does not seem that a constraint is needed at specification level that requires the intruder deduction system. Nevertheless, this result is showing that even if we totally exclude the entire Dolev-Yao intruder deduction system and we use a standard model checker we will

obtain the same result as if we were using the intruder on all the specification but one. Therefore the intruder rules seem not to be relevant to these web application case studies.

n	Spec	With Attacker rules	Without Attacker rules
1	Command_injection_Inj	ATTACK_FOUND	ATTACK_FOUND
2	Command_injection_NoInj	NO_ATTACK_FOUND	NO_ATTACK_FOUND
3	path_based_ac_Mutated_reach	ATTACK_FOUND	ATTACK_FOUND
4	path_based_ac_Mutated	NO_ATTACK_FOUND	NO_ATTACK_FOUND
5	path_based_ac	NO_ATTACK_FOUND	NO_ATTACK_FOUND
6	rbac_1_http_mutated	ATTACK_FOUND	NO_ATTACK_FOUND
7	rbac_1_http	NO_ATTACK_FOUND	NO_ATTACK_FOUND
8	rbac_1_mutated	NO_ATTACK_FOUND	NO_ATTACK_FOUND
9	rbac_1_no_ieqtom_mutated	NO_ATTACK_FOUND	NO_ATTACK_FOUND
10	rbac_1_no_ieqtom	NO_ATTACK_FOUND	NO_ATTACK_FOUND
11	rbac_1	NO_ATTACK_FOUND	NO_ATTACK_FOUND
12	rbac_3_mutated	NO_ATTACK_FOUND	NO_ATTACK_FOUND
13	rbac_3_no_ieqtom_mutated	NO_ATTACK_FOUND	NO_ATTACK_FOUND
14	rbac_3_no_ieqtom	NO_ATTACK_FOUND	NO_ATTACK_FOUND
15	rbac_3	NO_ATTACK_FOUND	NO_ATTACK_FOUND
16	stored_xss_cookie_mutated		
17	stored_xss_cookie		
18	xss_reflected_goal1_no_ieqtom	ATTACK_FOUND	ATTACK_FOUND
19	xss_reflected_goal1	ATTACK_FOUND	ATTACK_FOUND
20	xss_stored_goal1_malicioustom		
21	xss_stored_goal1_no_ieqtom_malicioustom		
22	xss_stored_goal1_no_ieqtom		
23	xss_stored_goal1		
24	xss_stored_goal2_maliciousTom	ATTACK_FOUND	ATTACK_FOUND
25	xss_stored_goal2_no_ieqtom_maliciousTom	ATTACK_FOUND	ATTACK_FOUND
26	xss_stored_goal2_no_ieqtom_nonsan	NO_ATTACK_FOUND	NO_ATTACK_FOUND
27	xss_stored_goal2_no_ieqtom	NO_ATTACK_FOUND	NO_ATTACK_FOUND
28	xss_stored_goal2_nonSan	ATTACK_FOUND	NO_ATTACK_FOUND
29	xss_stored_goal2	NO_ATTACK_FOUND	NO_ATTACK_FOUND
30	xss_stored_goal3_maliciousTom		
31	xss_stored_goal3_no_ieqtom_maliciousTom		
32	xss_stored_goal3_no_ieqtom		
33	xss_stored_goal3		
34	xss_stored_goal4_maliciousTom		
35	xss_stored_goal4_no_ieqtom_maliciousTom		
36	xss_stored_goal4_no_ieqtom		
37	xss_stored_goal4		

Table 12.1. SATMC deduction system test results

Experiment II — excluding the encryption/decryption rules.

The experiment has been performed using SATMC but using the standard 3.5 version instead of the beta version mentioned before. In order to exclude the encryption/decryption rules I have manually modified (deleting each occurrence of these rules) each Sate¹ file generated by SATMC while running the standard validation on each specification of our case study. The time threshold has been set to 300 seconds as for the previous experiment.

The test is summarized in Table 12.2 which reports either the SATMC output or an empty field for timeout for each specifications for the two considered cases: with and without encryption rules. I can observe that for each specification the result has been the same with or without the encryption/decryption rules. In particular, I have obtained the same results as in the previous experiment except that also for `xss_stored_goal2_nonSan.alan++` SATMC with no encryption rules reports the correct attack trace.

In these experiments the Dolev-Yao intruder seems not to be relevant for web applications when it comes to cryptographic rules. I however conjecture that message concatenation, and non-deterministic scheduling of Dolev-Yao attacker are needed on a more general web application case study. This could be used to speed up the search of goals and then to increase performances of model checkers while validating web applications.

I, therefore, want to focus on peculiarities of web applications when performing model-based testing and find which is the right way to model them. This could let me use a model checker to find attacks that are not strictly related to what the web application is implementing (e.g. protocol faults) but to the web application environment they are deep into.

¹ Sate files are generated by SATMC after performing a preliminary static analysis. These files contain the ASLan specifications and the intruder rules.

n	Spec	Attacker with Encryption rules	Attacker without Encryption rules
1	Command_injection_Inj	ATTACK_FOUND	ATTACK_FOUND
2	Command_injection_NoInj	NO_ATTACK_FOUND	NO_ATTACK_FOUND
3	path_based_ac_Mutated_reach	ATTACK_FOUND	ATTACK_FOUND
4	path_based_ac_Mutated	NO_ATTACK_FOUND	NO_ATTACK_FOUND
5	path_based_ac	NO_ATTACK_FOUND	NO_ATTACK_FOUND
6	rbac_1_http_mutated	ATTACK_FOUND	ATTACK_FOUND
7	rbac_1_http	NO_ATTACK_FOUND	NO_ATTACK_FOUND
8	rbac_1_mutated	NO_ATTACK_FOUND	NO_ATTACK_FOUND
9	rbac_1_no_ieqtom_mutated	NO_ATTACK_FOUND	NO_ATTACK_FOUND
10	rbac_1_no_ieqtom	NO_ATTACK_FOUND	NO_ATTACK_FOUND
11	rbac_1	NO_ATTACK_FOUND	NO_ATTACK_FOUND
12	rbac_3_mutated	NO_ATTACK_FOUND	NO_ATTACK_FOUND
13	rbac_3_no_ieqtom_mutated	NO_ATTACK_FOUND	NO_ATTACK_FOUND
14	rbac_3_no_ieqtom	NO_ATTACK_FOUND	NO_ATTACK_FOUND
15	rbac_3	NO_ATTACK_FOUND	NO_ATTACK_FOUND
16	stored_xss_cookie_mutated		
17	stored_xss_cookie		
18	xss_reflected_goal1_no_ieqtom	ATTACK_FOUND	ATTACK_FOUND
19	xss_reflected_goal1	ATTACK_FOUND	ATTACK_FOUND
20	xss_stored_goal1_malicioustom		
21	xss_stored_goal1_no_ieqtom_malicioustom		
22	xss_stored_goal1_no_ieqtom		
23	xss_stored_goal1		
24	xss_stored_goal2_maliciousTom	ATTACK_FOUND	ATTACK_FOUND
25	xss_stored_goal2_no_ieqtom_maliciousTom	ATTACK_FOUND	ATTACK_FOUND
26	xss_stored_goal2_no_ieqtom_nonsan	NO_ATTACK_FOUND	NO_ATTACK_FOUND
27	xss_stored_goal2_no_ieqtom	NO_ATTACK_FOUND	NO_ATTACK_FOUND
28	xss_stored_goal2_nonSan	ATTACK_FOUND	ATTACK_FOUND
29	xss_stored_goal2	NO_ATTACK_FOUND	NO_ATTACK_FOUND
30	xss_stored_goal3_maliciousTom		
31	xss_stored_goal3_no_ieqtom_maliciousTom		
32	xss_stored_goal3_no_ieqtom		
33	xss_stored_goal3		
34	xss_stored_goal4_maliciousTom		
35	xss_stored_goal4_no_ieqtom_maliciousTom		
36	xss_stored_goal4_no_ieqtom		
37	xss_stored_goal4		

Table 12.2. SATMC Encryption test results

Cross-Site Request Forgery

In a CSRF, an attacker violates the integrity of a user's session with a web application. In order to do that, an attacker injects HTTP requests via the user's browser. In general, the security policies of a browser permit web applications to send HTTP requests to any other web application. Thanks to these policies, an attacker is allowed to use resources, i.e. sending requests to these resources, not otherwise under his control.

In order to exploit a CSRF, and attack¹ a web application, mainly three parties have to get involved: an intruder, a client and a web server. The intruder is the entity that wants to find (and then to exploit) the vulnerability and attack the web application hosted on the web server. The web server is thus the entity that represents the web application host and, finally, the client entity is the honest agent who interacts with the web application (i.e. with the web server).

If the web application is vulnerable to CSRF, an attacker can trick the client to perform requests to the web server on his behalf. This attack scenario (depicted in Figure 13.1) can be summarized by the following steps:

1. The client logs in to the web application (authentication phase).
2. The web server sends a cookie (cookie exchange) to the client who will store it (within the web browser).
3. From this point on, the cookie will be automatically attached by the web browser to every request sent by the client to the web server (in message 3. of Figure 13.1 the client sends an honest request along with his cookie).
4. The intruder sends to the client a malicious link containing a request (Request') for the web application on the web server.
5. If the client follows the link, the web browser will automatically attach the cookie and will send the malicious request to the web server.
6. The web application cannot distinguish a request made by the intruder and forwarded by the client from one made and sent by an honest agent; therefore, it accepts the request.

It is important to observe that, from the description of CSRF I have given, an intruder sees the client as an "oracle". The intruder does not see the communication between

¹ In this context, with "attacking a web application" I mean that an intruder can perform requests to the web application that it should not be allowed to do.

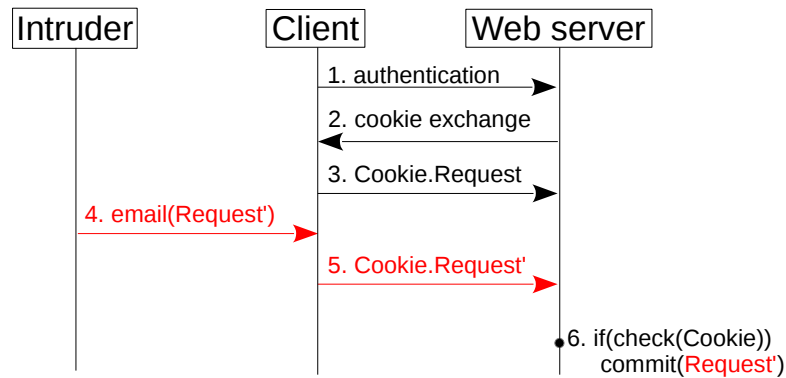


Fig. 13.1. CSRF Oracle Message Sequence Chart

the client and the web server but it will send a request to the client and wait for it to be executed, as showed in Figure 13.2.

13.1 CSRF protections

The state-of-the-art protections against CSRF attacks are mainly two (as reported in [13, 77]) and can be used together:

1. *Client confirmation*: the web server asks the client for a confirmation at every request the client sends to the web server.
2. *Secret validation token*: a secret, usually called *CSRF token* (e.g. a pseudo-random token), shared between the client and the web server, has to be attached to every request.

13.1.1 Client confirmation

Challenge-response is a protection option for CSRF in which the web application asks to the client a proof that he really wants to proceed with the request. In the case where the client has been tricked (by an intruder) to submit a request to the web application, when the web application asks for a confirmation, the client will not proceed with the request.

There are various implementation methods such as:

- CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart): a user is asked to type a sentence visualized as an image.
- Re-authentication: the web application asks for the authentication password.
- One-time token: the web application asks to the client a password that has been created to be valid for only one login session.

There is no trivial way for an intruder to bypass this protection, however it has an impact to the user experience. In fact, asking for confirmations to every request can, e.g. in the case of re-authentication, induce a user to choose easier password to speed-up the various confirmations. This has an obvious impact on the security of the overall web applications.

13.1.2 Secret validation token

With this approach, the web application server asks for additional information in each HTTP request. These informations are used to identify the client and then to check if he has enough permission to submit such a request. The main constraint for this validation token is to be hard to guess by an attacker. In fact, if a request does not have the right token, e.g. does not match the expected value, the web application must reject the request.

The main “weakness” of this approach is that it is often not used to protect the login phase. This is because in order to implement this protection there has to be a session to bind the token to and usually there is no session before the authentication/login phase. However, by creating “pre-sessions” it is possible to use validation tokens to protect the authentication phase.

There are many ways to implement this token:

- Given that the browser’s cookies are only exposed to related domain, a session identifier can be used as a validation token and, on every request, the server checks that the token matches the user’s session identifier. The only way for an attacker to bypass this defense is to guess the validation token. However, an attacker who owns a user’s session identifier can access the user’s account. The main disadvantage of using a session identifier as a validation token is that sometimes users reveals the content of a page to a third parties and if the page contains the user’s session identifier (in the form of CSRF token) an attacker can easily impersonate the user.
- The web application generates a random nonce and stores it as a user’s cookie. This CSRF token is then used by the client on every HTTP request and the web application validates that it matches the value stored. This is an unsafe method because it fails to protect against an attacker that can overwrite the nonce with his own CSRF token value (even if the web applications uses HTTPS).
- A refinement of the above method is to bind the token to the user session. On every request, the web application validates that the token is associated to the user’s session identifier. There is no security issue with this approach but the web application has to maintain a state table to validate these tokens.
- The most promising method is to use cryptography, instead of web application’s sessions, to bind the CSRF token to the session identifier.

A web application can use cryptography to protect against CSRF but many web applications fail to implement the secret token protection correctly.

13.1.3 Combining protections

In Figure 13.1 I report the message sequence chart (MSC) of a web application that uses both these CSRF protection mechanisms. In this way, the intruder cannot simply send a request to the client and wait for its execution. In fact, the client will not confirm the request and the browser will not automatically add the secret to the request.

My goal is to check if protections against CSRF, implemented in a web application, are strong enough; that is, to check if there is a way for the intruder to bypass

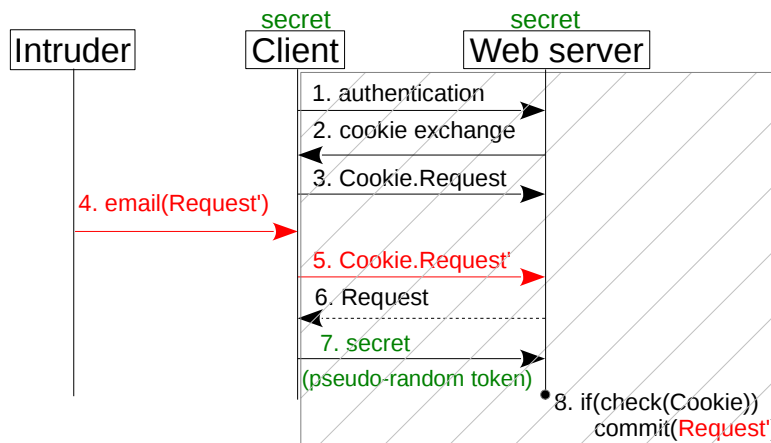


Fig. 13.2. CSRF from the intruder point of view and the barred part is not visible to the intruder

protections and force the web server to commit a rogue request that it is not allowed to do. Before defining my technique for specifying a web application with the focus on detecting CSRF I show a set of rules (guidelines) that a modeler should be aware of for defining a web application model without trivial flaws that can lead to a CSRF:

- The CSRF token has to be unique for each client.
- The CSRF token must be unique for each client-server interaction.
- The CSRF token must not be sent with the query string² in the URL.
- A request has to fail if the CSRF token is missing.

² The part of URL containing data to be passed to the web application.

CSRF in ASLan++: modeling and detection

In this chapter I describe a technique for modeling web applications to check for CSRF. I will use the DocumentRepository specification as a running example. Even if no attack has been detected on this case study it is illustrative for several reasons: it uses the usual client-server paradigm, it models cookies generation, storage and exchange (using a database) and the server handles login, commit, and malicious requests. Before going into the details of the modeling part I give the system description of the running example.

14.1 DocumentRepository description

The DocumentRepository ¹ system is a document repository that implements a document management system for the secure management and sharing of documents via web browser. Its main purpose is then to share and store different documents produced by various group of possibly different institutions. Suppose that both Alice and Bob (from two different institutions) are using the repository system. A typical scenario is the following:

- Alice logs in, via the login page, by providing her credentials (username and password).
- Alice is then forwarded to the system starting page where she can browse to the repositories list. She can now access to all public repositories and to private ones to which she has the permission.
- Alice clicks on one of the private repositories she has access to (repository *A*) and uploads a new document.
- Now Bob, who is the administrator of the repository *A* can download, edit or remove the file Alice has just uploaded and he can also edit Alice's permission on the private repository.

¹ The DocumentRepository system is a non public industrial case study within the SPaCioS project [100], already described in Section 4.3. I have then hidden the real name of the system and omitted some of the details.

14.2 ASLan++ modeling for CSRF detection

In order to check for CSRF I consider two entities: *Client/Oracle* entity and *Server* entity, as described in Chapter 13.

Client/Oracle.

In the Client entity I model a first authentication phase to obtain the cookie and logging in to the web application. First, in line 3, the client sends its credentials (username and password) and then the server, upon checking the received credentials are correct, sends back a new cookie to the Client (lines 7, 8).

```

1 % sends his/her name and password
2 %to the server's login service
3 Actor ->* Server: Actor.UserName.Password;
4
5 % the server's login service responds
6 %to the login request with a cookie
7 select { on (Server *->* Actor: ?Cookie &
8           ?Cookie=cookie(UserName,?,?): { } }

```

After this phase, the Client can perform requests to the server asking for services. When a client wants to send a request to the DocumentRepository system, it first loads the web page (usually using a web browser). The server produces the web page and sends it together with a CSRF token (i.e., a fresh pseudo-random token linked to the session ID of the Client). At specification level, skipping line 9 for the moment, I can model this mechanism by creating a variable Request that the Client wants to submit. When the Client sends this Request to the server (line 13), the latter will generate and send the token, CSRFToken, back to the Client (line 14). Now the Client sends (line 15) the Request together with the cookie and the CSRF token.

```

9 ? -> Actor: ?Request;
10 % load request page with the csrf token
11 % client asks for a web page and then
12 % server sends it to him including a csrf token
13 Actor *->* Server: Cookie.Request;
14 Server *->* Actor: ?CSRFToken;
15 Actor *->* Server: Cookie.Request.CSRFToken;

```

Between the authentication and the request submission parts, in line 9, I have added a message containing a variable Request. This message is sent from an unknown entity in order to model the scenario in which the Client receives a malicious email from a third party; the email contains a link to submit a request to the web application.

Finally, in line 15, the Client will receive from the server the confirmation that the request has been executed by the web application.

```

16 % the server's frontend sends back
17 % to the client the answer
18 Server *->* Actor: Request.?Answer;

```

Server:

The server entity accepts three different kinds of requests: authentication, request for a web page and request that it has to commit to the web application.

With *authentication request* a Client (if not already authenticated) sends to the server its username and password asking to log in (line 21). The server will check the received credentials (lines 22, 24) and, if they are correct, it will generate a cookie (line 31) that will be sent back to the Client (line 38).

```

19 % 1) login service receives the client
20 % request and generate a new session cookie
21 on((? ->* Actor: ?UserIP.?UserName.?Password
22 & !dishonest_usr(?UserName)) &
23 % checks if the data are available in the
    database
24 loginDB->contains((?UserName,?Password,
    ?Role)): {
25 % I have checked, using the password,
26 % that the client is legitimate.
27 % With the query, I extract the role of the
    legitimate client.
28
29 % creates the cookie and sends it back to the
    client
30 Nonce := fresh();
31 Cookie := cookie(UserName,Role,Nonce);
32 % adds the cookie into the DB associated
33 % with the name of the client
34 cookiesDB->add(Cookie);
35
36 % uses the IP address to communicate the
37 % cookie to the correct client
38 Actor *->* UserIP: Cookie;
39 }
```

The second type of request is a *web page request*. The Client asks for a web page before sending a request to the web application. The Client is already logged in and then it sends the request together with the cookie (line 44). The server will check the cookie (line 45) and generate a fresh token (line 47) that will send back to the Client (line 49).

```

40 % 2) with a cookie, a client makes
41 % a request to the frontend
42 % without the CSRF token and receives
43 % the respective token from the repository
44 on(?UserIP *->* Actor:
    cookie(?UserName,?Role,?Nonce).?Request &
45 cookiesDB->contains(cookie(?UserName,?Role,
    ?Nonce))): {
```

```

46
47         CSRFToken:=fresh();
48         csrfTokenDB->add((UserIP,Request,CSRFToken));
49         Actor *->* UserIP: CSRFToken;
50     }

```

The third case is when a Client sends a *request to the server* (line 54). The server checks both the token (line 57) and the cookie (line 62) and then commits the request (line 66).

```

51 % 3) a client makes a request
52 %     (along with a cookie) to the frontend
53 %     and receives the answer from the repository
54 on(?UserIP *->* Actor:
55     cookie(?UserName,?Role,?Nonce).
56     ?Request.?CSRFToken &
57
58 % checks if the token is the right one
59 csrfTokenDB->contains((?UserIP,?Request,
60     ?CSRFToken)) &
61
62 % checks if the client is allowed to do this
63 % request and the link client-cookie
64 checkPermissions(?UserName,?Request) &
65 cookiesDB->contains(cookie(?UserName,?Role,
66     ?Nonce))): {
67
68 % if the client has the right credential, then
69 % the request
70 % is executed and the answer is sent back to
71 % the Client
72 commit(Request);
73 Answer := answerOn(Request);
74 Actor *->* UserIP: Request.Answer;
75 }
76 }

```

Goal.

The goal is to check if there is a way for the intruder to commit a request to the web application:

```
csrf_goal: [](!commit(intruderRequest));
```

From the specification, the only way that the intruder has to commit a request is to bypass the CSRF protection (i.e., CSRF Token). To model that the intruder wants to submit a request that an honest agent does not, I have introduced a particular request (*intruderRequest*) within the Session entity as follows:

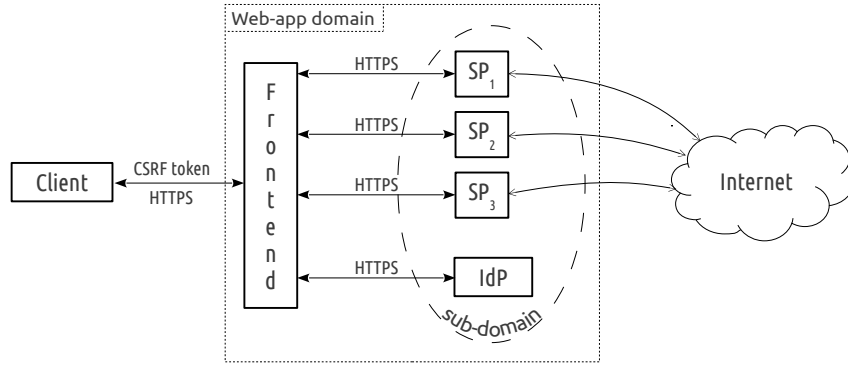


Fig. 14.1. Application scenario - Example

```

71 body { %% of the Environment entity
72   role1->can_exec(request1);
73   role1->can_exec(intruderRequest);
74   role2->can_exec(request2);
75   any UserIP. Session(UserIP, usr1, role1,
76     request1) where !dishonest(UserIP);
76   new Session(i, usr2, role2,
77     request1);
77 }

```

Validation.

The AVANTSSAR platform [4] has reported that the DocumentRepository specification is safe with respect to the modeled goal. This means that, given a bounded number of sessions, and considering a Dolev-Yao intruder, in this modeled scenario the three model checkers on which the AVANTSSAR platform relies on have not detected a CSRF.

14.3 A more complex example

In this section I present an example, depicted in Figure 14.1, to motivate the usage of my technique for more complex architectures.

The aim is to show that, after abstracting away unimportant implementation details, with my modeling technique it is possible to identify CSRF at design phase. I show a design schema of a Service Oriented Architecture (SOA) named Arch1. Arch1 (and a SOA in general) is a distributed system that offers functionalities that are not all hosted within the same server but are distributed on various hosts. Usually, a common interface (e.g., the AVANTSSAR platform web interface available from www.avantssar.eu) is offered to the client for communicating with the system. The architecture structure is then hidden to users who see the SOA as if it were a client-server architecture. It is also a common design choice to provide APIs for directly communicating with one (or a subset) of the services of the SOA so that expert

users can develop their own client (or use a customized one if provided). Languages such as WSDL [24] are widely used in SOAs for this purpose.

Architecture description.

Arch1 architecture is mainly composed by four parts: Client, Frontend, Service and Identity Providers (from now on SP and IdP respectively).

- The *client* entity represents the client browser. It can only communicate with the frontend via HTTPS, i.e., a secure channel: authenticated and encrypted.
- The *frontend* entity provides a common interface to communicate with the system. To avoid CSRF, for each HTML page loaded by the client a CSRFToken (Chapter 14) is attached by the frontend; the client will attach it to its request for guaranteeing freshness. Upon the receipt of the correct message, composed by request and token, the frontend will forward the request to the correct SP.
- *SPs* are the core of the system and provide the services of the SOA.
- *IdP* is the entity that handle the client authentication.

The frontend, SPs and IdP are all within the same web application domain, i.e., they are grouped in sub-domains that refer to the same domain. As a last remark on the architecture, each SP is also accessible from outside the SOA, and a client can directly communicate with a SP from the Internet.

The scenario I have modeled starts with an authentication phase in which the client logs in using the SAML-SSO protocol. After that, it can submit requests to the system by communicating with the frontend that acts as a proxy between the client and SPs. I have also modeled, as motivated in Chapter 13, that if the client receives from another (dishonest) entity a request for the Arch1 system, his browser attaches the cookie and sends the request. This behavior represents that the user clicks on a malicious email sent by a dishonest agent trying to exploit a CSRF on the system.

Authentication phase.

As already stated in Chapter 11, HTTP(S) is a stateless protocol and then cookies are used to ensure authentication. To store a cookie the client has to authenticate with the SOA. I have chosen one of the state-of-the-art authentication protocols, SAML-SSO[74] but I could have used OpenID or OAuth obtaining the same behavior at this level of abstraction. SAML-SSO uses an IdP to authenticate the credentials (e.g., username and password) given by a client. Here the client can only communicate with the frontend and then the frontend will act as a Proxy between the client and the IdP. The client provides his credentials to the frontend that forwards them to the IdP. The IdP, after validating the client's credentials, creates a cookie that is sent back to the Client via the frontend. Now the cookie is stored within the client's browser used and it will be attached to every request he will send to the SOA (because every part of the SOA is inside the same web domain).

Honest client behavior.

Once authenticated, the client has a cookie stored in his web browser. He loads a web page in which a CSRFToken is provided and he sends the request together with the token and the cookie to the frontend. The frontend forwards the message to the

correct SP that, through the frontend, will communicate the result of the commitment of the client's request. An SP will not directly check the cookie of the request but will ask the IdP to check if the cookie is a valid one.

Arch1 ASLan++ model and validation.

Due to page limit I cannot report the entire ASLan++ model but it follows the structure of the running example of Section 14.2.

I have used the AVANTSSAR platform for the verification of the specification obtaining the following attack trace.

```

MESSAGES :
1. frontend *->* <client> : n78(Csrftoken)
2. <?> ->* frontend : Client(80).Cookie(83).
                      Req(83).Csrftoken(84)
3. frontend *->* <sp> : Client(80).Cookie(83).
                      Req(83).Csrftoken(84)
4. <frontend> *->* sp : Client(80).Cookie(83).
                      Req(83).Csrftoken(84)
5. <?> -> client : intruderreq.sp
6. client ->* <sp> : client.client_cookie.
                      intruderreq
7. <?> ->* sp : client.client_cookie.intruderreq

```

I have assumed the Client has already logged in to the web application. In message 1. the frontend sends the CSRF Token to the Client (after having checked the cookie of the Client) In message 2. the Client sends an honest request to the frontend and in messages 3. and 4. the frontend forwards the request to an SP. In message 5. the intruder sends an email containing a malicious link to the Client with a dishonest request for a SP. The Client clicks on it and in message 6. the request is sent directly to the SP. In message 7. the intruder request is committed and a CSRF is performed.

The attack trace shows that the modeled architecture Arch1 is vulnerable to CSRF. It is clear that, even if protections against CSRF have been (correctly) implemented, the manual detection of CSRF is a difficult and time consuming task. My technique has permitted the automatic detection of CSRF in a complex architecture as Arch1 is. This extends the AVANTSSAR platform functionalities to check for CSRF.

Case studies and results

In this chapter I present results of applying my approach to three case studies. I have used the AVANTSSAR platform [4] to carry out the case studies and for the validation of the CSRF goal.

DocumentRepository.

The AVANTSSAR platform model checkers conclude that the specification (described in Chapter 13) is safe with respect to the CSRF goal (i.e., no attack trace has been found). This means that the CSRF protection (i.e. CSRF token) cannot be bypassed, in the modeled scenario (i.e., with a bounded number of sessions), by the Dolev-Yao intruder [32]. This result, which has been confirmed by an industrial partner, shows that the combination of SSL and a CSRF token do not permit the intruder to attack the web application using a CSRF.

WebAuth.

Authors in [2] have developed a methodology to detect web vulnerabilities using formal methods. In their most extensive case study, WebAuth [87], they show how they have found a CSRF. In order to compare [2] with my methodology I have then chosen to model the same case study.

WebAuth is an authentication system for web pages and web application developed by Stanford University. It consists of three main components: User Agent (UA), WebAuth-enabled Authentication Server (WAS) and WebAuth Key Distribution Center (WebKDC). The first time a UA attempt to access a web page he is redirected to the WebKDC that will ask to the UA for providing his authentication credential. An encryption identity is given to the UA. Now, the UA can use his new encrypted identity to obtain the web pages he wants to browse. The UA identity is stored in a cookie that the browser will “show” to the WAS in a user transparent way so the UA will simply browse the pages while the browser will use the cookie to retrieve them.

The result of the analysis shows a flaw in the authentication protocol, rather than a CSRF, in which the intruder convinces the UA to be communicating with the WAS while the UA is communicating with the intruder. In fact, in the attack reported in [2] the token that has to be shown to the WAS in order to retrieve the service is the same

used to start the authentication procedure and, due to the stateless property of HTTP, the WAS cannot detect if the two are different.

In order to detect a CSRF, I started from the protocol specification of WebAuth where the possibility of adding a CSRF token is not mandatory nor excluded. I have then modeled two versions: one with CSRF token exchange and the other one without. The model checkers return “NO_ATTACK_FOUND” (that means the specification is safe with respect to the CSRF goal defined) if the token is present, otherwise they report a CSRF as in the attack trace that follows.

```
MESSAGES :
[ . . . ]
18. UA (121) *->* <Actor (121)> : n119(Cookie).
                                intruderRequest
19. <UA (121)> *->* Actor (121) : n119(Cookie).
                                intruderRequest
20. Actor (121) *->* <UA (121)> : intruderRequest
```

From message 1 to message 17 there are the needed interactions between the UA and the system in order to obtain the cookie. Message 18 shows that the intruder sends to the UA a malicious message. In the real case it would be an email with a link containing a request that can be executed only from UA that has access to the system. The intruder is not logged in to the WebAuth application but he wants the honest agent AU to execute the action. In messages 19 and 20 the UA follows the link and then his browser automatically adds the cookie to the request hidden in the link. In message 21 the system replies with an acknowledge of the execution of the request. I can conclude that with my technique it has been possible to detect both an authentication flaw of the protocol underling the web application and CSRF, while the two were confused in [2].

EUBank.

I have analyzed a web application of one of the major European bank searching for CSRF. I have manually analyzed the web application and in particular the money transfer part. The scenario I have modeled can be summarized by the following steps:

1. *Login phase*: a client logs in the web application by providing two numerical codes, a client id and a numerical password over an HTTPS communication.
2. *Bank transfer set up*: the user fills in a form with all the necessary data for committing a bank transfer. Once committed, the web application asks for a confirmation. The user has to provide a numerical code that he can retrieve from his EUBank Pass, a hardware key that displays a numerical code freshly generated every sixty seconds.
3. *Bank transfer conclusion*: after checking the numerical code inserted by the user, the web application sends to the client a confirmation page with all the details of the bank transfer.

It is important to highlight that all the communication between the client and the server (bank) goes through a secure HTTPS channel, and even if no CSRF token is generated from the web application the EUBank Pass code is used also as a CSRF token

The first model I have implemented follows exactly the steps above, with the assumption that the client has his own EUBank Pass and no CSRF has been detected from the AVANTSSAR platform model checkers. Modeling a scenario in which an intruder has obtained the EUBank Pass key (e.g., through social engineering [95]), I obtain an attack trace reporting a CSRF on the web application. The attack trace is very similar to the abstract attack trace of CSRF of the WebAuth case study because I have assumed that the intruder knows the EUBank Pass. The only difference between the two abstract attack traces is that the intruder has to send the content of the EUBank Pass to the web server to confirm the bank transfer.

I have also manually tested it by transferring money from an EUBank account to another (of a different bank) simulating a CSRF exploitation. I have reported the attack (Figure 15.1) and the bank has confirmed it.

localhost/

Servizio Clienti

FAQ CONTATTI MODIFICA PIN ESCI X

HOME CONTI CARTE COMUNICAZIONI PROFILO NEGOZIO ONLINE

Nascondi

Inserimento dati Conferma 3 Ripetologo

BONIFICO DISPOSTO CORRETTAMENTE

ATTENZIONE: il bonifico verso altre Banche può essere annullato entro le ore 20.00 di oggi o - in caso di richiesta di esecuzione in data successiva a quella odierna - entro le ore 20 del giorno lavorativo precedente la data di esecuzione richiesta. Il bonifico su nostra Banca con data esecuzione corrispondente alla data odierna viene eseguito in tempo reale e non è annullabile. Può essere annullato solo in caso di richiesta di esecuzione in data successiva a quella odierna, entro le ore 20.00 del giorno lavorativo precedente la data di esecuzione richiesta. Per annullare il bonifico clicchi qui (e sceglia Bonifici e giroconti).

DATI ORDINANTE

N° rapporto Ordinante: ROCCHETTO MARCO

DATI BONIFICO

Beneficiario: MRCO ROCCHETTO

Indirizzo: Località: Prov. CAP:

IBAN:

Banca: ING DIRECT N.V. Sede: SEDE

Importo: 1,00 EUR Commissioni: 1,10

Causale: TEST

Data esecuzione Data inserimento Valuta beneficiario:

11.04.2013 11.04.2013 11:52:57 12.04.2013

Per effettuare bonifici senza inserire password dispositive attivi le Operazioni Veloci

MENU

CONTI CORRENTI

MONEYBOX CD

CONTI DI DEPOSITO

BONIFICI E GIROCONTI

Bonifico Italia

Bonifici periodici

Giroconto

Bonifico Europeo - SEPA

Bonifico Estero OnLine

Bonifico Estero multiplo

Trasferimento estero convenzionato

Ultimi bonifici/giroconti

Lista bonifici

Lista disposizioni estero

Lista bonifici SEPA

Lista disposizioni

IMPOSTE E TASSE

RICARICHE

BOLLETTE E UTENZE

ALTRI PAGAMENTI

ARCHIVIO PAGAMENTI

DOCUMENTI ONLINE

OPERAZIONI VELOCI

SMS PREMIUM

BANCA VIA CELLULARE

NEGOZIO ONLINE

STATO RICHIESTE

Fig. 15.1. EUBank CSRF attack. The bank transfer has been performed from a local Apache server as highlighted in the url field.

Discussion and related work

In Part IV, I have shown that a model-based technique for detecting CSRF related vulnerabilities is feasible and can be of help in the analysis of complex web applications, by leveraging existing symbolic techniques under the Dolev-Yao adversary models.

There exist several works that aim to perform model-based testing of Web applications, e.g., [2, 31, 96]. In particular I want to compare my techniques with works that consider CSRF vulnerabilities.

In [2], authors have presented a (formal) model-based method for the verification of Web applications. They propose a methodology for modeling Web applications and the results of the exploitation of the technique on five case studies, modeled in the Alloy [45] modeling language. Even if the idea is similar, they have defined three different intruder models that should find Web attacks while I have used the standard Dolev-Yao intruder. Also, the detailed way they have used to define the Web application models results in attack traces which are difficult to interpret. In contrast, I have chosen to abstract away from implementation details creating a more abstract modeling technique to easily define a Web application scenario, thus more amenable to human interpretation. The AVANTSAR platform [4] (a state-of-the-art formal analysis tool already described in Section 3.3) has permitted me to use the ASLan++ language and to obtain human-readable attack traces. As a final remark, I have also showed in Chapter 15 that authors in [2] have not found a CSRF on their most extensive case study, confusing an authentication flaw for a CSRF.

Another work close to mine is [21], in which authors have presented a tool named SPaCiTE that, relying on a model checker for the security analysis that uses ASLan++ specifications as input, generates potential attacks with regard to common Web vulnerabilities such as XSS, SQL injection and access control logic flaws. However, they have not explored CSRF. Given that SPaCiTE uses ASLan++ to define input specification, my approach should be easily integrated in the SPaCiTE tool. However, SPaCiTE provides a concretization phase based on libraries that maps high level actions (ASLan++ actions) to low level ones (browser actions) and, to integrate my approach in SPaCiTE, this requires further study.

In future work, I plan to investigate how to model further web vulnerabilities for the detection of more complex attacks. This is not a trivial task, in fact, the required level of details needed for modeling a specification for the detection of other vul-

nerabilities, e.g. XSS (Cross-Site Scripting), has a strong impact on the performance of the model checking techniques available and, most of all, the Dolev-Yao intruder is useless if the model of the web application has not the right abstraction level. I have performed several experiments on 37 ASLan++ specification provided by the SPaCIoS project to show that, with the right modeling phase, the Dolev-Yao intruder model can be useful to detect CSRF.

Conclusions and future work

Summary of contributions

Automated formal verification of the security of systems is one of the most studied fields of information security of the last few decades. Many different approaches and tools have been proposed together with a lot of speed up techniques (e.g., space reduction techniques). However, both theoretical and practical approaches to information security are far from being concluded. In particular, the constant growing of the complexity of systems (e.g., web applications) is introducing new and complex vulnerabilities that are often very difficult to discover by manual testing.

The main contributions of this thesis touch two different stages of the development process of systems, design time and testing (runtime). In particular, I have analyzed the state-of-the-art security techniques of both stages and proposed the following two approaches:

1. An interpolation-based method for the verification of security protocols.
2. A formal model-based technique for the automatic detection of CSRF.

The first method is called SPiM (Security Protocol interpolation Method, Part III) and is useful for the detection of security related flaws on security protocols (e.g., NSPK and NSL). The idea behind this work has been to take advantage of the use of Craig's interpolants [28] in order to speed up the search of goal locations (i.e., the search of security flaws) while model checking security protocols. This method has been implemented into a tool called SPiM tool (Chapter 8). With more details, having chosen the ASLan++ formal language (proposed by the AVANTSSAR project [4]) as the input language with which to define the behavior and the security properties of the system one wants to analyze (the specification of the protocol), this specification is then automatically translated into SiL (SPiM input Language) that is the input language accepted by the verification core of the Security Protocol interpolation Algorithm, SPiA. I have also proved the correctness of this translation by showing that the translation preserves all the important properties of the original specification (see Section 7.1 for more details).

SPiA checks the SiL specification against the security properties the modeler wants to check. SPiA searches for attack state combining the standard Dolev-Yao intruder model of [32] and calculating interpolants as a response to a search failure so that the model checker will not be blocked in the same way in future searches. This algorithm is an adaptation of the IntraLA algorithm proposed by McMillan in [61].

In particular, IntraLA has been developed for the verification of software and not of network protocols, and introducing the Dolev-Yao theory and switching from the analysis of sequential program to parallel ones (security protocols are sets of parallel program, one for each party involved in the communication) have been the two main contributions of the overall work.

Finally, I have showed, both theoretically and practically with SPiM, that Craig's interpolation method can be used to speed up the search of security protocols flaws via model checking.

While model checking has been widely used for design time verification of security properties of systems in general, it can also be very useful to check for security flaws on the implemented system (at runtime). This is called model-based security verification and in this thesis I have proposed a model-based method for the verification of web applications against a particular flaw, i.e., Cross-Site Request Forgery.

With this approach, I have showed that a web application specification can be formalized by using ASLan++. In particular, I have defined how to write an ASLan++ specification for a web application in order to search for CSRF (more details in Part IV) and showed the importance of the Dolev-Yao intruder model for this type of analysis. For this approach, I have run several experiments to show how the Dolev-Yao intruder can be misused and I also have supported my approach by means of several case studies. In my most extensive case study I have detected a CSRF attack in the web site of one of the major EU banks.

Future work

As for any research work there are several directions that could be deepened and links with other research work that could be very interesting to follow.

The most interesting one is the study of possible interactions between the interpolation-based speed-up technique proposed in SPiM and other speed-up techniques already implemented in other model checkers, e.g., constraint differentiation [69].

Model checkers like CL-AtSe [97] and Maude-NPA [35] implement optimizations based on rewriting the input rules of the specification. In particular, CL-AtSe has showed that a great speed up could be obtained from such optimizations and there is no evident incompatibility with the interpolation technique I have proposed in this thesis. It is, however, important to note that SPiM performs a forward search and an adaptation of the interpolation-based speed up technique to other search techniques (e.g., backward reachability based tools, like Maude-NPA) would require further study.

Furthermore, I do not see any incompatibility in using interpolation together with constraint differentiation or partial order reduction [79]. Constraint differentiation, implemented in OFMC [14], is a refinement of partial order reduction. It prunes the search space by analyzing the constraint of the state under validation with respect to states of previous paths. As interpolation works on reducing the search space by excluding some paths during the analysis (while, e.g., constraint differentiation prunes the state space by not considering the same state twice).

The only possible side effect that I foresee is that the number of paths pruned by interpolation could decrease when I use it in combination with other optimization techniques. In general, however, although I do not have experimental evidence yet, I expect that if enhanced with such techniques, SPiM could then reach even higher speed-up rates. I am currently working in this direction.

For what concerns the runtime security analysis of web applications, I have showed in Chapter 12 that defining a specification to search for web attacks (e.g., CSRF) could lead to the exclusion of the attacker during the analysis. I also compared my method with [2] showing that authors confused an authentication flaw with a CSRF in their most extensive case study.

This highlights that a model-based approach to search for web applications attacks requires a deep study of the intruder model one wants to use. One of the pos-

18 Future work

sible future directions, that I am currently working on, is the definition of a web attacker that could possibly take advantage (or even be an extension) from the Dolev-Yao intruder model to search for web attacks, e.g., CSRF, XSS.

References

- [1] Acunetix. Acunetix web vulnerability scanner. <http://www.acunetix.com/>.
- [2] D. Akhawe, A. Barth, P.E. Lam, J. Mitchell, and D. Song. Towards a Formal Foundation of Web Security. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*, pages 290–304, 2010.
- [3] Roberto M Amadio, Denis Lugiez, and Vincent Vanackère. On the symbolic reduction of processes with cryptographic functions. *Theoretical Computer Science*, 290(1):695–740, 2003.
- [4] Alessandro Armando, Wihem Arsac, Tigran Avanesov, Michele Barletta, Alberto Calvi, Alessandro Cappai, Roberto Carbone, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Gabriel Erzse, Simone Frau, Marius Minea, Sebastian Mödersheim, David von Oheimb, Giancarlo Pellegrino, Serena Elisa Ponta, Marco Rocchetto, Michaël Rusinowitch, Mohammad Torabi Dashti, Mathieu Turuani, and Luca Viganò. The AVANTSSAR Platform for the Automated Validation of Trust and Security of Service-Oriented Architectures. In *TACAS, LNCS 7214*, pages 267–282. Springer, 2012.
- [5] Alessandro Armando, David Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuellar, Paul Hanks Drielsma, Pierre-Cyrille Héam, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In *Proceedings of CAV'05, LNCS 3576*, pages 281–285. Springer, 2005.
- [6] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuellar, and Llanos Tobarra. Formal analysis of saml 2.0 web browser single sign-on: breaking the saml-based single sign-on for google apps. In *Proceedings of the 6th ACM workshop on Formal methods in security engineering*, pages 1–10. ACM, 2008.
- [7] Alessandro Armando and Luca Compagna. SATMC: a SAT-based model checker for security protocols. In *JELIA, LNAI 3229*, pages 730–733. Springer, 2004.
- [8] Alessandro Armando, Giancarlo Pellegrino, Roberto Carbone, Alessio Merlo, and Davide Balzarotti. From model-checking to automated testing of security

- protocols: Bridging the gap. In *Tests and Proofs*, pages 3–18. Springer, 2012.
- [9] AVANTSSAR. Deliverable 2.1: Requirements for modelling and ASLan v.1. www.avantssar.eu, 2008.
 - [10] AVANTSSAR. Deliverable 5.4: Assessment of the AVANTSSAR Validation Platform. <http://www.avantssar.eu/>, 2010.
 - [11] AVANTSSAR. Deliverable 2.3 (update): ASLan++ specification and tutorial, 2011. Available at <http://www.avantssar.eu>.
 - [12] Chetan Bansal, Karthikeyan Bhargavan, and Sergio Maffei. Discovering concrete attacks on website authorization by formal analysis. In *25th IEEE Computer Security Foundations Symposium, CSF 2012*, June 2012.
 - [13] Adam Barth, Collin Jackson, and John C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *Proceedings of the 15th ACM conference on Computer and communications security, CCS '08*, pages 75–88, New York, NY, USA, 2008. ACM.
 - [14] David Basin, Sebastian Mödersheim, and Luca Vigano. Ofmc: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, 2005.
 - [15] Beef: The browser exploitation framework project. <http://beefproject.com/>.
 - [16] Edward V Berard and Mark Twain. Abstraction, encapsulation, and information hiding. *E Berard Essays on Object-Oriented Software Engineering*, 1, 1993.
 - [17] Philippe Biondi. Scapy. <http://www.secdev.org/projects/scapy/>.
 - [18] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.
 - [19] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *CSFW*, pages 82–96. IEEE CS, 2001.
 - [20] Yohan Boichut, Pierre-Cyrille Héam, Olga Kouchnarenko, and F Oehl. Improvements on the genet and klay technique to automatically verify security protocols. In *Proc. AVIS*, volume 4, 2004.
 - [21] M. Büchler, J. Oudinet, and A. Pretschner. SPaCiTE – Web Application Testing Engine. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 858–859, 2012.
 - [22] Matthias Büchler, Johan Oudinet, and Alexander Pretschner. Security mutants for property-based testing. In *Tests and Proofs*, pages 69–77. Springer, 2011.
 - [23] Yannick Chevalier, Luca Compagna, Jorge Cuéllar, P Hanks Drielsma, Jacopo Mantovani, Sebastian Mödersheim, Laurent Vigneron, et al. *A high level protocol specification language for industrial security-sensitive protocols*. na, 2004.
 - [24] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, et al. Web Services Description Language (WSDL) 1.1, 2001.
 - [25] Steven R. Lavenhar Christoph Michael and Howard F. Lipson. Source Code Analysis Tools - Overview. <https://buildsecurityin.us-cert.gov/bsi/articles/tools/code/263-BSI.html>, 2009.

- [26] Chinotec Technologies Company. Paros - for web application security assessment. <http://www.parosproxy.org/>.
- [27] Véronique Cortier, Stéphanie Delaune, and Pascal Lafourcade. A survey of algebraic properties used in cryptographic protocols. *Journal of Computer Security*, 14(1):1–43, 2006.
- [28] William Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(03):269–285, 1957.
- [29] C.J.F. Cremers. The Scyther Tool: Verification, falsification, and analysis of security protocols. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, USA, Proc.*, volume 5123/2008 of *Lecture Notes in Computer Science*, pages 414–418. Springer, 2008.
- [30] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS, LNCS 4963*, pages 337–340. Springer, 2008.
- [31] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A Survey on Model-based Testing Approaches: A Systematic Review. In *WEASELTech '07*, pages 31–36. ACM, 2007.
- [32] D. Dolev and A. Yao. On the Security of Public-Key Protocols. *IEEE Transactions on Information Theory*, 2(29), 1983.
- [33] Adam Doupé, Marco Cova, and Giovanni Vigna. Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners. In Christian Kreibich and Marko Jahnke, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 6201 of *Lecture Notes in Computer Science*, pages 111–131. Springer Berlin Heidelberg, 2010.
- [34] Eclipse Foundation. *Eclipse - An Open Development Platform*, 2011. <http://www.eclipse.org/>.
- [35] Santiago Escobar, Catherine Meadows, and José Meseguer. Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties. In *FOSAD*, pages 1–50. Springer, 2007.
- [36] Santiago Escobar, Catherine Meadows, José Meseguer, and Sonia Santiago. State space reduction in the maude-nrl protocol analyzer. *Information and Computation*, 238:157–186, 2014.
- [37] Ettercap. <http://ettercap.sourceforge.net/>.
- [38] Firesheep. <http://codebutler.github.com/firesheep/>.
- [39] HP Fortify. Hp webinspect. https://www.fortify.com/products/web_inspect.html.
- [40] Grendel-scan. <http://grendel-scan.com/>.
- [41] Cenzic hailstorm professional. <http://www.cenzic.com/products/cenzic-hailstormPro/>.
- [42] Ian Hodkinson and Mark Reynolds. 11 temporal logic. *Studies in Logic and Practical Reasoning*, 3:655–720, 2007.
- [43] IBM. Rational appscan. <http://www-01.ibm.com/software/awdtools/appscan/>.
- [44] Immunity Inc. Immunity canvas. <http://www.immunitysec.com/products-canvas.shtml>.
- [45] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2012.

- [46] Girish Janardhanudu and Ken van Wyk. White Box Testing. <https://buildsecurityin.us-cert.gov/bsi/articles/best-practices/white-box/259-BSI.html>, 2009.
- [47] R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7(2):79–130, 1994.
- [48] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
- [49] V. Kodaganallur. Incorporating language processing into Java applications: a JavaCC tutorial. *IEEE Software*, 21(4):70–77, 2004.
- [50] Eleftherios Koutsoufios, Stephen North, et al. Drawing graphs with dot. Technical report, Technical Report 910904-59113-08TM, AT&T Bell Laboratories, Murray Hill, NJ, 1991.
- [51] Lapse: The security scanner for java ee applications. https://www.owasp.org/index.php/OWASP_LAPSE_Project.
- [52] D. Longley and S. Rigby. An automatic search for security flaws in key management schemes. *Computers and Security*, 11(1):75–89, 1992.
- [53] G. Lowe. Breaking and Fixing the Needham-Shroeder Public-Key Protocol Using FDR. In *TACAS, LNCS 1055*, pages 147–166. Springer, 1996.
- [54] Gavin Lowe. Casper: A compiler for the analysis of security protocols. *Journal of computer security*, 6(1):53–84, 1998.
- [55] PortSwigger Ltd. Burp suite. <http://portswigger.net/burp/>.
- [56] SensePost Pty Ltd. Wikto. <http://www.sensepost.com/labs/tools/pentest/wikto>.
- [57] Gordon Lyon. Nmap security scanner. <http://www.nmap.org/>, 2011.
- [58] Maltego. <http://www.paterva.com/web5/>.
- [59] Kenneth L McMillan. An interpolating theorem prover. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 16–30. Springer, 2004.
- [60] Kenneth L. McMillan. Applications of Craig Interpolants in Model Checking. In *TACAS, LNCS 3440*, pages 1–12. Springer, 2005.
- [61] Kenneth L McMillan. Lazy annotation for program testing and verification. In *Computer Aided Verification*, pages 104–118. Springer, 2010.
- [62] Kenneth L. McMillan. Interpolants from Z3 proofs. In *FMCAD*, pages 19–27, 2011.
- [63] Catherine A. Meadows. Formal verification of cryptographic protocols: A survey. In Josef Pieprzyk and Reihana Safavi-Naini, editors, *Advances in Cryptology ASIACRYPT’94*, volume 917 of *Lecture Notes in Computer Science*, pages 133–150. Springer Berlin Heidelberg, 1995.
- [64] J.K. Millen, S.C. Clark, and S.B. Freedman. The interrogator: Protocol security analysis. *Software Engineering, IEEE Transactions on*, SE-13(2):274–288, Feb 1987.
- [65] J.C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using mur phi;. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 141–151, May 1997.
- [66] Sebastian Modersheim. Algebraic properties in alice and bob notation. In *Availability, Reliability and Security, 2009. ARES’09. International Conference on*, pages 433–440. IEEE, 2009.

- [67] Sebastian Mödersheim and Luca Viganò. Secure pseudonymous channels. In *Computer Security—ESORICS 2009*, pages 337–354. Springer, 2009.
- [68] Sebastian Mödersheim and Luca Viganò. The Open-Source Fixed-Point Model Checker for Symbolic Analysis of Security Protocols. In *FOSAD 2008/2009*, LNCS 5705, pages 166–194. Springer, 2009.
- [69] Sebastian Mödersheim, Luca Viganò, and David Basin. Constraint differentiation: Search-space reduction for the constraint-based analysis of security protocols. *Journal of Computer Security*, 18(4):575–618, 2010.
- [70] N-Stalker. N-stalker web application security scanner. <http://www.nstalker.com/products/editions/>.
- [71] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, December 1978.
- [72] Tenable network security. Tenable nessus. <http://www.nessus.org/products/nessus>.
- [73] Thomas L Nielsen, Jens Abildskov, Peter M Harper, Irene Papaeconomou, and Rafiqul Gani. The capec database. *Journal of Chemical & Engineering Data*, 46(5):1041–1044, 2001.
- [74] OASIS. Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0. Available at http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security, 2005.
- [75] OWASP. OWASP WebGoat and WebScarab. OWASP, 2007. Available at <http://www.lulu.com/product/file-download/owasp-webgoat-and-webscarab/1889626>.
- [76] OWASP. OWASP WebGoat Project. https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project, 2011.
- [77] OWASP. OWASP Cross Site Request Forgery. [https://www.owasp.org/index.php/Cross-Site_Request_Forgery_\(CSRF\)](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)), 2013.
- [78] Lawrence C Paulson. The inductive approach to verifying cryptographic protocols. *Journal of computer security*, 6(1):85–128, 1998.
- [79] Doron Peled. Ten years of partial order reduction. In *Computer Aided Verification*, pages 17–28. Springer, 1998.
- [80] AVISPA project. Automated Validation of Internet Security Protocols and Applications. <http://www.avispa-project.org>.
- [81] Qualysguard it security. http://www.qualys.com/products/qg_suite/.
- [82] Rapid7. Metasploit framework. <http://www.metasploit.com/>.
- [83] Marco Rocchetto, Martn Ochoa, and Mohammad Torabi Dashti. Model-based detection of csrf. In Nora Cuppens-Boulahia, Frdric Cuppens, Sushil Jajodia, Anas Abou El Kalam, and Thierry Sans, editors, *ICT Systems Security and Privacy Protection*, volume 428 of *IFIP Advances in Information and Communication Technology*, pages 30–43. Springer Berlin Heidelberg, 2014.
- [84] Marco Rocchetto, Luca Viganò, Marco Volpe, and Giacomo Dalle Vedove. Using interpolation for the verification of security protocols. In *Security and Trust Management*, pages 99–114. Springer, 2013.
- [85] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The*. Pearson Higher Education, 2004.

- [86] Michaël Rusinowitch and Mathieu Turuani. Protocol insecurity with a finite number of sessions and composed keys is NP-complete. *Theor. Comput. Sci.*, 299(1-3):451–475, 2003.
- [87] Roland Schemers and Russ Allbery. Webauth v3 technical specification. <http://www.webauth.stanford.edu/protocol.html>, 2009.
- [88] Selenium. <http://seleniumhq.org/>.
- [89] Dawn Xiaodong Song. Athena: a new efficient automatic checker for security protocol analysis. In *Computer Security Foundations Workshop, 1999. Proceedings of the 12th IEEE*, pages 192–202, 1999.
- [90] Dawn Xiaodong Song, Sergey Berezin, and Adrian Perrig. Athena: a novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9(1):47–74, 2001.
- [91] Chris Sullo and David Lodge. Nikto. <http://www.cirt.net/nikto2>.
- [92] Nicolas Surribas. Wapiti. <http://wapiti.sourceforge.net/>, 2006.
- [93] Core Security Technologies. Core impact. <http://www.coresecurity.com/content/core-impact-overview>.
- [94] F.J. Thayer Fabrega, J.C. Herzog, and J.D. Guttman. Strand spaces: why is a security protocol correct? In *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on*, pages 160–171, May 1998.
- [95] Tim Thornburgh. Social Engineering: The “Dark Art”. In *Proceedings of the 1st Annual Conference on Information Security Curriculum Development, InfoSecCD ’04*, pages 133–135, New York, NY, USA, 2004. ACM.
- [96] Terry Tidwell, Ryan Larson, Kenneth Fitch, and John Hale. Modeling Internet Attacks. In *Proceedings of the 2001 IEEE Workshop on Information Assurance and security*, volume 59, 2001.
- [97] Mathieu Turuani. The CL-Atse Protocol Analyser. In *RTA, LNCS 4098*, pages 277–286, 2006.
- [98] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of modelbased testing. 2006. *Department of Computer Science The University of Waikato Private Bag*, 3105, 2006.
- [99] Kenneth R. van Wyk. Penetration Testing Tools. <https://buildsecurityin.us-cert.gov/bsi/articles/tools/penetration/657-BSI.html>, 2007.
- [100] L. Viganò. The SPaCIoS Project: Secure Provision and Consumption in the Internet of Services. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 497–498. IEEE, 2013.
- [101] David Von Oheimb and Sebastian Mödersheim. Aslan++ a formal security specification language for distributed systems. In *Formal Methods for Components and Objects*, pages 1–22. Springer, 2012.
- [102] J.M. Wing. A specifier’s introduction to formal methods. *Computer*, 23(9):8–22, Sept 1990.
- [103] Wireshark. <http://www.wireshark.org/>.

©2014 Marco Rocchetto

This research was supported by the AVANTSSAR Project FP7-ICT-2007-1 No.216471 and the SPACIOS Project FP7-ICT-2009-5 No.257876. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the AVANTSSAR Project, the SPACIOS Project and the Department of Computer Science of the University of Verona, Italy.