11-6-2015

A Comparative Study of Formal Verification Techniques for Authentication Protocols

Hernan Miguel Palombo
University of South Florida, hpalombo@mail.usf.edu

A Comparative Study of Formal Verification Techniques for Authentication Protocols

by

Hernan Miguel Palombo

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Co-Major Professor: Hao Zheng, Ph.D.
Co-Major Professor: Jay Ligatti, Ph.D.
Yao Liu, Ph.D.

Date of Approval:
October 2, 2015

Keywords: Formal methods, security, secrecy

DEDICATION

To my family.

ACKNOWLEDGMENTS

I am grateful to Dr. Hao Zheng and Dr. Jay Ligatti for their precious and constant help on this project.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Protocol verification is an exciting area of network security that intersects engineering and formal methods. This thesis presents a comparison of formal verification tools for security protocols for their respective strengths and weaknesses supported by the results from several case studies. The formal verification tools considered are based on explicit model checking (SPIN), symbolic analysis (Proverif) and theorem proving (Coq). We formalize and provide models of several well-known authentication and key-establishment protocols in each of the specification languages, and use the tools to find attacks that show protocols insecurity. We contrast the modelling process on each of the tools by comparing features of their modelling languages, verification efforts involved, and analysis results.

Our results show that authentication and key-establishment protocols can be specified in Coq's modeling language with an unbounded number of sessions and message space. However, proofs in Coq require human guidance. SPIN runs automated verification with a restricted version of the Dolev-Yao attacker model. Proverif has several advantages over SPIN and Coq: a tailored specification language, and better performance on infinite state space analysis.

CHAPTER 1

INTRODUCTION

Network security is a prominent and challenging field of computer science that cross-cuts different areas of engineering, information theory, psychology, and economics. In order to help secure communications over insecure networks, systems use cryptographic protocols. The design of such protocols tends to be error-prone. Even though protocol design has been a subject of extensive study by the research community, attacks have been found on protocols that were thought to be correct for many years [1]. In this context, formal methods are useful because they can provide stronger assurances that protocol designs satisfy security properties.

There are two common static analysis approaches, model checking and theorem proving, to formally verify that protocol designs are correct. Model checking uses state transition systems to model the behaviour of a system and then uses state exploration to find if an undesired state is reachable (i.e. a counter-example). On the other hand, theorem proving involves defining logical inference rules that describe the semantics of a system, and then using mathematical tools (e.g. induction) to prove different security-related theorems.

In this thesis, we look at both methods from a practical perspective. We choose well-established general-purpose tools for each method, and evaluate each technique by verifying models of well-known security protocols. Furthermore, we experiment with a domain-specific tool based on resolution logic, and compare the overall verification process and results. While model-checking tools may be easier to use, theorem proving provides a language support for more expressive specifications and can be used to prove security by construction. On the

other hand, specialized tools have the benefit of convenient built-in features, with more structured specifications and attacker models.

1.1 Background

1.1.1 Security Policies and Properties

A security policy is a rule that defines specific behaviour of a system. In general, a system can be described as "secure" if it satisfies a set of security policies. Accordingly, a policy must remain true in all possible execution traces. A policy is usually composed of one or more properties, each characterizing about a set of traces. Additionally, non-property polices also exist, and are much harder to verify because they must consider the set of all possible traces [2].

Two main categories of properties exist, safety and liveness [3, 4]. Safety properties can be informally described as "nothing bad will happen", while liveness properties could be explained as "something good will eventually happen".

In general, if a system has flaws, safety properties are a lot easier to disprove because they only require to find one execution trace that leads to a property violation. On the other hand, proving/disproving liveness properties is usually harder because the proof must always exhaust all possible traces.

For our study, we are interested in two safety properties that the protocols must enforce, namely secrecy and authentication. Secrecy ensures that data can only be seen by authorized users. The goal of authentication is to assert that an agent is really who s/he claims to be. The process of authentication over a network is the result of a series of message exchanges, after which one party can assert that s/he has initiated communication with the other. A more formal specification of these properties and the cryptographic primitives required for enforcement will be given in Chapter 3.

Although other related properties exist, secrecy and authentication are two of the most important building blocks for security. For example, authorization, a property that regulates

what actions agents can perform on domains and objects, requires authentication of users to determine if s/he is allowed to access the resources, and secrecy to ensure that unauthorized users cannot access private data. In addition, privacy, a property that ensures that some action's originator or agent's presence cannot be traced, can also be stated in terms of secrecy of the actions or presence on the network.

An earlier classification of security properties was based on confidentiality, integrity, and availability [5, 6, 7]. Confidentiality is a property related to reading, which states that access to assets should be granted only to authorized users. Integrity is a property related to writing, which ensures that data has not been modified by unauthorized agents. Availability is usually stated as a liveness property, and refers to the ability of a system to service requests or remain operable. Clarkson presents more details of this taxonomy and related work in [8].

Another approach to specify security policies is to define policies for access control. This class of policies regulate actions on objects by specifying which subjects and domains can operate on those objects (e.g. file system permissions, or network access to some host). Some of the most popular models used to specify access control policies are access control lists (ACLs), role-based access control (RBAC) [9], and policy-based access control (PBAC). Access matrices, as introduced by Lampson [10], are a standard way of visualizing access control policies, in which permissions are mapped to domains and objects.

Model-checking has been successfully applied to formalize and verify different access control systems. Among recent interesting studies, there are those of [11, 12, 13].

1.1.2 Cryptographic Protocols

Cryptography is the science that studies techniques for secure communication in the presence of third parties [14]. A cryptographic protocol defines a set of rules for communication between different entities, providing some security guarantees. These include initial, response, and termination requirements. Usually, the design of security protocols abstracts

the implementation of cryptographic operations and assumes perfect cryptography (e.g. encrypted ciphers cannot be guessed).

There are two high-level mechanisms for encryption, namely symmetric and asymmetric cryptography. The idea of symmetric (secret-key) cryptography is simple, the key must remain secret and only the parties that hold the key can encrypt or decrypt messages. Assuming perfect cryptography, a more interesting problem is the initiation process, i.e. establishing a shared-key, which is usually achieved using asymmetric encryption (public-keys).

Public-key cryptography is based on the use of trapdoor functions, i.e. a one-way function with a secret inverse function f^{-1} that allows the possessor of it to go back to f at any time [15]. Public-key cryptosystems usually consist of key generation, encryption, and decryption algorithms [16]. Compared to shared-key cryptography, using public-keys is more convenient for authentication services (authenticating many users using secret keys would require users to store a large number of keys that must remain secret). Notice, nonetheless, that using public keys requires some public key infrastructure for the exchange of public keys. Moreover, public-key cryptography can be used to generate digital signatures in order to maintain private communication. Nevertheless, public-key cryptography is rarely used to encrypt entire communications because it is significantly slower than secret-key cryptography. Therefore, one of the most common uses of public-key cryptography is to authenticate two parties and establish a shared secret key that can be used later to encrypt the rest of the communication.

RSA is a major cryptosystem implementation used to provide asymmetric cryptography. It relies on mathematical properties of numbers (specifically, the factoring problem) to allow an exchange between two parties who are able to use public and private keys to communicate securely over an insecure network. The security of this system is based on the assumption that computing the secret key is unfeasible, i.e. an NP-complete problem for the attacker.

1.1.3 Authentication and Key-Establishment

To preserve secrecy of data transmitted over an insecure network, either symmetric or asymmetric encryption may be used. In practice, though, asymmetric encryption is orders of magnitude slower and more expensive than its symmetric counterpart. Yet, establishing a shared secret key is an unavoidable problem. If a passive attacker can be assumed, exponential key exchange (as described by Diffie-Hellman [16]) might be used. However, as this is not the general case, key-establishment usually requires the execution of a protocol that relies on asymmetric encryption and an infrastructure for managing public-keys (such as one that uses certificate authorities). The focus of this thesis is on verification of protocols that establish authentication and secret keys. Thus, the verification process must present evidence that authentication cannot be masqueraded, and that the secret keys cannot be seen or altered by an attacker.

1.2 Motivation

1.2.1 Protocol Design Issues

Designing secure protocols is an error-prone process. Protocol designs must be analyzed very carefully to avoid logical flaws. It is difficult to reason about protocols because one must consider concurrency, distributed systems, cryptography, sound mapping of designs to implementation, and an unknown attacker. History shows that attacks have been found even many years after the protocols were introduced and widely deployed [1]. Table 1.1 shows important examples that demonstrate this point.

1.2.2 Protocol Verification

Even though protocol verification began more than 30 years ago, it is still an active area of research. There are two main approaches, model checking and theorem proving, and a vast number of verification tools exist for each method. Therefore, it becomes difficult for

Table 1.1: History of protocol design and flaws found

Year	Description
1978	Needham & Schroeder proposed authentication protocols for large networks [17]
1981	Denning & Sacco found attacks on the Needham-Schroeder protocols and proposed modifications [18]
1983	Dolev & Yao created the first threat model using formal algebra [19]
1987	Burrows, Abadi & Needham defined a logic for authentication [20]
1994	Hickman developed first version of SSL (issues in v1 and v2; fixed in v3) [21]
1994	Ylonen created SSH [22]
1995	Abadi & Needham proposed protocol design good practices [23]
1995	Lowe found an attack on the NS protocol (17 years later) [24]
2002	Stubblefield et al. described a WEP design flaw that allowed for a key recovery attack [25]
2002	Vaudenay described a design flaw and attacks that affected SSL/TLS, IpSec, WTLS and SSH protocols
2005	Gilgbert et al. found an attack on HB+ RFID authentication protocol [26]
2008	Armando used model-checking to show how to break the SAML-based single sign-on protocol for google apps [27]
2012	Kahya, Ghoualmi & Lafourcade found attacks on PKM, the security protocol used by WiMAX networks [28]
2014	A design issue was found on the SSL 3.0 protocol [29]
2014	Cao, et al. found design issues in Oauth and OpenID protocols [30]

protocol engineers to find the right tool to use. A primary goal of our study is to bring some light to this issue. We evaluate the different tools, and provide insights to help protocol engineers choose the right verification technique.

Moreover, the lack of case-studies has always been a weakness of formal methods which, in some way, has prevented its wider adoption by engineers without a strong formal mathematical background. This thesis intends to fill in this gap by providing a new set of protocol models in three different specification languages and the results of running each tool.

Additionally, researchers working on either model-checking or theorem-proving may benefit by learning how one approach compares to the other, applied to the specific domain of cryptographic protocol designs. Finally, we aim to use the results of our analysis to lay out future research opportunities by surveying some of the most relevant works in this area, which is abundant and challenging to mine.

1.2.3 Verification Tools

In order to evaluate the approaches of model-checking and theorem proving, we have chosen two of the most prominent existing tools, namely SPIN and Coq.

SPIN [31] is a mature model-checker (first released in the early 1980s by Holzmann), that has been adopted by industry, particularly in the hardware verification domain. It uses a finite state automata representation, which reflects the essence of the model-checking approach. SPIN has been used to verify security protocols [32, 33], although the number of available case-studies in this area is limited. In 2014, Ben Henda [34] presented an approach to encode generic attackers using the Dolev-Yao model [19]. This paper motivated our study; we wanted to find out if a generic approach could be extended to other tools, and how would these tools compare to each other.

On the other hand, Coq [35] is a well-established general purpose theorem prover. It has been used to verify security protocols [36, 37], but the number of published case-studies that use this tool for protocol verification is also rather small. This limitation was another motivation for us to choose this theorem-proving tool.

Another motivation is to compare and contrast general purpose tools against a domain-specific one. For this reason, we have chosen to use the symbolic model-checker Proverif [38]. One of Proverif's interesting features is that it allows to verify protocol runs for an unbounded number of sessions. Its powerful modeling language, the applied pi-calculus, based on [39], is also an attractive feature. More on this topic will be covered in Chapter 3.

1.2.4 Security Protocols

For analysis, we have chosen a representative sample of authentication and key-distribution protocols. Particularly, we focus on protocols with authentication and secrecy goals. The protocols used in this study are those proposed by:

- Needham and Schroeder (NSPK) [17],

- Denning and Sacco (DS) [18],

- Tatebayashi, Matsuzaki and Newman (TMN) [40],

- Diffie and Hellman (DH) [16].

The NSPK, one of the most analyzed protocols in the literature, has become a canonical example in protocol verification. In this thesis, we analyze both a simplified version and the full version. The DS protocol, also extensively studied, demonstrates the use of signatures with asymmetric encryption. The TMN protocol is one of the first authentication protocols designed for mobile networks. Apart from its historic connotation, it serves as an interesting example because of its use of Vernam encryption techniques, i.e. XOR functions, a technique commonly used in cryptography[1]. Finally, DH is one of the most significant protocols in the history of cryptography. Many protocol families have been derived from the idea of using exponentiation for an unauthenticated agreement of a shared key between two parties. Many widely-used protocols, such as SSL and TLS, rely on this technique for key-establishment.

1.3 Related Work

In 1996, Meadows [41] compared FDR [42] and NRL [43] model checkers by analyzing a model of the Needham-Schroeder public key protocol. FDR is a refinement checker that only works on finite-state systems. It was replaced by FDR2 in 1995, and more recently by FDR3. NRL is a logic-rewriting system that can verify infinite-state systems based on Prolog, which was later replaced by Maude-NPA. Our study is more comprehensive and current than Meadows' work. This thesis analyzes more protocols with different cryptographic primitives, and includes theorem proving in the analysis.

Among the most recent studies, Patel et al. [44] compared FDR, AVISPA, HERMES, Interrogator, NRL, Brutus, Murphi, Proverif, Athena, and Scyther model-checkers. Their study only evaluated models in Scyther (the modeling process was not evaluated for the

[1]Vernam encryption is used in one-time-pads, which are implemented in RC4, a popular stream cipher that is widely used over the Internet [14].

other tools), and the metrics used for comparing the verification process were limited to the following yes/no questions: public availability of the tool, falsification, bounded/unbounded verification, and termination. This thesis provides models for all the tools compared and does a more extensive analysis.

In 2011, [45] compared CSP/FDR with other techniques for finding attacks on protocols, namely manual proofs using Strand spaces and BAN logic, and ALSP (Action Language for Security Protocols). Notwithstanding, this thesis is focused on existing tools that do not require manual construction of proofs.

In [46], Avalle surveyed formal analysis of security protocols, but focused on implementations; more specifically it analyzed code generation and model extraction mechanisms and did not consider analysing models with the tools used in this thesis.

1.4 Contributions

The following is a list of the contributions of this thesis:

- It reviews the extensive related work of protocol verification with a focus on relevant case-studies. Despite the abundant literature, no previous surveys have approached the topic in this manner, from a practical and comparative perspective.

- It extends the number of case-studies of security protocols using some of the most prominent formal methods tools, and make the models publicly available[2]. We demonstrate with examples the benefits of applying formal methods to find protocol attacks, and provide source code of the models that can be used as a reference in the future by other engineers/researchers. We describe in detail how to encode protocol descriptions, attacker models, and security properties in a way that can be generalized for other protocols of the same class.

[2]http://myweb.usf.edu/~hpalombo/models/

- It provides a comparison of the modeling and verification process using the two most common formal techniques, model checking and theorem proving. As a further matter, we generalize our encoding approach by showing how to formalize all the necessary cryptographic primitives for security protocol verification using any formal tool.

- It analyzes pros and cons of using specialized versus general-purpose tools, and discuss the modeling process and type of results that can be obtained with each.

Although others have attempted to compare different verification tools, we are not aware of other studies that have compared modeling and analysis of both model-checking and theorem-proving, general-purpose and tailored tools with the level of detail presented here. For all of the reasons mentioned above, our work provides a valuable and unique contribution.

In summary, three new findings are presented in the following chapters that form the core of this thesis:

1. We can generalize the process of modeling security protocols to build models for, at least, three different classes of verification tools.

2. The results of the experiments show that well-known attacks can be found using any of the three tools, but different attack models must be considered in some cases.

3. The three tools can be compared in terms of the suitability of their modeling languages to encode cryptographic protocols, and the characteristics of the verification results.

The modeling framework presented in this thesis is applied to several case studies and interesting conclusions are derived after comparing the results. The case studies considered in this thesis are a representative class of authentication and key establishment protocols. Our comparison does a thorough analysis of the results that can be obtained for different attacker and protocol models, and sets a road that may direct to future research.

1.5 Outline

Chapter 2 presents the reader with background on static analysis tools, and describes model-checking and theorem proving techniques. Chapter 3 explains how to model protocol specifications, from informal narrations to formal specification logics. It also shows how to encode cryptographic primitives, attacker models, authentication and secrecy properties. Chapter 4 describes the models of four protocols (NSPK, DS, TMN, and DH) using each of the three different tools, and shows the results of the verification process. Chapter 5 presents a comparison of the modeling and verification process for each of the tools. Finally, Chapter 6 discusses some of the issues with symbolic protocol models, explores opportunities in code generation and model extraction techniques, and concludes this thesis by setting the grounds for future work.

CHAPTER 2

BACKGROUND OF VERIFICATION TOOLS

2.1 Static Security Enforcement

There are two major approaches for security enforcement: static and dynamic analysis. Two common examples of dynamic mechanisms are runtime monitors and dynamic typing. Despite the benefits of real-time monitoring, the performance penalty introduced by these methods can be significant and, in many cases, static analysis is a better solution.

As Schneider explains in [2], runtime monitors cannot check for some non-property policies, such as one that states that information always flows from x to y, because this condition depends on all possible execution traces. On the other hand, a model checker can sometimes verify such information-flow policies by searching for a counter-example, i.e. a trace that leads to an invalid state in which information does not flow from x to y.

Static enforcement is a broad term that may refer to techniques that range from code analysis to type-checking, model-checking, and theorem proving. In this thesis, we limit our scope to model checking and theorem proving. Yet, type-checking is closely related to theorem-proving. For instance, the calculus of constructions (CoC), the underlying logics of Coq, is a theory of types that serves both as a typed programming language and a foundation for constructive proofs [47]. In standard programming languages, type-checking is fully automated, and achieves soundness by restricting allowable programs and sacrificing completeness. On the other hand, human-guided proof construction using interactive theorem provers does not have this limitation.

2.2 Model Checking

More than three decades ago [48], model checking has emerged as an alternative technique to manual proof construction for asserting properties about systems. Properties are specified in some form of propositional calculus, for example, linear temporal logic (LTL) [49, 50], or computational tree logic (CTL) [51]. Systems are modelled as transition graphs, where each node represents a possible state. For many practical systems, though, the state space may become extremely large, leading to the state-explosion problem, i.e. the number of states in a system grows exponentially as the number of variables increases.

Depending on the representation of the state space, model checking is either explicit or symbolic. Explicit model checking is based on enumerating all possible states of a program, resulting in a directed graph representing all reachable states and their transitions. Then, the decision procedure explores each path until, either a counter-example is found (property is unsatisfiable), or all the paths have been explored (property is satisfied). Until the 90s, explicit model checking was a popular technique in formal methods research. This method works well for checking LTL properties on small systems. However, when larger systems are considered, the memory requirements caused by the state explosion problem are so large that explicit model checking becomes impractical.

To cope with the state explosion problem, many optimization techniques were developed and integrated into existing tools for asynchronous systems. Particularly, partial order reduction prunes the state space by eliminating non relevant states/transitions to properties under verification [52]. Bit-state hashing is another technique that focuses on state space coverage instead of completeness. It limits memory usage by indexing each state into a hash table of bits [53].

SPIN features these and other optimization techniques, which make it a prominent general-purpose model-checker that is still widely used today[1]. Another popular explicit

[1] http://spinroot.com/spin/Workshops/

model-checker is Java Path-finder (JPF) [54], which started as a Java to Promela[2] translator, and then developed its own model-checking engine. While Promela is closer to the C language, JPF is naturally tied to verification of Java programs.

Symbolic representation is an alternative approach that can be applied to verify larger systems. Symbolic model-checkers encode transition systems as boolean formulas. Early tools were implemented using binary decision diagrams, while tools developed later use boolean satisfiability solvers. Some of the most popular symbolic model checkers are NuSMV, which was first published as SMV by McMillan in 1993 [55]. More recent symbolic model checkers are based on SMT solving [56] applied to system encodings with various theories on first order logic.

Despite the improvement over explicit representations, symbolic model checking cannot deal very well with recursion. Bounded model checking tries to solve this problem by imposing a fixed bound on recursion. This technique became popular in the early 2000s. Although it is generally incomplete because decision procedures may not terminate as the bound increases, in practice it is enough to disprove properties since counter-examples can often be detected with a small bound.

In general, model checkers' modeling languages provide features to describe state-transition systems. For example, Promela's labels and *goto* statements allow users to specify state transitions, and if blocks can be used to capture non-deterministic selection behaviour. Even symbolic model-checkers use this representation. For instance, the symbolic model checker Uppaal has a graphical interface that allows users to draw networks of timed automata. A different approach was taken by the bounded model checker Alloy [57], which has an input modeling language based on set relations.

Table 2.1 shows related work in which security protocols were verified using model checking techniques. One of the early studies that used a model checker to verify security protocols is described in [58], where the Needham-Schroeder, Tatebayashi-Matsuzaki-Newman,

[2]Promela is SPIN's modeling language.

Table 2.1: Case studies using model checkers to verify security protocols

Work	Description
[58]	Mitchell et al. used Murphy to verify the several authentication protocols
[32]	Merz analyzed a simplified version of the NSPK protocol using SPIN
[33]	Maggi and Sisto used SPIN to verify the NSPK protocol
[59]	Cheminod et al. verified protocols from the SPORE repository [60] using the model checkers FDR, STA, S3A, and OFMC
[61]	Cremers compared execution time of verification in Avispa, FDR, Proverif and Scyther
[62]	Dupressoir et al. used VCC to verify the RPC and Otway-Rees protocols

and Kerberos protocols are analyzed using Murphy [63]. Several studies have modelled the Needham-Schroeder public key exchange protocol and verified it using SPIN [32, 33]. In [62] Dupressoir et al. used the general-purpose C model checker VCC [64] to prove secrecy and authentication on the RPC and Otway-Rees protocols.

Moreover, Cheminod et al. compared four model checkers (FDR, STA, S3A, and OFMC) to verify a number of protocols in [59, 60]. Their results only showed which tools can find known attacks, but the study did not analyze the modeling process for each of the tools. In [61] execution time was compared for several model checkers (the tools in Avispa, FDR, Proverif and Scyther) by verifying the NSPK, EKE, and TLS protocols to find known attacks. Proverif shows the best performance among the tools considered.

2.3 Theorem Proving

Proof construction has followed two major currents in computer science: automated provers, and proof assistants. Automated provers offer a "push-button" solution, i.e. they take a proposition and give a *yes/no* (or *ran-out-of-time*) answer [65]. On the other hand, interactive theorem provers[3], also known as proof assistants, automate some basic aspects of building proofs while they still depend on human guidance for more challenging tasks.

[3]For simplicity, throughout the thesis we refer to interactive theorem provers using the general term theorem provers.

In this thesis, we focus on comparing a model-checker, a resolution verifier, and a theorem prover. Therefore, the rest of this section will have a focus on theorem proving.

Theorem provers can be used in two ways, either to prove a system's security or to show its insecurity. Showing insecurity implies demonstrating that an attack is possible, i.e. prove that a conjecture can be derived given a logical deductive system. Proving security is a harder task, in which proofs must be completed by construction.

Proof development is a NP-complete problem, and human interaction is required. Moreover, the learning curve is steep. In order to be able to construct proofs, engineers must have strong foundations in logic and mathematical tools such as induction.

Nevertheless, theorem provers facilitate the proof construction process by providing an environment with helpful resources and libraries. For example, Coq has a tactics library that consists of predefined –and extensible– functions to automate some steps in a proof. In addition, proofs are grouped in theories, and these are built in a modular way. Theories are organized in hierarchical structures that can be included when writing new proofs.

Theorem provers allow to verify infinite protocol runs and message spaces, as opposed to model-checking techniques, in which protocol sessions are always bounded or the algorithms may not terminate. While model-checkers verify models by searching for counter-examples, interactive theorem provers use type theory to derive facts about terms.

Some of the most popular proof assistants that have been used to prove security properties of systems are Isabelle [66], Coq [35], LEGO [67], PVS [68] and ACL2 [69]. Table 2.2 summarizes relevant works in this area.

2.4 Specialized Tools

Several model checkers have been specially designed to verify security protocols. For example, SATMC is a bounded model checker that uses a SAT-solver to verify protocol specifications written in the ASLan language. Internally, SATMC uses NuSMV to generate SAT encodings for the LTL formulae and MiniSAT to solve the SAT problems [74].

Table 2.2: Case studies using theorem provers to verify security protocols

Work	Description
[36]	Bolignano used Coq to prove security properties of electronic commerce protocols
[70]	Dutertre and Schneider used PVS to prove secrecy and authentication of the Needham-Schroeder Public Key Authentication Protocol
[71]	Paulson used Isabelle to prove secrecy and authentication of the Needham-Schroeder Public Key Authentication Protocol
[72]	Meng et al. presented an automated proof of resistance of denial of service attacks using events
[73]	Meier et al. added semantic rules to the Isabelle theorem prover to handle security protocols and presented an algorithm for mechanic construction of proofs
[37]	Cheng et al. described a program logic for verifying secure routing protocols (Coq)

Furthermore, it has been integrated into several frameworks, such as the AVISPA project [75], which provides several back-end model-checkers for protocols specified in HLPSL, a high-level protocol specification language. Among those model-checkers, OFMC [76] is an on-the-fly model checker for infinite-state systems. It uses lazy and demand-driven search techniques for modeling a Dolev-Yao intruder. Another framework is the Casper/CSP/FDR approach, which combines the Casper specification language [77] to translate a protocol's high-level description into a Communication Sequential Process (CSP) description; CSP is the input language to FDR3 [78], a refinements checker. All of the tools previously described can find attacks for a bounded number of sessions.

Protocol verification for an unbounded number of sessions has been shown to be an undecidable problem [79, 80]. There are two approaches to deal with this issue, either to abstract the models (e.g. Proverif), or to allow non-termination [81]. Proverif uses resolution techniques with over-approximations, which may report false attacks. On the other hand, Maude-NPA [81] is based on a term-rewriting algorithm with several heuristics to reduce the possibility of non-termination. Other specialized tools that have been developed for security protocols are HERMES [82], CL-Atse (Constraint-Logic-based Attack Searcher) [83], TA4SP (Tree-Automata-based Automatic Approximations for the Analysis of Security Protocols) [84], and Scyther [85]. Among more recent studies using specialized tools,

Kusters and Truderung [86] used ProVerif to analyze protocols with Diffie-Hellman exponentiation. Kahya et al. [28] presented a formal analysis of the PKM protocol using Scyther. Cheval and Blanchet [87] proved anonymity in the private authentication protocol [88] using ProVerif. Elbaz [89] analyzed and verified a key agreement protocol over cloud computing using Scyther.

Even though some studies have analyzed security protocols using different tools, none of those studies have examined a common modeling framework that can be applied to verify protocols using model-checking, theorem proving, and specialized tools. In order to fill this gap, in the following chapter we describe a generalized modeling framework.

CHAPTER 3

FORMALIZATION OF SECURITY PROTOCOLS

Security protocols can be specified at different levels of abstraction. Protocol narrations are a natural way of describing interactions between agents who participate in a protocol [90]. Nevertheless, narrations only describe expected message exchanges, and leave some agents' actions unspecified. Therefore, more explicit formalisms are needed in order to verify correctness of such protocols.

In this chapter we describe some of the issues with informal protocol narrations. We then survey relevant formalisms for specification of security protocols. We present a generalized modeling approach that we use for comparison of the different verification tools. Later, we explain how to model protocols and properties for each of the verification tools considered in this study. We conclude the chapter with a discussion of diverse approaches to protocol instantiation.

3.1 Informal Protocol Narrations

Traditionally, high-level specifications of security protocols have been written using protocol narrations [91], which describe a correct execution trace as a sequence of message exchanges between the parties involved in the protocol. For example,

$$\begin{aligned} &1.\ A \rightarrow B:\ \{N_A, B\}_{PK_B} \\ &2.\ B \rightarrow A:\ \{N_A\}_{PK_A} \end{aligned} \tag{3.1}$$

means that A sends a message to B containing a fresh nonce N_A generated by A and B's identity, encrypted with B's public key PK_B. Then B replies to A by sending the received nonce encrypted with the public key of A, (PK_A). The numbers at the beginning of the lines show the order in the exchange sequence and are incremented every time a message is sent/received. Messages can be extended by adding values separated by commas.

Most cryptographic operations can be represented in this format. Nonetheless, protocol narrations are sometimes ambiguous. In order to capture agents' behavior precisely, we need to add important details that happen between transitions. For instance, B sends Message 2 in Protocol 3.1 only if the name identifier in Message 1 matches his own name. These constraints are important to reduce the message space and avoid incorrect results, e.g. false attacks.

Another source of ambiguity is the implicit initial knowledge. Apart from knowledge about its own identity, an agent may know some information about other agents at the start of the protocol. For example, in some cases, it is desirable to assume that a client's registration phase has completed successfully before the protocol starts. Therefore, we assume that the server trusts its own copy of A's public key. This assumption is part of an assumed trust base. The consequence of this assumption is that the server is able to detect a forged message that is supposed to come from A if it is encrypted with another key that is not PK_A. This assumption can be modeled as an action through which the server checks if the incoming message contains PK_A before continuing with the protocol. Neither the honest agent's initial knowledge nor their intermediate actions are captured by protocol narrations. More ambiguities are due to an implicit attacker model, i.e. the attacker capabilities are not specified in protocol narrations.

Several authors have attempted to translate protocol narrations into process algebras for verification. Bodei et al. [92] describe how to translate protocol narrations into the Lysa-calculus, a formalism similar to the spi-calculus but without channels. Briais and Nestmann [93] have attempted to develop translators from protocol narrations into the pi-calculus.

In all cases, however, the translation requires human intervention to disambiguate implicit assumptions that the input notation does not capture, and the translation cannot be fully automated.

3.2 Formal Modeling Languages

There are different approaches to formalizing protocols for the purpose of verifying their security properties. Common formalisms used to model security protocols are BAN logic [20], CSP (Communicating Sequential Processes) [94], and Strand spaces [95]. A comparative case study of BAN-logic, CSP, and Strand spaces applied to the Needham-Schroeder protocol can be found in [96].

BAN logic is a formalism that describes protocols using a deductive system of beliefs, which gained some popularity in the early 90s. However, the protocol's "idealization" [97] and facts that must be assumed may leave out important details, and make BAN proofs unsound [98]. BAN proofs can only show authentication properties, assuming confidentiality and integrity of the underlying cryptographic system. Most importantly, the logic does not have precise semantics for its operations [99].

CSP is a process algebra developed by Hoare [94] in the late 70s. CSP semantics are more general, i.e. it can be used to describe any communication process. Inspired by parallel hardware implementations, Hoare's goal was to include parallelism and non-determinism as structural features of a language. A failures-divergence refinement checker (FDR) was later introduced by Roscoe in [100] as a model checker for CSP programs. Despite the benefits of the tool, modeling in CSP still remains a difficult task. Later, Lowe developed CASPER [101], a compiler to translate specifications in a high-level language into CSP.

Strand spaces, developed by Guttman et al. [95], represent sequences of events in security protocols. Strands may represent actions of both honest agents and attackers connected by causal events. Despite being useful to prove correctness of many protocols, the mechanics of proofs using strand spaces are very detailed and somewhat complicated [102]. This

modeling framework was implemented later in Athena [103], a specialized model-checker for security protocols.

The pi-calculus is a functional language similar to the lambda-caluclus designed to describe concurrent processes [104]. It provides channel primitives, which can be used to represent communications. Nevertheless, the pi-calculus does not have cryptographic primitives built in, and encryption must be encoded (this is achieved with some effort in [39, 105]). Based on this fact and the idea that scoping is the basis of security property enforcement, several variants of the pi-calculus have been proposed and used in the context of security [39] [106]. Proverif models can be written in the applied pi-calculus [107], an extension of the pi-calculus with function symbols that can represent data structures and cryptographic operations.

In explicit state model checkers, protocols are represented using automata. Their modeling languages reflect the underlying formalism. In SPIN's modeling language, Promela, a system is the composition of each agent's automaton, which is described as a process and details the agent's participation in the protocol. Send and receive actions are synchronized events on shared channels that are primitive language constructs. An attacker is usually represented by an independent automaton, and the state transition diagram of the entire system is generated by the interleaving of all possible transitions.

Theorem-provers use an inductive approach to specify protocols. For example, Coq's modeling language is based on the theory of types, which has features such as inductive definitions and type inference. A protocol description has a set of rules that correspond to actions of agents and attackers. Security properties are stated as lemmas which are then proved inductively. The same approach is used in Isabelle and other theorem-provers.

3.3 A Generalized Modeling Approach

Despite differences in modeling frameworks, a general framework for modeling and verification of security protocols should support (a) formal protocol description, (b) attacker

modeling, and (c) property specification. Moreover, depending on the verification technique, the framework must provide a way to instantiate the protocol in order to discover potential attacks. Figure 3.1 shows a high-level view of the modeling process.

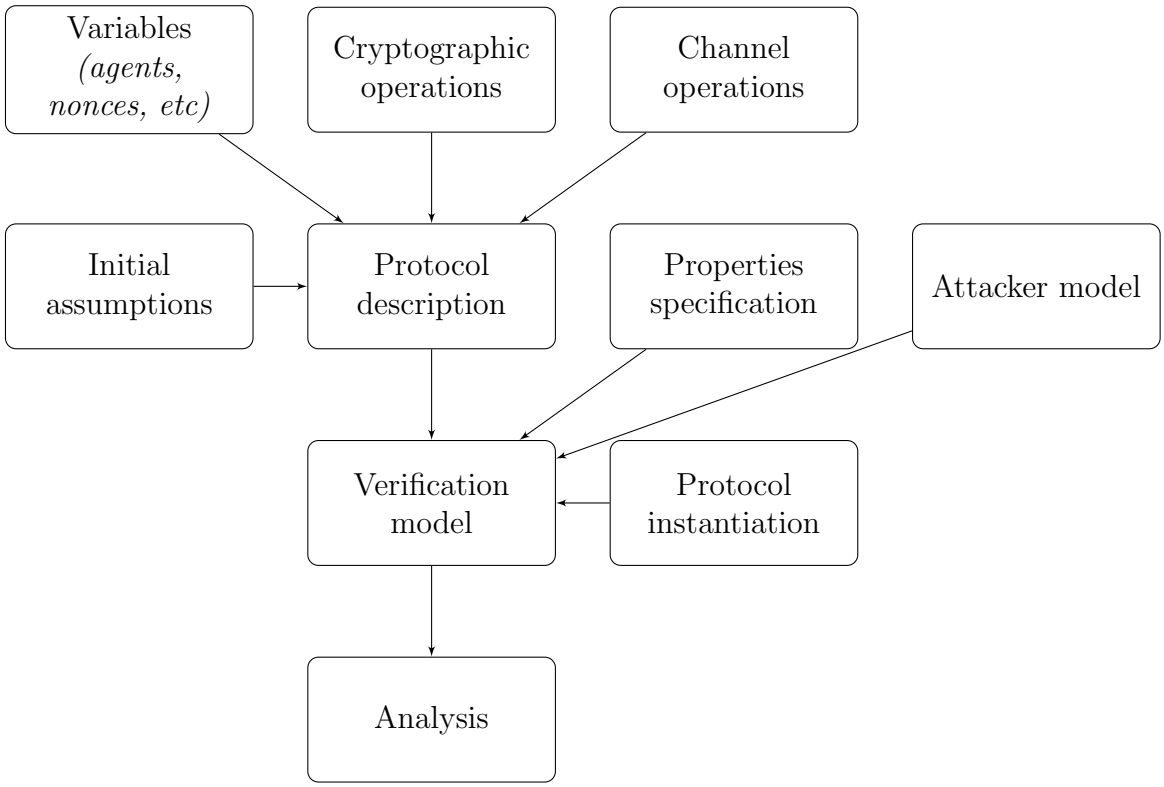

Figure 3.1: Modeling cryptographic protocols for verification

A formal protocol description is defined with a set of initial assumptions, and sequences of operations that represent interactions between agents. Moreover, it requires other preliminary definitions: a set of values that form messages (agents, nonces, etc), some communication channels, a set of input/output functions (send, receive) that operate on those channels, and a set of cryptographic operations.

Some typical cryptographic operations are encryption, digital signatures, hashing, and Diffie-Hellman exponentiation. To represent cryptosystems, the framework should provide a way to generate keys, encrypt, and decrypt messages. Simmilarly, digital signature operations require key generation, sign and signature verification functions.

Initial assumptions are another important aspect of the modeling process. Unambiguous models must explicitly define the variables that each agent knows before the start of the protocol. In general, most models assume that every agent has access to all public keys and cryptographic functions. On the other hand, nonces and secret keys are kept private to their respective owners.

The attacker model refers to the attacker's initial knowledge and all the attacker's potential actions. For example, a Dolev-Yao (DY) attacker [19] may be able to spy, intercept, or inject messages at will. A more restrictive attacker can be specified, e.g. a passive eavesdropper, significantly reducing the message space.

A model must also include some security properties to be checked, sometimes called verification goals. Proofs of (in)security will be bound to the properties specification. Therefore, formulation of the properties is an important matter.

3.3.1 Communication on Shared Channels

In general, communications in authentication and key-establishment protocols occur on shared channels. Therefore, real source and destination of messages cannot be guaranteed. In order to capture the possibility that an attacker may intercept a message, we abstract away the identity of the addressee in all the send actions. Similarly, we abstract the sender's identity from receive operations to consider attackers message injection capabilities.

$1.\ A \to X:\ \{N_A, X\}_{PK_X}$
$1'.\ Y \to B:\ \{N_Y, B\}_{PK_B}$
$2'.\ B \to Y:\ \{N_A\}_{PK_A}$
$2.\ X \to A:\ \{N_A\}_{PK_A}$

(a) Interlocutors abstraction with strong restrictions

$1.\ A \to X:\ \{N_A, X'\}_{PK_{X'}}$
$1'.\ Y \to B:\ \{N_{Y'}, B\}_{PK_B}$
$2'.\ B \to Y'':\ \{N_{Y'}\}_{PK_{Y'}}$
$2.\ X'' \to A:\ \{N_A\}_{PK_A}$

(b) Restrictions relaxation

Figure 3.2: Modeling communication for different attacker models

The augmented message sequence for Protocol Narration 3.1 is shown in Figure 3.2a. Noteworthily, our abstraction maintains the relations between message elements. For example, the identity X in the body of message $1'$ must match the public key used to encrypt the message. Moreover, this formalization further restricts the message space by requiring that the addressee of message 1 by A matches the identity in the body of the message, and the public key used to encrypt the message. We call this restriction a strong one. A more relaxed version is shown in Figure 3.2b, in which addressor/addressee may not match the message elements (e.g. X and X' in message 1), and there is no relation of identities between different messages in the sequence (e.g. X and X'' in messages 1 and 2 respectively). Ultimately, the decision to use strong or relaxed restrictions will depend on the assumptions about the protocol and the attacker model. For instance, to model a more powerful attacker, a relaxed protocol description should be used. In the case studies presented in Chapter 4, we try more restrictive attacker models first and, if no attacks are found, the message restrictions are relaxed to extend the message space. The remainder of this chapter explains how to model security protocols and their properties for each of the verification tools considered in this thesis.

3.4 Modeling in SPIN, Proverif, and Coq

In this section, we describe how to use the general modeling framework to develop protocol models and property specifications for SPIN, Coq, and Proverif. Our ideas are rooted in the works by [34, 108, 38]. Full syntax of specification languages for these tools can be found in [109, 110, 38]. As a general convention in all our models, we capitalize types, use all upper-case letters for constants, and lower-case for function names. Next, we discuss general aspects of models for each of these tools.

3.4.1 Cryptographic Operations

A main requirement for specifying security protocols is to encode cryptographic operations. Our assumption of perfect cryptography allows us to abstract away the implementation details, and focus on the functional behavior of cryptographic constructs.

In Figure 3.3a we show how to represent asymmetric and symmetric key generation by defining functions that map agents to public, secret, and shared keys. Figure 3.3b shows common cryptographic operations used by authentication and key establishment protocols.

For all agents x,y:

$pk(x)$ is the public key of x,

$sk(x)$ is the secret key of x, and

$ss(x, y)$ is the shared secret key between x and y.

(a) Key generation

For all agents x,y and messages m:

$aenc(m, pk(x))$	(asymmetric encryption)
$adec(aenc(m, pk(x)), sk(x)) = m$	(asymmetric decryption)
$senc(m, ss(x, y))$	(symmetric encryption)
$sdec(enc(m, ss(x, y)), ss(x, y)) = m$	(symmetric decryption)
$sig(m, sk(k))$	(signatures)
$verif\_sig(sig(m, sk(k)), pk(k)) = m$	(signature verification)
$hash(m)$	(hash functions)
$vernam(a, b)$	(Vernam encryption)
$vernam(vernam(a, b), a) = b$	(Vernam decryption)
$exp(exp(g, x), y) = exp(exp(g, y), x)$	(DH exponentiation, where g is an agreed value)

(b) Common operations

Figure 3.3: Definitions of cryptographic functions

At a high level, any language with functions will suffice for encoding cryptographic operations. However, there are some subtleties. For example, some operations, such as the Diffie-Hellman exponentiation, describe equivalence properties between functions. Moreover, encoding some operations using standard types requires careful thought.

From a computation perspective, we can think of cryptographic operations having one of two types, constructors or destructors [38]. Encryption, signature, and hashing functions are constructors, as they introduce new terms. On the other hand, destructors eliminate terms. Decryption and signature verification are two examples of destructor functions.

```
1 Inductive key : Set :=
2   PK : agent -> key
3   | SK : agent -> key
4   ...
5 Inductive message : Set :=
6   Name : agent -> message
7   | Key : key -> message
8   | Enc : message -> key -> message
9   ...
```

Figure 3.4: Inductive types in Coq

We define one-way functions as having no body, so they cannot take an evaluation step. This representation of cryptographic operations is easy to implement in Coq and Proverif. In Coq, inductive types have constructors which can represent one-way functions that will be carried around as predicates. We specify constructors for keys and messages (Figure 3.4), and also for cryptographic functions (Figure 3.5).

```
1    known : message -> Prop :=
2      | aenc : forall m a, known m -> known (Enc m (PK(a)))
3      | adec : forall m a, known (Enc m (PK(a))) -> known (Key (SK a)) ->
           known m
```

Figure 3.5: Cryptographic functions in Coq

Proverif allows the user to define constructor and destructor functions, so we use them for operations like encryption and decryption (Figure 3.6).

```
1 fun pk(skey): pkey.
2 fun aenc(bitstring, pkey): bitstring.
3 reduc forall x: bitstring, y: skey; adec(aenc(x,pk(y)),y) = x.
```

Figure 3.6: Cryptographic keys and functions in Proverif

In SPIN, we must make careful decisions to minimize the state space, so in the models we use constants wherever possible. For example, agents, nonces, public and secret keys are labelled indices of a user-defined array. Then, to find if a value is an agent's public or secret key, we define macros that just offset the function's input value. An example can be seen on Figure 3.7.

```
1 #define PK(x) x - j
2 #define SK(x) x - (j + 3)
3
4 mtype = {...A, B, I, ...(j other constants)..., PKA, PKB, PKI, SKA, SKB,
      SKI...}
```

Figure 3.7: Key constructors in SPIN

A call to $PK(x)$, where x is an agent A, B, or I, will return PKA, PKB, or PKI respectively. Moreover, decryption is achieved by evaluating the last field of an incoming message (Figure 3.8).

```
1 proctype Responder(mtype b) {
2   ...
3   comm?msg, eval(PK(b));
4   ...
```

Figure 3.8: Decryption in SPIN

The receive operation (denoted by "`?`") of message `msg` on channel `comm` will only step if the last field of the message matches the public key of `b`.

Furthermore, we use constants to abstract certain properties of operations that depend on equivalences. For instance, Figure 3.9 shows how Diffie-Hellman exponentiation can be

hard-coded using a macro definition. Thorough this representation, we are able to prove some attacks (e.g. a MITM) that work for –at least– a bounded number of sessions.

```
1  # define IsDHKey(x) (x <= 3)
2
3  mtype = {NULL, Sec, A, B, I, Ga, Gb, Gi, Kab, Kai, Kbi};
4
5  # define DHKey(eny, x, k)  if \
6   :: (x==A) && (eny==Gb) -> k=Kab\
7   :: (x==B) && (eny==Ga) -> k=Kab\
8   :: (x==A) && (eny==Gi) -> k=Kai\
9   :: (x==I) && (eny==Ga) -> k=Kai\
10  :: (x==B) && (eny==Gi) -> k=Kbi\
11  :: (x==I) && (eny==Gb) -> k=Kbi\
12  :: else k = NULL\
13 fi
```

Figure 3.9: Diffie-Hellman exponentiation in SPIN

In Proverif, we can define functions as equivalence relations. This is useful to represent properties like the Diffie-Hellman exponentiation (Figure 3.10).

```
1 const g: G.
2 fun exp(G, exponent): G.
3 equation forall x: exponent, y: exponent;
4   exp(exp(g, x), y) = exp(exp(g, y), x).
```

Figure 3.10: Diffie-Hellman exponentiation in Proverif

In Coq, we define constants to represent the calculated values in Diffie-Hellman exchange (in the same manner as it was done in SPIN).

3.4.2 Attacker Models

The most commonly used attacker model is the one presented by Dolev-Yao (the DY model) [19], which assumes perfect cryptographic properties (cryptographic primitives treated as black-boxes), and a non-deterministic behavior of the intruder. Verification of models with DY attackers is done symbolically, as opposed to computational models that use probabilistic methods and take into consideration algorithmic complexity of cryptographic

operations. In the computational model, messages are bitstreams, cryptographic primitives are functions that operate on them, and the attacker is modeled as a any probabilistic polynomial-time Turing machine [111].

The assumption of perfect cryptography in the DY model implies that the attacker cannot guess secret keys. Typical attacker actions in this model are eavesdropping, injection, and interception of messages. The DY attacker model is simple and powerful; its behavior is non-deterministic, and the message space is infinite.

Ideally, verification with unbounded settings is desired. However, it is not always possible on every tool. For example, it is not possible to model an attacker with unbounded storage space in SPIN. Therefore, the capabilities of each attacker model must be explicitly defined. In our modeling framework, a DY attacker with unlimited storage and message generation capability is denoted as class A_1. Moreover, we define a restricted attacker class that can generate messages of a fixed length as A_2. Finally, we refer to a class of attackers that can generate fixed-length messages and have bounded storage as A_3.

Our modular approach to defining classes of attackers could be easily extended. For example, we could define an attacker class A_4 for attackers that can only read but cannot inject or intercept messages (passive attackers). Verification of a protocol model using different classes of attackers can lead to more fine-grained results. For example, we may be able to show that authentication holds when the attacker has limited storage capabilities even if the property fails in the case of an attacker with unbounded storage. On the other hand, a number of attacks may be found even in the presence of restricted attackers. The case studies in Chapter 4 demonstrate that attacks can be found for several protocols even when considering limited message generation and storage capabilities of the attacker.

Another important decision when modeling an attacker is how to represent knowledge. It is not possible to allow the attacker to have infinite storage capabilities if model-checking is used. On the other hand, when using theorem proving, one can use induction to generalize proofs for an unbounded attacker's knowledge vector.

```
1  Inductive send : agent -> message -> Prop :=
2    inject  : forall m, known m -> send I m
3    ...
4  with known : message -> Prop :=
5     (* initial knowledge *)
6     | name : forall a, known (Name a)
7     | nonce : forall Y, known (Nonce (I,Y))
8     | know_ski : known (Key (SK I))
9     | know_pk : forall x, known (Key (PK x))
10    | know_ssi : forall x, known (Key (K (x,I)))
11    (* actions *)
12    | spy : forall m, receive I m -> known m
13    | decomp_l : forall m m', known (Pair m m') -> known m
14    | decomp_r : forall m m', known (Pair m m') -> known m'
15    | compose  : forall m m', known m -> known m' -> known (Pair m m')
16    | aenc  : forall m a, known m -> known (Enc m (PK a))
17    | adec : forall m a, known (Enc m (PK a)) -> known (Key (SK a)) ->
         known m
18    | senc : forall m a, known m -> known (Key (K(a,I))) -> known (Enc m
         (K (a,I)))
19    | sdec : forall m a b, known (Enc m (K (a,b))) -> known (Key (K (a,b)
         )) -> known m .
```

Figure 3.11: Generic attacker model in Coq

In Coq, we use an inductive type *known* to represent the attacker actions and knowledge vector (Figure 3.11). In Proverif, the attacker is implicit, so there is no need to model it. In SPIN, the attacker is specified as a separate process that can spy, intercept, or inject messages (Figure 3.12a). Furthermore, we initialize the knowledge vector to represent the facts that are initially assumed to be known by the attacker (Figure 3.12b).

3.4.3 Property Specification

Cryptographic protocols are designed to enforce specific security properties. In general, security goals, i.e. properties that need to be enforced, are usually stated in the protocol specification. Property requirements are usually embedded in protocol specification documents, which are written in natural language and may include some informal protocol narrations and ad hoc annotations [91].

```
proctype attacker() {
  mtype m = NULL, k = NULL;
  mtype prev_m, prev_k;
  do
    :: comm?m, k ->
       atomic {
        AddToKnowledge(m, k);
        prev_m = m; prev_k = k;
        if
        /* intercept */
        :: skip
        /* spy */
        :: comm!m, k
        fi
       }
    /* inject */
    :: RandMessage(m, k) ->
       atomic {
         IsValidMessage(m, k, prev_m, prev_k) -> comm!m, k
       }
  od
}
```

(a) Attacker process

```
init {
  atomic {
    Knows[NULL - 1] = 1;
    Knows[A - 1] = 1;
    Knows[B - 1] = 1;
    Knows[I - 1] = 1;
    Knows[Ni - 1] = 1;
    Knows[PKi - 1] = 1;
    Knows[SSKi - 1] = 1;
    ...
  }
}
```

(b) Attacker's initial knowledge

Figure 3.12: Generic attacker model in SPIN

The goals of authentication and key establishment protocols are mainly secrecy and authentication. Formalization of such properties has been thoroughly studied before [2, 112, 91]. According to Abadi [107], secrecy may be specified as a predicate on behaviours or as an equivalence relation on traces. When defined as an equivalence property, secrecy means that the attacker cannot distinguish between two traces with different secrets.

```
# define SecInv     ( !Knows[SEC - 1] )

active proctype SecMonitor() {
  atomic {
    do
      :: !SecInv -> assert(SecInv)
    od
  }
}
```

(a) in SPIN

```
Lemma secrecy : known I (Secret (SEC)).
```

(b) in Coq

```
free SEC: bitstring [private].
query attacker(SEC).
```

(c) in Proverif

Figure 3.13: Secrecy property

To check a security protocol for secrecy, we use an invariant condition specifying that the secret token is not known by the attacker. In SPIN, this invariant condition is checked by an active process monitor that must remain true in all states (Figure 3.13a). In Coq, we define it as a lemma which states that the constant `SEC` of type `Secret` cannot be known by the intruder `I` (Figure 3.13b). In Proverif we declare a private free variable of type `bitstring`[1], and then check if the fact `attacker(SEC)` is ever true (Figure 3.13c).

Authentication is usually defined as a safety property [38] below.

> "If B reaches the end of the protocol and he believes that he has shared the key k with A, then A was indeed his interlocutor and she has shared k."

This definition of authentication implies two things: (1) there is a correspondence between the event that B has finished the protocol with A sending k, and A has started the protocol with B receiving k, and (2) the relationship is injective (i.e. one-to-one). The

[1]Free variables marked as private are global variables which aren't initially known by the intruder [38].

injective property implies that there is no man-in-the-middle. Figure 3.14 shows the events sequence.

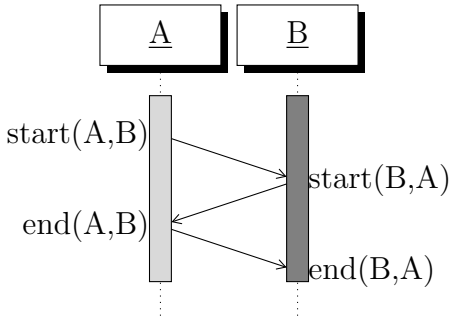

Figure 3.14: Authentication based on events correspondence

From the message sequence, we can draw several conclusions: (1) $end(B, A)$ should always happen after $start(A, B)$, and (2) $end(A, B)$ should always happen after $start(B, A)$. Therefore, checking for authentication requires verifying that two events happen in chronological order, i.e. the client requests authentication to the server ($start(A, B)$), and the server authenticates the client ($end(B, A)$). If only one of these conditions is met but not the other, or if they occur in incorrect order, the authentication process is compromised.

This is modelled by defining two global variables, one for the request and one for the response, defining the authentication invariant as described, and checking that this invariant is true on every reachable state. Figure 3.15 shows how this is done in each of the languages. In SPIN, the macro function `Event` can be specified just in terms of the event type, `START` or `END`, and the participant agents, `A` and `B`, since the number of runs is fixed (Figure 3.15a). Conversely, the same property in Proverif must include the key being exchanged to identify the corresponding session (Figure 3.15c). In Coq, authentication failure is shown by proving lemma $auth\_fail$ (Figure 3.15b), which states that event `end I B A` is true. This `end` function is analogous to the one shown in Figure 3.14, with an extra first argument to indicate the agent with whom `A` has initiated the protocol (in this case `I`).

```
bit START_AB = 0;
bit END_AB = 0;
bit START_BA = 0;
bit END_BA = 0;
# define Event(e,x,y) if \
        :: ((e==START_AB) && (x == A) && (y == B)) -> START_AB = 1 \
        :: ((e==START_BA) && (x == A) && (y == B)) -> START_BA = 1 \
        :: ((e==END_AB) && (x == A) && (y == B)) -> END_AB = 1 \
        :: ((e==END_BA) && (x == A) && (y == B)) -> END_BA = 1 \
        :: else skip \
        fi
# define AuthInv    ( (!END_AB || START_BA) && \
            (!END_BA || START_AB) )

active proctype AuthMonitor()
{
  atomic {
    do
      :: !AuthInv -> assert(AuthInv)
    od
  }
}
```

(a) in SPIN

```
Lemma auth_fail : end I B A.
```

(b) in Coq

```
event start(pkey, pkey, bitstring).
event end(pkey, pkey, bitstring).

query x: pkey, y: pkey, z: bitstring; inj-event(end(x,y,z)) ==> inj-
    event(start(x,y,z)).
```

(c) in Proverif

Figure 3.15: Authentication property

3.5 Protocol Instantiations

To instantiate different runs of a protocol, we can think of each step in the protocol as a function application that takes an instance of a sender agent, a receiver agent, and a message, and outputs a message to be sent. The send and receive actions are modelled as separate functions. This approach is used when modeling protocols inductively. Moreover, in Coq, a

sequence of function applications represent traces. Consequently, a proof of insecurity uses inversion of function application to derive that an attack trace is possible.

On the other hand, agents can be modelled as independent processes, each defining a sequence of statements representing actions. In this case, send and receive operations are input/output events on one or more channels. This approach is common in model checking.

```
1 init {
2  ...
3  run processA(A);
4  run processB(B);
5  run attacker();
6 }
```

(a) in SPIN

```
1 process
2  ...
3  ((!processA(skA, pkB)) | (!processB(skB, pkA)))
```

(b) in Proverif

Figure 3.16: Protocol instantiations

In SPIN, protocols are instantiated by running the agents and attacker processes (Figure 3.16a). The state space is the parallel composition of automata representing agents and attacker actions, which are synchronized by channel events. The search algorithm traverses the state space looking for states that violate the security invariants, in which case, attack traces are reported. Figure 3.16b shows the parallel composition of unbounded instances of `A` and `B` in Proverif. Internally, traces are represented symbolically. Processes defined in the applied pi-calculus are translated into Horn clauses, and then facts about property violations are checked using resolution techniques.

The next Chapter shows how to encode protocols and prove attacks using the generalized framework just described.

CHAPTER 4

CASE STUDIES

In order to demonstrate the generalized modeling framework presented in Chapter 3, and to compare SPIN, Proverif, and Coq (Chapter 5), in this chapter we describe four case studies. Authentication and key establishment protocols proposed by Needham-Schroeder [17], Denning-Sacco [18], Tatebayashi-Matsuzaki-Newman [40], and Diffie-Hellman [16] are modeled and verified with the three verification tools. Next, we describe each of the protocols, their simplified versions, known attacks, and the proposed solutions.

4.1 Needham-Schroeder Public-Key (NSPK) Protocol

The NSPK protocol [17] has become a canonical example for key establishment protocol verification. It provides mutual authentication using asymmetric encryption. In addition, secret nonces N_A and N_B can be used to generate a shared key. The protocol narration is illustrated in Figure 4.1a. The most important communication in the protocol occurs in Steps 3, 6, and 7. To simplify our analysis process, we first consider a reduced version as shown in Figure 4.1b.

The security goals of the protocol are the secrecy of nonces N_A and N_B, and mutual authentication of agents A and B. Authentication means that if B finishes the protocol with A and he believes that he has shared the nonce N_b with A, then A was indeed his interlocutor and she has the shared nonce N_B (authentication of A to B). Moreover, if A finishes the protocol with B and she believes that she has shared the nonce N_A with B, then B was indeed her interlocutor and he has the shared nonce N_A (authentication of B to A).

1. $A \rightarrow S : A, B$
2. $S \rightarrow A : \{PK(B), B\}_{SK(S)}$
3. $A \rightarrow B : \{A, N_A\}_{PK(B)}$
4. $B \rightarrow S : B, A$
5. $S \rightarrow B : \{PK(A), A\}_{SK(S)}$
6. $B \rightarrow A : \{N_A, N_B\}_{PK(A)}$
7. $A \rightarrow B : \{N_B\}_{PK(B)}$

(a) Original version

1. $A \rightarrow B : \{A, N_A\}_{PK(B)}$
2. $B \rightarrow A : \{N_A, N_B\}_{PK(A)}$
3. $A \rightarrow B : \{N_B\}_{PK(B)}$

(b) Simplified version

Figure 4.1: The Needham-Schroeder public key exchange protocol (NSPK)

Nearly 17 years after it was first published, Lowe [24] found that this version of the protocol is vulnerable to a man-in-the-middle attack. An attacker can initiate a session with A and relay the messages to B, making B believe that he is communicating with A. The attack is described in figure 4.2a.

1. $A \rightarrow I : \{A, N_A\}_{PK(I)}$
2. $I \rightarrow B : \{A, N_A\}_{PK(B)}$
3. $B \rightarrow A : \{N_A, N_B\}_{PK(A)}$
4. $A \rightarrow I : \{N_B\}_{PK(I)}$
5. $I \rightarrow B : \{N_B\}_{PK(B)}$

(a) Attack

1. $A \rightarrow B : \{A, N_A\}_{PK(B)}$
2. $B \rightarrow A : \{B, N_A, N_B\}_{PK(A)}$
3. $A \rightarrow B : \{N_B\}_{PK(B)}$

(b) Fixed-version

Figure 4.2: An attack on NSPK and a proposed solution

The result of the attack is that the intruder knows the nonces N_A and N_B, i.e. secrecy fails, and the authentication between A and B also fails (because the interlocutor of A is not B and the interlocutor of B is not A). A proposed solution is to include the identity of B in the second message (Figure 4.2b), i.e. $\{B, N_A, N_B\}_{PK(A)}$.

In order to experiment with different models and state spaces, we encoded both the simplified and full versions of the protocol. As far as modeling, the complexity added on the latter is insignificant.

4.2 Denning-Sacco (DS) Authentication

Denning and Sacco [18] proposed several alternatives to the Needham-Schroeder key distribution protocols. Figure 4.3a shows the full version of one of these alternatives in which a server distributes public keys to A (message 2), which are then forwarded along with a fresh key to B (message 3). Message 4 shows how the fresh key is used to share a secret message between A and B. Figure 4.3b shows a simplified version Denning-Sacco's protocol in which public-key distribution is initially assumed.

1. $A \rightarrow S : A, B$
2. $S \rightarrow A : \{A, PK(A), T\}_{SIG(S)}, \{B, PK(B), T\}_{SIG(S)}$
3. $A \rightarrow B : \{A, PK(A), T\}_{SIG(S)}, \{B, PK(B), T\}_{SIG(S)}, \{\{k, T\}_{SIG(A)}\}_{PK(B)}$
4. $B \rightarrow A : \{Sec\}_k$

(a) Original version

1. $A \rightarrow B : \{\{k\}_{SIG(A)}\}_{PK(B)}$
2. $B \rightarrow A : \{Sec\}_k$

(b) Simplified version

Figure 4.3: The Denning-Sacco key distribution protocol (DS)

The problem, again, relies on the ambiguity of implicit assumptions in the protocol specification. If an attacker I is able to convince A to initiate a conversation while impersonating B, i.e. A uses I's public key to encrypt the first message instead of B's, then the attacker is able to see the secret key and, consequently, any secret message encrypted with such key (Figure 4.4a). This man-in-the-middle attack can be fixed by adding A and B's identities to the first message (Figure 4.4b).

4.3 Tatebayashi-Matsuzaki-Newman (TMN) Protocol

The TMN protocol [40] was designed for key exchange in mobile networks. The full version of the protocol is shown in Figure 4.5a. Again, to simplify the analysis process, we

1. $A \rightarrow I : \{\{k\}_{SIG(A)}\}_{PK(I)}$
2. $I \rightarrow B : \{\{k\}_{SIG(A)}\}_{PK(B)}$
3. $B \rightarrow I : \{Sec\}_k$
4. $I \rightarrow A : \{Sec\}_k$

(a)

1. $A \rightarrow B : \{\{A, B, k\}_{SIG(A)}\}_{PK(B)}$
2. $B \rightarrow A : \{Sec\}_k$

(b)

Figure 4.4: An attack on DS and a proposed solution

show a reduced version that omits the use of timestamps (T_A and T_B) and shared secret keys ($SS(A, S)$ and $SS(B, S)$) in Messages 1 and 3. In the final step, the server sends the Vernam encryption using the nonces N_A and N_B. Figure 4.5b shows the simplified version.

1. $A \rightarrow S : B, \{T_A, SS(A, S), N_A\}_{PK(S)}$
2. $S \rightarrow B : A$
3. $B \rightarrow S : A, \{T_B, SS(B, S), N_B\}_{PK(S)}$
4. $S \rightarrow A : B, V(N_A, N_B)$

(a) Original version

1. $A \rightarrow S : B, \{N_A\}_{PK(S)}$
2. $S \rightarrow B : A$
3. $B \rightarrow S : A, \{N_B\}_{PK(S)}$
4. $S \rightarrow A : B, V(N_A, N_B)$

(b) Simplified version

1. $I(A) \rightarrow S : B, \{N_I\}_{PK(S)}$
2. $S \rightarrow B : A$
3. $B \rightarrow S : A, \{N_B\}_{PK(S)}$
4. $S \rightarrow I(A) : B, V(N_B, N_I)$

(c) An attack

Figure 4.5: The Tatebayashi-Matsuzaki-Newman protocol (TMN)

Several attacks were found on the TMN protocol [113]. An attack happens when an intruder who impersonates A, injects Message 1 and intercepts Message 4. Since the intruder knows the nonce N_I and has received the Vernam encryption $V(N_B, N_I)$, s/he can learn N_B (by applying the XOR function).

4.4 Diffie-Hellman Key-Exchange (DH)

Diffie-Hellman key exchange [16] was one of the first protocols to establish secure communication using public-keys on public channels. It provides a simple and elegant way for two unauthenticated parties to agree on a shared key. In DH, each party calculates the exponentiation of G^{N_i} modulo P, where P and G are values that both parties have agreed upon, i is an agent participating in the protocol, and N_i is a random value generated by i. After exchanging the calculated values, each party can compute the shared key based on the homomorphic property of exponentiation (Figure 4.6a).

1. $A \rightarrow B : P, G, (G^{N_A})\ mod\ P$
2. $B \rightarrow A : (G^{N_B})\ mod\ P$
3. A and B compute the keys: $k = (G^{N_B})^{N_A} = (G^{N_A})^{N_B}$
4. $A \rightarrow B : Sec_k$

(a) Original version

1. $A \rightarrow B : G^{N_A}$
2. $B \rightarrow A : G^{N_B}$
3. $A \rightarrow B : Sec_k$

(b) Simplified version

1. $A \rightarrow I : G^{N_A}$
2. $I \rightarrow B : G^{N_I}$
3. $B \rightarrow I : G^{N_B}$
4. $I \rightarrow A : G^{N_I}$
5. $A \rightarrow I : Sec_k$

(c) A man-in-the-middle attack

Figure 4.6: Diffie-Hellman key exchange protocol

To reduce the state space and optimize verification, we can use a simplified version of the protocol, as shown in Figure 4.6b, and still capture the protocol semantics that are relevant in the symbolic model. In this case, we can assume that the shared values P and G are public, and then demonstrate that a man-in-the-middle attack is possible. A possible

attack is shown in Figure 4.6c, in which an attacker impersonates the honest agents to listen and possibly alter communication between them.

4.5 The Code for the NSPK Protocol

In this Section, we show how to write code for the agent interactions of the simplified NSPK protocol in the modeling languages of each tool. In SPIN and Proverif, we translate the protocol narration into a composition of agent processes. In Coq, we use constructors of inductive types to define allowed protocol transitions.

```
1  proctype processA(mtype a) {
2    mtype x, nx;
3    atomic {
4      RandInterlocutor(x);
5      Start(a, x);
6      comm!Nonce(a), a, PK(x); /* Msg 1 */
7    }
8    atomic {
9      comm?eval(Nonce(a)), nx, eval(PK(a)); /* Msg 2 */
10     End(a, x);
11   }
12   comm!NULL, nx, PK(x); /* Msg 3 */
13 }
14
15 proctype processB(mtype b) {
16   mtype ny, y;
17   atomic {
18     comm?ny, y, eval(PK(b)); /* Msg 1 */
19     IsAgent(y);
20     Start(y, b);
21   }
22   comm!ny, Nonce(b), PK(y); /* Msg 2 */
23   atomic {
24     comm?eval(NULL), eval(Nonce(b)), eval(PK(b)); /* Msg 3 */
25     End(y, b);
26   }
```

Figure 4.7: Promela code for the NSPK protocol

Figure 4.7 shows the process definitions for the simplified version of the NSPK protocol in SPIN. The initiator process starts by randomly choosing between `B` and `I` as an interlocutor (line 4), immediately followed by the event `Start` (line 5), and the first step in the Protocol

Narration 4.1b (line 6). Then, the second message is received (line 9), and the end of the authentication process is declared (line 10). Finally, the nonce `Nb` is sent back to `B` on line 12. Similarly, the receiver follows the protocol description in 4.1b.

```
1  let processA(pkB: pkey, skA: skey) =
2    in(c, pkX: pkey);
3    event startAX(pkX);
4    new Na: bitstring;
5    out (c, aenc((Na, pk(skA)), pkX)); (* Msg 1 *)
6    in(c, m:bitstring); (* Msg 2 *)
7    let (=Na, NX: bitstring) = adec(m, skA) in
8    out(c, aenc(NX, pkX)); (* Msg 3 *)
9    if pkX = pkB then event endYB(pk(skA));
10   out(c, senc(secANa, Na));
11   out(c, senc(secANb, NX)).
12
13 let processB(pkA: pkey, skB: skey) =
14     in(c, m: bitstring); (* Msg 1 *)
15     let (NY: bitstring, pkY: pkey) = adec(m, skB) in
16     event startBY(pkY);
17     new Nb: bitstring;
18     out(c, aenc((NY, Nb), pkY)); (* Msg 2 *)
19     in(c, m3: bitstring); (* Msg 3 *)
20     if Nb = adec(m3, skB) then
21     if pkY = pkA then event endXA(pk(skB));
22     out(c, senc(secBNa, NY));
23     out(c, senc(secBNb, Nb)).
```

Figure 4.8: Proverif code for the NSPK protocol

The Proverif code for the same protocol is shown in Figure 4.8. Agent `A` receives the public key of its interlocutor on line 2. The protocol starts on line 3 and ends on line 9 by signalling the corresponding events. Lines 10-11 send two arbitrary values (`secANa` and `secANb`) encrypted with the secret nonces `Na` and `Nb`. Proverif verifies the secrecy of the nonces by checking whether the values can be decrypted by the attacker.

In Coq, we define the protocol transitions as constructors of the inductive type `send` (Figure 4.9). Agent `A`'s interlocutor, agent `X`, is declared on line 1. In this model, `A` can only start a session with `X`; yet, it is sufficient to show the attack in 4.2a. On the other hand, `B` can start a session with any agent `Y` (line 5 of Figure 4.9). To uniquely identify nonces, they are represented as tuples *(x,y)*, where x is the agent that generated the value, and y is the

```
1 Variable X:agent.
2
3 Inductive send : agent -> message -> Prop :=
4   Tinit   : send A (Enc (P (Nonce (A,X)) (Name A)) X)
5 | T1 : forall Y d, receive B (Enc (P (Nonce d) (Name Y)) B)
6     -> send B (Enc (P (Nonce d) (Nonce (B,Y))) Y)
7 | T2 : forall d, receive A (Enc (P (Nonce (A,X)) (Nonce d)) A)
8     -> send A (Enc (Nonce d) X)
```

Figure 4.9: Coq's code for the NSPK protocol

intended addressee. The protocol narrations for the other protocols are translated into code in a similar manner.

4.6 Experimental Results

We specified models and security properties (secrecy and authentication) for all protocols using SPIN, Proverif, and Coq. In Proverif and Coq, we specify the attacker model representing the class A_1 (Section 3.4.2), a DY attacker with an unbounded message space. As we explained in Chapter 3, attackers with an unbounded message space cannot be modeled in SPIN. Therefore, we specify attacker class A_2 that can generate fixed-length messages and has bounded storage.

The machine used for verification was an Intel Core 2 Quad CPU running at 2.4 GHz, with 4GB RAM, and a 64-bit version of Linux. To verify a model in SPIN, a process analyzer must be compiled first. We compiled SPIN's process analyzer using the *-O2* (allow optimizations) and the *BFS* (Breadth first search) options. The process analyzer was run with the option *-E* to ignore any potential deadlock[1] and check only for reachability properties. We ran Proverif with default options, and used *coqc* and *coqchk* to compile and run Coq proofs in batch mode. The results obtained for each protocol are shown in Figure 4.1.

[1]In SPIN, deadlock occurs if the model does not reach a valid end state. To verify secrecy and authentication, we are not particularly interested in validity of end states. We are strictly concerned with reachability of states that violate the properties.

Table 4.1: Verification execution time in SPIN, Proverif and Coq

	SPIN		**Proverif**	**Coq**
Attacker class	A_3	A_2	A_1	A_1
Protocol				
NSPK (full)	5.8s	–	0.03s	0.3s
NSPK	0.4s	4.15s	0.02s	0.2s
DS	0.06s	0.07s	0.02s	0.2s
TMN	0.1s	0.1s	0.02s	0.2s
DH	0.08s	0.08s	0.01s	0.2s

4.6.1 SPIN

When the full version of the NSPK protocol with attacker A_2 is verified for attacks, SPIN runs out of memory. We try to run verification with bitstate hashing but it does not solve the problem. Consequently, we modify the attacker's message generation procedure to a more restricted version of the DY attacker (as suggested by Ben Henda [34]). In this attacker model, denoted as A_3, the message generation procedure selects new values based on previous message values that are adjacent in the known vector. Although the attacker A_3 does not randomly select messages to be injected, SPIN is able to detect an attack. Verification of the other protocols in SPIN completed successfully with both A_2 and A_3.

For the A_3 attacker, it takes SPIN less than 1 second to find each of the attacks shown in Figures 4.2a, 4.4a, 4.5c, and 4.6c. However, the full version of the NSPK protocol takes significantly longer. Moreover, using attacker A_2, SPIN runs out of memory. These results reveal that the state explosion is an unavoidable problem in SPIN.

Figure 4.10 shows a NSPK's man-in-the-middle attack found by SPIN. This attack corresponds to the attack shown in Figure 4.2a, and violates the authentication property. SPIN provides a trace showing that `B` finishes the protocol believing that he is interacting with `A` but in reality he is communicating with the attacker `I`.

```
1  proc 0 = AuthMonitor
2  proc 1 = :init:
3  using statement merging
4  proc 2 = Initiator
5  proc 3 = Receiver
6  proc 4 = Intruder
7  q\p   0   1   2   3   4
8    1   .   .   comm!Na,A,PKi
9    1   .   .   .   .   comm?Na,A,PKi
10   1   .   .   .   .   comm!Na,A,PKb
11   1   .   .   .   comm?Na,A,PKb
12   1   .   .   .   comm!Na,Nb,PKa
13   1   .   .   comm?Na,Nb,PKa
14   1   .   .   comm!NULL,Nb,PKi
15   1   .   .   .   .   comm?NULL,Nb,PKi
16   1   .   .   .   .   comm!NULL,Nb,PKb
17   1   .   .   .   comm?NULL,Nb,PKb
18 spin: rns.pml:42, Error: assertion violated
19 spin: text of failed assertion: assert((((!(IniCommitAB)||RecRunningAB)
       &&(!(RecCommitAB)||IniRunningAB)))
20 spin: trail ends after 73 steps
```

Figure 4.10: NSPK's authentication failure (man-in-the-middle attack) found by SPIN

4.6.2 Proverif

We created models of the protocols in Proverif using the typed version of the applied pi-calculus. Although all Proverif programs must be well-typed, types are ignored by default during verification [38]. This means that the attacker can generate even ill-typed messages. Enforcing types during verification reduces the state space and, therefore, may be a useful option for verification of protocols that do not terminate.

```
1 RESULT not attacker(secANa[]) is true.
2 RESULT not attacker(secANb[]) is true.
3 RESULT not attacker(secBNa[]) is false.
4 RESULT not attacker(secBNb[]) is false.
5 RESULT inj-event(endYBNb(x_955,y_956)) ==> inj-event(startBYNb(y_956,
     x_955)) is true.
6 RESULT inj-event(endXANa(x_1999,y_2000)) ==> inj-event(startAXNa(y_2000,
     x_1999)) is false.
7 RESULT (even event(endXANa(x_3140,y_3139)) ==> event(startAXNa(y_3139,
     x_3140)) is false.)
```

Figure 4.11: NSPK secrecy and authentication failure output by Proverif

We ran verification with default options on Proverif and found attack traces in an average of 0.02 seconds (Figure 4.1). After each verification run, Proverif outputs more details than SPIN on its results. An interesting example is shown on Figure 4.11, in which secrecy and authentication properties fail for the NSPK protocol. More specifically, lines 3 and 4 show that B can leak the nonces, and line 6 shows that B's authentication of A fails. In fact, line 7 confirms that even the non-injective relation does not hold. B can finish the protocol believing that he is communicating with A, even if A has never run. Proverif also provides a derivation of attacks, as well as attack traces in the pi-calculus.

```
1  new skA creating skA_2938 at {1}
2  out(c, pk(skA_2938)) at {3}
3  new skB creating skB_2936 at {4}
4  out(c, pk(skB_2936)) at {6}
5  in(c, pk(a_2934)) at {8} in copy a_2935
6  event(startAX(pk(a_2934))) at {9} in copy a_2935
7  new Na creating Na_2937 at {10} in copy a_2935
8  out(c, aenc((Na_2937,pk(skA_2938)),pk(a_2934))) at {11} in copy a_2935
9  in(c, aenc((Na_2937,pk(skA_2938)),pk(skB_2936))) at {20} in copy a_2933
10 event(startBY(pk(skA_2938))) at {22} in copy a_2933
11 new Nb creating Nb_2939 at {23} in copy a_2933
12 out(c, aenc((Na_2937,Nb_2939),pk(skA_2938))) at {24} in copy a_2933
13 in(c, aenc((Na_2937,Nb_2939),pk(skA_2938))) at {12} in copy a_2935
14 out(c, aenc(Nb_2939,pk(a_2934))) at {14} in copy a_2935
15 in(c, aenc(Nb_2939,pk(skB_2936))) at {25} in copy a_2933
16 event(endXA(pk(skB_2936))) at {28} in copy a_2933
17 The event endXA(pk(skB_2936)) is executed in session a_2933.
18 A trace has been found.
```

Figure 4.12: Proverif's attack trace showing NSPK authentication failure

An attack trace that shows authentication failure appears on Figure 4.12. Each line corresponds to the pi-calculus statement that is executed, and is annotated with the line number in the model. Moreover, a session identifier is appended at the end of each line. Lines 1-4 model the distribution of the public keys, assumed on the protocol model. On line 5, a new session is created between `A` and `I` (session `a_2935`), and `A` receives the public key of its interlocutor. The event on line 6 indicates that `A` starts the protocol with `X`, represented by `pk(a_2934)`, which in this case is the attacker. On line 7, `A` creates the nonce `Na_2937`,

and sends it on line 8, initiating the first step in the protocol. The execution continues following the steps described on Figure 4.2a. The name `a_2933` is used to represent the session between the attacker and `B`. On line 16, B ends the protocol believing that he has completed the run with A but in reality he has been interacting with the attacker.

Table 4.2: Verification results of secrecy queries and injective properties in Proverif

Protocol	**Proverif Results**
NSPK	Secrecy fails since the attacker can learn N_A and N_B from B. Authentication of A by B fails, even when A is not running the protocol.
DS	Neither secrecy nor authentication can be guaranteed. Authentication fails even when the other party is not running the protocol.
TMN	Secrecy of keys generated by A and B can be leaked by A or S. Moreover, A's authentication of B is compromised, even when B is not running the protocol.
DH	Secret s can be learned by the attacker.

In Proverif, different properties can be verified on a single run. For this reason, following the approach explained by Blanchet [38], we expand our definition of secrecy to obtain more information about the conditions in which this property is satisfied or fails. For instance, the parties in the NSPK protocol use nonces N_A and N_B to generate a shared secret key that will be used later in communication. We send four messages (two on each agent) and add the corresponding queries to reveal which nonce is leaked, and who is the agent that may send this information to the attacker. For example, at the end of process A we send the secrets `secANa` and `secANb` as two separate messages encrypted with `Na` and `Nb`. The queries `attacker(secANa)` and `attacker(secANb)` check if the attacker can learn the secrets. These secrets can only be leaked if the attacker knows `Na` and `Nb`, which were used as keys to encrypt the respective messages. Figure 4.2 summarizes the results obtained in Proverif after verification of the more detailed secrecy queries and the injective properties of authentication, as described in Chapter 3 for all protocols.

4.6.3 Coq

In Coq, we manually derived a proof of the secrecy and authentication failure lemmas. Figure 4.13 shows the proof of authentication failure for the NSPK protocol.

```
1  Lemma ns_auth_fail : receive I B (Enc (Nonce (B,A)) B).
2  apply link with I.
3  apply inject.
4  apply aenc; auto.
5  apply adec with I; auto.
6  apply spy.
7  apply link with A.
8  apply T2.
9  apply link with I.
10 apply inject.
11 apply spy.
12 apply link with B.
13 apply T1.
14 apply link with I.
15 apply inject.
16 apply aenc; auto.
17 apply adec with I; auto.
18 apply spy.
19 apply link with A.
20 apply Tinit.
21 Qed.
```

Figure 4.13: NSPK Authentication failure proof in Coq

`Tinit`, `T1` and `T2` are transitions, `inject` and `spy` are attacker actions, and `aenc` and `adec` are asymmetric encryption and decryption operations. A special rule `link` is used to relate send and receive operations. The proof advances by consistently using the tactic `apply` on the inductive constructors to derive a premise from an implication.

Finally, the proofs were compiled with *coqc* and the theories were checked using *coqchk*. The execution time for each protocol is shown in Figure 4.1. In all cases, Coq was slower than Proverif and, in some cases, faster than SPIN. Since *coqchk* recursively verifies the theory's dependent libraries, the results may be influenced by the number of libraries loaded for each proof.

CHAPTER 5

COMPARATIVE ANALYSIS

In this Chapter, we compare SPIN, Proverif, and Coq from two different perspectives. First, we compare expressiveness of modeling styles for some key language features, which play an important role in representations of security protocols. In particular, we analyze programming style, communication model, encoding of cryptographic primitives, protocol narrations, attacker models, and properties specification. Second, we review the tools in terms of verification problems they can solve. The metrics we use are message space, number of sessions, and attacker models. In addition, we consider correctness of results, automation and guarantee of termination.

5.1 Modeling and Specification

Modeling and property specification are essential components of the verification process. If a model does not accurately capture the semantics of a system, the results will be worthless. Similarly, incorrect specification of properties may lead to false results. Therefore, it is important to analyse each of the verification tools in terms of how precisely they can capture the semantics of cryptographic protocols, i.e. compare language expressiveness. To formally measure expressiveness of different languages, different approaches have been taken. A survey of formal frameworks to compare programming languages can be found in [114]. In the last two decades, there has been a trend to compare program representations based on language translations that preserve observational equivalence [115]. However, as Gorla and Nestmann point out, this approach is not informative enough in regards to the actual quality of a representation [116].

Our approach to comparing the modeling languages of SPIN, Coq, and Proverif is less formal, but has more practical implications. Our conclusions are based on the results of protocols verified in Chapter 4. First, we discuss programming styles and how they influence model specifications. Then, we compare how each of the key elements of security protocols is described in each of the specification languages. Table 5.1 shows a comparison of the programming styles and the techniques used in each case, which we analyze in the following sections.

Table 5.1: Comparison of language features in SPIN, Proverif, and Coq

	SPIN	**Proverif**	**Coq**
Programming style	imperative	functional	functional
Communication model	channels	channels	dependent types
Cryptographic primitives	function macros	constructors/ destructors	inductive types
Protocol narration	roles	roles	set of rules
Attacker model	process	–	set of rules
Properties specification	LTL/monitor process	queries	lemmas

5.1.1 Programming Style

SPIN's modeling language, Promela, has an imperative style similar to the C programming language. Systems can be described in terms of processes, states and transitions between them. An advantage of state-based programming is that it is relatively easy to describe systems with a few possible states. That is the case of authentication and key establishment protocols, which usually consist of a small number of interactions between agents that have different roles. In Chapter 4, we have defined roles as processes and interactions as guarded statements with side-effects. The verification results show that state exploration is a fruitful method as long as the protocols remain small. For example, SPIN finished verification -and found attacks- for the reduced versions of the NSPK, DS, TMN, and DH protocols. However, SPIN cannot find attacks on the full version of the NSPK

protocol because it runs out of memory. SPIN fails due to the state explosion problem. This result suggests that explicit state representation may not work for larger protocols.

There are two ways to deal with this problem in SPIN. First, the state space can be manually pruned by eliminating steps that are not essential to prove the desired properties. In Chapter 4 we described how to apply this simplification technique and find attacks for the selected protocols. Second, we can restrict the attacker model to reduce the state space. For the NSPK protocol, we model the attacker A_3, whose message generation function does not consider all possible messages. For the NSPK protocol, we show in Section 4.6 that we can still find Lowe's attack using the simplified attacker model. However, verification using A_3 attacker model considers less attacks. Consequently, the results are less interesting compared to Proverif and Coq, which can verify the protocols using attackers that are more powerful.

We also use processes to define users interactions in Proverif. In contrast to SPIN, we can define more expressive models and rely on Proverif's internal abstractions to reduce the state space. For all the protocols discussed in Chapter 4, the Proverif models were approximately 30% less verbose than the SPIN models. Verbosity was measured by comparing lines-of-code in each model. From a modeling perspective, Proverif's higher-level specifications are more attractive than SPIN models which must be manually optimized to avoid the state explosion problem. Nevertheless, due to the over-approximations of Proverif's decision procedure, the tool cannot guarantee the correctness of the attacks it finds.

The applied pi-calculus, Proverif's default input language, has a functional style with some imperative features. There are pros and cons of adopting the applied pi-calculus for specification of security protocols. Functional features such as scoping are useful to define secrecy of values [107]. Notwithstanding, reasoning about side-effects (e.g. channel operations) is not always so straightforward as compared to Promela. For instance, consider when one wants to assign several variables to the evaluation of different functions to use later in a send operation. In SPIN, this is achieved by a simple sequence of variable assignments,

followed by a channel operation. In Proverif, one must use nested *let* environments, which are a less natural way of describing a sequence of operations.

Coq's specification language, Gallina, is a descendent of the family of ML functional programming languages. It combines in one language the proof representation and the computation building the proofs [117]. In Gallina, protocol transitions are specified as inductive definitions, which are similar to inference rules. This logical representation intuitively captures the semantics of transitions using type quantifiers and implication, two language features that are not present in Proverif's pi-calculus. Additionally, Gallina has built-in type inference, a feature that is not present in Proverif or SPIN.

5.1.2 Communication Model

In order to model concurrent systems, specification languages provide different mechanisms to describe communication over a network. In the case of cryptographic protocols, channels are a convenient and intuitive way of representing communications between agents. Moreover, protocol logic in process algebras is separated in different blocks, which define a sequence of actions by each of the participant agents. In Chapter 4 we showed how to encode agents as processes that communicate over shared channels in SPIN and Proverif. Although SPIN also allows the use of shared variables, channels are a more natural way to represent communication in cryptographic protocols. For the class of protocols modeled in this thesis, unbuffered synchronized channels are sufficient. To model the protocol interactions, SPIN allows users to define any number of processes that may start concurrently using the keyword `active`. Alternatively, processes may be instantiated by other processes. In our modeling framework we choose the latter option (Figure 3.16a).

Proverif's pi-calculus allows only one top-level process definition. Therefore, processes representing agent actions are specified using `let` environments (Figure 4.7). Then, the top-level process instantiates a parallel composition of agent processes (Figure 3.16b). Channels

in Proverif have similar semantics to rendezvous channels in SPIN; transitions only occur when send and receive operations are synchronized.

In Coq, communications are defined in terms of actions, which are grouped by inductive type definitions. Some separation of logic can be achieved by defining different types for send, receive, and attacker-related actions. However, some definitions are coupled with actions of different type. For example, the constructor representing the attacker's action `inject`, shown in Figure 3.11, has an argument of type `known` and returns a term of type `send`. In this case, `inject` is a constructor of type `send` because of its return type, but its definition is dependent on its argument of type `known`. Types that may contain expressions referring to other types are called dependent types [118]. The deductive system used in Coq requires careful thought. In our modeling framework, inference rules capture the behavior of honest agents and attackers from a global view of the protocol; specification is not clearly separated by roles as in SPIN or Proverif.

Furthermore, in order to model Dolev-Yao attackers, the recipients of sent messages and originators of received messages must be left unspecified, or use variables to allow the attacker along with honest agents as the recipients/originators. Our approach for modeling such variables is described in Section 3.3.1. When one defines protocol transitions in Coq, one must pay special attention to what agents will be involved in the transition. Naturally, defining protocols as interactions between roles in SPIN or Proverif is more intuitive and easier to understand.

5.1.3 Cryptographic Primitives

Cryptographic functions are not built-in to the specification languages of SPIN, Proverif, or Coq. Nonetheless, cryptographic functions can be encoded in the generalized framework discussed in Chapter 3. This approach must consider constructs that are specific to each of the languages. Cryptographic functions in Proverif are defined as constructor/destructor functions. This representation is simple and intuitive.

In Coq, protocol transitions describe cryptographic operations using properties of inductive message constructors. For instance, a message can be either a message m or an encryption of a message *Enc m*. Suppose there is a transition in which an agent X receives an encrypted message m, decrypts it, and sends it concatenated with its own identity. This transition can have *Enc m* as an argument and *(m,X)* as a return type. In this case, an explicit representation of the encryption function is not needed. Despite this specification shortcut for protocol transitions, explicit cryptographic functions must be defined for the attacker model.

In Promela, cryptographic functions can be specified using either macros or inline functions. In any case, SPIN compiler replaces every function call with the body of the function definition. Moreover, neither syntactic construct defines a new variable scope. In general, both constructs operate in a similar manner, except that inline functions cannot be used in assignments. For this reason, we use function macros in our framework. The subtleties of Promela's semantics require careful thought to capture exact protocol semantics. We use atomic blocks to guarantee that protocol transitions and events will execute as a single step. In addition, cryptographic functions must be encoded efficiently; they shall use a minimal number of states to represent operations, and use constants wherever possible. The case studies in Chapter 4 demonstrate that less efficient representations suffer from the state explosion problem.

5.1.4 Attacker Modeling

Perhaps the most interesting aspect of the comparison between SPIN, Proverif, and Coq, is the way in which symbolic attacker models are specified. In SPIN, an attacker is defined as a process, similarly to honest agents. The main difference to honest agents is that the attacker's interaction in the protocol must occur at random instances. One way to achieve this is to use Promela's control-flow blocks, such as `if` or `do`, which provide a way to define action branches that get selected non-deterministically. For attacker model A_3, we

use `do` blocks; for attacker A_2, instead, we use `select` statements which are more elegant syntactic constructs for the same purpose.

In Proverif, the attacker is implicit, that is, models do not need to specify attacker actions. Proverif checks the protocol using the symbolic attacker introduced by Dolev and Yao [19] although a few customizations are also possible. For example, setting the attacker to *passive* restricts the attacker to one that can only read and compute messages, but not inject messages. As we have mentioned in Section 3.4.2, extending our framework to handle passive attackers is not hard. For the Proverif models, setting the attacker to passive will be enough. Moreover, setting the option `keyCompromise` adds the initial assumption that the attacker has obtained the secrets of some sessions, and tells Proverif to check if other session secrets can be compromised.

The attacker in Coq is modeled by adding a dependent type `known` that has constructors for each of the attacker actions. An additional constructor must be added to the type *send* in order to allow the attacker to inject messages. Coq's attacker model, as shown on Figure 3.11, is explicit and straightforward. Specifying a passive attacker is not too hard, as removing the constructor *inject* will suffice.

5.1.5 Property Specification

In SPIN, correctness properties can be specified in either linear temporal logic (LTL) or as assertions embedded within the code. For cryptographic protocols, both secrecy and authentication can be specified directly in the model. For instance, in our case studies, we defined two special processes to monitor that the properties always remained true.

In Proverif, secrecy is usually specified as a reachability property. It is checked by a clause that queries if the attacker has obtained the secret. Authentication is specified as an observational equivalence property, expressed as an implication that relates start and end functions which are triggered accordingly. In the case of Proverif, properties are always specified in the model. In our models, we use Proverif's built-in syntax to check for injective

events that specify authentication properties. Injective events are also useful to check other properties. For example, nested correspondences can be used to verify that messages are received in the correct order. Moreover, Proverif can check for observational equivalence properties. These properties assure that two processes are indistinguishable for an attacker. Properties that fall into this category are non-interference, offline guessing attacks, and the decisional Diffie-Hellman assumption, among others [38].

In Coq, we specify properties as lemmas. To show that a protocol is insecure, we prove a lemma demonstrating that the property fails. The secrecy lemma states that the secret is in the attacker's knowledge vector. Similarly to SPIN and Proverif specification, authentication is defined as an implication of events relating the start and end of the protocol. In Coq, lemmas about security properties are separated from the actual model of the protocol. This approach provides better modularity and decoupling than property specification in SPIN or Proverif.

5.2 Analysis of Verification Results

In order to further contrast SPIN, Proverif and Coq, we compare important characteristics of the verification results that can be obtained as proofs of protocol (in)security. Table 5.2 shows the different factors that we considered for the comparison.

Table 5.2: Comparison of verification results in SPIN, Proverif, and Coq

	SPIN	**Proverif**	**Coq**
Number of sessions	bounded	unbounded	unbounded
Symbolic attacker	restricted	unrestricted	unrestricted
Message space	bounded	approximated	unbounded
Soundness	yes	yes	yes
Completeness	no	no	yes
Correctness of attacks	yes	maybe	yes
Automation	high	medium	low
Guaranteed termination	yes	no	no

We compare these three tools by examining models of protocols for an unbounded number of sessions and unrestricted symbolic attackers with an unrestricted message space. Soundness guarantees that if no attacks are found, then the protocol is guaranteed to satisfy the property under consideration [119]. Completeness is achieved if the tool can recognize all sound protocols as correct. Correctness of attacks refers to false negative results. In some cases, verification may report an attack that cannot occur. Automation, from a user's perspective, is another important metric for cost-benefit analysis. We consider a tool fully automated if it can take a model of a protocol as input and output a result indicating if the protocol meets the specification or not. Finally, it is important to consider if the analysis is guaranteed to terminate, or if it may have an open ending.

5.2.1 Number of Sessions

It has been shown that protocol security is undecidable for protocols with an unbounded number of sessions [80]. Each tool deals with this problem differently. SPIN's modeling language only allows specification of bounded processes. Proverif allows specification of unbounded number of sessions but it cannot guarantee termination. An unbounded number of sessions can be modeled in Coq using inductive definitions with type quantifiers. Nevertheless, Coq's approach cannot be fully automated; it always requires human intervention.

5.2.2 Attacker Model and Message Space

An unrestricted symbolic attacker has full control of the network, s/he can spy or intercept messages at will [19]. Moreover, the attacker can generate any number of messages and has unlimited storage capabilities. Dealing with unrestricted attackers is an undecidable problem [120]. The attackers that can be represented in SPIN are a restricted version of the DY attacker. In SPIN models, the attacker can only store a small number of messages. In fact, our A_3 attacker can only store the last message received. Even in the bounded setting, adding more storage capabilities to our case studies causes the verification to run out of

memory quickly due to the state explosion problem. Proverif's attacker representation is built-in to the tool and, thus, its specification is implicit in the models. Internally, Proverif considers an attacker that can generate an unbounded message space. To cope with the undecidability issue, Proverif performs safe abstractions when it translates protocols into Horn clauses [38]. These approximations preserve soundness but may lead to incomplete results. In fact, on ocassion Proverif outputs that a property "cannot be proved".

In Chapter 3, we showed how to model an unrestricted attacker in Coq. This attacker is able to generate an unbounded number of messages and has infinite storage capabilities. The attacker's message space in Coq is unbounded, since all attacker actions are inductively defined. The implications of adding expressiveness to the attacker model are similar to what happens when we consider unbounded number of sessions. In this case, analysis using attackers with an unbounded message space is an undecidable problem [120]. Consequently, verification cannot be fully automated. This is demonstrated in Coq by the fact that all proofs of the protocols, analyzed in Chapter 4, require human guidance to be completed.

5.2.3 Correctness of Attacks

In general, correctness of verification tools is measured in terms of soundness and completeness. Traditionally, soundness has been stated with respect to the tool's results. In particular, a technique is called sound if the properties that it proves are always true [119]. On the other hand, completeness shows that the tool can prove the properties on all models that are considered sound.

Using these definitions, SPIN and Proverif results are all sound but incomplete. That is, if we can prove that a protocol model satisfies a set of properties, then the result is correct. However, in general there are protocols and properties that neither SPIN nor Proverif can prove correct, even though they may be. Our case study for the full version of the NSPK demonstrates this fact in SPIN, since verification could not complete. Moreover, the source

of incompleteness is coupled with the state explosion problem; modeling unbounded sessions and unrestricted DY attackers is not feasible in SPIN.

Although Proverif accepts protocol models in the unbounded space, its resolution procedure may report false attacks and, therefore, the technique is incomplete. In this thesis, verification for the NSPK protocol reported a valid attack; however, a different model of the same protocol presented in [121] reports a false attack.

In Coq, proofs can be constructed for protocols with unbounded sessions and unrestricted attackers. A proof of protocol insecurity is useful to demonstrate that an attack trace is derivable. In general, Coq's proofs can provide greater confidence that a protocol is secure. Indeed, Coq can be used to derive user-defined lemmas that are not provable by the other tools.

5.2.4 Automation

In this thesis, the automation metric describes the amount of interaction required from the user in the verification process. Our definition of an automated tool does not consider the modeling phase. In every case, modeling a protocol is a manual process, and the effort involved depends on the protocol at hand.

In terms of the verification process, SPIN is highly automated. We verified the four models and obtained a yes/no answer for all the protocols without requiring any user interaction. In addition, SPIN output message sequence diagrams detailing the attack traces.

Verification using Proverif is somewhat automated. In our case studies, Proverif provided a positive result or an attack trace for all the protocols that we verified; i.e. interaction was not required to obtain a pass or fail answer. Nevertheless, Proverif cannot always decide if a protocol is secure or not (as shown in [121]). In such a case, the user has to manually examine the results to conclude if an attack derivation corresponds to a real attack or if it is a false negative. Even when Proverif finds an attack trace, translation from Proverif's output to a more user-friendly format, e.g. a message sequence diagram, must be done manually.

Proof construction in Coq requires considerable effort. Although proof tactics provide automation of trivial tasks, the process is highly dependent on human guidance. To prove that a property is false, finding a counter-example will suffice. This is the case in the protocols analyzed in Chapter 4. Nevertheless, proving that a protocol is correct with respect to some properties specification may be a more challenging task. Overall, theorem proving has a steep learning curve. Even with the help provided by Coq, finding attacks for the case studies in Chapter 4 requires more effort than in SPIN or Proverif.

5.2.5 Termination

Constructing proofs in Coq may be hard, time consuming and, in many cases, one may not be able to prove some properties. In such cases, the open questions are, *is it possible to prove the protocol's security property? Is there a possible attack for this protocol?* If an attack is found, we know that the protocol is insecure. But if no proof of security exists, there is no certainty about answering the security questions.

On the other hand, SPIN is guaranteed to terminate for verification of a protocol with a bounded number of sessions. However, the state explosion problem could prevent obtaining a result if the system runs out of memory. This is the case for the full version of the NSPK protocol analyzed in Chapter 4. Despite this fact, in general, key establishment and authentication protocols can be abstracted to models that capture the essential semantics while keeping the state space relatively small. We have abstracted the models and showed that some attacks were found.

Proverif does not guarantee termination due to the infinite sessions and state space considered. For the class of protocols that we studied, however, non-termination is not an issue. In fact, the results obtained demonstrate that Proverif is more efficient than the other tools in terms of execution time.

5.3 Chapter Summary

In this chapter we have analyzed SPIN, Proverif, and Coq for modeling and verification of key establishment and authentication protocols. Promela, SPIN's modeling language, has an imperative style that resembles the C programming language. Models for Proverif can be described using the applied pi-calculus, a process algebra that has been specifically designed for security specification. From a modeling perspective, we have compared the different language constructs that were used in the case studies of Chapter 4. SPIN and Proverif provide channels which are convenient ways of representing communication between agents that have specific roles. In Coq, Gallina's inductive definitions with quantified and dependent types, on the other hand, are useful language features that can be used to represent cryptographic properties of protocols.

We have shown that choosing the right tool always implies a trade-off. In general, it is easier to model a system using SPIN's state representation than using the pi-calculus or logical inference rules. However, SPIN can only verify a bounded number of sessions and message space. Moreover, Proverif verifies an unbounded number of sessions and message space at the cost of potentially not terminating or reporting false attacks. In practice, however, these issues may avoidable by modifying the models or appealing to other techniques [121]. On the other hand, Coq theories with security proofs are not bounded and can represent an unrestricted symbolic attacker. We have shown that protocol insecurity can be proved in Coq and that the results are sound. On the other hand, a proof that demonstrates that the properties hold may be harder to construct but has the benefit of completeness. This result is not obtainable in SPIN or Proverif.

CHAPTER 6

CONCLUSION

6.1 Thesis Summary

In this thesis, we have presented a generalized framework for modeling security protocols and specification properties for verification using the model-checkers SPIN and Proverif, and the theorem prover Coq. We have shown how to systematically translate informal protocol narrations into formal models to be used as input into each of the tools.

Moreover, we have formally analyzed several examples of a particular class of cryptographic protocols using three alternative methods. We used SPIN to analyze explicit state models written in Promela; symbolic models in the pi-calculus were verified using Proverif; and finally we proved lemmas that show protocol insecurity using the theorem prover Coq. Our experiments extend the number of case-studies of protocol analysis using formal methods, which are relatively scarce compared to other fields.

The most important contribution of this study is the comparison of the three tools presented in Chapter 5. We have analyzed the modeling language features and compared the verification process for each of the tools based on important metrics for several case studies that were presented in Chapter 4. While the most comprehensive results can be obtained with theorem proving, our results show that model checking is a practical alternative that offers higher levels of automation. SPIN does a good job at analysing simple protocols in the bounded case; it finds attacks for all the protocols that we verify. Proverif automatically finds attacks in the unbounded case, a result that is not achievable in SPIN and requires considerable manual effort in Coq.

6.2 Future Work

One of the issues with formal verification of security protocols using a symbolic model for cryptographic primitives and the attacker is that strong assumptions must be made. In the symbolic model, cryptography is unbreakable and the attacker has unlimited computational ability. A computational model of security which considers limitations of the attacker has been proposed since the 1980s [122, 111], and has gained recent popularity [123, 124, 125, 126].

In the computational model, verification considers computability of the adversary, and the probability that the keys will be compromised. For example, symmetric encryption is considered secure if the attacker has a minimal probability of distinguishing between two encrypted values of the same length [127].

In general, protocols that are found secure in the symbolic model are not always safe in the computational model [128]. In other words, attacks found using the computational model can be mapped to the symbolic model, but mapping in the inverse direction is not always achievable. For proving security of protocols in the computational model, two approaches have been taken. The first one, computational soundness, focuses on proving that a security proof in the symbolic model corresponds to a proof in the computational approach [129].

The second approach pioneered by Laud [130], called the direct approach, aims to prove security directly in the computational model. A survey of related literature of both methods can be found in [131]. Blanchet extended Laud's ideas and implemented Cryptoverif [132], a verification tool for the computational model. Another recently proposed tool is EasyCrypt [123], which checks proof sketches and compiles them into verifiable proofs in CertiCrypt [133] (a framework that uses Coq for game-proof construction). The main issues with these recent tools is that they can only be applied to some general security properties, e.g. secrecy and authentication, so the applicability of these tools to verify other security problems yet remains an open question. Moreover, the number of case studies in computational verification of security protocols is still limited.

Another research trend is to bridge the gap between verifiable models and actual system implementations. There are two approaches on this line of work, model extraction and code generation. Avalle et al. [46] published a comprehensive survey of recent work in these areas. Model extraction is based on the classical theory of abstractions [134]; implementation details are extricated and properties proved on the simplified model. In general, programs must be annotated to allow the algorithms to identify which parts of the program can be abstracted. Automated code generation follows a model-driven software engineering approach, a secure high-level model is iteratively refined into a concrete implementation. Some tools also provide a proof asserting that the implementation is sound with respect to the abstract model.

In any case, neither model extraction nor code generation always find logical flaws in protocols. An example is the logical flaw found on the TLS protocol, which was previously claimed to be verified on implementations [46]. The problem was that all studies considered a reduced version of the protocol to cope with verification limitations, i.e. the proofs were sound but not complete. Extending verification methods to deal with larger models seems like an arduous task. Nevertheless, some recent work has been done in composability [135, 136] of formal methods to deal with this issue.

Moreover, most of the existing work in model extraction and code generation focuses on symbolic models. Linking implementations to computational proofs would provide greater trust. Some preliminary work has been done on this field [137, 138], but the complexity of dealing with the abstract models may be preventing this technique to be adopted by non-experts [46]. A recent work uses Java both as a modeling and implementation language (JavaSPI) [139], however the approach may generate code that is slow and not optimal. Future work, in this case, could focus on translating protocol models into C language, for instance, or optimizing the Java code generated by JavaSPI.

In this thesis, we considered a restricted set of protocols, namely those for authentication and key establishment. Authentication and secrecy are the two most interesting security properties of the protocols we studied. Nonetheless, other classes of protocols exist

for specific applications (e.g. smart-cards, e-commerce, e-voting or e-passports). Only a few approaches have been focused on verification of application specific properties [140]. In addition, some protocols aim to enforce security properties other than authentication and (syntactic) secrecy; for example anonimity, indistinguishability (strong secrecy), and unlinkability in privacy protocols. On the theoretical side, Bruso et al. [141] have addressed a first issue with these properties, defining these security properties more precisely. On the practical side, more case studies would pave the road to maturity in this area.

LIST OF REFERENCES

[1] Ross Anderson and Roger Needham. Programming satan's computer. In *Computer Science Today*, pages 426–440. Springer, 1995.

[2] Fred B Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, 2000.

[3] Leslie Lamport. Proving the correctness of multiprocess programs. *Software Engineering, IEEE Transactions on*, (2):125–143, 1977.

[4] Bowen Alpern and Fred B Schneider. Defining liveness. *Information processing letters*, 21(4):181–185, 1985.

[5] International Organization for Standardization. Information processing systems: Open systems interconnection?basic reference model. part 2: Security architecture. iso 7498-2, 1989.

[6] Karl Rihaczek. The harmonized itsec evaluation criteria. *Computers & Security*, 10(2):101–110, 1991.

[7] Butler W. Lampson. Computer security in the real world. In *Annual Computer Security Applications Conference*, 2000.

[8] Michael Ryan Clarkson. *Quantification and Formalization of Security*. PhD thesis, Cornell University, 2010.

[9] Ravi S Sandhu, Edward J Coyne, Hal L Feinstein, and Charles E Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[10] Butler W Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24, 1974.

[11] Samrat Mondal and Shamik Sural. Security analysis of temporal-rbac using timed automata. In *Information Assurance and Security, 2008. ISIAS'08. Fourth International Conference on*, pages 37–40. IEEE, 2008.

[12] Graham Hughes and Tevfik Bultan. Automated verification of access control policies using a sat solver. *International journal on software tools for technology transfer*, 10(6):503–520, 2008.

[13] Vincent C Hu, D Richard Kuhn, Tao Xie, and JeeHyun Hwang. Model checking for verification of mandatory access control models and properties. *International Journal of Software Engineering and Knowledge Engineering*, 21(01):103–127, 2011.

[14] Bruce Schneier. *Applied cryptography: protocols, algorithms, and source code in C*. john wiley & sons, 1996.

[15] Ronald L. Rivest. Cryptography. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume 1, pages 717–755. Elsevier, 1990.

[16] Whitfield Diffie and Martin E Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976.

[17] Roger M Needham and Michael D Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[18] Dorothy E Denning and Giovanni Maria Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, 1981.

[19] Danny Dolev and Andrew C Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, 1983.

[20] Michael Burrows, Martin Abadi, and Roger M Needham. A logic of authentication. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, volume 426, pages 233–271. The Royal Society, 1989.

[21] Kipp Hickman and Taher Elgamal. The ssl protocol. *Netscape Communications Corp*, 501, 1995.

[22] Tatu Ylonen. Ssh–secure login connections over the internet. In *Proceedings of the 6th USENIX Security Symposium*, volume 37, 1996.

[23] Martín Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. *IEEE transactions on Software Engineering*, 22(1):6–15, 1996.

[24] Gavin Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information processing letters*, 56(3):131–133, 1995.

[25] Adam Stubblefield, John Ioannidis, Aviel D Rubin, et al. Using the fluhrer, mantin, and shamir attack to break wep. In *NDSS*, 2002.

[26] Henri Gilbert, Matthew Robshaw, and Herve Sibert. Active attack against hb+: a provably secure lightweight authentication protocol. *Electronics Letters*, 41(21):1169–1170, 2005.

[27] 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.

[28] Noudjoud Kahya, Nacira Ghoualmi, and Pascal Lafourcade. Formal analysis of pkm using scyther tool. In *Information Technology and e-Services (ICITeS), 2012 International Conference on*, pages 1–6. IEEE, 2012.

[29] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This poodle bites: Exploiting the ssl 3.0 fallback, 2014.

[30] Yinzhi Cao, Yan Shoshitaishvili, Kevin Borgolte, Christopher Kruegel, Giovanni Vigna, and Yan Chen. Protecting web-based single sign-on protocols against relying party impersonation attacks through a dedicated bi-directional authenticated secure channel. In *Research in Attacks, Intrusions and Defenses*, pages 276–298. Springer, 2014.

[31] Gerard J Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[32] Stephan Merz. Model checking: A tutorial overview. In *Modeling and verification of parallel processes*, pages 3–38. Springer, 2001.

[33] Paolo Maggi and Riccardo Sisto. Using spin to verify security properties of cryptographic protocols. In *Model Checking Software*, pages 187–204. Springer, 2002.

[34] Noomene Ben Henda. Generic and efficient attacker models in spin. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, pages 77–86. ACM, 2014.

[35] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The coq proof assistant reference manual: Version 6.1. 1997.

[36] Dominique Bolignano. Formal verification of cryptographic protocols. In *Proceedings of the third ACM Conference on Computer and Communication Security*, 1996.

[37] Chen Chen, Limin Jia, Hao Xu, Cheng Luo, Wenchao Zhou, and Boon Thau Loo. A program logic for verifying secure routing protocols. In *Formal Techniques for Distributed Objects, Components, and Systems*, pages 117–132. Springer, 2014.

[38] Bruno Blanchet, Ben Smyth, and Vincent Cheval. Proverif 1.88 pl1: Automatic cryptographic protocol verifier, user manual and tutorial. 2014.

[39] Martín Abadi and Andrew D Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proceedings of the 4th ACM conference on Computer and communications security*, pages 36–47. ACM, 1997.

[40] Makoto Tatebayashi, Natsume Matsuzaki, and David B Newman Jr. Key distribution protocol for digital mobile communication systems. In *Advances in Cryptology-CRYPTO89 Proceedings*, pages 324–334. Springer, 1990.

[41] Bruce Schneier. Cryptographic design vulnerabilities. *Computer*, 31(9):29–33, 1998.

[42] FDR2 User Manual. Failures-divergence refinement. 2000.

[43] Catherine Meadows. The nrl protocol analyzer: An overview. *The Journal of Logic Programming*, 26(2):113–131, 1996.

[44] Reema Patel, Bhavesh Borisaniya, Avi Patel, Dhiren Patel, Muttukrishnan Rajarajan, and Andrea Zisman. Comparative analysis of formal model checking tools for security protocol verification. In *Recent Trends in Network Security and Applications*, pages 152–163. Springer, 2010.

[45] Suvansh Lal, Mohit Jain, and Vikrant Chaplot. Approaches to formal verification of security protocols. *arXiv preprint arXiv:1101.1815*, 2011.

[46] Matteo Avalle, Alfredo Pironti, and Riccardo Sisto. Formal verification of security protocol implementations: a survey. *Formal Aspects of Computing*, 26(1):99–123, 2014.

[47] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and computation*, 76(2):95–120, 1988.

[48] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[49] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[50] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):455–495, 1982.

[51] Edmund M Clarke and E Allen Emerson. *Design and synthesis of synchronization skeletons using branching time temporal logic*. Springer, 1982.

[52] Doron Peled. Ten years of partial order reduction. In *Computer Aided Verification*, pages 17–28. Springer, 1998.

[53] Robert Morris. Scatter storage techniques. *Communications of the ACM*, 11(1):38–44, 1968.

[54] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[55] Kenneth L McMillan. *Symbolic model checking*. Springer, 1993.

[56] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using smt solvers instead of sat solvers. *International Journal on Software Tools for Technology Transfer*, 11(1):69–83, 2009.

[57] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[58] John C Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using murφ. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 141–151. IEEE, 1997.

[59] Manuel Cheminod, Ivan Cibrario Bertolotti, Luca Durante, Riccardo Sisto, and Adriano Valenzano. Experimental comparison of automatic tools for the formal analysis of cryptographic protocols. In *Dependability of Computer Systems, 2007. DepCoS-RELCOMEX'07. 2nd International Conference on*, pages 153–160. IEEE, 2007.

[60] Security protocols open repository, 2003.

[61] Cas JF Cremers, Pascal Lafourcade, and Philippe Nadeau. Comparing state spaces in automatic security protocol analysis. In *Formal to Practical Security*, pages 70–94. Springer, 2009.

[62] François Dupressoir, Andrew D Gordon, Jan Jürjens, and David A Naumann. Guiding a general-purpose c verifier to prove cryptographic protocols. *Journal of Computer Security*, 22(5):823–866, 2014.

[63] David L Dill, Andreas J Drexler, Alan J Hu, and C Han Yang. Protocol verification as a hardware design aid. In *ICCD*, volume 92, pages 522–525. Citeseer, 1992.

[64] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In *Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009.

[65] Benjamin C Pierce, Chris Casinghino, Michael Greenberg, Vilhelm Sjoberg, and Brent Yorgey. Software foundations. *Course notes, online at http://www. cis. upenn. edu/˜ bcpierce/sf*, 2(3.1):3–2, 2010.

[66] Lawrence C Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media, 1994.

[67] Zhaohui Luo and Robert Pollack. *LEGO proof development system: User's manual*. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science, 1992.

[68] Sam Owre, John M Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *Automated Deduction?CADE-11*, pages 748–752. Springer, 1992.

[69] Matt Kaufmann, J Strother Moore, and Panagiotis Manolios. *Computer-aided reasoning: an approach*. Kluwer Academic Publishers, 2000.

[70] Bruno Dutertre and Steve Schneider. *Embedding CSP in PVS: an application to authentication protocols*. Queen Mary and Westfield College, Department of Computer Science, 1997.

[71] Lawrence C Paulson. *Mechanized proofs of security protocols: Needham-Schroeder with public keys*. University of Cambridge, Computer Laboratory, 1997.

[72] Bo Meng, Wei Huang, and Zimao Li. Automated proof of resistance of denial of service attacks using event with theorem prover. *Journal of Computers*, 8(7):1728–1741, 2013.

[73] Simon Meier, Cas Cremers, and David Basin. Efficient construction of machine-checked symbolic protocol security proofs. *Journal of Computer Security*, 21(1):41–87, 2013.

[74] Alessandro Armando, Roberto Carbone, and Luca Compagna. Satmc: a sat-based model checker for security-critical systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 31–45. Springer, 2014.

[75] Luca Compagna. Sat-based model-checking of security protocols. *Phd, Universita di Genova, Italy, and University of Edinburgh, UK*, 2005.

[76] David Basin, Sebastian Mödersheim, and Luca Vigano. *An on-the-fly model-checker for security protocol analysis*. Springer, 2003.

[77] Gavin Lowe. Casper: A compiler for the analysis of security protocols. *Journal of computer security*, 6(1):53–84, 1998.

[78] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and Andrew W Roscoe. Fdr3a modern refinement checker for csp. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 187–201. Springer, 2014.

[79] Michaël Rusinowitch and Mathieu Turuani. Protocol insecurity with finite number of sessions is np-complete. 2001.

[80] Nancy Durgin, Patrick Lincoln, John Mitchell, and Andre Scedrov. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12(2):247–311, 2004.

[81] Santiago Escobar, Catherine Meadows, and José Meseguer. Maude-npa: Cryptographic protocol analysis modulo equational properties. In *Foundations of Security Analysis and Design V*, pages 1–50. Springer, 2009.

[82] Liana Bozga, Yassine Lakhnech, and Michaël Périn. Hermes: An automatic tool for verification of secrecy in security protocols. In *Computer Aided Verification*, pages 219–222. Springer, 2003.

[83] Mathieu Turuani. The cl-atse protocol analyser. In *Term Rewriting and Applications*, pages 277–286. Springer, 2006.

[84] Yohan Boichut, Nikolai Kosmatov, and Laurent Vigneron. Validation of prouvé protocols using the automatic tool ta4sp. *TFIT*, 6:467–480, 2006.

[85] Cas JF Cremers. The scyther tool: Verification, falsification, and analysis of security protocols. In *Computer Aided Verification*, pages 414–418. Springer, 2008.

[86] Ralf Kusters and Tomasz Truderung. Using proverif to analyze protocols with diffie-hellman exponentiation. In *Computer Security Foundations Symposium, 2009. CSF'09. 22nd IEEE*, pages 157–171. IEEE, 2009.

[87] Vincent Cheval and Bruno Blanchet. Proving more observational equivalences with proverif. In *Principles of Security and Trust*, pages 226–246. Springer, 2013.

[88] Martın Abadi and Cédric Fournet. Private authentication. *Theoretical Computer Science*, 322(3):427–476, 2004.

[89] Hazem A Elbaz. Analysis and verification of a key agreement protocol over cloud computing using scyther tool. *International Journal of Cloud Computing and Services Science (IJ-CLOSER)*, 3(6), 2015.

[90] Colin Boyd and Anish Mathuria. *Protocols for authentication and key establishment*. Springer Science & Business Media, 2013.

[91] Martn Abadi. Security protocols and their properties. In *Foundations of Secure Computation, NATO Science Series*, pages 39–60. IOS Press, 2000.

[92] Chiara Bodei, Mikael Buchholtz, Pierpaolo Degano, Flemming Nielson, and H Riis Nielson. Automatic validation of protocol narration. In *Computer Security Foundations Workshop, 2003. Proceedings. 16th IEEE*, pages 126–140. IEEE, 2003.

[93] Sébastien Briais and Uwe Nestmann. A formal semantics for protocol narrations. *Theoretical Computer Science*, 389(3):484–511, 2007.

[94] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[95] FJ Thayer Fabrega, Jonathan C Herzog, and Joshua D Guttman. Strand spaces: why is a security protocol correct? In *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on*, pages 160–171. IEEE, 1998.

[96] CJF Cremers, S Mauw, and EP De Vink. Formal methods for security protocols: three examples of the black-box approach. *NVTI newsletter*, 7, 2003.

[97] Dan M Nessett. A critique of the burrows, abadi and needham logic. *ACM SIGOPS Operating Systems Review*, 24(2):35–38, 1990.

[98] Colin Boyd and Wenbo Mao. On a limitation of ban logic. In *Advances in CryptologyEUROCRYPT93*, pages 240–247. Springer, 1994.

[99] Paul S Grisham, Charles L Chen, Sarfraz Khurshid, and Dewayne E Perry. Validation of a security model with the alloy analyzer, 2006.

[100] Bill Roscoe et al. Model- checking csp. 1994.

[101] Gavin Lowe. Casper: A compiler for the analysis of security protocols. In *Computer Security Foundations Workshop, 1997. Proceedings., 10th*, pages 18–30. IEEE, 1997.

[102] Peter Ryan and Steve A Schneider. *The modelling and analysis of security protocols: the csp approach*. Addison-Wesley Professional, 2001.

[103] 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. IEEE, 1999.

[104] Robin Milner. *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.

[105] Michael Baldamus, Joachim Parrow, and Björn Victor. Spi calculus translated to π-calculus preserving may-tests. In *Logic in Computer Science, 2004. Proceedings of the 19th Annual IEEE Symposium on*, pages 22–31. IEEE, 2004.

[106] Steve Kremer and Robert Kunnemann. Automated analysis of security protocols with global state. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 163–178. IEEE, 2014.

[107] Martín Abadi. Security protocols: Principles and calculi. In *Foundations of security analysis and design IV*, pages 1–23. Springer, 2007.

[108] Christine Paulin-Mohring. Laser 2011 summer school. https://www.lri.fr/~paulin/LASER/. Accessed: 2015-04-23.

[109] Gerard J Holzmann. *The SPIN model checker: Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.

[110] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, et al. The coq proof assistant reference manual–version 7.2. Technical report, Technical Report 0255, INRIA, 2002.

[111] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299, 1984.

[112] Gavin Lowe. A hierarchy of authentication specifications. In *Computer Security Foundations Workshop, 1997. Proceedings., 10th*, pages 31–43. IEEE, 1997.

[113] Choonsik Park, Kaoru Kurosawa, Tatsuaki Okamoto, and Shigeo Tsujii. On key distribution and authentication in mobile radio networks. In *Advances in CryptologyEUROCRYPT93*, pages 461–465. Springer, 1994.

[114] Matthias Felleisen. On the expressive power of programming languages. In *ESOP'90*, pages 134–151. Springer, 1990.

[115] John C Mitchell. On abstraction and the expressive power of programming languages. *Science of Computer Programming*, 21(2):141–163, 1993.

[116] Daniele Gorla and Uwe Nestmann. Full abstraction for expressiveness: History, myths and facts. *Mathematical Structures in Computer Science*, pages 1–16, 2014.

[117] Susmit Sarkar. *A dependently typed programming language, with applications to foundational certified code systems*. ProQuest, 2009.

[118] Adam Chlipala. Certified programming with dependent types, 2011.

[119] Bruno Blanchet. Automatic verification of correspondences for security protocols. *arXiv preprint arXiv:0802.3444*, 2008.

[120] Zhiyao Liang and Rakesh M Verma. Improving techniques for proving undecidability of checking cryptographic protocols. In *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*, pages 1067–1074. IEEE, 2008.

[121] Tom Chothia, Ben Smyth, and Chris Staite. Automatically checking commitment protocols in proverif without false attacks. In *Principles of Security and Trust*, pages 137–155. Springer, 2015.

[122] Andrew C Yao. Theory and application of trapdoor functions. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*, pages 80–91. IEEE, 1982.

[123] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology–CRYPTO 2011*, pages 71–90. Springer, 2011.

[124] Michael Backes, Ankit Malik, and Dominique Unruh. Computational soundness without protocol restrictions. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 699–711. ACM, 2012.

[125] Gilles Barthe, Juan Manuel Crespo, Benjamin Grégoire, César Kunz, Yassine Lakhnech, Benedikt Schmidt, and Santiago Zanella-Béguelin. Fully automated analysis of padding-based encryption in the computational model. In *Proceedings of the 2013*

ACM SIGSAC conference on Computer & communications security, pages 1247–1260. ACM, 2013.

[126] Gergei Bana and Hubert Comon-Lundh. A computationally complete symbolic attacker for equivalence properties. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 609–620. ACM, 2014.

[127] Mihir Bellare, Anand Desai, Eron Jokipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption. In *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, pages 394–403. IEEE, 1997.

[128] Bruno Blanchet. Security protocol verification: Symbolic and computational models. In *Proceedings of the First international conference on Principles of Security and Trust*, pages 3–29. Springer-Verlag, 2012.

[129] Martin Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption)*. *Journal of cryptology*, 15(2):103–127, 2002.

[130] Ilja Tshahhirov and Peeter Laud. Digital signature in automatic analyses for confidentiality against active adversaries. In *NordSec*, pages 29–41, 2005.

[131] Véronique Cortier, Steve Kremer, and Bogdan Warinschi. A survey of symbolic methods in computational analysis of cryptographic systems. *Journal of Automated Reasoning*, 46(3-4):225–259, 2011.

[132] Bruno Blanchet. Cryptoverif: Computationally sound mechanized prover for cryptographic protocols. In *Dagstuhl seminar Formal Protocol Verification Applied*, page 117, 2007.

[133] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. *ACM SIGPLAN Notices*, 44(1):90–101, 2009.

[134] Edmund M Clarke, Orna Grumberg, and David E Long. Model checking and abstraction. *ACM transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542, 1994.

[135] Suzana Andova, Cas Cremers, Kristian Gjøsteen, Sjouke Mauw, Stig F Mjølsnes, and Saša Radomirović. A framework for compositional verification of security protocols. *Information and Computation*, 206(2):425–459, 2008.

[136] Anupam Datta, Ante Derek, John C Mitchell, and Arnab Roy. Protocol composition logic (pcl). *Electronic Notes in Theoretical Computer Science*, 172:311–358, 2007.

[137] Karthikeyan Bhargavan, Cédric Fournet, Ricardo Corin, and Eugen Zalinescu. Cryptographically verified implementations for tls. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 459–468. ACM, 2008.

[138] David Cadé and Bruno Blanchet. From computationally-proved protocol specifications to implementations. In *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on*, pages 65–74. IEEE, 2012.

[139] Matteo Avalle, Alfredo Pironti, Davide Pozza, and Riccardo Sisto. Javaspi: A framework for security. *Developing and Evaluating Security-Aware Software Systems*, page 225, 2012.

[140] Nina Moebius, Kurt Stenzel, and Wolfgang Reif. Formal verification of application-specific security properties in a model-driven approach. In *Engineering secure software and systems*, pages 166–181. Springer, 2010.

[141] Mayla Brusó, Konstantinos Chatzikokolakis, Sandro Etalle, and Jerry Den Hartog. Linking unlinkability. In *Trustworthy Global Computing*, pages 129–144. Springer, 2013.