Automated Formal Methods for Security Protocol Engineering

Alfredo Pironti, Davide Pozza, Riccardo Sisto
Dip. di Automatica e Informatica, Politecnico di Torino, Italy

ABSTRACT

Designing and implementing security protocols are known to be error-prone tasks. Recent research progress in the field of formal methods applied to security protocols has enabled the use of these techniques in practice. The objective of this chapter is to give a circumstantial account of the state-of-the-art reached in this field, showing how formal methods can help in improving quality. Since automation is a key factor for the acceptability of these techniques in the engineering practice, the chapter focuses on automated techniques and illustrates in particular how high-level protocol models in the Dolev-Yao style can be automatically analyzed and how it is possible to automatically enforce formal correspondence between an abstract high-level model and an implementation.

1. INTRODUCTION

Security protocols enable distributed interactions to occur securely even over insecure networks. Well known examples are the protocols for secure authentication or key exchange that we use daily.

With the growth of connectivity over the internet, there is an increasing demand for secure distributed ICT systems, which in turn is rapidly widening the spread and scope of security protocols. Web services, grid computing, electronic commerce and SCADA systems for remote control are just few examples of the many emerging distributed applications that need security. In addition to the bare data secrecy and authenticity goals, which characterize the most classical protocols, new different goals such as non-repudiation or secure transactions in electronic commerce systems have recently started to be considered as desirable, with several new protocols being proposed. The role of standards is fundamental in this field, because distributed applications rely on interoperability. However, the variegated needs of applications may sometimes call for proprietary solutions as well, when standards do not (yet) cover needs adequately. So, tasks such as designing and implementing security protocols are becoming less esoteric and more common, either as part of new standards development or as part of new products development.

These tasks are generally quite critical, because of the delicate role security protocols normally play in protecting valuable assets. Furthermore, despite their apparent simplicity, security protocols are very difficult to get right, even when developed and reviewed by experts, because they add the difficulty of taking into account all the possible operations of malicious parties to the ones of concurrent operation in a distributed environment. It is then widely recognized that the rigorous approach of formal methods plays a key role in developing security protocol designs and implementations at the desired quality level.

Although using formal methods is still considered difficult and requires expertise, research on formal methods in general, and on their application to security protocols in particular, has recently made much progress. Therefore, difficulty is progressively mitigated by the greater automation level and user friendliness that can be achieved. This progress is also being acknowledged by development process standards and evaluation standards, such as the Common Criteria for Information Technology Security Evaluation (2009), which prescribe the use of formal methods for attaining the highest assurance level, required for the most critical system components. It can be expected that in the near future the role of these more rigorous practices will further increase, as the demand for critical components increases.

The objective of this chapter is to give a circumstantial account of the state-of-the-art formal techniques that can help in improving the quality of security protocol designs and implementations in practice. The chapter aims to show what can still be done in practice, using the most promising available research results that do not require excessive expertise from users, thus being affordable. The intended focus is then on those techniques that have already been studied in depth and that can offer acceptable user-friendly automated tool support, demonstrated by research prototype tools. The newest theoretical research trends will just be mentioned, in order to show how the research in this field is moving on.

2. BACKGROUND

A security protocol can be defined as a communication protocol aimed at reaching a goal in a distributed environment even in the presence of hostile agents that have access to the network.

Examples of security goals are user authentication (that is, proving a user's identity to another remote user) and secrecy in data exchange (that is, transferring data in such a way that only the intended recipients can read transmitted data).

Like any other communication protocol, a security protocol involves a set of actors, also called principals or agents, each one playing a protocol role and exchanging protocol messages with the other protocol actors. However, differently from normal communication protocols, security protocols are designed in order to reach their goals even in the presence of hostile actors who can eavesdrop and interfere with the communication of honest agents. For example, an attacker agent is normally assumed to be able to intercept and record protocol messages (passive attacker), and even alter, delete, insert, redirect, reorder, and reuse intercepted protocol messages, as well as freshly create and inject new messages (active attacker). The goals of the protocols are normally reached by using cryptography, which is why these protocols are also named cryptographic protocols.

The logic of a cryptographic protocol is often described abstractly and informally without getting into the details of cryptography. This informal description is also called the "Alice and Bob notation", because protocol roles are usually identified by different uppercase letters and are associated to agent identities with evocative names in function of their roles, such as for example (A)lice for the first protocol participant, (B)ob for the second one, (E)ve for an eavesdropper and (M)allory for a malicious active attacker.

For example, the core part of the Needham & Schroeder (1978) public key mutual authentication protocol can be described in Alice and Bob notation as

1: A→B: {A,NA}KBpub
2: B→A: {NA,NB}KApub
3: A→B: {NB}KBpub

This protocol became notorious because of a flaw that was discovered by Lowe (1996) several years after its publication. The protocol aims at guaranteeing each participant about the identity of the other participant and at establishing a pair of shared secrets between them. A protocol description in Alice and Bob notation is a sequence of rules, each one describing a protocol message exchange in the form X→Y: M where X is the sender, Y is the intended recipient and M is the message. For example, the first rule specifies that agent Alice (A) sends to agent Bob (B) message {A,NA}KBpub. This writing stands for an encrypted pair where the components of the pair are Alice's identity A and a freshly generated nonce NA. The pair is encrypted with KBpub, that is Bob's public key. The (unreached) goal is that, after having completed the message exchange, Alice and Bob are certain about each other's identity, and the nonces NA and NB have been shared between each other, but have been kept secret to anyone else.

Another semi-formal notation often used to represent the logic of security protocols is UML sequence diagrams. For example, the Needham & Schroeder (1978) protocol described above can be equivalently represented by the UML sequence diagram of Figure 1. Each actor has a vertical lifeline that extends from top to bottom, and exchanged messages are represented by arrows, whose caption is the content of the

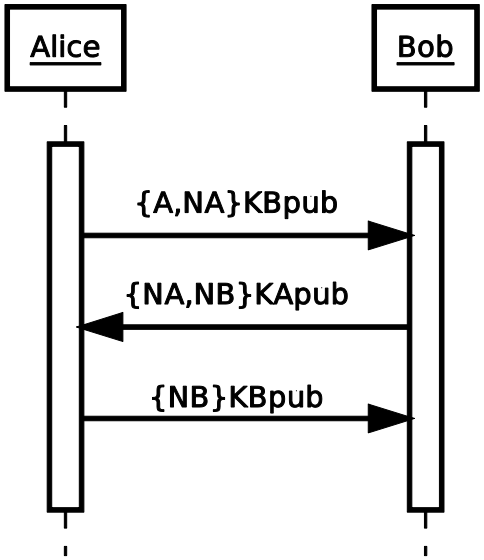

Figure 1 Needham-Schroeder public key protocol.

message. Both "Alice and Bob" notation and UML sequence diagrams are effective in representing a typical protocol scenario, while error handling or conditional execution are better represented by more formal and refined languages, such as the ones described in the rest of this chapter.

An attack on a security protocol is a protocol run scenario where the protocol does not reach the intended goal because of the hostile activity of an attacker. For example, if the protocol has been designed to ensure the secrecy of a secret datum D, an attack on the protocol is a protocol run in which an attacker gets the datum D (or partial knowledge on it). An attack is relevant only if it may occur with a non-negligible probability and using realistic resources. For example, the probability that an attacker who does not know a key guesses the value of the key is normally negligible because of the key length and because of its pseudo-random nature. The word "attack" often implicitly refers to run scenarios that may occur with non-negligible probability and using realistic resources.

Some attacks on security protocols exploit errors in protocol logic and are independent of details such as cryptographic algorithms or message encoding. These attacks can be described in terms of the abstract protocol descriptions shown above. For example, the attack discovered on the Needham-Schroeder protocol (Lowe, 1996) can be represented via the UML sequence diagram of Figure 2. The attack works out when a malicious agent Mallory (M) can convince Alice to exchange a nonce with him, so that he can use the messages from Alice to talk with Bob, while making Bob think that he is talking with Alice. Hence, Mallory is able to obtain Bob's "secret" nonce that was intended for Alice.

Other attacks cannot be described abstractly in this way, because they depend on details or interactions of particular cryptosystems.

Attacks on security protocols can be categorized according to the protocol properties they break (e.g. secrecy, authentication). Moreover, in a more technical way they can be categorized by the categorization of the kinds of weaknesses or flaws they exploit (Gritzalis, Spinellis & Sa, 1997 and Carlsen, 1994). According to the latter, the main classes can be described as follows.

Attacks based on **Cryptographic Flaws** exploit weaknesses in the cryptographic algorithms, by means of cryptanalysis, in order to break ideal cryptographic properties (e.g. determining the key used to encrypt a message from the analysis of some encrypted messages).

Attacks based on **Cryptosystem-Related Flaws** exploit protocol flaws that arise from the bad interaction between the protocol logic and specific properties of the chosen cryptographic algorithm. An example can be found in the Three-Pass protocol (Shamir, Rivest & Adleman, 1978) which requires the use of a commutative function (such as the XOR function) to perform encryption. Because of the reversible nature

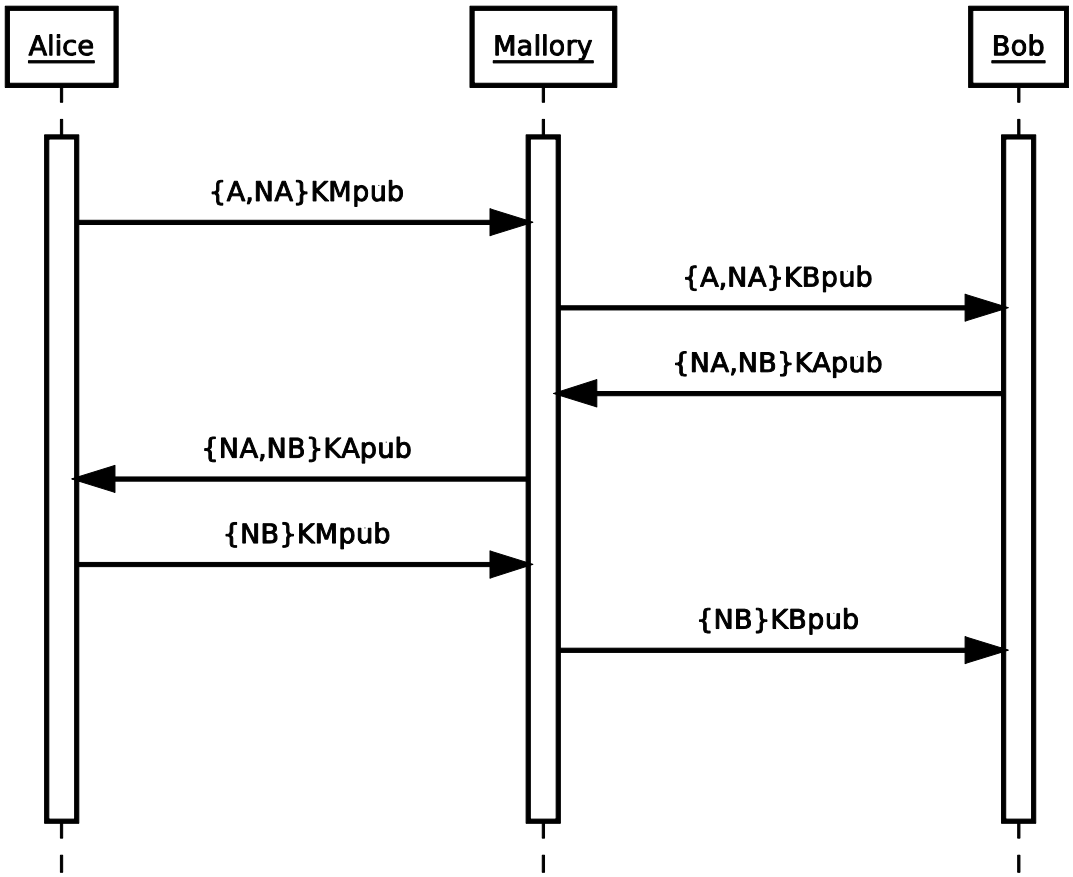

Figure 2 An attack on the Needham-Schroeder public key protocol.

of the XOR function and of the way this function is used to encrypt the messages of the protocol, it is possible for an attacker to decrypt some encrypted data by XORing intercepted encrypted messages of the protocol session. Another example can be found in some implementations of the widely deployed SSH protocol (Ylonen, 1996), where some bits of the encrypted plaintext can be recovered by an attacker, by injecting maliciously crafted packets when encryption is in CBC mode (Albrecht, Paterson & Watson, 2009).

Attacks based on **Elementary Flaws** exploit protocol weaknesses that are considered elementary in the sense that the protocol completely disregards some kind of protection. For example, the attack described in Burrows, Abadi, & Needham (1990) on the CCITT X.509 protocol (Marschke, 1988) can be classified as an attack of this kind. Here, a message is encrypted and then a signature is applied to the encrypted part. Since the signature itself is not protected in any way, the attacker can simply replace it with the attacker's signature in order to appear as the originator of the message to the sender.

Attacks based on **Freshness Flaws** (or **Replay Attacks**) arise when one or more protocol participants cannot distinguish between a freshly generated message and an old one that is being reused. Thus, the attacker can simply replay an old message making the victim believe that it is a newly generated one. An example of a protocol containing this kind of flaw is the Needham & Schroeder (1978) secret key protocol, where the third message of the protocol can be replayed by an attacker, so that an old key (possibly compromised) is reused for the communication.

Guessing Attacks concern protocols that use messages containing predictable information encrypted with weak keys, either because poorly chosen (e.g. obtained by user passwords) or because generated by weak random number generators. These flaws can be exploited by analyzing old sessions of protocol runs in

order to gain knowledge on the characteristics and on the internal status of pseudo-random generators. Another way to exploit these vulnerabilities is by performing dictionary-based attacks in order to guess keys, when the key space can be determined a priori (i.e. the most-likely combinations of bits forming the key can be determined). An example of a guessing attack is possible on the Needham & Schroeder (1978) secret key protocol, in addition to the reply attack explained above. The guessing attack can occur because keys are obtained from user-chosen passwords

Attacks based on **Internal Action Flaws** exploit protocol flaws that occur when an agent is unable to or misses to perform the appropriate actions required to securely process, receive, or send a message. Examples of such actions are: performing some required computation (such as a hash), performing checks aimed at verifying the integrity, freshness or authenticity of a received message, or providing required fields (such as a digest) inside a message. Often, these flaws are due to incomplete or wrong protocol specifications. An example of this flaw can be found in the Three-Pass protocol (Shamir, Rivest & Adleman, 1978) because, in the third step, the check that the received message must be encrypted is missing.

Oracle Attacks happen when an attacker can use protocol participants as "oracles", in order to gain knowledge of some secret data, or in order to induce them to generate some data that the attacker could not generate on his own. Note that such data could be obtained by the attacker by interacting with one or more protocol sessions, even in parallel, while impersonating several agents. It is worth noting that these attacks may be possible because of the simultaneous presence of multiple protocols with unforeseen dependencies. Even if each protocol is safe, it may be that their combination is flawed because it enables some oracle attack. This may happen, for example, when the same keys are used by different protocols. For example, the Three-Pass protocol (Shamir, Rivest & Adleman, 1978 and Carlsen, 1994) is subject to this kind of attack in addition to the attack mentioned above.

Type Flaw Attacks happen when an attacker cheats a protocol agent by sending him a message of a type different than the expected one, and the receiver agent does not detect the type mismatch and uses message data in an unexpected way. An example of this attack is possible on the Otway-Rees protocol (Otway & Rees, 1987), where an attacker can reply parts of the first protocol message so that they are interpreted as a legitimate field of the last protocol message and in this way cheat Alice by inducing her to believe to have exchanged a secret key with Bob while in fact she has not.

It is also possible to have attacks that exploit bugs in protocol implementations rather than bugs on protocols. For example, this occurs when an attacker exploits a divergence between the behavior of the implementation and the behavior prescribed by the protocol.

Attacks can furthermore be divided into two other broad categories: the attacks that can be performed by a passive attacker; and the attacks that require an active attacker. An attack performed by a passive attacker can be carried out without altering message exchange, simply by eavesdropping the exchanged messages. With this respect the Needham-Schroeder attack is an active attack, because the attacker must interact with the protocol agents by placing himself between the two. The most common forms of active attacks are the replay attack and the man in the middle (MITM) attack. In the former, the attacker replays an old message to an honest protocol actor that cannot distinguish between a freshly generated message and the old one. In the latter, the attacker intercepts all the messages exchanged by protocol actors (i.e. the attacker is in the middle of the communication), and relays intercepted messages and/or fabricates and injects new ones. In this way, actors are induced to believe that they are directly talking to each other, while, in the reality, the attacker is successfully impersonating the various protocol actors to the satisfaction of the other actors.

Security protocols normally reach their goals after a few message exchanges. However, despite this apparent simplicity, their logic flaws can be subtle and difficult to discover and to avoid during design. The difficulty comes not only from the bare use of cryptography, which is common to other cryptography applications. The extra complexity of security protocols comes from the unbounded number of different possible scenarios that arise from the uncontrolled behavior of dishonest agents, who are not constrained

to behave according to the protocol rules, combined with the possibility to have concurrent protocol sessions with interleaved messages.

Without rigorous protocol models and automated tools for analyzing them it is difficult to consider all the possible attack scenarios of a given protocol. This has been demonstrated by the case of protocols that were designed by hand and discovered to be faulty only years after their publication, as it happened with the Needham-Schroeder authentication protocol. It is particularly significant that the attack on this protocol was discovered with the help of formal modeling and automated analysis (Lowe, 1996).

A rigorous, mathematically-based approach to modeling, analyzing and developing applications is called a formal method. The mathematical basis of formal methods opens the possibility to even prove facts about models. For example, a formal model of a security protocol and its environment can be mathematically proved to fulfill a given property, such as the inability of attackers to learn the protocol secret data.

Of course, the results about protocol models are as significant as the model is. When a proof of correctness is obtained on a very abstract model, where many low-level details are abstracted away, attacks that rely on such low-level aspects cannot be excluded. On a more refined model, where such low-level details are taken into account, a proof of correctness can show those attacks are impossible. Unfortunately, as the complexity of models increases, a fully formal model and a proof of correctness become more difficult to achieve, and in any case a gap remains between formal models on one side and real protocols with real actors implemented in usual programming languages on the other side.

Formal methods can be supported by software tools. For example, tools can be available for searching possible attacks in protocol models or for building correctness proofs about models. When a formal method is supported by automatic tools, it is said to be an automated formal method.

Having automated formal methods is normally considered to be a must for formal method acceptability in production environments, because of the prohibitive level of expertise that is normally needed for using non-automated formal methods. One of the problems with automation is that as the complexity of models increases beyond a very abstract level, all the main questions about protocol correctness become undecidable, which prevents tools from always being able to come out with a proof of correctness automatically.

Traditionally, there have been two different approaches to rigorously model and analyze security protocols. On one hand are those models that use an algebraic view of cryptographic primitives, often referred to as “formal models”; on the other hand are those models that use a computational view of such primitives, generally referred to as “computational models”.

The algebraic view of cryptography is based on perfect encryption axioms: (1) The only way to decrypt encrypted data is to know the corresponding key; (2) Encrypted data do not reveal the key that was used to encrypt them; (3) There is sufficient redundancy in encrypted data, so that the decryption algorithm can detect whether a ciphertext was encrypted with the expected key; (4) There is no way to obtain original data from hashed data; (5) Different data are always hashed to different values; (6) Freshly generated data are always different from any existing data and not guessable (with non-negligible probability); (7) A private (resp. public) key does not reveal its public (resp. private) part.

Under these assumptions, cryptography can be modeled as an equational algebraic system, where terms represent data, constructors represent different encryption or data construction mechanisms, and destructors represent the corresponding decryption or data extraction mechanisms. For example, the constructor symEnc(M,k), the destructor symDec(X,k) and the equation symDec(symEnc(M,k),k) = M represent an algebraic system for symmetric encryption. As another example, the constructor H(M) represents hashing. The absence of a corresponding destructor represents non-invertibility of the cryptographic hash function.

Formal models allow simple and efficient reasoning about security protocol properties, because of their high level view on cryptography. However, there are two main drawbacks. One is that some cryptographic functions, mostly relying on bitwise operations, like for example XOR functions, are difficult to be represented in an equational algebraic model, or they could lead to undecidable models. The second one is that, because of the high abstraction level, there are several possible flaws that may be

present in the protocol implementation, but that are not caught in the formal model. For example, this happens because the algorithms implementing a particular cryptographic function do not satisfy some of the ideal assumptions made: as an instance, any function generating n bit nonces cannot satisfy assumption (6) after 2^n runs.

Computational models, in contrast, represent data as bit strings and use a probabilistic approach to allow some of the perfect encryption assumptions to be dropped. Assuming a bounded computational power (usually polynomial) for an attacker, the aim is to show that, under some constraints, the probability of an assumption being violated is negligible, that is, it is under an acceptable threshold. Technically, this is achieved by showing that an attacker cannot distinguish between encrypted data and true random data. Computational models can be used to deal with more cryptographic primitives, and to model more low level issues than formal models. However, they require more resources during protocol analysis, and usually the proofs are more difficult to be automated.

Historically, formal and computational models have evolved separately: the former focusing on the usage of basic cryptographic functions, and working at a higher level; the latter considering more implementation details and possible issues. Recent ongoing work starting from the seminal research of Abadi & Rogaway (2002), is trying to reconcile the two views and to relate them.

3. STATE OF THE ART

3.1. Security Protocol Engineering Issues

This subsection briefly accounts, categorizes and discusses the main issues that can lead to get flawed security protocols, starting from their design phase down to the implementation and application deployment phases. As shown below, flaws are constantly discovered, even in the recent past, which advocates for the use of more reliable engineering methodologies for security protocols. The next subsections present the state of the art of such methodologies based on automatic formal methods.

When a new security protocol is being developed, design is the first phase where flaws can be introduced. Regardless of the severity of weaknesses that they can cause, these flaws are in general very critical, because they will apply to all deployed implementations. The use of formal modeling and formal verification is an effective way to mitigate this issue, because it improves the understanding of the protocol and guarantees its correctness, up to the detail level that is captured by the formal model. Although there is evidence that the use of formal methods improves quality of the outcome artifact, the cost-effectiveness of their adoption, especially when used only once in the design phase, is still uncertain (Woodcock, Larsen, Bicarregui & Fitzgerald, 2009). For this reason, best practices and design guidelines that protocol experts have defined after a careful analysis of the most common mistakes made in designing security protocols (e.g. Abadi & Needham, 1996 and Yafen, Wuu, & Ching-Wei, 2004) are another important resource for protocol designers.

Implementing a security protocol is another development phase where flaws can be introduced, due to divergences between the protocol specification and its implementing code. Such divergences are often caused by programming mistakes (e.g. OpenSSL Team, 2009) or by unfortunate interpretations of an ambiguous specification (e.g. Albrecht et al., 2009). Again, formally specifying the security protocol avoids interpretation errors, and in principle enables implementations to be (semi-automatically) derived, thus reducing the probability of introducing mistakes.

Other programming mistakes introduce so-called vulnerabilities. They are security problems that may affect any software that receives data from a public channel, not just security protocols. They derive from unhandled or unforeseen input data that may cause the protocol implementation to crash or to produce nasty effects on the host where the protocol implementation is running. Notable examples are stack overflows that can be exploited in order to run arbitrary code on the host where the software runs. Vulnerabilities do not directly imply a failure of reaching the protocol goals, but they can be exploited for violating other security policies or for compromising the whole host.

3.2. Automating Formal Protocol Analysis

When done by hand, formal analysis of security protocol models can be an error prone task, as often the security proofs require a large number of steps that are difficult to track manually.
The final goal of automating such techniques is to give formal evidence (i.e. a mathematical proof) of the fact that a protocol model satisfies or does not satisfy certain security properties under certain assumptions. The mathematics and technicalities of these techniques require specific expertise which may not be part of the background knowledge of security experts. Then, automation and user-friendliness of such techniques and tools are key issues in making them acceptable and useable in practice.
Research in this field has always paid highest attention to these issues. For example, as highlighted by Woodcock et al. (2009), automation in formal methods has recently improved so much that in 2006 the effort of automating the proof of correctness for the Mondex project, a smartcard-based electronic cash system, was just 10% of the effort that would have been required in 1999. In fact, in 1999 the system was proven correct by hand; in 2006 eight independent research groups were able to get automatic proofs for the same system.
Several different automation techniques have been explored, depending on the form of the input model. In the next subsections, the most common forms used to express security protocols models and properties are introduced, and their related verification techniques are explained.

3.2.1. Logics of Beliefs (BAN Logic)

Logics of beliefs (e.g. Burrows, Abadi, & Needham, 1990) are very high-level models for reasoning on security protocols where perfect cryptography is assumed, and only some logical properties are tracked. Technically, they are modal logic systems designed to represent the beliefs of protocol actors during protocol executions. For example, the formula "P believes X" means that actor P believes that predicate X is true, in the sense that P has enough elements to rightly conclude that X is true. Thus, P may behave like X was true. Another formula is "fresh(X)", which means X is fresh data, that is it has not been previously used in the protocol (not even in previous sessions). If X is a newly created nonce, fresh(X) is assumed to hold just after its creation and the actor who just created the nonce believes it fresh (P believes fresh(X)). When a protocol session executes and messages are exchanged, the set of beliefs and other predicates for each actor updates accordingly. This is described in belief logics by a set of inference rules that can be used to deduce legitimate beliefs. Security properties can then be expressed by some predicates on actors' beliefs that must hold at the end of a protocol session.
More specifically, a protocol is described by using the Alice and Bob notation, and assumptions about initial beliefs of each actor are stated. For example, before protocol execution, Alice may already trust a particular server, or be known to be the legitimate owner of a particular key pair. Then, protocol execution is analyzed, so that beliefs of actors are updated after every message is exchanged. For example, after A→B: M has been executed, "A believes A said M" is true, where "A said M" means that A has sent a message including M. The BAN logic inference rules also allow new facts to be inferred from those that are already known. At the end of the protocol, actors' beliefs are compared with the expected ones: if they do not match, then the desired property is not satisfied.
Note that BAN logic can be very useful in protocol comparisons. Indeed, it is possible to compare the initial assumptions required by different protocols that achieve the same security goals. Moreover, it may be possible to find redundancies in the protocol, for example when an actor is induced by a received message to believe a fact that the actor already knows.
A BAN logic model can be verified by means of theorem proving. The theorem proving approach consists of building a mathematical proof in a formal deduction system in order to show that a desired security property holds in the given model. Since manually building formal proofs is a difficult and error-prone task, automation can be particularly effective. Some proof assistants, called theorem provers, can automatically solve simple lemmas, and they also interactively help the user in searching for the complete proof. Some of them, called automatic theorem provers, can even find complete proofs automatically.

Using theorem proving with belief logics is straightforward, because these logics are natively defined as propositional formal systems. However, using the bare logic is not so user-friendly and easy for a non-expert: the protocol, the assumptions made and the desired properties must all be expressed in the belief logic language; then, a proof of the desired properties must be found.
In order to simplify the use of belief logics, researchers have developed tools that automatically translate simple user-friendly descriptions of protocols (in Alice and Bob notation) and their initial assumptions and desired properties into expressions in the belief logic language (Brackin, 1998). Moreover, as it has been shown by Monniaux (1999), belief logics, such as BAN and GNY, are decidable, i.e. it is possible to build an algorithm that automatically decides, in finite time and space, if a proof for a given proposition exists and, in positive case, finds it. In conclusion, the whole process of translating user-friendly descriptions into a belief logic such as BAN, finding out if the desired properties can be proved and, in positive case, delivering a proof can be automated. In addition, this task is computationally feasible and takes seconds for typical protocols (Monniaux, 1999).
Of course, belief logics such as BAN and GNY have several limitations, because they only apply to authentication properties and they do not catch some kinds of flaws, such as exposure to attacks based on oracles. Nevertheless, experimental studies conducted by Brackin (1998) on the Clark & Jacob (1997) collection of authentication protocols have shown that only a small fraction of known flaws on the high-level descriptions of those protocols are not caught by this approach. Then, this is a fast mechanized analysis that can detect several logical bugs related to authentication in high-level protocol descriptions. It can be used as a preliminary low-cost analysis, before conducting other more expensive and more accurate analyses.
Belief logics have been extensively developed in the past and are now considered mature, so recent works mainly focus on their application (e.g. Qingling, Yiju & Yonghua, 2008) rather than on their development.

3.2.2. Dolev-Yao Models

The Dolev & Yao (1983) model of security protocols is currently widely used in many research works (e.g. Hui & Lowe, 2001, Abadi & Gordon, 1998, Bengtson, Bhargavan, Fournet, Gordon, & Maffeis, 2008, Fábrega, Herzog, & Guttman, 1999, Durgin, Lincoln, & Mitchell, 2004, Chen, Su, Liu, Xiao, 2010) and implemented in many protocol verification tools (e.g. Durante, Sisto, & Valenzano, 2003, Viganò, 2006, Mödersheim & Viganò, 2009, Blanchet, 2001). Since its introduction, it has gained popularity because it is a simple, high level modeling framework, yet effective in modeling common protocol features and reasoning about many common security properties.
Like logics of beliefs this way of modeling protocols stems from an abstract algebraic view of cryptography, but it is more accurate because it tracks exchanged messages rather than beliefs. In the Dolev-Yao model, a protocol is described by a discrete state-transition system made of communicating parallel sub-systems representing protocol actors. The transition system of each actor describes the actor behavior in terms of sent and received messages. For honest agents this is the behavior prescribed by the protocol, while for attacker agents it can be any behavior.
Many network models with apparently different features can be considered, but in general they can be shown to be equivalent, because one system can be encoded, or simulated, by the others. For example, one possible network model proposed by Schneider (1996) considers the network and the attacker as separate sub-systems, where the attacker is a protocol participant that has special actions, such as forging and eavesdropping messages. Other models (Ryan & Schneider, 2000) instead consider the intruder as the medium itself. Finally, association models (Voydock & Kent, 1983) group all securely connected actors into one “super-actor”, leading back to the basic idea of just having “super-actors” communicating only through an untrusted network. Usually, one chooses the network model that will make proving a particular security property as easy as possible.
While Dolev-Yao models can be formalized in different ways, from process calculi, to graphs and automata, to refinement types, the ideas and assumptions of the underlying model are always the same. Normally, protocol actors do not communicate directly over private channels, as this would make security

protocols useless; instead, they communicate over the shared untrusted network. Exceptions to this can be modeled, and in this case it is said that two or more actors share a private secure channel, to which the attacker has no access. As an algebraic model of cryptography is considered, many models have a set of data types and a set of constructors, destructors and equations that represent cryptographic functions and their properties. Some models come with a fixed set of cryptographic primitives and associated semantics, while others are extensible, thus enabling a wide range of protocols to be modeled. Being algebraic models, all of them can only represent the ideal properties of cryptographic primitives, so that, for example, cryptographic flaw attacks cannot be represented in these models. Moreover, being an active attacker always able to drop any message, it turns out that liveness properties, such as "an acknowledgment is eventually received", cannot be checked within Dolev-Yao models. This is not a big issue after all, since many common security properties, such as secrecy and authentication, can be defined as safety properties, and thus fully handled in the Dolev-Yao models.

Summing up, the general features of Dolev-Yao models make them amenable to represent protocol flaws based on wrong usage of cryptographic primitives that are independent of the particular cryptosystems used, while such models are not tailored to spotting other protocol flaws or flaws in the cryptosystems themselves. The power of the intruder is believed to be enough to express its worst behavior, under the assumptions of perfect cryptography (for example, it can never guess nonces or keys, nor even partially recover a key from a given ciphertext). Moreover, it is worth pointing out that many other implementation details are usually not captured by Dolev-Yao models, so protocol flaws that may arise because of these details cannot be caught too. For instance, side channels are usually not modeled. For example, one attacker could infer some information by looking at power consumption of certain devices during computation, or data transmitted over a secured channel could be inferred by looking at the data rate at which they are sent.

3.2.3. Representing Dolev-Yao Models as Process Calculi

Among the different formalisms that have been developed to specify security protocols at the Dolev-Yao level, process calculi are among the most used ones, hence they are taken as an example here. Specifically, a variant of the applied pi calculus, which is one of the most user-friendly process calculi, is shown as an example.

Process calculi are algebraic systems designed to represent communicating processes and their exchanged data, with a well-defined formal semantics. Because of their abstraction level, they are a good tool for formally describing communication protocols in the Dolev-Yao style.

For example, Spi calculus is a process calculus for security protocols (Abadi & Gordon, 1998) that has been designed as a security-aware extension of the pi-calculus (Milner, 1999). It is close to a programming language, in that it enables explicit representation of input/output operations as well as checks on received messages. This is amenable to the derivation of implementations, since operations in the specification can be mapped into corresponding operations in the implementation.

The applied pi calculus (Abadi & Fournet, 2001) is similar to the Spi calculus but it is based on an equational system where constructors, destructors and equations can themselves be specified, making the language easily extensible. In this chapter, the slightly extended and machine readable version of the applied pi calculus as it is accepted by a popular verification tool (Blanchet, 2009) is presented as an example. From now on, in this chapter the expression "applied pi calculus" will refer to this extended and machine readable version.

The applied pi calculus syntax is composed of terms, formally defined in Table 1, and processes, formally defined in Table 2. A term can be an atomic name (e.g. 'a'), a variable (e.g. 'x') that can be bound to any other term once, or a (constructor) function application $f(M_1, M_2, \ldots, M_n)$. For each constructor, corresponding destructor functions and equations can be defined separately, so as to specify an equational system that models the properties of cryptographic operations. For convenience, a syntactic sugar is introduced for tuples of terms, denoted by $(M_1, M_2, \ldots, M_n)$, which can be encoded within the standard syntax by corresponding constructor and destructor functions.

M, N ::=	**Terms**
a, b, c	name
x, y, z	variable
$f(M_1, M_2, \ldots, M_n)$	function application

Table 1 Applied pi calculus terms.

P, Q ::=	**Processes**
0	null process
P | Q	parallel composition
!P	replication
new a; P	restriction
if M = N then P else Q	conditional
let $x = g(M_1, M_2, \ldots, M_n)$ in P else Q	destructor application
in(c,x); P	message input
out(c,M); P	message output
event $f(M_1, M_2, \ldots, M_n)$; P	auxiliary authenticity event

Table 2 Applied pi calculus processes.

The semantics of processes can be described as follows. The null process '0' does nothing. The parallel process P | Q executes P and Q in parallel, while the replication process !P behaves as an unbounded number of instances of P running in parallel. The restriction process 'new a; P' creates a fresh name 'a', that is a name not known to anyone else, including the attacker. The conditional process 'if M = N then P else Q' behaves like P if M and N are the same term (modulo the equational theory), otherwise it behaves like Q; if Q is '0' the else branch can be omitted. The destructor application process 'let $x = g(M_1, M_2, \ldots, M_n)$ in P else Q' tries to compute the result of the application of the destructor function. If this result can be computed (i.e. an equation can be applied), the result is stored into variable 'x' and the process behaves like P, else it behaves like Q; again the else branch can be omitted if Q is '0'. With the tuples syntactic sugar, the destructor application process can also be used in a compact form of tuple-splitting process 'let $(x_1,x_2,\ldots,x_n) = M$ in P else Q'. The input process in(c,x); P receives a message from channel 'c' and stores it into variable 'x'. The output process out(c,M); P outputs the message M on channel 'c', then behaves like P. Finally, the auxiliary process 'event $f(M_1, M_2, \ldots, M_n)$; P' emits special events needed to establish correspondence properties such as authentication (more on this later).

A typical applied pi calculus protocol specification defines one process for each agent, and a "protocol instantiation" process, describing how protocol agents interact with each other in a protocol run. In such models, security properties can be specified as predicates that must hold for all the runs of a particular protocol. For example, secrecy can be defined as a reachability property, asking whether there exists one run where the attacker can get to know some data D that should be kept secret. The special correspondence events mentioned above enable reasoning about authentication properties. Suppose Alice wants to authenticate to Bob, meaning that at the end of a protocol run Bob can be sure he has been talking with Alice. This can be modeled by the following property: "Each time Bob ends a protocol session apparently with Alice, agreeing upon some session data D, Alice had previously started a protocol session with Bob, agreeing upon the same session data D." This is achieved by defining two events,

namely *running* and *finished*. Bob emits a finished(D) event after the protocol run and Alice emits a running(D) event as soon as the session is started and the relevant session data D are available. The authentication property is then expressed requiring that in each trace of the protocol each time a finished(D) event occurs, a corresponding running(D) event (referencing the same data D) has taken place in the past.

Communicating Sequential Processes (CSP) is another process calculus that has been used to model security protocols (Ryan & Schneider, 2000). Each process models a protocol actor and is described by the events it will emit on a particular communication channel. Essentially, message exchange is performed by honest agents by emitting send.A.B.M events, meaning that actor A sends message M apparently to B, and receive.B.A.M events, meaning that actor B receives message M apparently from A.

Differently from the applied pi calculus, in CSP the checks performed on the received data are always implicitly represented in the input operation. When emitting the receive.B.A.M event, one agent is willing to receive a message only if the received data pattern-match against M. This means that, during message reception, it is implicitly assumed by the CSP model that "all possible checks" will be performed on the received data (Ryan & Schneider, 2000). While this behavior is acceptable at an algebraic level, because pattern matching can be implemented by syntactic matching, it becomes non trivial to preserve the same semantics down to executable implementations of CSP processes, which makes this modeling framework not so amenable to deal with implementations of security protocols.

3.2.4. Automated Verification of Dolev-Yao Models by Theorem Proving

Security protocols expressed as Dolev-Yao models can be verified by theorem proving. Essentially, the protocol specification is translated into a logic system where facts such as "message M has been transmitted" or "the attacker may know message M" are represented. Therefore, proving that a security property holds amounts to proving that a particular assertion (for example, "the attacker may know secret S") cannot be derived in the formal system. However, due to the undecidability of security properties such as secrecy and authentication in Dolev-Yao models (Comon & Shmatikov, 2002), it is not possible to develop an automatic procedure that in finite time and space always decides correctly whether or not a security property holds for any given protocol model.

Three main possible ways to cope with this undecidability problem have been explored. One is restricting the model (e.g. by bounding the number of parallel sessions), so as to make it decidable. This approach, which has been extensively used for state exploration techniques, has the drawback of reducing the generality of the results. A second possibility is to make the verification process interactive, i.e. not fully automatic. This is the approach taken for example by Paulson (1998). Here the obvious drawback is that, although libraries of reusable theorems can reduce the degree of interactivity, user expertise in using a theorem prover is needed. Moreover, there is no guarantee of eventually getting a result. A third possibility, that suffers from the same latter limitation, is using semi-decision procedures, which are automatic but may not terminate, or may terminate without giving a result, or may give an uncertain result. This approach has been adopted, for example, by Song, Berezin, & Perrig (2001) and Blanchet (2001). In particular, the ProVerif tool (Blanchet, 2001, Blanchet, 2009) is also publicly available and is one of the most used tools in the class of automatic theorem provers for Dolev-Yao models. It is based on a Prolog engine and accepts protocol descriptions expressed either directly by means of Prolog rules that are added to the ProVerif formal system, or in the more user-friendly applied pi calculus described in Table 1 and Table 2, which is automatically translated into Prolog rules. ProVerif does not terminate in some cases. When it terminates, different outcomes are possible. ProVerif may come up with a proof of correctness, in which case the result is not affected by uncertainty, i.e. the protocol model has been proved correct under the assumptions made. ProVerif may sometimes terminate without being able to prove anything. In this case, however, it is possible that ProVerif indicates potential attacks on the protocol. This ability is particularly useful to understand why the protocol is (possibly) flawed and is generally not provided by other theorem provers. Note however that the attacks produced by ProVerif may be false positives, because of some approximations it uses. That said, on most protocols the tool

terminates giving useful results, which makes it one of the most used automated tools now available for verifying security protocols.
As an example, the Needham-Schroeder protocol specified in the applied pi calculus, as accepted by ProVerif, follows

```
1a: A(KA,KBPub,AID) :=
2a:       new NA;
3a:       out(cAB, pubenc((AID,NA), KBPub);
4a:       in(cAB, resp);
5a:       let resp_decr = pridec(resp, Pri(KA)) in
6a:       let (xNA,xNB) = resp_decr in
7a:       if xNA = NA then
8a:       out(cAB, pubenc(xNB, KBPub));
9a:       0

1b: B(KB,KAPub,AID) :=
2b:       in(cAB, init);
3b:       let init_decr = pridec(init, Pri(KB)) in
4b:       let (xAID, xNA) = init_decr in
5b:       if xAID = AID then
6b:       new NB;
7b:       out(cAB, pubenc((xNA,NB), KAPub);
8b:       in(cAB, resp);
9b:       let xNB = pridec(resp, Pri(KB)) in
10b:      if xNB = NB then
11b:      0

1i: Inst() :=  new KA; new KB; out(cAB, (Pub(KA), Pub(KB))); (
2i:                !A(KA,Pub(KB),AID) | !B(KB,Pub(KA),AID)
3i:       )
```

As usual, the two protocol actors are described by two separate processes, A and B, while a third process called Inst models the protocol sessions, by specifying concurrent execution of an unbounded number of instances of A and B processes.
In the example, role A is parameterized by a key pair KA, containing its public and private keys (denoted Pub(KA) and Pri(KA) respectively), by KBPub, i.e. B's public key, and by AID, i.e. A's identifier. At line 2a, A creates the nonce NA and, at line 3a, AID and NA are sent to B, encrypted under B's public key, over channel cAB. At line 4a A receives on channel cAB the response from B and stores it in the *resp* variable, which is then decrypted at line 5a. Since *resp* is supposed to be an encryption made with A's public key, it is decrypted with A's private key, and the result of decryption is stored in the *resp_decr* variable. At line 6a, *resp_decr* is split into two parts that are stored in variables xNA and xNB: the former should store A's nonce, while the latter should store B's nonce. Since A's nonce is known, at line 7a it is checked that the received value xNA matches with the original NA value. If this is the case, at line 8a A sends the received B's nonce encrypted with B's public key over channel cAB.
In applied pi calculus, fresh data (that is data created with the "new" operator) are neither known nor guessable by the attacker, while global data (such as AID or the communication channel cAB) are assumed to be known by the attacker. In particular this means that the attacker can actively or passively control the communication channel cAB, or try to cheat B sending him A's identity; however, the attacker does not know agents' private keys (while the public ones are known because they are sent in clear over the public channel before starting the protocol, at line 1i).

In applied pi calculus, the attacker is the environment, i.e. it is not modeled explicitly, but it is assumed that the attacker can be any applied pi calculus process running in parallel with Inst().
As explained above, the Needham-Schroeder protocol should ensure that at the end of a session A and B are mutually authenticated. In particular, authentication of A to B is the property that does not hold in the protocol. In order to express this property, so that it can be verified by the ProVerif tool, the specification must be enriched with the special *running* and *finished* events. Authentication of A to B can be expressed by adding an *event running(AID, NA, XNB, Pub(KA), KBPub)* statement in the A process, just before the third message is sent, that is between lines 7a and 8a. The corresponding *event finished(AID, xNA, NB, KAPub,Pub(KB))* statement is added to the B process at the end of the session, that is between lines 10b and 11b.
When ProVerif analyzes the enriched specification, it can automatically state that the authentication property is false. Moreover, an attack trace is returned, showing how the attacker can in fact break the property. The returned attack trace is essentially the same as the one shown in Figure 2.
Conversely, the authentication of B to A holds. When this property is expressed by properly enriching the specification with the *running* and *finished* events, ProVerif can prove that such property is true.

3.2.5. Automated Verification of Dolev-Yao Models by State Exploration

Another available technique is state exploration, which works by extensively analyzing all possible protocol runs. Then, rather than looking for a correctness proof, the analysis looks for violations of the desired security properties in the model runs. If a run of the model is found in which one of the desired security properties is violated, it can be concluded that the protocol does not satisfy that property; furthermore, the run that leads to the violation constitutes an attack on the protocol. For instance, let us assume a desired property of a protocol is the inability of the attacker to learn a certain secret. If a run of the model is found leading to a state in which the attacker knows the secret, then an attack has been found, which also shows by a counterexample that the desired property does not hold. If instead the model runs are exhaustively searched without finding any violation of the desired properties, it can be concluded that the model satisfies these properties. This conclusion is in principle equivalent to a proof of correctness like the one that can be obtained by a theorem prover. In practice, however, the aforementioned undecidability issue prevents the analysis from always getting to a proof of correctness. Indeed, Dolev-Yao models cannot be searched exhaustively by naïve state exploration because they are infinite, which is their main source of undecidability. The usual approach followed by state exploration tools is to bound the model so as to turn it into a finite one, which can be searched exhaustively. In this way, the analysis always terminates, either finding out one or more counterexamples, or without finding any flaw. In the second case, no full correctness proof has been reached, because the model had been simplified, but nonetheless something new is known: no flaws are present in the reduced model. This increases the confidence in the protocol correctness with respect to what was known before running the analysis. Some researchers have found that if a protocol satisfies some particular conditions, even a non-exhaustive finite search can be enough to give a correctness proof (e.g. Lowe, 1998, Arapinis, Delaune & Kremer, 2008). Unfortunately, real protocols not always meet such conditions. Another approach to get to a full security proof, used for example in the Maude-NPA by Escobar, Meadows, & Meseguer (2009), is to combine state exploration with inductive theorem proving techniques.
Although state exploration cannot always get to a proof of correctness, its main values are that it can be fully automated and that, on finite models, it always terminates with some useful information, either of correctness or with a counterexample. Moreover, its ability to find attacks is very important because it lets the user diagnose why a protocol is faulty. In contrast, with theorem proving tools, it may happen that a proof is not found but at the same time no hint about possible flaws in the protocol is given.
State exploration analysis for security protocols can be implemented in several different ways. Some researchers have built prototype tools specifically tailored for the analysis of security protocols, such as the NRL Protocol Analyzer (Meadows, 1996) and its next-generation Maude-NPA (Escobar et al., 2009), OFMC (Basin, Mödersheim, & Viganò, 2005) and its homonymous successor (Mödersheim, & Viganò, 2009), S3A (Durante et al., 2003) and many others. Other researchers instead have shown how general-

purpose state exploration tools such as model checkers can be used for the same purpose. Among the model checkers that have been experimented for analyzing security protocols, we can mention FDR (Lowe, 1996), and Spin (Maggi & Sisto, 2002).

The first attempts at state exploration worked with strictly finite models, obtained by imposing a double restriction: on one hand a bounded number of protocol sessions, on the other hand a bounded message complexity (obtained, for example, by assuming the maximum number of operations an attacker applies to build messages is bounded). Later on, it was discovered that it is possible to relax the second restriction, i.e. the one on message complexity, without losing decidability. Of course, relaxing this restriction leads to infinite-state models, because each time one of the protocol agents inputs some data, the attacker can send to that agent one of infinitely many different messages (all the ones the attacker can build using its current knowledge, which are infinite if no bound is put on message complexity). Nevertheless, the infinite number of states and transitions can be partitioned into a finite number of equivalence classes. Each class can be described using free variables, each of which can be instantiated into infinitely many different terms. The key point is that the analysis can be conducted without instantiating variables immediately, so that classes of states and of transitions are represented symbolically in the analysis by means of symbolic entities that include uninstantiated variables. This technique, initially introduced by Huima (1999), has led to the construction of state exploration tools that can successfully analyze protocol models with the only restriction of having a bounded number of sessions. Some tools, such as OFMC (Mödersheim, & Viganò, 2009), offer the user the possibility of avoiding to introduce an a-priori bound on the number of sessions and, using a lazy evaluation approach, they can effectively search the state space even in this case. Of course, because of undecidability, the search may not terminate if the number of sessions is left unbounded.

One problem that still remains with all state exploration tools is state explosion: the number of states and state paths to be explored increases exponentially with the number of protocol sessions. Therefore, it is typically practical to analyze models with only few sessions.

The different features of state exploration and theorem proving tools, which are somewhat complementary, suggest that good results can be obtained by a combined use of the two. For instance, a state exploration tool can be used initially, because of its ability to report attacks. When some confidence in the protocol correctness has been achieved, it is then possible to switch to a theorem prover, which can provide the final correctness proof without any limitation on the number of sessions.

Among the efforts towards the possibility of integrated use of different analysis tools, the most relevant one started with the AVISPA project (Viganò, 2006), followed by the AVANTSSAR project, aiming at the specification and verification of security-aware systems by means of state-of-the-art state exploration tools.

3.2.6. Automated Verification of Dolev-Yao Models by Type Checking

A novel technique, that complements security protocol specifications described by means of process calculi with refinement type systems, is implemented in the F7 framework (Bengtson et al., 2008, Bhargavan, Fournet, & Gordon, 2010). A refinement type is denoted by {x:T|C(x)}, where x is a variable, T is a type name, and C(x) is a logic formula that can depend on x itself. The refinement type {x:T|C(x)} is a subtype of the classical type {x:T}; a value M of {x:T|C(x)} is such that M is a value for {x:T}, and C(M) is true.

For example, the type {k:Key | MayMAC(k)} is the type of cryptographic keys that can be used to perform MAC operations over protocol data.

In F7, a protocol is specified by using a concurrent lambda-calculus, which is then annotated with refinement types. The used lambda-calculus has no embedded primitives for cryptography; instead they are provided as a library that implements a symbolic, algebraic model of such primitives, thus behaving like a Dolev-Yao model. Moreover, the lambda-calculus has special “assume C(x)” and “assert C(x)” expressions, that can be used by honest agents to specify authentication properties. Given a value M, the “assume C(M)” expression means that the proposition C(M) is true, while the “assert C(M)” can be used to test whether the fact C(M) can be derived during a protocol run.

In order to describe the attacker, a universal type Un is defined. This type can be subtype or supertype of any other type, and represents data known by the attacker. Then, the specification of an attacker is any process A typed as Un. When Un is used as a subtype of another type T, then T is said to be of tainted kind, meaning that values of T may come from the attacker; when Un is used as a supertype of T, then T is said to be of public kind, meaning that values of T may be sent to the attacker.
A specification is safe if no assertion can ever fail at run-time, despite the best effort of the attacker to let an assertion fail. In order to check whether a specification is safe, standard type checking and partial type inference algorithms can be used. Because of undecidability issues, it may be impossible to find a typing even for a correct protocol, and hence to find a proof of correctness.

3.2.7. Automatic Verification of Computational Models

As explained in section 2, computational models are more refined than the abstract formal models described above, because they take cryptanalysis issues into account. As a consequence, formal automatic verification of such models is more difficult. Two strategies have been essentially investigated in order to prove security properties on computational models. On one hand, the "indirect" strategy consists of proving the security properties on a more abstract formal model, and then to show that a corresponding computationally sound model is implied to be secure too. On the other hand, the "direct" strategy consists of proving the security properties directly in the computationally sound environment.
For the indirect approach, existing tools in the formal model domain can be re-used, although, in order to be able to prove the computational soundness of a formal model, the latter often needs to be enriched by many details that usually harden verification. Furthermore, once the formal model has been proven secure, it is still needed to show that it is computationally sound. Such computational soundness proof has usually to be developed manually for each protocol model and each security property, although recent research by Comon-Lundh & Cortier (2008) aims at providing some general results that can be applied on certain classes of security properties.
For the direct approach, some first attempts at applying automated theorem proving to computational models have recently been documented. The most relevant work is the one by Blanchet (2008), who has built and made available a prototype prover called CryptoVerif that is sound in the computational model. The tool accepts a process calculus inspired by the applied pi calculus. However, some significant semantic differences exist between the CryptoVerif input language and the applied pi calculus, so that it is currently not possible to feed both ProVerif and CryptoVerif with the same models. The CryptoVerif tool operates by interpreting the model as a "game", where the protocol is secure when the attacker has negligible probability of winning the game (that is, of breaking the security property). This tool can find proofs automatically in some cases, but undecidability prevents it from always deciding in finite time if a proof exists. With respect to ProVerif and other tools that operate on Dolev-Yao models and that have been proved useful in practice, this one is newer, so that no enough experience reports are currently available to shed light on its practical applicability.

3.3. Formally Linking Protocol Specification and Implementation

Up until now, the chapter mainly focused on formal, abstract specifications of protocols and their verification. However, real implementations of security protocols, implemented in a programming language, may significantly differ from the verified formal specification, so that the real behavior of the protocol differs from the abstractly modeled and verified one, possibly enabling attacks that are not possible according to the formal specification.
In order to ensure that the formal model is correctly refined by the implementation, two development methodologies can be used, namely model extraction (Bhargavan, Fournet, Gordon, & Tse, 2006, Jürjens, 2005, Goubault-Larrecq & Parrennes, 2005) and code generation (Pozza, Sisto & Durante, 2004, Pironti & Sisto, 2007, Tobler & Hutchison, 2004, Jeon, Kim, & Choi, 2005). These approaches are depicted in Figure 3 and detailed in the next subsections.

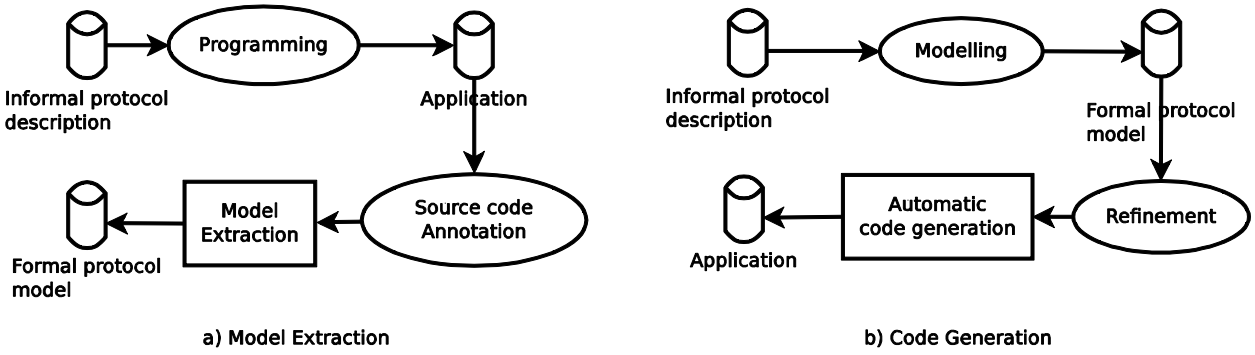

Figure 3 a) Model extraction and b) code generation approaches.

3.3.1. The Model Extraction Approach

In the model extraction approach depicted in Figure 3a, one starts by manually developing a full blown implementation of a security protocol from its specification, and enriches the source code with annotations about its intended semantics. This enriched code is then used to automatically extract an abstract formal model. The extracted formal model can then be verified (not depicted), in order to check the desired security properties, by one of the techniques presented in the previous sections. This approach has the advantage of allowing existing implementations to be verified without changing the way applications are currently written, except when annotations in the source code are required, in order to specify the link between implementation data and abstract terms. Usually, a full model of the protocol implementation is too complex to be handled. Therefore, to make verification feasible, many approaches extract an over-approximated model, where some details are soundly abstracted away, and only the relevant protocol logic parts are represented. These over-approximations could lead the analysis to report false positives, when the abstract model is flawed, but that flaw is ruled out in the code, by some implementation details that are abstracted in the formal model. However, when the over-approximations are sound, it is ensured that any flaw in the code is still present in the abstract model, and can thus be found (in other words, no flaw is missed when extracting the model).

In the work by Bhargavan, Fournet, Gordon, & Tse (2006), applications are written in F#, a dialect of the ML functional language. Then the fs2pv tool translates the F# program into a ProVerif model, that can be checked for security properties. The translation function implemented by fs2pv is proven to be sound under a Dolev-Yao attacker, meaning that the extracted ProVerif model preserves all the secrecy and authentication faults of the F# code, when a Dolev-Yao attacker is assumed. Actually, not all the F# code that constitutes the implementation of the security protocol is translated into the ProVerif model. Indeed, the F# cryptographic libraries are assumed to be correct, and are symbolically represented as Dolev-Yao constructors and destructors in the abstract model.

By using the fs2pv model extraction tool, Bhargavan, Fournet, & Gordon (2006) developed a provably correct reference implementation for WS-Security. Although this case study showed how the tool could be used in practice, it also showed some major drawbacks of this methodology. Functional languages such as ML (or its dialect F#) are not very common in the programming practice; furthermore, some constraints on the input F# code actually only allowed newly written code to be verified, instead of existing applications. Moreover, small changes in the F# code lead to different ProVerif models, some of which can easily be verified, while for others the verification process may diverge, requiring fine tuning of the original F# code, in order to enable verification. Nevertheless, the outcome of this case study is a fully functional provably correct implementation of the WS-Security standard, that can be used as a reference implementation, or even directly reused by other applications.

In the work by Goubault-Larrecq & Parrennes (2005) a similar approach is developed, but the source code of the implementation is the popular C language. In order to link the C data structures to symbolic Dolev-Yao terms, the user is required to annotate the source code with trust assertions. Then, a simplified

control flow graph is extracted from the annotated C source code, and translated into a set of first-order logic axioms. The obtained axioms, together with a logical formalization of the requested security properties, are finally checked with an automated theorem prover. Since a full operational model of a C program would be too complex to be handled, some sound over approximations are made by the tool. This means that security faults will be caught, but some false positives can arise.

3.3.2. The Code Generation Approach

In the code generation approach depicted in Figure 3b, one starts by deriving an abstract formal model from the informal protocol description, and refines such high-level formal model with low-level implementation details that would be not captured otherwise. The abstract model and its refinement information are then used to automatically generate the final implementation code. Also (not depicted), the abstract formal model can be used to check the desired security properties on the protocol.

In general, abstract models dealing with Dolev-Yao attackers use symbolic representations of terms, meaning that data encoding as bit strings is usually abstracted away. Moreover, since perfect encryption is assumed, the cryptographic algorithms are usually not represented. However, the code generation approach should allow the developer to specify these details, in order to generate interoperable applications that can conform to existing standards or specifications.

In this approach, once the formal model has been verified, tool automation is rather important in order to avoid manually introduced errors during the refinement phase form the formal model to the derived implementation. Like with the model extraction approach, tools should be trusted or verifiable too. Even when some implementation code must be manually written, it is important to ensure that this manually written code cannot introduce security faults. It must be pointed out that the code generation approach only allows new applications to be generated, and cannot deal with existing ones, thus not stimulating software reuse (only the general purpose libraries and the models can be reused).

Several attempts have been made in this field. For example, two independent works both named spi2java (Pozza et al. 2004, Tobler & Hutchison, 2004) allow the programmer to start from a verified spi calculus specification, and to get the Java code that implements it. On one hand, the framework by Tobler & Hutchison (2004) uses Prolog to implement the Spi to Java translator, thus facilitating the verification of the translator correctness. On the other hand, the framework by Pozza et al. (2004) comes with a formal definition of the Spi to Java translation (Pironti & Sisto, 2010), and enables interoperability of the generated applications (Pironti & Sisto, 2007), also giving sufficient conditions under which some manually written parts of the implementation are safe.

The spi2java tool by Pozza et al. (2004) has been used to develop an SSH Transport Layer Protocol (TLP) client (Pironti & Sisto, 2007). The client has been tested against third party servers, thus showing that an implementation adhering to the standard can be obtained. Although showing that the approach is practically useful, the case study also stressed some of its shortcomings. For example, half of the code composing the SSH TLP client was manually written. Although there exist sufficient conditions stating that no flaws can arise from that code, the manual effort to derive the application was still considerable. Moreover, the code is not as efficient as some other popular SSH TLP implementations, and the developer has no way to modify the generated code to improve code efficiency, without losing the guarantees about its correctness.

Another tool for security protocol code generation is AGC-C# (Jeon et al., 2005), which automatically generates C# code from a verified CASPER script. Unlike other works, this tool does not support interoperability of the generated code. Moreover, it accepts scripts that are slightly different from the verified Casper scripts. Manual modifications of the formal model are error prone, and the generated code starts from a model that is not the verified one.

3.3.3. Discussion

Both model extraction and code generation approaches present a trade-off between the ability of proving security properties, and the possibility of writing applications in the way they are commonly written. In

the model extraction approach, a functional language with formal semantics (like for example ML) is much simpler to reason about than an imperative language without formal semantics (like for example C); in the code generation approach, code optimizations are not allowed, in favor of achieving higher confidence about the correctness of the generated code. Note that, in principle, the code generation approach can ensure by construction that the automatically generated applications are also free from low-level errors like buffer overflows or integer overflows while the model extraction approach cannot.
It is finally worth pointing out that the tools described in these sections are research prototype tools, which aim is to show the feasibility of an approach. As also stated by Woodcock et al. (2009), industrial adoption of these techniques could make them better engineered, i.e. more usable and efficient.

4. FUTURE RESEARCH DIRECTIONS

The pervasiveness of networks made security protocols so widespread and tailored to the different applications that some of them cannot be adequately modeled and checked within the Dolev-Yao and BAN frameworks. For example, popular applications, like e-commerce, may require security goals that are not safety properties and thus cannot be checked in the presence of an active Dolev-Yao attacker. For example, in an e-commerce protocol, the attacker may cheat in completing a purchase, thus gaining by paying less than agreed with the merchant. It seems that these scenarios are better modeled with a game-theoretical approach (Gu, Shen, & Xue, 2009): each protocol actor is a potential attacker, that tries to cheat to maximize its profit (or, to "win" the game). Then, a security protocol is fair, if there exists no winning strategy for any player, meaning that at the end of the game every actor ended with a fair, agreed profit.
As hinted above, basic Dolev-Yao models have no concept of timing, nor of other side-channel information, such as power consumption or resource usage. In order to solve this issue, more refined models should be considered. However, this would both increase the complexity of specifications, making them harder to understand and write, and the complexity of verification. Again, a compositional approach could mitigate this problem. Note that being able to model and trace resource usage could also mitigate denial of service (DoS) attacks: if a security protocol is designed such that, during a DoS attack session, the attacked agent uses less resources than the attacker, it becomes unworthy for the attacker to complete its job (Meadows, 2001). Unfortunately, this does not hold for distributed DoS, where the attacker controls different machines, thus having much more computational power than the attacked agent. Finally, it must be noted that resource usage can be implementation dependent, making it non-trivial to model this aspect in an abstract and generic way.
Dolev-Yao models assume perfect cryptography, meaning that all possible bad interactions between the protocol and the cryptographic algorithms are not considered. Since recent works provide correctness proofs for some cryptographic algorithms (e.g. Bresson, Chevassut, & Pointcheval, 2007) (which also highlight their limitations), the next foreseeable step in this direction is to merge the two worlds, providing models and correctness proofs of security protocols, down to the cryptographic algorithm level.
Two indicators of field maturity show that there is still some research work to be done. The first indicator is the presence of standard frameworks or methodologies. As a matter of facts, no commonly agreed standard exists in protocol specification or verification: each problem stated above is solved with special approaches and tools, and their interoperability is quite limited. Indeed, very few de-facto standards for generic security protocols exist; the ISO/IEC CD 29128 (2010) standard is currently under development, but it is not trivial to foresee whether it will be widely adopted or not.
The second indicator is the usage of formal methods in the release of a security protocol standard. As the tool support is often not completely automatic and user-friendly, and it requires some expertise and implies a steep learning curve, formal methods are usually deemed too expensive for the average security protocol standard. The result is that often standards are released without the support of formal methods, and the latter are applied after the standard is published (see the AVISPA project, Viganò, 2006), making it hard to fix issues highlighted by their application. It is believable that as soon as more automatic and user-friendly tools will emerge, and the presence of networks of computers will become even more

pervasive and dependable, formal methods will be adopted, in order to fulfill the call for some form of certification of correctness.

In turns, this will require that automatic formal methods will be able to scale to handle large protocols, and even full security-aware applications, that make use of different protocols simultaneously. Unfortunately, composition of security protocols is not linear (Cremers, 2006), meaning that combining two protocols does not guarantee that the outcome protocol preserves the security properties of the twos. Indeed, combining protocols may have a disruptive effect, actually even breaking some of the expected security properties. As a practical example, an e-commerce website may wish to use SSL/TLS in order to setup a secure channel with the customer, and then use some electronic payment protocol (EPP) to redirect the customer to the bank website, in order to perform the purchase. Although each protocol could be proven secure in isolation, flaws may be found when using them together. When dealing with protocol composition, one (negative) result is already available: for each security protocol, a made-up attack-protocol can be found, which breaks security of the original protocol (Kelsey, Schneier, & Wagner, 1997). In principle, this means that it is not possible to check the interaction of one protocol with any other protocol, because it would be always possible to find a corresponding attack-protocol. Nevertheless, it is still meaningful to select a group of interacting protocols (SSL/TLS and EPP in the example) and check their interactions. Combining protocols greatly increases their complexity, making formal verification harder to apply, because of the required resources. Since this issue has a practical impact, it is believable that some research will be motivated in finding some compositional properties for security protocols, or some methodologies that would allow modular verification of security protocols. For example, Cortier, & Delaune (2009) propose to tag security protocols that share the same secrets, so that they execute like each protocol is running in isolation, because their messages cannot be interchanged. This approach does not apply to existing protocols, but it could be taken into account in the design of new protocols.

As an example of new aspects of security protocols that researchers are trying to include in formal analysis tools, we can mention open-ended protocols. Usually, security protocols have a fixed number of participants and a fixed message structure; only the number of sessions and the steps made by the attacker to create a message can be unbounded. However, there exist protocols, such as group key exchange protocols, where the number of participants is unbounded, or protocols where the message exchange sequence may contain loops with an unbounded number of cycles. These protocols are known as open-ended protocols. For example, any protocol dealing with certificate chains is an open-ended protocol, because it faces a potentially unbounded number of certificate checks to be performed, each involving message exchanges in order to check the certificate revocation lists. Few works (e.g. Küsters, 2005) have explicitly addressed this problem, probably because classic protocols were deemed more urgent to address. Nevertheless, the increasing number of Internet-connected mobile devices is rising the need of open-ended protocols, making if foreseeable that more research work dealing with open-ended protocols verification will be developed.

5. CONCLUSION

This chapter has introduced the main problems that have to be addressed for engineering security protocols and how formal methods and related automated tools can now help protocol designers and implementers to improve quality. The chapter has stressed the progress that has been made in this field in the last years. While in the past formal security proofs required so high levels of expertise that only few researchers and field experts could develop them, the availability of fully automated tools has now enabled a wider number of protocol developers to get advantage of formal methods for security protocols. Given the vast scope of available solutions, attention has been focused just on the most popular and most representative ones, without exhaustiveness claims. The chapter has covered not just the formal protocol design techniques, but also the techniques that can be used in order to enforce strict correspondence between formal protocol specifications and their implementations.

REFERENCES

Abadi, M., & Fournet, C. (2001). Mobile values, new names, and secure communication. In *Symposium on Principles of Programming Languages* (pp. 104-115).

Abadi, M., & Gordon, A. D. (1998). A Calculus for Cryptographic Protocols: The Spi Calculus. *Research Report 149*.

Abadi, M., & Needham, R. (1996). Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22 , 122-136.

Abadi, M., & Rogaway, P. (2002). Reconciling two views of cryptography (The computational soundness of formal encryption). *Journal of Cryptology*, 15 (2), 103-127.

Albrecht M.R., Watson G.J. & Paterson K.G. (2009). Plaintext Recovery Attacks Against SSH. In *IEEE Symposium on Security and Privacy* (pp. 16-26).

Arapinis, M., Delaune, S., & Kremer, S. (2008). From One Session to Many: Dynamic Tags for Security Protocols. In *Logic for Programming, Artificial Intelligence, and Reasoning* (pp. 128-142).

Basin, D. A., Mödersheim, S., & Viganò, L. (2005). OFMC: A symbolic model checker for security protocols. *International Journal of Information Security*, 4 (3), 181-208.

Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A. D., & Maffeis, S. (2008). Refinement Types for Secure Implementations. In *IEEE Computer Security Foundations Symposium* (pp. 17-32).

Bhargavan, K., Fournet, C., & Gordon, A. D. (2006). Verified Reference Implementations of WS-Security Protocols. In *Web Services and Formal Methods* (pp. 88-106).

Bhargavan, K., Fournet, C., & Gordon, A. D. (2010). Modular verification of security protocol code by typing. In *Symposium on Principles of Programming Languages* (pp. 445-456).

Bhargavan, K., Fournet, C., Gordon, A. D., & Tse, S. (2006). Verified Interoperable Implementations of Security Protocols. In *Computer Security Foundations Workshop* (pp. 139-152).

Blanchet, B. (2001). An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *IEEE Computer Security Foundations Workshop* (pp. 82-96).

Blanchet, B. (2008). A Computationally Sound Mechanized Prover for Security Protocols. *IEEE Transactions on Dependable and Secure Computing*, 5 (4), 193-207.

Blanchet, B. (2009). Automatic Verification of Correspondences for Security Protocols. *Journal of Computer Security*, 17 (4), 363-434.

Brackin, S. (1998). Evaluating and improving protocol analysis by automatic proof. In *IEEE Computer Security Foundations Workshop* (pp. 138-152).

Bresson, E., Chevassut, O., & Pointcheval, D. (2007). Provably secure authenticated group Diffie-Hellman key exchange. *ACM Transactions on Information and System Security (TISSEC)*, 10 (3).

Burrows, M., Abadi, M., & Needham, R. (1990). A Logic of Authentication. *ACM Transactions on Computer Systems*, 8 (1), 18-36.

Carlsen, U. (1994). *Cryptographic protocol flaws: know your enemy*. In *IEEE Computer Security Foundations Workshop* (pp. 192-200).

Chen, Q., Su, K., Liu, C., Xiao, Y., (2010). Automatic Verification of Web Service Protocols for Epistemic Specifications under Dolev-Yao Model. In *International Conference on Service Sciences* (pp. 49-54).

Common Criteria for Information Technology Security Evaluation (2009), and the Common Methodology for Information Technology Security Evaluation. Retrieved from http://ww.commoncriteriaportal.org/index.html.

Clark, J., & Jacob, J. (1997). A Survey of Authentication Protocol Literature: Version 1.0 (Technical Report).

Comon-Lundh, H., & Cortier, V. (2008). Computational soundness of observational equivalence. In *ACM conference on Computer and communications security* (pp. 109-118).

Comon, H., & Shmatikov, V. (2002). Is it Possible to Decide whether a Cryptographic Protocol is Secure or Not? *Journal of Telecommunications and Information Technology*, 4/2002 , 5-15.

Cortier, V., & Delaune, S., (2009). Safely composing security protocols. *Formal Methods in System Design*, 34 (1), 1-36.

Cremers, C. J. F. (2006). Feasibility of Multi-Protocol Attacks. In *Availability, Reliability and Security* (pp. 287-294).

Dolev, D., & Yao, A. C.-C. (1983). On the security of public key protocols. *IEEE Transactions on Information Theory*, 29 (2), 198-207.

Durante, L., Sisto, R., & Valenzano, A. (2003). Automatic testing equivalence verification of spi calculus specifications. *ACM Transactions on Software Engineering and Methodology*, 12 (2), 222-284.

Durgin, N. A., Lincoln, P., & Mitchell, J. C. (2004). Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12 (2), 247-311.

Escobar, S., Meadows, C., and Meseguer, J. (2009). Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties. In *Foundations of Security Analysis and Design* (pp. 1-50).

Fábrega, F. J. T., Herzog, J. C., & Guttman, J. D. (1999). Strand Spaces: Proving Security Protocols Correct. *Journal of Computer Security*, 7 (2/3), 191-230.

Goubault-Larrecq, J., & Parrennes, F. (2005). Cryptographic Protocol Analysis on Real C Code. In *Verification, Model Checking, and Abstract Interpretation* (pp. 363-379).

Gritzalis, S., Spinellis, D., & Sa, S. (1997). Cryptographic Protocols over Open Distributed Systems: A Taxonomy of Flaws and Related Protocol Analysis Tools. In *International Conference on Computer Safety, Reliability and Security* (pp. 123-137).

Gu, Y., Shen, Z., & Xue, D. (2009). A Game-Theoretic Model for Analyzing Fair Exchange Protocols. In *International Symposium on Electronic Commerce and Security* (pp. 509-513).

Hui, M. L., & Lowe, G. (2001). Fault-preserving simplifying transformations for security protocols. *Journal of Computer Security*, 9 (1/2), 3-46.

Huima, A. (1999). Efficient Infinite-State Analysis of Security Protocols. In *Workshop on Formal Methods and Security Protocols*.

ISO/IEC CD 29128. (2010). *Verification of Cryptographic Protocols.* Under development.

Jeon, C.-W., Kim, I.-G., & Choi, J.-Y. (2005). Automatic Generation of the C# Code for Security Protocols Verified with Casper/FDR. In *International Conference on Advanced Information Networking and Applications* (pp. 507-510).

Jürjens, J. (2005). Verification of low-level crypto-protocol implementations using automated theorem proving. In *Formal Methods and Models for Co-Design* (pp. 89-98).

Kelsey, J., Schneier, B., & Wagner, D. (1997). Protocol Interactions and the Chosen Protocol Attack. In *Security Protocols Workshop* (pp. 91-104).

Küsters, R. (2005). On the decidability of cryptographic protocols with open-ended data structures. *International Journal of Information Security*, 4 (1-2), 49-70.

Lowe, G. (1996). Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. *Software - Concepts and Tools*, 17 (3), 93-102.

Lowe, G. (1998). Towards a Completeness Result for Model Checking of Security Protocols. In *IEEE Computer Security Foundations Workshop* (pp. 96-105).

Maggi, P., & Sisto, R. (2002). Using SPIN to Verify Security Properties of Cryptographic Protocols. In *SPIN Workshop on Model Checking of Software* (pp. 187-204).

Marschke, G. (1988). The directory authentication framework. In *CCITT Recommendation X.509*.

Meadows, C. A. (1996). The NRL Protocol Analyzer: An Overview. *Journal of Logic Programming*, 26 (2), 113-131.

Meadows, C. A. (2001). A cost-based framework for analysis of denial of service in networks. *Journal of Computer Security*, 9 (1), 143-164.

Milner, R. (1999). Communicating and Mobile Systems: the Pi-Calculus. Cambridge University Press.

Mödersheim, S., & Viganò, L., (2009). The Open-Source Fixed-Point Model Checker for Symbolic Analysis of Security Protocols. In *Foundations of Security Analysis and Design* (pp. 166-194).

Monniaux, D. (1999). Decision procedures for the analysis of cryptographic protocols by logics of belief. In *IEEE Computer Security Foundations Workshop* (pp. 44-54).

Needham, R., & Schroeder, M. (1978). Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21 (12), 993-999.

OpenSSL Team (2009). OpenSSL Security Advisor. Available at http://www.openssl.org/news/secadv_20090107.txt

Otway, D., & Rees O. (1987) Efficient and timely mutual authentication. *Operating Systems Review*, 21(1):8–10.

Paulson, L. (1998). The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6 (1-2), 85-128.

Pironti, A., & Sisto, R. (2007). An Experiment in Interoperable Cryptographic Protocol Implementation Using Automatic Code Generation. In *IEEE Symposium on Computers and Communications* (pp. 839-844).

Pironti, A., & Sisto, R. (2010). Provably Correct Java Implementations of Spi Calculus Security Protocols Specifications. *Computers & Security*, 29 (3), 302-314.

Pozza, D., Sisto, R., & Durante, L. (2004). Spi2Java: Automatic Cryptographic Protocol Java Code Generation from spi calculus. In *Advanced Information Networking and Applications* (pp. 400-405).

Qingling, C., Yiju, Z., & Yonghua, W., (2008). A Minimalist Mutual Authentication Protocol for RFID System & BAN Logic Analysis. In *International Colloquium on Computing, Communication, Control, and Management* (pp. 449-453).

Ryan, P., & Schneider, S. (2000). The modelling and analysis of security protocols: the CSP approach. Addison-Wesley Professional.

Schneider, S. (1996). Security properties and CSP. In *IEEE Symposium on Security and Privacy* (pp. 174-187).

Shamir, A., Rivest, R., & Adleman, L. (1978). Mental Poker (Technical Report). Massachusetts Institute of Technology.

Song, D. X., Berezin, S., & Perrig, A. (2001). Athena: A Novel Approach to Efficient Automatic Security Protocol Analysis. *Journal of Computer Security*, 9 (1/2), 47-74.

Tobler, B., & Hutchison, A. (2004). Generating Network Security Protocol Implementations from Formal Specifications. In *Certification and Security in Inter-Organizational E-Services*. Toulouse, France.

Viganò, L. (2006). Automated Security Protocol Analysis with the AVISPA Tool. *Electronic Notes on Theoretical Computer Science*, 155, 61-86.

Voydock, V. L., & Kent, S. T. (1983). Security mechanisms in high-level network protocols. *ACM Computing Surveys*, 15 (2), 135-171.

Woodcock, J., Larsen, P. G., Bicarregui, J., & Fitzgerald, J. (2009). Formal methods: Practice and experience. *ACM Computing Surveys*, 41 (4), 1-36.

Yafen, L., Wuu, Y., & Ching-Wei, H. (2004). Preventing type flaw attacks on security protocols with a simplified tagging scheme. In *Symposium on Information and Communication Technologies* (pp. 244-249).

Ylonen, T. (1996). SSH - Secure login connections over the internet. In *USENIX Security Symposium* (pp. 37-42).

ADDITIONAL READING SECTION

Abadi, M. (1999). Secrecy by typing in security protocols. Journal of the ACM, 46 (5), 749-786.

Abadi, M. (2000). Security protocols and their properties. In *Foundations of Secure Computation* (pp. 39-60).

Abadi, M., & Blanchet, B. (2002). Analyzing security protocols with secrecy types and logic programs. *ACM SIGPLAN Notices*, 37 (1), 33-44.

Aura, T. (1997). Strategies against replay attacks. In *IEEE Computer Security Foundations Workshop* (pp. 59-68).

Bodei, C., Buchholtz, M., Degano, P., Nielson, F., & Nielson, H. R. (2005). Static validation of security protocols. *Computer Security*, 13 (3), 347-390.

Bugliesi, M., Focardi, R., & Maffei, M. (2007). Dynamic types for authentication. *Journal of Computer Security*, 15 (6), 563-617.

Carlsen, U. (1994, Jun). Cryptographic protocol flaws: know your enemy. In *Computer Security Foundations Workshop* (pp. 192-200).

Clarke, E. M., Jha, S., & Marrero, W. (2000). Verifying security protocols with Brutus. *ACM Transactions on Software Engineering and Methodology*, 9 (4), 443-487.

Coffey, T. (2009). A Formal Verification Centred Development Process for Security Protocols. In Gupta, J. N. D. & Sharma, S. (Ed.), *Handbook of Research on Information Security and Assurance* (pp. 165-178). IGI Global.

Crazzolara, F., & Winskel, G. (2002). Composing Strand Spaces. In *Foundations of Software Technology and Theoretical Computer Science* (pp. 97-108).

Denker, G., & Millen, J. (2000). CAPSL integrated protocol environment. In *DARPA Information Survivability Conference and Exposition* (pp. 207-221).

Donovan, B., Norris, P., & Lowe, G. (1999). Analyzing a library of security protocols using Casper and FDR. In *Workshop on Formal Methods and Security Protocols*.

Durgin, N. A., & Mitchell, J. C. (1999). Analysis of Security Protocols. In *Calculational System Design* (pp. 369-395).

Gong, L. (1995). Fail-stop protocols: An approach to designing secure protocols. In *Dependable Computing for Critical Applications* (pp. 44-55).

Gordon, A. D., Hüttel, H., & Hansen, R. R. (2008). Type inference for correspondence types. In *Security Issues in Concurrency* (pp. 21-36).

Gordon, A. D., & Jeffrey, A. (2003). Authenticity by Typing for Security Protocols. *Journal of Computer Security*, 11 (4), 451-521.

Haack, C., & Jeffrey, A. (2006). Pattern-matching spi-calculus. *Information and Computation*, 204 (8), 1195-1263.

Hubbers, E., Oostdijk, M., & Poll, E. (2003). Implementing a Formally Verifiable Security Protocol in Java Card. In *Security in Pervasive Computing* (pp. 213-226).

ISO/IEC 15408 - Security techniques - Evaluation criteria for IT security. (2005).

Jürjens, J. (2002). UMLsec: Extending UML for Secure Systems Development. In *The Unified Modeling Language* (pp. 412-425).

Kocher, P. C., Ja_e, J., & Jun, B. (1999). Differential power analysis. In *Advances in Cryptology - CRYPTO* (pp. 388-397).

Lodderstedt, T., Basin, D. A., & Doser, J. (2002). SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *The Unified Modeling Language* (pp. 426-441).

Lowe, G. (1997). A hierarchy of authentication specifications. In *Computer Security Foundations Workshop* (pp. 31-43).

Luo, J.-N., Shieh, S.-P., & Shen, J.-C. (2006). Secure Authentication Protocols Resistant to Guessing Attacks. *Journal of Information Science and Engineering*, 22 (5), 1125-1143.

Marschke, G. (1988). The Directory Authentication Framework. *CCITT Recommendation X.509*.

Meadows, C. (2003). Formal Methods for Cryptographic Protocol Analysis: Emerging Issues and Trends. (Technical Report).

Meadows, C. (2004). Ordering from Satan's menu: a survey of requirements specification for formal analysis of cryptographic protocols. Science of Computer Programming 50 (1-3), 3-22.

Mitchell, J. C., Mitchell, M., & Stern, U. (1997, May). Automated analysis of cryptographic protocols using MurΦ. In *IEEE Symposium on Security and Privacy* (pp. 141-151).

Pironti, A., & Sisto, R. (2008a). Formally Sound Refinement of Spi Calculus Protocol Specifications into Java Code. In *IEEE High Assurance Systems Engineering Symposium* (pp. 241-250).

Pironti, A., & Sisto, R. (2008b). Soundness Conditions for Message Encoding Abstractions in Formal Security Protocol Models. In *Availability, Reliability and Security* (pp. 72-79).

Roscoe, A. W., Hoare, C. A. R., & Bird, R. (1997). The theory and practice of concurrency. Prentice Hall PTR.

Ryan, P. & Schneider, S. & Goldsmith, M. & Lowe, G. & Roscoe, B. *(Ed.) (2001). The Modelling and Analysis of Security Protocols*, Addison-Wesley.

Stoller, S. D. (2001). A Bound on Attacks on Payment Protocols. In *IEEE Symposium on Logic in Computer Science* (p. 61).

Wen, H.-A., Lin, C.-L., & Hwang, T. (2006). Provably secure authenticated key exchange protocols for low power computing clients. *Journal of Computer Security*, 25 (2), 106-113.

Woo, T. Y. C., & Lam, S. S. (1993). A Semantic Model for Authentication Protocols. In *IEEE Symposium on Security and Privacy* (pp. 178-194).

Xiaodong, S. D., David, W., & Xuqing, T. (2001). Timing analysis of keystrokes and timing attacks on SSH. In *USENIX Security Symposium* (pp. 25-25).

KEY TERMS & DEFINITIONS

Security Protocol: communication protocol aimed at achieving a goal over an unsecure network
Formal Method: mathematically-based method for specifying and analyzing systems
Formal Model: description of a system with unambiguous, mathematically defined semantics
Formal Verification: providing evidence (by a mathematical proof) of a property holding on a formal model
Theorem Proving: formal verification technique based on direct building of mathematical proofs
State Space Exploration: formal verification technique based on exploration of all the possible behaviors of a model
Model Extraction: technique for automatically extracting a formal model from an implementation
Code Generation: technique for automatically refining a formal model into an implementation