Introduction to Formal Methods for the Analysis and Design of Cryptographic Protocols

Chapter 2

Introduction to Formal Methods for the Analysis and Design of Cryptographic Protocols

DANIELE BRINGHENTI[1], RICCARDO SISTO[1], FULVIO VALENZA[1], JALOLLIDDIN YUSUPOV[2]

This chapter introduces formal methods, and how they can be applied to cryptographic protocols, in order to detect their weaknesses early and achieve high assurance about their correctness. Both design and implementation phases will be covered. The chapter starts with an introduction to cryptographic protocols in section 2.1, and to formal methods in section 2.2. Then, section 2.3 presents the various ways formal methods can be used throughout the cryptographic protocols lifecycle. Afterwards, in section 2.4, the different types of formal models that can be used for cryptographic protocols are detailed, while the next two sections focus, respectively, on the main formal verification techniques that can be used for cryptographic protocols at design level (section 2.5) and the frameworks for cryptographic protocol engineering (section 2.7). Finally, section 2.8 draws conclusions.

2.1 Cryptographic protocols

A communication protocol is a set of rules about communication, designed to achieve some goals in a distributed system. Such rules govern, for instance, the format of exchanged messages, and the procedures that each process in the system should follow

[1]D. Bringhenti, R. Sisto, and F. Valenza are with the Dipartimento di Automatica e Informatica, Politecnico di Torino, Turin, Italy (e-mail: daniele.bringhenti@polito.it, riccardo.sisto@polito.it ,fulvio.valenza@polito.it).

[2]J. Yusupov is with the Department of Automatic Control and Computer Engineering, Turin Polytechnic University in Tashkent, Tashkent, Uzbekistan (e-mail: jaloliddin.yusupov@polito.uz).

to reach the goals, leaving other implementation details free. If each process in the system follows the protocol rules, the goals should be reached, independently of the implementation choices made when implementing each process (this property is called interoperability).

Cryptographic protocols aim to guarantee security proprieties in a distributed system against the antagonistic actions of attackers who threaten the system. Many of these security properties have been defined in recent years. The standard X.800 [29] classifies such proprieties into five macro categories and fourteen sub-categories. Tables 2.1 and 2.2, summarize this classification. A precise definition of these and even other properties can be found in the RFC-2828 [77] glossary. In addition to the 5 macro-categories defined in X.800, a sixth one is worth to mention, which is deserving incresing attention. It is the class of privacy-related specific properties, i.e. security properties that aim specifically to guarantee privacy, i.e., "the right of individuals to control what information related to them may be collected and stored and by whom and to whom that information may be disclosed"[77]. The most notable examples of such properties are anonymity (i.e., the concealment of a user's identity) and untraceability (i.e., the concealment of the identity of who made some action).

As their name suggests, cryptographic protocols usually reach their goals by means of cryptography. However, the more general term security protocols is also in use in literature.

Table 2.1: Macro Categories of Security Proprieties (as defined in X.800)

Authentication: guarantee that a communicating entity is who it claims to be (e.g., making sure that the other peer engaged in a point-to-point communication is who it clims to be or that the origin of some data is the claimed one). Several mechanisms/technologies can be used for authentication like passwords, tokens, signatures, challenges.

Access Control: protection against unauthorized use of resources. This is usually achieved by authenticating the user who is trying to use a resource and checking that user's authorization before granting access.

Data confidentiality: protection from unauthorized disclosure of data; (e.g., guarantee that confidential information is only available to the authorized partners of a communication). This is usually achieved by encrypting data.

Data integrity: protection against unauthorized changes to data, including both intentional change or destruction and accidental change or loss. This is usually achieved by making unauthorized changes to data detectable (e.g., by means of digest checking or signature checking).

Non-repudiation: protection against false denial of involvement in an association (e.g., assurance that a signed document cannot be repudiated by the signer). This is usually achieved by employing digital signatures.

Table 2.2: Sub-categories of security proprieties (as defined in X.800)

Peer entity authentication: confirmation of the identities of one or more of the entities connected to one or more of the other entities.

Data origin authentication: corroboration of the source of some data.

Connection confidentiality: confidentiality of all user-data exchanged on a connection.

Connectionless confidentiality: confidentiality of all user-data exchanged in a single connectionless message exchange (e.g., confidentiality of a single block of data, like a packet).

Selective field confidentiality: confidentiality of selected fields within the user-data exchanged on a connection or in a single connectionless message exchange.

Traffic flow confidentiality: confidentiality of the information which might be derived from the observation of traffic.

Connection integrity: integrity of all user-data exchanged on a connection or detection of any modification, insertion, deletion or duplication of any data.

Connection integrity with recovery: integrity of all user-data exchanged on a connection or detection of any modification, insertion, deletion or duplication of any data with recovery attempted.

Selective field connection integrity : integrity of selected fields within the user data exchanged on a connection or detection of any modification, insertion, deletion, or duplication of the data in a selected set of fields.

Connectionless integrity: integrity of a single connectionless data exchange or detection of any modification, insertion, deletion, or duplication of the single message.

Selective field connectionless integrity: integrity of selected fields within the data in a single connectionless data exchange or detection of any modification, insertion, deletion, or duplication of the single message.

Non-repudiation with proof of origin: protection against any attempt by the sender to falsely deny sending the data or its contents.

Non-repudiation with proof of delivery protection against any subsequent attempt by the recipient to falsely deny receiving the data or its contents.

We use cryptographic protocols in everyday computing, like log in to a system (e.g., Kerberos authentication), or conduct a secure e-commerce transaction (e.g., TLS). Such real-life protocols are often specified as international standards, but not always: sometimes, custom protocols are defined and used for particular purposes. Real-life cryptographic protocols are characterized by the full complexity of a real protocol, including all the aspects that are necessary for interoperability (e.g., precise defintion of message formats, options, etc.). In the world of research, instead, it is common to focus on more abstract and simpler protocol models, where some details, such as the precise binary message formats, are left out for simplicity (this is the case, for example, of the Needham-Schroeder Public Key Authentication Protocol that will be presented below as an example). Such models, that we can call theoretical protocols, catch only the essence of the protocol rules, and they can be used to reason about the basic protocol mechanisms and the security properties they can enforce, before defining the full-fledged protocol specification intended for real applications. In 1978, Roger Needham and Michael Schroeder initiated a large body of work on the design and analysis of cryptographic protocols with their proposal of two theoretical authentication protocols [64], one based on symmetric encryption, and the other one based on asymmetric encryption. Later on, it was discovered that both these protocols were flawed, and new theoretical protocols were proposed to fix them. This process of error and fix continued, witnessing how difficult it is to detect flaws in these protocols, finally leading to the standard real-life protocols Kerberos [65] (based on symmetric keys, and widely used in the Windows and mac-OS OSs) and SSL/TLS [39, 73] (based on asymmetric keys and certificates, and widely used in the web). A classical survey of such theoretical protocols was produced by Clark and Jacob [30]. The survey presents not only a description of the classical theoretical protocols but also the known attacks that were discovered after their publication.

The standard real-life cryptographic protocols are typically reviewed many times by academy researchers, industry experts, and standards organizations (e.g., IETF, ETSI, ISO, IEEE) before being standardized. New protocols are subjected to security threat modeling and analysis to ensure that they offer protection against commonly known attack patterns. When these protocols are deployed in practice, their robustness is monitored, and over time their security issues are worked out. When one of these standard protocols is evaluated as insecure, a more secure version is made available (e.g., TLS 1.1/1.2/1.3 to replace TLS 1.0), or new protocols are designed to replace the aging ones (e.g., AES replaced the aging DES/3DES).

A cryptographic protocol uses communication and cryptographic primitives to achieve its goals. A cryptographic protocol involves two or more communicating actors, in literature commonly named as principals. Their communications may occur on communication channels of various kinds (e.g., Ethernet, Wi-Fi, TCP/IP communications, public/private networks) by message exchanges. Multiple sessions of a cryptographic protocol can run concurrently.

One or more roles are associated with each principal, depending on the reasons why the principal participates in the protocol. For each role, the protocol defines a set of rules that all the principals playing that role have to stick to. When a principal executes a role during the effective communication, it is commonly named agent in the literature.

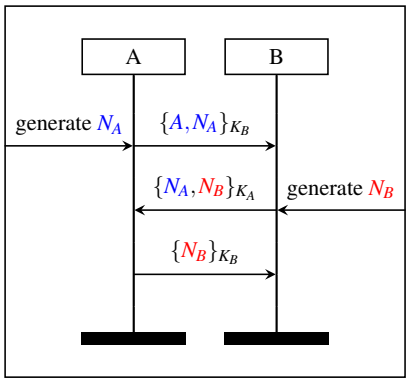

Figure 2.1: Needham-Schroeder Public-key Authentication Protocol

One significant yet simple example of a theoretical cryptographic protocol is one of the two authentication protocols proposed by Roger Needham and Michael Schroeder in 1978 [64] and known as the Needham-Schroeder Public Key (NSPK) protocol. Many existing protocols are derived from the NSPK protocol, including the widely used Kerberos authentication protocol suite.

The aim of the protocol is to establish mutual authentication between principals A (the initiator) and B (the receiver). In the original formulation of the protocol, A and B obtain each other's public key by interacting with a public key server. Here, we present a simplified version, in which it is assumed that A and B already know each other's public key. As shown in Figure 2.1, the protocol is composed of three messages:

1. in the first step, A creates a random value (i.e., a nonce or a challenge) N_A and encrypts this value along with its name A with the public key K_B of B;

2. when B receives such a message, B decrypts it, generates another random value N_B and responds to the challenge by encrypting both nonces, with the public key of A (i.e., K_A), in order to show to A that B was really able to decrypt the original message;

3. in the last step, A sends back N_B to B, encrypted by K_B, in order to show that it was able to decrypt the second message.

At the end of the session, A should have the guarantee that the other party is really B, and B have the guarantee that the other party is A. Moreover, the two nonces N_A and N_B should be shared secrets, only known by A and B, and can be used to establish a session key to encrypt further messages.

Simple cryptographic protocols are usually described by means of the so-called Alice and Bob notation, which specifies the sequence of messages exchanged, with the indication of who is the sender and who is the intended recipient of each message. For

the NSPK protocol, this description reads as follows.

$$\begin{aligned} A \rightarrow B &: \{A, N_A\}_{K_B} \\ B \rightarrow A &: \{N_A, N_B\}_{K_A} \\ A \rightarrow B &: \{N_B\}_{K_B} \end{aligned}$$

A second simple example of a theoretical cryptographic protocol is the one-way authentication based on X.509 certificates, which is described in the X.509 standard [49, 48]. This protocol can be used when a principle A sends some data to another principle B, in order to provide the following security guarantees to the recipient B: (a) the received data were generated by A, were not modified, and are timely (i.e. they are not a reply of already sent data) (b) the received data were intended for B (c) a certain identified part of the received data is confidential (i.e., only known to A and B). The protocol starts from the assumption that A and B know each other's public key (which can be obtained securely by means of X.509 certificates) and it includes a single message sent from A to B:

$$A \rightarrow B : A, T_A, N_A, B, X, \{Y\}_{K_B}, \{h(T_A, N_A, B, X, \{Y\}_{K_B})\}_{K_A^{-1}}$$

where T_A and N_A are, respectively, a time stamp and a nonce, generated by A, X represents the non-confidential data to be sent, Y the confidential data to be sent, K_B is B's public key, K_A^{-1} is A's private key, and $h()$ is a cryptographic hash function. Here, the encrypted hash is a signature that will be checked by B before accepting the message as valid.

In the execution of a cryptographic protocol, an actor is legitimate if allowed to execute it, illegitimate if not allowed to. Besides, an actor is honest if he/she does not deviate from the protocol specification, dishonest he/she deviates, possibly preventing the protocol from reaching the expected security goals. Dishonest actors may be legitimate: dishonest because deviating from the protocol specification, legitimate because allowed to execute the protocol.

It is possible to divide attackers into two classes: passive and active. Passive attackers just intercept, copy and inspect protocol messages, while active ones can also interfere with the protocol, by deleting, altering, reordering and redirecting protocol messages, as well as by forging new protocol messages and inserting them into the conversation.

It is also possible to categorize the attacks on security protocols depending on the weaknesses they exploit [71, 45, 27]. Table 2.3 reports a non-exhaustive list of the most common classes of attacks. Moreover, it is important to underline that the above classification is not a partition in a mathematical sense, i.e., that some attacks may fall into several classes.

As an example of an attack on a cryptographic protocol, in Figure 2.2 we show the man-in-the-middle attack to the NSPK Authentication Protocol, which was discovered by Lowe in 1995 [56], 17 years after the protocol was initially defined! In the mean time, the flawed version had been used as the basis of commercial products, which, of course, were vulnerable to this attack. In order to break the protocol, an attacker E persuades A to initiate a session with it. Then, it relays the messages to B and convinces

Table 2.3: Most common classes of attacks

Cryptographic flaw attacks: attacks that try to break idealistic properties of cryptographic operations by exploiting weaknesses of the cryptographic algorithms. Analyzing an encrypted message to extrapolate the secret key is an example of a cryptographic flaw attack. These attacks can be refined to take collateral information into account, leaked by a cryptographic primitive or by its particular usage within a security protocol [51].
Internal action flaw attacks: attacks that exploit the absence of some protocol operations that are crucial to guarantee a security property. This kind of weakness falls within the so-called internal action flaw class and may be due both to protocol design errors or to implementation mistakes [24].
Type flaw attacks: attacks based on the absence of proper message type checking. In this kind of attack, the attacker sends a protocol actor a message of different type than what expected, and the actor fails to detect the type mismatch, so misinterpreting the message contents or behaving in an unexpected way [40, 47].
Replay attacks: attacks that are based on re-sending an intercepted message, exploiting the weakness that some actor is not able to distinguish between a fresh message and an old message that is being re-used [50].
Man-in-the-middle attacks: attacks where the attacker stands in the middle between two honest actors and breaks the protocol while relaying messages from one actor to the other [32].
Oracle attacks: attacks where the attacker uses a protocol actor as an oracle to get some information that the attacker could not generate on its own. Then, this information can be used by the attacker to forge new messages that get injected into another parallel protocol session [81].

B that B is communicating with A, by executing the following steps (as illustrated in Figure 2.2:

1. A sends N_A to E, who decrypts the message with K_E;
2. E relays the message to B, pretending that A wants to communicate with B;
3. B sends N_B
4. E relays it to A
5. A decrypts N_B and confirms it to E, who learns it;
6. E re-encrypts N_B, and convinces B that she's decrypted it.

At the end of the attack, B falsely believes that A started a session with B, and that N_A and N_B are known only to A and B, while in fact A started a session with E and E knows N_A.

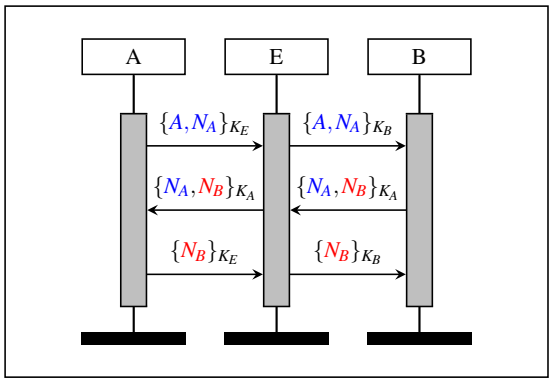

Figure 2.2: Man-in-the-middle attack on the Needham-Shroeder Public Key Authentication protocol

Focusing now on the standard, real-life cryptographic protocols, a significant example widely spread in applications such as web, email, and instant messaging, is the SSL/TLS protocol, which aims primarily at providing authentication, privacy and data integrity between two principals. This protocol will be referenced throughout the chapter.

The Secure Sockets Layer (SSL) protocol was firstly defined by Netscape in 1994. However, the first official release of SSL, version 2.0, was out in 1995, because version 1.0 of SSL was never released because it had serious security flaws. While, the updated and final version of the SSL protocol, SSL 3.0, was released in November 1996.

In 1999, the Internet Engineering Task Force (IETF) introduced the Transport Layer Security (TLS) protocol as an upgrade to SSL v3. The TLS 1.0 RFC document (RFC 2246) states the differences between TLS 1.0 and SSL 3.0, while the TLS 1.1 RFC document (RFC 4346) describes minor updates to TLS 1.0 released in April 2006. In August 2008, the TLS 1.2 (RFC 5246) was released with the following main changes: (i) adding cipher-suite-specified pseudorandom functions (PRFs); (ii) adding AES cipher suites; (iii) removing IDEA and DES cipher suites. The current version of TLS, TLS 1.3, was released in August 2018 (RFC 8446), where several unsafe technologies were removed, and the protocol was streamlined for better performance.

The SSL/TLS protocol runs in the application layer. It is composed of two parts: the TLS Record and the TLS Handshake protocol. Specifically, the cryptographic parameters of the session state are produced by the TLS Handshake Protocol, which operates on top of the TLS Record Protocol. When a TLS client and server first start communicating, they agree on a protocol version, select cryptographic algorithms, optionally authenticate each other, and use public-key encryption techniques to generate shared secrets.

In this chapter, for simplicity, we focus only on the TLS Handshake protocol, with reference to TLS 1.0. This Protocol involves the following phases:

- Exchange hello messages to agree on algorithms, exchange random values, and check for session resumption.

- Exchange the necessary cryptographic parameters to allow the client and server to agree on a premaster secret.
- Exchange certificates and cryptographic information to allow the client and server to authenticate themselves.
- Generate a master secret from the premaster secret and exchanged random values.
- Provide security parameters to the record layer.
- Allow the client and server to verify that their peer has calculated the same security parameters and that the handshake occurred without tampering by an attacker.

Specifically, the full handshake protocol messages of TLS 1.0 are presented below and in Figure 2.3, in the order they must be sent. The only message which is not bound by these ordering rules is the Hello Request message, which can be sent at any time, but it should be ignored by the client if it arrives in the middle of a handshake. The character * is used to flag optional or situation-dependent messages that are not always sent.

A full specification of each message and of the protocol state machine would be too complex to be reported here. They can be found in the protocol standard (RFC 2246).

1. When a client first connects to a server it is required to send the **client hello** as its first message;
2. The server will the **server hello** in response to a client hello message when it was able to find an acceptable set of algorithms;
3. The Server sends **certificate**, whenever the agreed-upon key exchange method is not an anonymous one. This message will always immediately follow the server hello message;
4. The **server key exchange message** is sent by the server only when the server certificate message (if sent) does not contain enough data to allow the client to exchange a premaster secret;
5. A non-anonymous server can optionally request a **certificate** from the client, if appropriate for the selected cipher suite;
6. The **server hello done message** is sent by the server to indicate the end of the server hello and associated messages. After sending this message the server will wait for a client response;
7. The **Client certificate** is only sent if the server requests a certificate. **Client key exchange message** will immediately follow the client certificate message, if it is sent. Otherwise it will be the first message sent by the client after it receives the server hello done message;

8. The **Certificate verify** message is used to provide explicit verification of a client certificate;

9. A **finished message** is always sent immediately after a change cipher spec message to verify that the key exchange and authentication processes were successful. It is essential that a change cipher spec message be received between the other handshake messages and the Finished message.

2.2 Formal methods

Formal methods [82] are rigorous, mathematically-based techniques used to analyze, design and implement computer-based systems (hardware, software, complex ICT systems). Their mathematical foundation stands in the mathematical models used to represent the systems to analyze, design, or implement. Such models use mathematical concepts, such as sets, functions, relations, etc., to represent system structure or behavior, finally providing unambiguous and precise representations. Formal methods are generally contrasted with informal methods, i.e., the ones that do not employ mathematical models at all, relying on empirical approaches for design, implementation, and verification. Examples of informal methods are the development techniques based on expressing security assumptions, requirements, and algorithms as a mix of English text and time diagrams showing the most typical behavioural scenarios, and that include, as verification techniques, tests manually derived from the requirements, and manual reviews. There are also intermediate methods, classified as semi-formal, that combine rigorous mathematical models with informal ones. For example, UML [66] is a semi-formal modelling language because it includes parts that are formal (e.g., state charts), and parts that are not (e.g., use case diagrams).

One of the main reasons for using formal methods is that they enable mathematical proofs of the correctness of the employed system models, thus finally providing high correctness assurance for the systems that are designed and implemented based on them. Another reason is to provide unambiguous specifications of systems or their requirements in the various stages of development. In fact, an informal model, not having a mathematical model behind, may be subject to slightly different interpretations. Having an unambiguous reference point is particularly important when different parts of the system are developed by different persons or when different stages of the development are performed by different developers. Accordingly, formal methods include formal specification techniques, i.e., mathematically-based languages or notations that can be used to describe systems or their properties unambiguously through formal models, and formal verification techniques, i.e., algorithms that can be used to formally prove the correctness of such models and of the development process.

Developing a system by means of formal methods usually consists of developing several formal specifications. For example, the starting point can be a high-level formal requirements specification, i.e., an unambiguous specification of the requirements of the system to be implemented. From this specification, a high-level system specification can be derived, which provides a high-level description of how the system is organised and works. Then, this high-level description can be refined into other, more

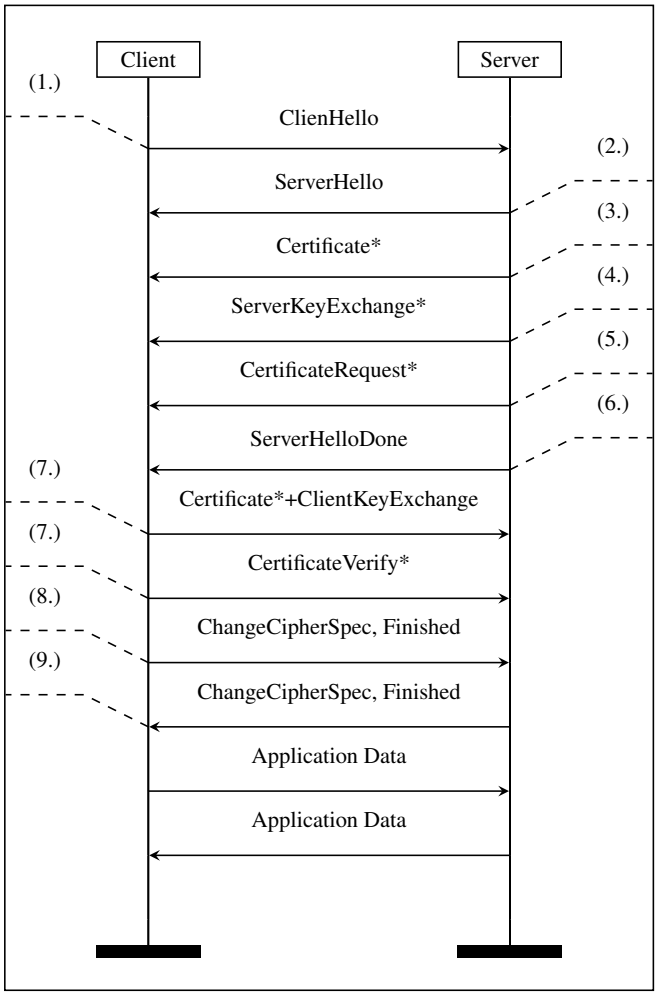

Figure 2.3: Message flow for a full TLS 1.0 handshake

detailed, formal descriptions of the system, its components, and their properties. The refinement process continues until the formal models finally obtained are so detailed that an implementation can be directly derived from them.

In such a development process, formal verification can be used to check the correctness of the models that are developed and of the process itself. This is done by analysing each formal specification, in order to check its internal consistency, i.e., absence of contradictions in the model, as well as by comparing different formal specifications in order to check that they are correctly related. For example, the high-level system specification must satisfy the high-level formal requirements expressed by the formal requirements specification. Checking that this is true is an example of formal verification. If both specifications are formal, i.e., formulated in unambiguous mathematical notation, it is even possible to obtain a formal proof of this fact. Another example is when we have two different formal models of the system, at different detail levels. In this case, formal verification can be used to check that the more detailed model is coherent with the less detailed one, i.e., it does not diverge from the corresponding higher-level description.

Formal models are key references not only for formal verification, but also for other development activities. For example, a formal requirements specification can be used as a reference for building a set of tests that aim to check that the system implementation fulfils the system requirements (requirements-based testing). Or, a detailed formal specification of a system module can be used as a reference for developing the implementation of that model. Usually, all these activities are performed automatically or semi-automatically, in order to avoid human errors and to further increase the confidence that the development process is correct, i.e., that the final implementation satisfies the system requirements formulated initially.

2.3 Formal methods for cryptographic protocols

Formal methods are particularly important for protocols, for example as a way to enable protocol interoperability, i.e., the possibility for different implementations of the same protocol, developed by different software providers, to work together and to achieve the protocol goals. When designing a protocol, a set of formal requirements is formulated, expressing the goals the protocol is required to achieve. Then, a formal protocol specification is formulated, describing the protocol features (messages and procedures) unambiguously. The detail level of this description is chosen so that only the constraints that are strictly necessary for achieving interoperability and the desired goals are expressed, while leaving all the other details free to be chosen by the developers of implementations. Formal protocol specifications can be used as the reference protocol descriptions, e.g., when issuing the protocol standard (e.g., the EGP standard [1] uses a state machine based formal specification). At this stage, protocol designers can also perform a formal verification in order to prove that the protocol specification is enough to guarantee interoperability of implementations and the fulfilment of the desired protocol goals. Of course, a protocol implementation must be coherent with the protocol specification, otherwise, if it deviates from what is prescribed by the protocol, interoperability with other implementations may no longer hold, and the protocol goals

may not be reached. In order to limit this kind of problem, an implementer can use automatic code generation tools that perform the refinement from formal specification to implementation automatically or semi-automatically [76, 78].

Formal methods represent an important opportunity especially for the design and implementation of cryptographic protocols. In this section, the main motivations why the application of formal methods can improve cryptographic protocol engineering are first illustrated. Then, the effective ways how formal methods can be leveraged for cryptographic protocols are discussed.

2.3.1 Motivation

The specification, design and implementation of a cryptographic protocol are complex tasks, difficult to manage by human beings without the support of automated or semi-automated tools. Two main reasons motivate why cryptographic protocol engineering is characterized by such a high complexity. The first reason is common to all distributed systems: it is the intrinsic parallel and asynchronous nature of cryptographic protocols. The second one is more specific to cryptographic protocols: it is the large set of possible different behaviors of the attackers.

Concerning the first motivation, cryptographic protocols have been defined as communication protocols that ensure security properties among multiple parties. The concurrent nature of cryptographic protocols is explicitly implied from this definition. Multiple sessions of the same protocol might be active in parallel, one/more for each participant to the communication, and these sessions may share data such as cryptographic keys and other material. For this reason, attackers may exploit weaknesses by playing with two or more parallel sessions, by injecting data obtained from one session into another session. An example of attack that exploits this kind of bad interaction among parallel processes of the same cryptographic protocol is the so-called oracle attack. In this attack, the attacker would use one of the participants to the communications to generate some information that the attacker would not be able to get by itself. The attacker could then create a new message, based on this information, and send it to a different parallel process of the same protocol.

Therefore, the security of a protocol must be analyzed by considering what happens when an arbitrary number of sessions are executed in parallel. It is well known that this kind of analysis is difficult because the number of possible interleavings of actions from multiple sessions blows up rapidly with the number of parallel processes involved. If the implementation of a protocol is tested in a traditional way without the support of formal verification, some vulnerabilities that might manifest only when some messages are received in a specific order could be overlooked. Instead, formal methods guarantee that all the possible cases are analyzed. Such an exhaustive analysis has thus become a fundamental component for communication protocol engineering, and it is even more important for cryptographic protocols, since security is directly involved.

The unconstrained behavior of an intruder is the second motivation for leveraging formal methods in this field. Traditional testing requires to predict and manually simulate scenarios where an attacker might try to exploit some possible vulnerabilities of a cryptographic protocol. However, a similar approach would never be able to consider a large number of cases, as each one should be specifically planned and tested.

Instead, in principle, only the computational complexity of what an attacker may do can reasonably be constrained. All behaviors that are computationally feasible should be considered, including those that do not follow the protocol rules, and the different number of such behaviors is huge compared to what a test could exercise. Formal methods can consider all possible cases, or a large number of them, rather than just some of them, so having more possibility to find the corner cases that an attacker could exploit.

2.3.2 Role of formal methods in cryptographic protocol engineering

Formal methods can contribute to cryptographic protocol engineering in four main ways: protocol specification, protocol verification, protocol implementation verification, and protocol monitoring. All these contributions apply to both standard protocols (i.e, protocols that are developed when a new standard has to be established) and to custom protocols.

Protocol specification

As already mentioned, a formal protocol specification can provide the necessary unambiguous, complete, and consistent definition of the protocol rules, which is essential in order to avoid possible misunderstandings of the specifications by different implementers and consequent failure of guaranteeing the expected security properties. A formal protocol specification can be expressed by means of any language that has a formal, i.e. mathematically-based, syntax and semantics. Typical examples are state machines, coupled with language-independent data representations, such as for example the language accepted by the NRL protocol analyzer [60], or other formalisms based on process algebras, such as for example the spi calculus [3] and the applied pi calculus [22]. Protocol specifications can be expressed at different abstraction levels. For example, one in which messages are expressed as abstract data types and one in which messages are expressed as bit strings. More information about such different types of models is provided in section 2.4

In addition to specifying the protocol rules, including the behavioral rules of each protocol role and the rules for building the exchanged data representations, formal protocol specifications provide an unambiguous and precise description of the security properties that the protocol is expected to enforce, i.e. the protocol requirements. While, as discussed, most of the properties belong to some standard classes, in order to leverage formal methods it is not enough to just define the classes of properties enforced by the protocol, but it is necessary to give all the details of such properties, so that they cannot be misinterpreted. For example, a single authentication property generally involves not only ensuring that when an actor A successfully concludes a session of the protocol with another actor, A knows the identity of the other actor, but also that the two actors agree on some data, which usually must remain secret. All this has to be expressed by means of mathematical notation. For example, the Proverif protocol verification tool [22] takes, as its input, a script that includes a protocol description written in the applied pi calculus formal language, and security properties expressed in a tool-specific formal language. Moreover, each security property may hold or not under

certain environmental conditions and assumptions about what the attacker is supposed to be able to do, which also has to be specified in order to have a fully formal specification. The Proverif script, for example, also includes statements specifying what the intruder is initially assumed to know.

The starting point for developing formal protocol (and requirements) specifications is generally an informal idea, which itself is not yet suitable to the application of formal methods. This idea gets formalized and encoded in the selected formal language. Of course, formalization is a human activity, which is subject to errors. For this reason, formal protocol specification is tightly coupled with formal protocol verification, which is necessary to provide adequate correctness assurance.

Protocol verification

Having formally defined the protocol security requirements and the protocol rules, possibly at different detail levels, formal protocol verification checks that these specifications are internally consistent and coherent with one another. For example, one can verify that each specification follows the syntax and semantic rules of its specification language (internal consistency), and that if each honest actor of the protocol follows the specified rules of the protocol, the specified security properties hold under certain assumptions about what the attacker can do (coherence between requirements specification and protocol specification). Another form of coherence verification is the verification that a more abstract protocol specification is consistent with the corresponding low-level specification.

A noteworthy observation must be made for providing a correct interpretation of what verification represents exactly. In formal methods, the terms "validation" and "verification" have different meanings. On one side, validation is the operation through which it is checked if a model fulfills user requirements and it is never formal, since user requirements are originally informal and subjective. On the other side, verification checks the internal consistency of the model and compares the model with others at higher level; it can be a formal process, if the involved models are formal as well. To this end, in protocol verification the models are always compared to other models or the formal specification of the requirements, while the subjective and informal requirements are not considered anymore.

Under this important assumption, protocol verification at this stage (i.e., design stage, when abstract models are defined early before the actual real-world implementation) can be performed with different techniques. Some of the most common techniques used for formal verification of cryptographic protocols are model checking [74], theorem proving [69], type systems [52, 44] and game theory [15]. Further information about these techniques will be provided in Section 2.5. The advantage provided by performing formal verification at this stage in cryptographic protocol engineering is that the internal model consistency and the fulfillment of the security requirements are formally verified before the development of the real-world implementation, thus saving time and human resources.

Protocol implementation verification

As it has been explained for protocol verification, the existence of intermediate levels of abstraction between the formal representation of the requirements and the concrete implementation allows to notice errors in the protocol formulation before developers spend time on writing the source code. However, protocol verification at the model level is not sufficient to have the guarantee of protocol correctness. The reason is that, when the developers write the implementation, they might introduce programming errors and bugs (e.g., errors due to the differences between high-level modeling languages and the typical low-level programming languages), and as a result the expected protocol behavior is altered, leading to protocol vulnerabilities.

Protocol implementation verification aims to fill the gap between protocol design and implementation by avoiding inconsistencies between them. To this end, automated or semi-automated tools are exploited. According to the categorization proposed by Avalle et al. in [8], the two main techniques investigated in the literature are automatic code generation and automatic model extraction. In the former technique, which follows the well-known software engineering paradigm of model-based development, an abstract formal model is developed first, by hand, and the real-world implementation is automatically derived from it on the basis of some user-specified implementation choices. In the latter, the protocol implementation code is written first, by hand, even without necessarily having a formal model yet, and a formal model is derived automatically from the implementation code, enriched with some annotations that guide the model extractor in understanding the protocol rules that are implemented. In both cases, there is a formal model which can be used to perform protocol verification, and to detect protocol flaws. In both cases, the typical human mistakes made in the implementation phase are avoided, because the mapping between abstract model (which is formally verified) and implementation is done automatically by a tool. Additional information about protocol implementation verification is provided in Section 2.6.

Protocol monitoring

Formal methods can be used to perform *runtime verification*, i.e. to monitor the execution of a process implementing a cryptographic protocol. With respect to protocol model or implementation verification, runtime verification has access to practical and more detailed information on the runtime behavior of the process.

As for traditional protocol formal verification, the security properties and the protocol rules must be formally expressed. The specification languages that are used for runtime verification in protocol monitoring generally allow for the possibility to express a temporal evolution of events (i.e., a security property might be guaranteed after a certain sequence of events happened in the communication among the participants). The most common family of specification languages used for this purpose is temporal logic: examples are linear temporal logic , interval temporal logic, and signal temporal logic. The survey by Cassar et al. [28] details how these three types of temporal logic can be used in the context of protocol monitoring.

The typical protocol monitoring setup is made by three components: the process running the protocol, the monitor and the instrumentation mechanism. The monitor

is the process that observes the protocol execution and can establish if the security properties formally expressed are satisfied, after its behavior has been observed for a sufficient interval of time. The instrumentation mechanism, instead, records the information related to the behavior of the process running the protocol, decides which pieces of information should be made visible for the monitoring activity, and sends them to the monitor in the form of an ordered stream called "execution trace". The monitor retrieves this ordered stream of events and applies formal verification techniques.

Another possibility is to derive automatically the monitoring specifications from the same kind of protocol specification used as a basis for protocol implementation.

2.4 Types of formal models used for cryptographic protocols

This section focuses on how cryptographic protocols are modeled formally, and on the different classes of models that are used.

The formal models used for cryptographic protocol engineering are characterized by different abstraction levels. The more the model is abstract, the more it is simple to analyze and easy to manage. However, at the same time, the more a model is abstract the less number of possible attacks on the protocol it can take into account. For this reason, different models at different abstraction levels are generally used in different phases of a cryptographic protocol design and implementation lifecycle.

There exist mainly three kinds of models that have been successfully applied for rigorously analysing and implementing cryptographic protocols: belief models, symbolic models, and computational models. Belief models are the most abstract ones. They are suited especially for analyzing the logic of a protocol in terms of the beliefs that the principals can legitimately achieve through the protocol. Symbolic models represent the actual behavior of a cryptographic protocol in terms of states and actions of the actors, using abstractions that preserve the most important logical flaws that may affect the protocol. Finally, computational models are even more precise, being able to represent a larger class of attacks on cryptographic protocols, but at the expense of higher verification complexity. In the remainder of this section, the different flavors of such classes of models will be detailed.

2.4.1 Belief models

Belief logics are logic systems that have been useful in the design and analysis of cryptographic protocols, starting with the pioneering work done by Burrows, Abadi, and Needham on BAN logic [25], and continuing with several related logics: GNY [43], AT [5], SVO [80]. More details about these different logic systems can be found in the survey by Syverson and Cervesato [79]. A belief logic is used to determine how the legitimate beliefs of agents in a protocol run evolve when the protocol events occur. For example, in BAN-like logics, statements like the following ones can be used:

- A said M (protocol agent A sent a message containing M in a protocol run);

- A sees M (protocol agent A received a message containing M in a protocol run and A could get M, e.g., by decrypting the message with a key known by A, and use M);
- A believes P (protocol agent A has right to believe that statement P is true);
- N is fresh (N has never been sent in any previous message of the protocol);
- $\stackrel{K}{\rightarrow} A$ (K is A's public key);
- $A \stackrel{K}{\leftrightarrow} B$ (K is a shared key known only by A and B).

A logic of beliefs includes axioms and inference rules that describe how the beliefs of agents may evolve, i.e. how each agent can deduce new legitimate beliefs. For example, if $\{M\}_{K^{-1}}$ represents the encryption of M with the private key corresponding to public key K, the following inference rule

$$\frac{A \text{ believes } \stackrel{K}{\rightarrow} B,\ A \text{ sees } \{M\}_{K^{-1}}}{A \text{ believes } B \text{ said } M} \tag{2.1}$$

specifies that if agent A has right to believe that K is B's public key and A sees $\{M\}_{K^{-1}}$, then A is also entitled to believe that B said M (because, as K is B's public key, B is the only agent that could encrypt M with the private key corresponding to K).

The logic can be used to determine the set of legitimate beliefs that each agent can deduce after a run of the protocol, starting from an initial set of assumed beliefs. For example, if we assume that agent A knows B's public key K, we can assume the initial belief "A believes $\stackrel{K}{\rightarrow} B$". Then, if the protocol prescribes that B sends $\{M\}_{K^{-1}}$ to A, after A actually receives this message, we can conclude that "A believes B said M", because of inference rule (2.1).

Indeed, belief logics usually do not model protocol messages but rather the *meaning* conveyed by such messages, considering only the encryoted ones (cleartext messages can be forged or modified by the attacker hence they do not contribute to making legitimate beliefs). More precisely, a protocol is modeled by what is called the idealized view of the protocol, which includes the meaning conveyed by each protocol message, and the encryption key, rather than the message itself. For example, this is the idealized view of the NSPK protocol presented in [25]:

$$\begin{aligned} A \rightarrow B &: \{N_A\}_{K_B} \\ B \rightarrow A &: \{\langle A \stackrel{N_B}{\rightleftharpoons} B\rangle_{N_A}\}_{K_A} \\ A \rightarrow B &: \{\langle A \stackrel{N_A}{\rightleftharpoons} B\rangle_{N_B}\}_{K_B} \end{aligned}$$

where $\langle X\rangle_{N_A}$ is a statement that combines another statement X with N_A, which is a secret that is used to prove the identity of whoever utters $\langle X\rangle_{N_A}$, while $A \stackrel{Y}{\rightleftharpoons} B$ means that Y is a secret known only to A and B. In this view, the meaning of the first message is that A communicates N_A to B, while the meaning of the second protocol message, which is $\{N_A, N_B\}_{K_A}$, is that B communicates to A that N_B is a secret known only to

them and N_A proves B's identity to A. The last message has a similar meaning, but with inverted roles.

The final beliefs can be compared with the expected ones in order to check whether security properties are satisfied or not. For example, if the aim of the protocol is to agree about a shared key, at the end we would like to establish the beliefs "A believes $A \overset{K}{\leftrightarrow} B$" and "$B$ believes $A \overset{K}{\leftrightarrow} B$", but also "$A$ believes B believes $A \overset{K}{\leftrightarrow} B$" and "$B$ believes A believes $A \overset{K}{\leftrightarrow} B$".

One of the advantages of this model is that belief logics are fairly simple and decidable. Hence, when a protocol has been properly formalized, if its logic is correct, it is possible to automatically prove the expected beliefs, and the resulting proofs are usually short and simple. However, when a proof for a protocol cannot be found, there is no evidence that an attack on the protocol is possible. Moreover, aspects such as how messages are built and how they are processed when received, and which real cryptosystems are used are not modeled. Hence, a protocol proved correct may still have weaknesses in the unmodeled aspects. For this reason, belief logics can be used just to get an initial assessment of a protocol, in order to validate its logic before continuing with more sophisticated analyses.

Another issue with belief logics is that subtle errors may be introduced in the formalization of the initial beliefs and of the protocol itself, which is difficult to detect and may lead to wrong conclusions about the protocol correctness. For example, the NSPK protocol was proved correct using the BAN logic in 1990 [25]. However, later on, after Lowe found a man-in-the-middle attack on the same protocol [57], Alves-foss showed in [7] that this attack was not detected when developing the above cited BAN logic proof because of a wrong formalization of the protocol used in that proof. In the same paper, he also showed that the protocol properties could be proved by the BAN logic under the fixed formalization. Similar errors when applying BAN logic are documented in [23].

2.4.2 Symbolic models

The foundation for the symbolic analysis of security protocols was laid by Dolev and Yao [36]. They introduced a model in which cryptographic primitives and other operations on data are represented as algebraic function symbols that are assumed to satisfy certain properties, and the adversary has full control over the network but is constrained to compute only through the defined function symbols.

In this abstraction, the protocol messages are algebraic terms built from the defined function symbols. This model assumes perfect cryptography, which means making a number of assumptions about the properties that cyptographic operations have. For example, the only way to decrypt encrypted data is to know the corresponding decryption key; encrypted data do not reveal the key that has been used to encrypt them; encrypted data have enough redundancy so that decryption can detect whether a wrong key has been used to attempt decryption; there is no way to compute the inverse of a cryptographic hash function and different messages are hashed to different values.

For instance, by following this approach, shared-key encryption is basically modeled by two function symbols, *enc* and *dec*, where $enc(x,y)$ represents the encryption

of x under key y, $dec(x,y)$ represents the decryption of x with key y, and the following equality holds:

$$dec(enc(x,y),y) = x \tag{2.2}$$

As this is the only equality assumed about these functions, one can decrypt $enc(x,y)$ only when one has the key y, according to the perfect cryptography assumption.

The attacker's capabilities include the possibility to eavesdrop messages transmitted on public channels, and use any cryptographic function on the eavesdropped data. In addition, an attacker can forge new messages and send them (i.e. insert) on public channels. The attacker can also participate in the protocol as any other legitimate actor and it can have its own keys.

The model can represent the behavior of an arbitrary number of parallel protocol sessions involving legitimate actors and the attacker. Generally, the attacker is considered as the communication medium, to emphasise that it has complete control on the communications (see Figure 2.4). Each protocol role is modeled by specifying its operational behavior in the form of a state transition model where not only message send and receive operations are represented, but also how received messages are manipulated and checked.

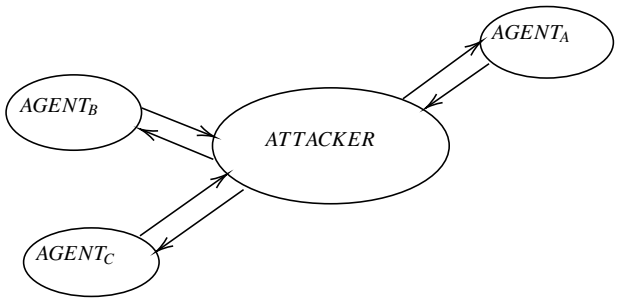

Figure 2.4: Representation of a Dolev-Yao model with the attacker as the medium

For example, the A role of the NSPK protocol in one protocol session can be modeled by the state-transition model represented in Figure 2.5. Then, the state-transition models of the protocol roles in multiple sessions and of a possible attacker are combined together to form the overall state transition model. When building this model, some assumptions are made about what the attacker initially knows. Then, during the execution of the protocol, the attacker may get additional knowledge from the eavesdropped messages. The overall state transition model is unbounded, because of the unbounded number of parallel sessions that may be started and the unbounded number of ways the attacker has to manipulate messages (e.g., an attacker who knows m and k may decide to send m encrypted with k any number of times, i.e., $enc(m,k)$, or $enc(enc(m,k),k)$, etc., and all these messages are considered as different according to the symbolic model).

In this kind of model, there are two different main ways of formalizing security properties. The first way is to express security properties as trace properties. A trace

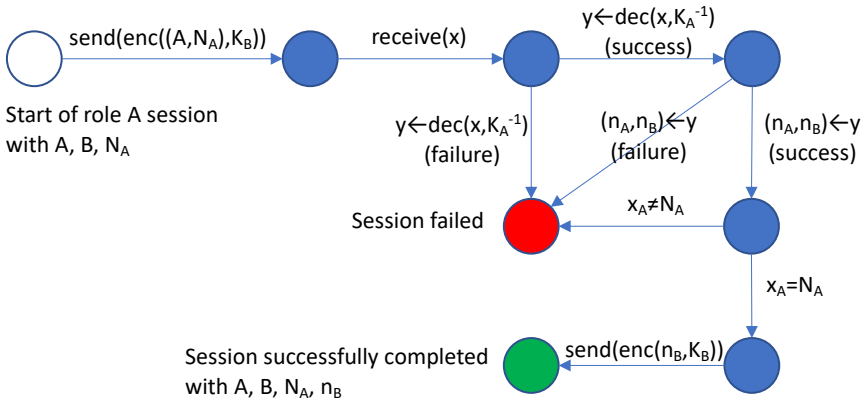

Figure 2.5: Symbolic model of *A*'s role in the NSPK model

is the record of a specific run of the protocol, i.e., a path in the overall state-transition model graph. For each trace, a given security property is true or false, and the property is considered to hold for the protocol if it holds for all its traces. For example, a way to formalize the secrecy of a given N is to require that N is never known by the attacker. If there is a trace which arrives at a state in which the attacker knows N, the property is false for that trace, and, hence, for the protocol, and the trace shows how an attacker can attack the protocol.

A second way to formalize security properties in symbolic models is to leverage some concept of equivalence, i.e. state the unobservability of some differences between different versions of the protocol. For example, the secrecy of N could be formalized by requiring that an attacker cannot distinguish a version of the protocol in which N is transmitted from a second version that only differs from the previous one because, instead of N, another value N' is sent. This second way of expressing secrecy is stronger than the previous one, based on trace properties: if a protocol satisfies the equivalence-based secrecy of N, then, under the same assumptions, it also satisfies the trace-based secrecy of N, but the reverse is not necessarily true.

Symbolic models are precise enough to detect a large number of protocol flaws, including for example the possibility of man-in-the-middle attacks, replay attacks, and oracle attacks. For example, a symbolic model was used by Lowe to find the man-in-the-middle attack on the NSPK protocol [57]. Symbolic models are suitable for automated verification using different techniques, as it will be detailed in section 2.5. However, their complexity (they are infinite-state) makes the usual security properties undecidable on them, as it was shown by Durgin et al. in [37]. Although this theoretical result implies the impossibility of building algorithms that can automatically decide whether an arbitrary input protocol model satisfies a given security property, it still admits the construction of automated procedures that can prove or disprove security properties on symbolic models in some cases. Indeed, as it will be discussed in section

2.5, the state-of-the-art verification techniques for symbolic cryptopgraphic protocol models can prove or disprove the usual security properties in most cases, making this approach the most used today.

Like belief models, symbolic models use abstractions that disregard some aspects of a real protocol, such as the features of the real cryptographic operations, and for this reason they do not consider possible weaknesses related to such aspects.

2.4.3 Computational models

Computational models were developed at the beginning of the 1980s by Goldwasser, Micali, Rivest [42, 41] and Yao [83], laying their foundations on the computability and computational complexity theories. Messages are modeled as bit strings, and cryptographic primitives as functions from bit strings to bit strings. An attacker is modeled as any probabilistic Turing machine having polynomially bounded resources to perform an attack. This is the same model generally used by cryptographers.

In this kind of models, the impossibility of some operations (e.g., the computation of the inverse of a cryptographic hash function), which characterizes perfect cryptography of symbolic models, is replaced by a probabilistic model, according to which the probability that such operations are completed successfully by a probabilistic polynomial-time Turing machine is negligible. This property usually is motivated by a widely accepted computational complexity assumption, e.g., that a certain mathematical function cannot be computed in polynomial time. Consequently, security proofs on computational models do not aim to prove the impossibility for an attacker to achieve a certain goal, rather to prove that the probability that an attacker achieves the goal is negligible. Such proofs generally exploit complexity-theoretic reductions, i.e., implications of the form: “if there exists an attacker that runs in polynomial time and has non-negligible success probability, then there exists an algorithm that solves a hard computational problem in polynomial time with non-negligible probability, which is assumed to be false”.

Computational models are less abstract and, hence, more realistic, than symbolic ones. For example, they can model cryptosystem-related weaknesses, in addition to the weaknesses also captured by symbolic models. However, proofs turn out to be more complex and difficult to automate. For this reason, proofs on computational models have been traditionally developed manually, until Laud made a first attempt to mechanize them in 2003 [54], followed by more general techniques developed in the next years [53, 18]. Another approach that has been developed to mechanize such proofs is to infer security properties in the computational model from corresponding properties automatically proved in the symbolic model. This kind of inference is possible under certain assumptions. The study of these assumptions started from the seminal work done by Abadi and Rogaway in [4], and continued with other studies that led to a number of approaches and tools to obtain security proofs in the computational model indirectly from corresponding proofs automatically built for the symbolic model. A survey by Cortier et al. [34] presents these techniques thouroghly. Another detailed account of the literature about direct and indirect techniques to automate security proofs for computational models has been provided by Blanchet in [20].

Although today there are automated theorem provers that can build such proofs

automatically, as computational models are more complex than symbolic ones, undecidability affects them as well, and such provers may not reach a result in some cases.

2.5 Formal verification of cryptographic protocols at design level

This section explains the main techniques used for formal verification of cryptographic protocols.

Formal verification is an imperative step in the design of cryptographic protocols and provides a rigid and thorough means of evaluating the correctness of cryptographic protocols. There exists a number of different formal verification techniques to verify cryptographic protocols. In this section, the main different approaches are presented: model checking, theorem proving, type systems, and game theory. The next chapter of this book will provide a more detailed analysis of formal verification techniques for cryptographic protocols.

2.5.1 Model Checking

Model checking approaches to the analysis of cryptographic protocols have proved remarkably successful [68]. Model checking [31] uses a state exploration technique, in which the state space of a state transition model is defined and then explored by the tool to determine if there are any paths through the space corresponding to violations of the intended properties of the modeled system. For instance, Fiterău-Broştean et al. [38] formalized several security and functional properties drawn from the SSH RFC specifications [55], and then they verified these properties by performing model checking on state-transition models extracted from different SSH implementations. In the context of cryptographic protocol verification, model checking is especially suitable for symbolic models, as they are based on state-transition semantic models. By using this technique, it is possible to look for the possible attacks on the modeled protocol [11, 74]. However, since the search space to be explored is infinite, because of the infinite different ways an attacker can combine eavesdropped data and because of the unbounded number of protocol sessions, a naive approach to model checking is not exhaustive and this kind of search alone cannot guarantee security when no attack is found. This problem has been addressed by research in the context of Dolev-Yao models, and a number of more sophisticated state exploration techniques have been developed that at least partially remove this limitation. Such techniques exploit the fact that an infinite number of different data or execution sequences are in fact equivalent. Exploiting this fact, under certain circumstances, by using these techniques, it is possible to achieve even a proof of security, but this is not possible in all cases, because, as already mentioned, the formal verification problem in the Dolev-Yao model has been shown to be undecidable.

The application of model checking to security protocols has been surveyed extensively by Basin et al. in [11]. For the formal verification of symbolic models, different model checking techniques have been used. Initially, explicit state space exploration by means of general purpose model checkers, such as FDR [57], murphi [62], and

spin [58], was proposed. These first attempts were limited to models with a bounded number of sessions and length of messages. Then, more sophisticated tools have been developed, capable of analyzing models without these restrictions. For example, OFMC [12] uses lazy, demand-driven search techniques to analyze even infinite-state symbolic models, while Tamarin [61] analyzes the state space implicitly, leveraging constraint solving based on satisfiability checkers. If the checker fails, Tamarin gives the user the possibility to try to complete the proof interactively.

2.5.2 Theorem proving

A totally different approach to formal verification is theorem proving, which consists of developing a security proof in a deductive system.

This approach is very obvious when using belief logics to model and verify protocols, because in that case the logic system is already defined by the belief logic and by the protocol specification rules. As belief logics are decidable, theorem proving can be automated and it always leads to a positive or negative result. For example, using BAN logic, and the formalization of the NSPK protocol in that logic, as presented by Burrows et al. in [25], a proof that the NSPK protocol satisfies some security properties can be constructed.

For what concerns symbolic models, formal verification by theorem proving is possible by building a formal system that represents the protocol rules and the attacker's capabilities in a way that is equivalent to (i.e., sound and complete w.r.t.) the symbolic model of the protocol. However in this case, apart from the difficulty of building the logic system, which can be solved by doing it automatically staring from other state-transition-oriented representations, the undecidability of the verification problem prevents theorem provers to always find a proof or disproof of a security property. Different solutions have been proposed to overcome this problem. One is to resort to manual or semi-automatic theorem provers. For example, Paulson shows in [69] how Isabelle, a general purpose higher-order logic (HOL) theorem prover can be used to verify security properties in cryptographic protocols. Another solution, adopted for example by Blanchet for Proverif [19], is to develop a proof search algorithm that is fully automatic but only in some cases it provides a result (proof or disproof), while in other cases it fails to return a result. In some cases (e.g., in the work by Havelund [46]), theorem proving and state exploration techniques have been combined together.

One valuable advantage of using theorem proving for symbolic models is that, if successful, it provides a formal proof of correctness. However, when the prover is not automatic, an extremely high expertise is necessary in order to find proofs, while fully automatic solutions exist but they sometimes fail to provide a result. Among the success stories of theorem proving, we can mention that Paulson succeeded in 1999 to prove some security properties of the TLS protocol using a symbolic model and his interactive proving method [70], while more recently, in 2017, some security properties have been proved on TLS 1.3 by Bhargavan et al. [14].

2.5.3 Type systems

A type system is a lightweight formal method originally developed for reasoning about programs. A type checker which exploits a type system is typically used within a compiler in order to prove the absence of some errors related to data types in the program.

In the early days of programming, type systems were used only to ensure certain basic correctness properties of programs such as the arguments to primitive arithmetic operations are always numbers and differentiating between a string and integer value in the memory. Later on, type systems have been exploited to perform also other types of verification, and not only for programming languages.

In the field of cryptographic protocol verification, the use of type systems has been explored, for example, by Gordon and Jeffrey [52, 44], who developed a series of type systems for verifying authenticity in security protocols, and by Cortier et al. [33], who addressed privacy properties (e.g. anonymity) by means of a similar technique. These approaches reduce the problem of verifying properties in security protocols to the type checking problem. Intuitively, a type system can be used to over-approximate the behavior of a protocol, in such a way that a positive check of its types can be used to prove some security properties.

A notable example of protocol verification performed by means of this technique is the verification of an interoperable implementation of TLS, written in F#, by means of the F7 type system [16]. In this work, TLS is implemented by a set of modules. Initially, these modules are just typed interfaces, with types that capture the security properties. Then, these modules are refined by adding their concrete implementations.

2.5.4 Game theory

Another approach, which has been used to find proofs automatically for computational protocol models, is the automation of game-based cryptographic proofs. A game-based proof is organized as a sequence (or tree) of games, sometimes also named experiments. Each game is characterized by a certain probability an attacker has to win the game. On one end of a chain of games we have a game that describes the competition between a protocol to be verified and a computationally bounded attacker which tries to break it. On the other end of the chain we have another game, for which the probability the attacker has to win is known (e.g. it is known it is negligible). The games in the chain are such that a transition from one game to the next one is characterized by a particular relation between the win probabilities of the two games. For example, the probability an attacker has to win the second game is less than the probability it has to win the first one. Or, the difference of win probability between consecutive games is negligible. If one such chain is found, it is a proof, based on reductions.

Even though cryptographic proofs based on game reductions are powerful, they are complex, and can easily become involved and difficult to manage manually. For this reason, their automation has been addressed by research, and tools that implement it are available. Among them, we can mention cryptoverif [18, 21] and easycrypt [10].

Among the notable results obtained with this approach, we can mention the verification of the TLS Handshake protocol [15, 14].

2.6 Techniques for linking design models to implementations

Belief, symbolic, and computational models are characterized by a higher level of abstraction than the corresponding real-world protocol implementations. Therefore, this existing gap between abstract formal models and their implementations might have a negative impact on the security level that the usage of formal methods should guarantee for cryptographic protocols.

Among the multiple potential problems derived from this gap, a first issue is that the complexity of the control flow and data types for a real-world protocol implementation is typically much higher than the complexity of the techniques that are used for the definition of abstract models at design level. As such, logical and coding errors might be introduced in the implementation of the protocol; if those errors are not identified in the testing phase before release, they might create a discrepancy between the model and the effective behavior of the protocol. Another problem is that programming languages that can be used for the real implementation deeply differ from each other, e.g., in terms of data structures and operations to manage them. These low-level mechanisms do not find a direct correspondence to the high-level abstract models defined with formal methods. This missing link might be another cause of errors that would not be easily identified.

OpenSSL and OpenSSH are real-life examples where the implementation step introduced errors on a protocol that has been proven secure when the abstract model has been previous analyzed. They are open-source implementations of two widespread cryptographic protocols, respectively SSL (i.e., the name that TLS originally had when developed by Netscape Communications) and SSH. The original protocols have been widely studied and formally verified in literature (e.g., Morissey et al. succeeded in providing a modular and generic proof of security for the application keys established through the TLS handshake protocol [63]). However, their implementations receive several security patches per year, due to low-level implementation bugs that could not be envisioned when designing the models. For example, a notorious OpenSSL bug that has been exploited by man-in-the-middle attacks is the one described in [67]. Several functions inside OpenSSL incorrectly checked the result after calling the EVP_VerifyFinal function, allowing a malformed signature to be treated as a good one rather than as an error. This fault cannot be found by applying verification techniques at design level, because in the semantics of the model the way the code handles return values is evidently overlooked. Instead, a user enumeration vulnerability flaw was found in OpenSSH, through version 7.7 [35]. The vulnerability occurs because the bailout for an invalid authenticated user is not delayed until after the packet containing the request has been fully parsed.

These considerations clarify that the gap between language-agnostic abstract models and real-world protocol implementations with a specific programming language cannot be easily filled through a manual translation of the model into the software program. For this reason, automated techniques have been developed, so that the gap between them is reduced, and the possibility to have flaws introduced in the implementation phase is also reduced. According to the categorization proposed by Avalle et

al. in [8], these techniques are mainly classified into two families (i.e., automatic code generation and model extraction), depending on the starting point for the automated process: abstract model or code implementation. A more detailed presentation of the research in this field can be found in the survey written by Avalle et al. in [9].

2.6.1 Automatic code generation

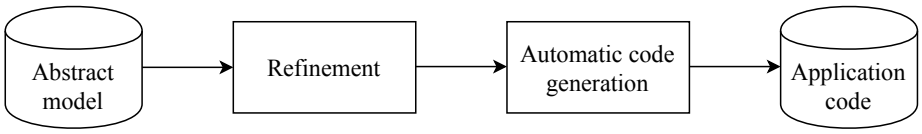

Figure 2.6: Workflow of automatic code generation

Figure 2.6 depicts the general workflow that is followed by automatic code generation to link the abstraction of formal models and the specificity of a real software implementation. As the workflow suggests, automatic code generation requires the definition of an abstract model as first step, and later the code of the corresponding implementation is produced by an automated tool.

In more detail, starting from an informal description of the cryptographic protocol (i.e., a description of the expected behavior and the security properties that the protocol must fulfill), the user designs a formal, abstract model (e.g., belief, symbolic or computational model) which respects the description, and the model is formally verified. Nevertheless, before a software implementation is generated by employing an automatic tool, the abstract model must go though a refinement operation. More specifically, it is necessary to enrich the model with the implementation choices that will drive the generation of the code. The reason is that a design choice might correspond to multiple alternative implementation choices, among which the programmer typically chooses the most suitable one according to the protocol requirements. These implementation choices must be thus provided to the tool, so that they can be followed in the code generation. At that point, the abstract model enriched by these indications is enough for the employed tool to automatically generate the protocol implementation. An example that follows this procedure is the technique described by Cadé et Blanchet in [26], where the source code is automatically derived from computational models.

Approaches based on automatic code generation must be characterized by two essential features. The first one is that the function that is employed in the automatic code generation for the mapping of the abstract model to the concrete implementation must preserve the abstract security properties of the model. In fact, joining the formal proof of soundness for the code generation function together with the proof of correctness for the security properties of the abstract model (e.g., through other formal verification techniques), then the correctness of the security properties is consequently proved on the concrete implementation. The second feature, instead, is the variety and nature of implementation choices that the user of the automated tool can express. The optimal scenario would be the possibility to express a high range of implementation choices, to guarantee the interoperability of the implementation generated by the tool with the

other third party existing implementations. However, there might be cases where some restrictions are placed so that certain aspects of the output software are bounded to choices taken by the developer of the tool itself.

The automation provided by code generation brings over many advantages with respect to naive approaches based on a manual programming of the concrete implementation. Firstly, a central problem related to the gap between abstract model and real-world implementation, i.e., the introduction of programming errors and bugs in the cryptographic protocol, is solved by construction. With the formal proof of the soundness for the code generation function, in fact, the behavior that is shown by the corresponding implementation is proved to be the same as expected, and no security vulnerabilities due to the typical manual coding errors are introduced. Secondly, it is possible to avoid information leakage through side channels by construction as well. Finally, another benefit is that the human user does not have to know the low-level artifacts of a programming language, since they are automatically managed by the tool.

At the same time, some drawbacks are present in this approach. The most evident drawback is that the user is required to know the abstract modeling language that is used for the formalization of the model. If on one side the user must deal with a language at a higher level, on the other side reaching a good level of expertise on a modeling language might be not immediate. Moreover, since the code is automatically generated, all the optimizations an expert developer might introduce are absent, and the efficiency of the resulting software might be not the most suitable one for the target application.

2.6.2 Model extraction

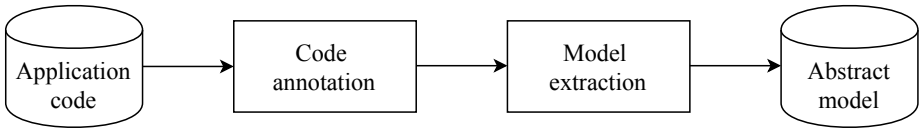

Figure 2.7: Workflow of model extraction

Figure 2.7 depicts the general workflow that is followed by model extraction. The starting and ending points of this process are the opposite of those that are present in the approaches based on automatic code generation.

First of all, software developers write the application code, i.e., they directly write the real-world implementation of the cryptographic protocol. The source code is later enriched with annotations, whose purpose is to provide additional specifications about formalisms that cannot be conveyed through the constructs of traditional programming languages. Then, an automated tool performs the extraction of an abstract model from this annotated source code, discarding the elements of the source code that are not relevant for the security properties that must be guaranteed by means of the cryptographic protocol. The extracted model is finally verified with a formal prover, so that the formal correctness of the model is guaranteed. As an example, a technique that follows this workflow is described by Aizatulin et al. in [6], where the abstract model is extracted from a source code written in C language.

A characteristic feature of model extraction is the degree of coverage for the programming language that is used to write the source code of the application. The degree of coverage represents the percentage of programming language features that can be used to write the application and that are handled by the automated tool for the extraction of the abstract model. Concerning this aspect, traditional programming languages such as C, C++, C#, Java, or F# are characterized by a syntax with a high complexity, so it is impossible that all the elements of their syntax are covered by techniques based on model extraction. To this end, for each programming language a subset of features has been identified as the features that are properly handled by the majority of this kind of tools.

The most evident benefit of model extraction is that the starting point of the process is the application code written by a human being. As such, an expert developer can introduce manifold optimizations which can improve the efficiency of the protocol implementation when employed in the real world; this was evidently not feasible for the automatic code generation, where at most a limited number of choices could be specified to drive the generation of the code. Moreover, this technique hides the difficulty of learning and managing abstraction languages to the users. Even though programming languages such as C++ and Java themselves have a rich and complex syntax, the developers can limit themselves to learn these languages, because they can consider the formalization of the abstract model as a black box represented by the tool employed for model extraction. Additionally, the tools which are available for finding errors in the definition of a high-level model are much less advanced than the debuggers for common programming languages. In light of this gap, the possibility to use state-of-the-art analyzers of the source code of applications grants the developer a more user-friendly and complete control on the product they are developing.

Nevertheless, model extraction is not exempted from some disadvantages. From a theoretical point of view, this technique might be applied on legacy protocol implementations to get the corresponding abstract model. However, this possibility usually collides with the language features that are used for the legacy application and the restricted subset that is allowed for the effective employment of techniques based on model extraction. In fact, it cannot be expected that legacy code was originally thought for this purpose. Therefore, a manual intervention (e.g., elimination of constructs that are not accepted, or introduction of annotations) is nearly always required in this context. Furthermore, the principles on which model extraction is based are the opposite of the traditional precepts of model-driven software engineering. In that field, the waterfall model, where the code is derived from high-level descriptive requirements through multiple stages of refinement, is the most common pattern that has been used for years in software development. A reverse approach thus puts a bigger emphasis and complexity on the coding activity, which is not guided by a recognized software engineering standard.

2.7 Frameworks for cryptographic protocol engineering

The development of a robust and strong cryptographic protocol is a burdensome challenge if it is completely managed in a manual way. The support provided by specialized frameworks for cryptographic protocol engineering is the most suitable resource to reduce the presence of flaws and vulnerabilities in the protocol implementations, which might be exploited by attackers to penetrate into a networked system.

In the following, two of the most important frameworks for cryptographic protocol engineering are presented. The first one is SPEAR II, proposed by Saul and Hutchison in [76] to enable a user-friendly interaction with multiple tools for formal analysis according to a so-called multidimensional approach. The second is instead JavaSPI, an environment proposed by Sisto et al. in [78] to allow the user to create a symbolic model of a cryptographic protocol by using a subset of Java features that correspond to the modeling language applied π-calculus [2].

2.7.1 Spear II

Multidimensional cryptographic protocol engineering is a unified approach where multiple dimensions (i.e., classes of techniques and tools, characterized by a common goal) are effectively combined in a single application, which provides a comprehensive way to specify, develop and formally analyze a protocol. The main reason that led to the birth of the multidimensional approach is that most of the available techniques for formal analysis cannot detect all the flaws and vulnerabilities that might be present in the design and implementation of a cryptographic protocol, if they are singularly applied. Their combination could simply overcome this problem, because each technique would compensate the drawbacks of the others, and at the end a protocol engineer would get a higher assurance of the protocol correctness.

An example of framework that has been developed according to the principles of multidimensional cryptographic protocol engineering is SPEAR II [76], acronym for "Security Protocol Engineering and Analysis Resources II", and evolution of the original SPEAR I proposed by Bekmann et al. in [13], which was just a proof-of-concept lacking the user-friendliness and completeness that characterize its successor. A human user can interact with the graphical elements of this framework, which offers a great variety of techniques based on formal methods for each stage of the development of a cryptographic protocol, i.e., from the specification of the security properties that it must fulfill until the development of the real-world implementation. In the following, the main dimensions, corresponding to the most important modules of SPEAR II, will be described, so as to show how all the different stages of protocol engineering are managed throughout a multidimensional approach with respect to the application of traditional formal techniques.

Cryptographic protocol design. A dimension that is traversal to all the other components of SPEAR II is the protocol design. The modeling phase is eased thanks to the GYPSIE environment, better described by Saul and Hutchison in [75], which offers a graphical user interface through which the protocol specification is performed at three

different levels of abstraction, so that the user can perform operations of varied complexity according to their expertise. More specifically, the high-level view describes the complete flow of messages among the participants of the protocol communication, and the user-friendliness is provided by graphical and textual languages such as SDL (Specification and Description Language) and MSC (Message Sequence Chart). The navigator view offers another high-level prospective of the message flow of the cryptographic protocol, throughout a tree view which offers a full integration with the high-level view, so that drag-and-drop operations between graphical elements of the two views are allowed. Then, the component view works at a lower level of abstraction, because it allows the formal specification of each message of the flow, with another representation based on a hierarchical tree. Finally, the output of the GYPSIE environment is a protocol specification as simple text, LaTeX or a Prolog-like language.

Code generation. The code generation module can automatically generate the Java source code of a real protocol implementation, starting from the output of the GYPSIE environment, i.e., the abstract model describing the message flow of the protocol. Even though such a technique does not present any novelty by itself, the integration of automatic code generation in a unified environment improves the quality of all the other stages of protocol engineering. For example, the additional specifications driving the generation of the source code might be specified with interactive menus in GYPSIE, and if they do not result to be the most suitable ones according to performance and bench-marking measurements, they can be easily modified.

Performance analysis. The performance evaluation of the cryptographic protocol is based on a message rounds calculator. The input of this module is the protocol specification, i.e., the message flow of the communication, which is produced by the GYPSIE environment. Starting from this flow, the message rounds calculator establishes which messages could be sent in parallel for improving the protocol efficiency. In SPEAR II, the performance analysis can be carried out both on synchronous message rounds, if a participant to the communication can send a message only if it has received all the previous ones that were targeting it, or asynchronous (also called optimal) message rounds, if the communication among the participants may be asynchronous. Overall, the results of this performance evaluation can become one of the specifications driving the development of the real-world implementation, because the source code would be directly written considering not only formal aspects of the protocol, but also its future performance. This feature represents an additional way to bridge the gap between the model and its implementation.

Security logics. In SPEAR II, the formal analysis of the protocol model specified through GYPSIE is performed with GYNGER, a Prolog-based analyzer proposed by Mathuria et al. in [59], based on the GNY belief logic, whose access is eased by the presence of a visual GNY environment. GYNGER exploits a forward-chaining approach for the application of GNY inference rules; as such, the GNY inference rules that cannot be adapted for forward-chaining are not implemented in this framework. In particular, 16 of the 88 GNY inference rules are not implemented because of this restriction. However, their absence does not impact the outcome of the formal analysis, because those rules are not needed to perform this type of analysis. Then, GYNGER receives as input the message flow of the protocol (i.e., the model), alongside with the GNY rule set, the initial assumptions and the target goals. If a target goal is success-

fully reached, a formal proof is generated by the tool, with the indication of which statements were used to derive that goal. This proof is expressed with an English-style language, to further increase the user-friendliness of GYNGER.

Other dimensions. Other dimensions are related to the simulation of use cases where the protocol might be effectively used. For example, the *Meta-Execution* dimension aids the user in correctly modeling the protocol by leveraging it in simulated scenarios, so that the user can directly understand possible problems in the model before the generation of the code. Instead, the *Attack Analysis* dimension is composed by an attack construction engine, whose purpose is to simulate cyber-attacks and help the user understand if the security properties that the protocol should guarantee are effectively guaranteed also under attack.

2.7.2 JavaSPI

JavaSPI [78] is a framework for cryptographic protocol engineering, based on a model-driven approach for the automatic generation of the protocol source code. Inheriting and extending the main characteristics of its predecessor Spi2Java developed by Pironti and Sisto [72], the purpose of JavaSPI is to simplify the modeling of a protocol, allowing the users to use the Java language for both the protocol design and implementation. This also enables the possibility to simulate the execution of the protocol by running a Java debugger on the model, before the development of the corresponding implementation. In the remainder of this section, the workflow of JavaSPI will be detailed, with the aim to underline the improvements it brings over for the traditional protocol engineering.

First, the specification of a protocol symbolic model is made by using a subset of the Java programming language, corresponding to the features that are necessary to reach the same expressiveness as applied π-calculus, a modeling language derived from the Dolev-Yao abstraction. In greater details, two subsets of the Java language are identified and incorporated in this framework. On one side, the "core language" includes the minimum number of Java constructs that are really needed for the mapping to π-calculus, and it is the language that is used by the internal modules of JavaSPI for the derivation of formal proofs or the automatic management of the model, i.e., it is not directly employed by a human user. On the other side, the "extended language" extends, as the name suggests, the core language by introducing additional Java features and it is the subset that is effectively exploited by the users for the specification of the protocol model. All the additional features that are included in the extended language can be reconduced to the subset present in the core language, though. Therefore, this redundancy (i.e., the existence of two separate subsets of the Java language) might be theoretically avoided, since the core language would be enough for the creation of any model with a correspondence in π-calculus. However, in that case the users would have a more limited language to use, and this would impact the user-friendliness of the framework, since experienced Java developers should excessively restrain their knowledge and limit their capabilities.

The formal verification of security properties for the model specified in the core or extended language of JavaSPI is performed by means of the automated and efficient theorem prover ProVerif [17]. Since this theorem prover works on an abstract model

formulated with the applied π-calculus language, a translation from the Java-based specification is needed, and it is performed by the Java-ProVerif module of JavaSPI. Nonetheless, the level of abstraction is the same for both the core language of JavaSPI and the applied π-calculus required by ProVerif. Therefore, the translation is straightforward and consists in a simple conversion of the language syntax. However, the input that is received by ProVerif is not exclusively made by the model expressed in π-calculus. In fact, the user should introduce annotations to the Java-based model, representing the security properties that the cryptographic protocol must enforce. In particular, the user can specify two different types of security properties by means of annotations: the first type is represented by secrecy properties for the data exchanged by the parties involved in the protocol communications, whereas the second type is called correspondence of events and includes properties such as party authentication. These annotations are then translated into queries for ProVerif, so that ProVerif can provide a formal proof of the corresponding properties if this proof can be reached, or it can disprove them by providing counterexamples.

JavaSPI also offers a semi-automatic generator for deriving the protocol source code from the abstract model. This module, called Java-Java generator because both the model and the real implementation are written in the Java language, is not fully automated, because a manual refinement of the abstract model must be performed by the user. More specifically, some annotations must be added to the code. These annotations have a different purpose with respect to those needed for the formal verification of security properties: in particular, they consist in the implementation choices that the Java-Java generator must take in the development of the application code. Within this approach, information specifically focused on the implementation and information exclusively related to the abstract model are separated. As such, during the specification of the model, the user can avoid to care about the future implementation, since additional details will be provided later on by means of annotations, and he can thus only focus on the definition of a correct protocol behavior. The separation of the two main tasks of protocol engineering, i.e., model design and protocol implementation, is made even more evident thanks to the fact that the developers of JavaSPI have proved a set of soundness theorems such that, if a security property is verified for the abstract model by employing ProVerif, then the same property is guaranteed to be fulfilled in the implementation derived with the Java-Java generator.

In light of all the analyzed features of JavaSPI, the conclusion is that this framework optimally counterbalances the typical drawbacks of approaches based on automatic code generation. Users are not required to know π-calculus, and all the implementation choices are separated from the model design because they are conveyed through annotations. Finally, in the whole workflow, the users have the possibility to find problems in the protocol specification in multiple ways before the generation of the source code (e.g., symbolically executing the Java model with a debugger, or running the ProVerif tool), so that protocol engineering becomes less troublesome than how it traditionally was.

2.8 Conclusions

Cryptographic protocols are widely used in many applications today. However, their design and implementation are notoriously very difficult to get right. In this chapter we have first outlined the main features of cryptographic protocols and analyzed the main reasons for this difficulty. Starting from these considerations, we have showed how formal methods can help cryptographic protocol designers and implementers to avoid errors as much as possible, by enabling accurate verifications based on mathematical models, and even proofs of correctness.

The typical engineering activities related to design and implementation of a cryptographic protocol have been dissected, together with the support tools based on formal methods that are available. Specifically, we have analyzed the typical lifecycle that starts from the elicitation and formalization of security requirements, followed by protocol design and finally implementation. We have also shown how each phase can be supported by verification tools that operate at different levels of abstraction, including the capability of linking descriptions at different levels. Finally, we have illustrated examples of frameworks that combine together different types of tools, operating at different abstraction levels.

Bibliography

[1] Exterior Gateway Protocol formal specification. RFC 904, April 1984.

[2] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. *SIGPLAN Not.*, 36(3):104–115, January 2001.

[3] Martin Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[4] Martin Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 20(3):395–395, 2007.

[5] Martín Abadi and Mark R. Tuttle. A semantics for a logic of authentication (extended abstract). In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '91, page 201–216, New York, NY, USA, 1991. Association for Computing Machinery.

[6] Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. Extracting and verifying cryptographic models from c protocol code by symbolic execution. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, page 331?340, New York, NY, USA, 2011. Association for Computing Machinery.

[7] Jim Alves-foss. The use of belief logics in the presence of causal consistency attacks. In *In Proceedings of the Nineteenth National Computer Security Conference*, pages 406–417, 1997.

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

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

[10] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. *EasyCrypt: A Tutorial*, pages 146–166. Springer International Publishing, Cham, 2014.

[11] David Basin, Cas Cremers, and Catherine Meadows. *Model Checking Security Protocols*, pages 727–762. Springer International Publishing, Cham, 2018.

[12] David Basin, Sebastian Mödersheim, and Luca Viganò. Ofmc: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, 2005.

[13] JP Bekmann, P De Goede, and ACM Hutchison. Spear: Security protocol engineering and analysis resources. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, pages 3–5, 1997.

[14] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *IEEE Symposium on Security and Privacy (S&P'17)*, pages 483–503, San Jose, CA, May 2017. IEEE. Distinguished paper award.

[15] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Santiago Zanella-Béguelin. Proving the tls handshake secure (as it is). In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014*, pages 235–255, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[16] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing tls with verified cryptographic security. In *2013 IEEE Symposium on Security and Privacy*, pages 445–459, 2013.

[17] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Proceedings of the 14th IEEE Workshop on Computer Security Foundations*, CSFW '01, page 82, USA, 2001. IEEE Computer Society.

[18] Bruno Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207, October–December 2008. Special issue IEEE Symposium on Security and Privacy 2006. Electronic version available at http://doi.ieeecomputersociety.org/10.1109/TDSC.2007.1005.

[19] Bruno Blanchet. Automatic verification of correspondences for security protocols. *J. Comput. Secur.*, 17(4):363–434, December 2009.

[20] Bruno Blanchet. Mechanizing game-based proofs of security protocols. In Tobias Nipkow, Olga Grumberg, and Benedikt Hauptmann, editors, *Software Safety and Security - Tools for Analysis and Verification*, volume 33 of *NATO Science for Peace and Security Series – D: Information and Communication Security*, pages 1–25. IOS Press, May 2012. Proceedings of the summer school MOD 2011.

[21] Bruno Blanchet. Mechanizing game-based proofs of security protocols. In Tobias Nipkow, Olga Grumberg, and Benedikt Hauptmann, editors, *Software Safety and Security - Tools for Analysis and Verification*, volume 33 of *NATO Science for Peace and Security Series – D: Information and Communication Security*, pages 1–25. IOS Press, May 2012. Proceedings of the summer school MOD 2011.

[22] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1–2):1–135, October 2016.

[23] Colin Boyd and Wenbo Mao. On a limitation of ban logic. In Tor Helleseth, editor, *Advances in Cryptology — EUROCRYPT '93*, pages 240–247, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[24] Jakub Breier, Dirmanto Jap, Xiaolu Hou, and Shivam Bhasin. On side channel vulnerabilities of bit permutations in cryptographic algorithms. *IEEE Transactions on Information Forensics and Security*, 15:1072–1085, 2020.

[25] Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36, feb 1990.

[26] David Cadé and Bruno Blanchet. Proved generation of implementations from computationally secure protocol specifications. *Journal of Computer Security*, 23(3):331–402, 2015.

[27] U. Carlsen. Cryptographic protocol flaws: know your enemy. In *Proceedings The Computer Security Foundations Workshop VII*, pages 192,193,194,195,196,197,198,199,200, Los Alamitos, CA, USA, jun 1994. IEEE Computer Society.

[28] Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. A survey of runtime monitoring instrumentation techniques. *Electronic Proceedings in Theoretical Computer Science*, 254:15–28, Aug 2017.

[29] CCITT (Consultative Committee on International Telegraphy and Telephony). *Recommendation X*.435*: Message Handling Systems: EDI Messaging System*, 1991.

[30] John Andrew Clark and Jeremy Lawrence Jacob. *A survey of authentication protocol literature: Version 1.0*. Citeseer, 1997. Query date: 14/01/2011.

[31] Edmund M Clarke Jr, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model checking*. MIT press, 2018.

[32] Mauro Conti, Nicola Dragoni, and Viktor Lesyk. A survey of man in the middle attacks. *IEEE Communications Surveys Tutorials*, 18(3):2027–2051, 2016.

[33] Véronique Cortier, Niklas Grimm, Joseph Lallemand, and Matteo Maffei. A type system for privacy properties. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 409–423, New York, NY, USA, 2017. Association for Computing Machinery.

[34] Véronique Cortier, Steve Kremer, and Bogdan Warinschi. A survey of symbolic methods in computational analysis of cryptographic systems. *J. Autom. Reason.*, 46(3–4):225–259, apr 2011.

[35] CVE. CVE-2018-15473. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-15473, 2018. [Online; accessed 03-December-2021].

[36] D. Dolev and A. C. Yao. On the security of public key protocols. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, SFCS '81, page 350–357, USA, 1981. IEEE Computer Society.

[37] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In *Workshop on Formal Methods and Security Protocols (FMSP'99)*, July 1999.

[38] Paul Fiterău-Broştean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits Vaandrager, and Patrick Verleg. Model learning and model checking of ssh implementations. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017, page 142–151, New York, NY, USA, 2017. Association for Computing Machinery.

[39] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101, August 2011.

[40] Han Gao, Chiara Bodei, and Pierpaolo Degano. A formal analysis of complex type flaw attacks on security protocols. In José Meseguer and Grigore Roşu, editors, *Algebraic Methodology and Software Technology*, pages 167–183, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[41] Shafi Goldwasser and Silvio Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, page 365–377, New York, NY, USA, 1982. Association for Computing Machinery.

[42] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, 1988.

[43] L. Gong, R. Needham, and R. Yahalom. Reasoning about belief in cryptographic protocols. In *Proceedings. 1990 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 234–248, 1990.

[44] Andrew D Gordon and Alan Jeffrey. Typing correspondence assertions for communication protocols. *Theoretical Computer Science*, 300(1-3):379–409, 2003.

[45] S. Gritzalis and D. Spinellis. Cryptographic protocols over open distributed systems: A taxonomy of flaws and related protocol analysis tools. In Peter Daniel, editor, *Safe Comp 97*, pages 123–137, London, 1997. Springer London.

[46] Klaus Havelund and Natarajan Shankar. Experiments in theorem proving and model checking for protocol verification. In Marie-Claude Gaudel and James Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 662–681, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[47] J. Heather, G. Lowe, and S. Schneider. How to prevent type flaw attacks on security protocols. In *Proceedings 13th IEEE Computer Security Foundations Workshop. CSFW-13*, pages 255–268, 2000.

[48] International Standard Organization (ISO). International standard iso/iec 9594-8:2020, information technology - open systems interconnection - part8: The directory: Public-key and attribute certificate frameworks, 2020.

[49] ITU. Recommendation x.509 (10/19), information technology - open systems interconnection - the directory: Public-key and attribute certificate framework, 2019.

[50] Anca D. Jurcut, Tom Coffey, and Reiner Dojen. Design guidelines for security protocols to prevent replay and parallel session attacks. *Computers and Security*, 45:255–273, 2014.

[51] Geir M Køien. Why cryptosystems fail revisited. *Wireless Personal Communications*, 106:85–117, 2019.

[52] David E Langworthy, Gavin Bierman, Andrew D Gordon, Donald F Box, Bradford H Lovering, Jeffrey C Schlimmer, and John D Doty. Type system for declarative data scripting language, February 3 2015. US Patent 8,949,784.

[53] P. Laud. Symmetric encryption in automatic analyses for confidentiality against active adversaries. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, pages 71–85, 2004.

[54] Peeter Laud. Handling encryption in an analysis for secure information flow. In *Proceedings of the 12th European Conference on Programming*, ESOP'03, page 159–173, Berlin, Heidelberg, 2003. Springer-Verlag.

[55] Chris M. Lonvick and Tatu Ylonen. The Secure Shell (SSH) Authentication Protocol. RFC 4252, January 2006.

[56] Gavin Lowe. An attack on the needham-schroeder public-key authentication protocol. *Inf. Process. Lett.*, 56(3):131–133, 1995.

[57] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

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

[59] AM Mathuria, Reihaneh Safavi-Naini, and PR Nickolas. On the automation of gny logic. In *Proc. 18th Australian Computer Science Conf.*, volume 17, pages 370–379, Glenelg, South Australia, 1995.

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

[61] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 696–701, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[62] J.C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using mur/spl phi/. In *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No.97CB36097)*, pages 141–151, 1997.

[63] Paul Morrissey, Nigel P. Smart, and Bogdan Warinschi. A modular security analysis of the TLS handshake protocol. In *Advances in Cryptology - ASIACRYPT 2008, 14th International Conference on the Theory and Application of Cryptology and Information Security, Melbourne, Australia, December 7-11, 2008. Proceedings*, volume 5350 of *Lecture Notes in Computer Science*, pages 55–73. Springer, 2008.

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

[65] Dr. Clifford Neuman, Sam Hartman, Kenneth Raeburn, and Taylor Yu. The Kerberos Network Authentication Service (V5). RFC 4120, July 2005.

[66] OMG. OMG Unified Modeling Language (OMG UML), Version 2.5.1, 2017.

[67] OpenSSL. Incorrect checks for malformed signatures. https://www.openssl.org/news/secadv/20090107.txt, 2008. [Online; accessed 03-December-2021].

[68] Reema Patel, Bhavesh Borisaniya, Avi Patel, Dhiren Patel, Muttukrishnan Rajarajan, and Andrea Zisman. Comparative analysis of formal model checking tools for security protocol verification. In Natarajan Meghanathan, Selma Boumerdassi, Nabendu Chaki, and Dhinaharan Nagamalai, editors, *Recent Trends in Network Security and Applications*, pages 152–163, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[69] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *J. Comput. Secur.*, 6(1–2):85–128, January 1998.

[70] Lawrence C. Paulson. Inductive analysis of the internet protocol tls. *ACM Trans. Inf. Syst. Secur.*, 2(3):332–351, aug 1999.

[71] Alfredo Pironti, Davide Pozza, and Riccardo Sisto. Automated formal methods for security protocol engineering. *Cyber Security Standards, Practices and Industrial Applications: Systems and Methodologies*, page 138, 01 2011.

[72] Alfredo Pironti and Riccardo Sisto. An experiment in interoperable cryptographic protocol implementation using automatic code generation. In *Proceedings of the 12th IEEE Symposium on Computers and Communications (ISCC 2007), July 1-4, Aveiro, Portugal*, pages 839–844. IEEE Computer Society, 2007.

[73] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[74] Peter YA Ryan. The design and verification of security protocols. *Technical Report DRA/CIS3/SISG/CR/96/1.0, Defense Research Agency*, 1996.

[75] Elton Saul and Andrew Hutchison. In *A Generic Graphical Specification Environment for Security Protocol Modelling"*, pages 311–320, Boston, MA. Springer US.

[76] Elton Saul and Andrew Hutchison. Enhanced security protocol engineering through a unified multidimensional framework. *IEEE J. Sel. Areas Commun.*, 21(1):62–76, 2003.

[77] R. Shirey. Rfc2828: Internet security glossary. Technical report, USA, 2000.

[78] Riccardo Sisto, Piergiuseppe Bettassa Copet, Matteo Avalle, and Alfredo Pironti. Formally sound implementations of security protocols with javaspi. *Formal Aspects Comput.*, 30(2):279–317, 2018.

[79] Paul Syverson and Iliano Cervesato. The logic of authentication protocols. In Riccardo Focardi and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design*, pages 63–137, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[80] P.F. Syverson and P.C. van Oorschot. On unifying some cryptographic protocol logics. In *Proceedings of 1994 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 14–28, 1994.

[81] Ilaria Venturini. Oracle attacks and covert channels. In Mauro Barni, Ingemar Cox, Ton Kalker, and Hyoung-Joong Kim, editors, *Digital Watermarking*, pages 171–185, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[82] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4), oct 2009.

[83] Andrew C. Yao. Theory and application of trapdoor functions. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, page 80–91, USA, 1982. IEEE Computer Society.