Security Protocols: Principles and Calculi
Tutorial Notes

Martín Abadi

Microsoft Research
and
University of California, Santa Cruz

Abstract. This paper is a basic introduction to some of the main themes in the design and analysis of security protocols. It includes a brief explanation of the principles of protocol design and of a formalism for protocol analysis. It is intended as a written counterpart to a tutorial given at the 2006 International School on Foundations of Security Analysis and Design.

1 Introduction

Over the last 30 years, work on security protocols has led to a number of ideas and techniques for protocol design, to a number of protocol implementations, and also to many attacks. Gradually, it has also led to mature techniques for analyzing protocols, and to an understanding of how to develop more robust protocols that address a range of security concerns.

These notes are based on a tutorial on security protocols given at the 2006 International School on Foundations of Security Analysis and Design. The tutorial was an introduction to the following topics:

- security protocols (informally),
- some design principles,
- a formal calculus for protocol analysis: the applied pi calculus,
- automated proof methods and tools, such as ProVerif.

The slides from the tutorial are available on-line [1]. These notes are essentially a summary of the material presented there. They do not aim to provide a balanced survey of the entire field, nor to explain some of its advanced points, which are covered well in research papers. Instead, the notes aim to introduce the basics of security protocols and the applied pi calculus. They may help readers who are starting to study the subject, and may offer some perspectives that would perhaps interest others.

The next section is a general description of security protocols. Section 3 gives an example, informally. Section 4 explains a few principles for the design of protocols. Section 5 is an introduction to informal and formal protocol analysis. Sections 6 and 7 present the applied pi calculus and the ProVerif tool, respectively. Section 8 revisits the example of Section 3. Section 9 concludes by discussing some further work.

2 Security Protocols

Security protocols are concerned with properties such as integrity and secrecy. Primary examples are protocols that establish communication channels with authenticity and confidentiality properties—in other words, communication channels that protect the integrity and secrecy of the data sent between the intended protocol participants. Other examples include protocols for commerce and for electronic voting.

This section describes security protocols in a little more detail. It introduces a protocol due to Needham and Schroeder, which serves as a running example in the rest of these notes.

2.1 Cryptography

In distributed systems, security protocols invariably employ some cryptography [65], so they are sometimes called cryptographic protocols. We call them security protocols in order to emphasize ends over means, and also in order to include some exchanges in which cryptography is not needed or not prominent. For the purposes of these notes, only very simple cryptography is needed.

We focus on symmetric cryptosystems (such as DES and AES). In these, when two principals communicate, they share a key that they use for encryption and for decryption. Therefore, symmetric cryptosystems are also called shared-key cryptosystems. When integrity is required, as it often is, a shared key also serves for producing and for checking message authentication codes (MACs). A MAC is basically a signature, with the limitation that only the principals that know the corresponding shared key can check it, and that any of these principals could produce it. So, although any principal that knows the shared key could convince itself that some other principal has produced a given MAC, it may not be able to convince a judge or some other third party.

In SSL [46] and other practical protocols, multiple shared keys are often associated with each communication channel. For instance, for each point-to-point connection, we may have two shared keys for encryption, one for communication in each direction, and two for MACs, again one per direction. However, these four keys may all be derived from a shared master key. Furthermore, protocols often rely on both symmetric cryptosystems and asymmetric cryptosystems (such as RSA). While we may not explicitly discuss them, many of these variants do fall within the scope of the methods presented in these notes.

As in many informal protocol descriptions, below we assume that the encryption function provides not only secrecy but also authenticity. In other words, we proceed as though the encryption function includes a built-in MAC.

2.2 Other Machinery

There is more to the mechanics of protocols than cryptography. Moreover, protocols exist and should be understood in the context of other security machinery, such as auditing and access control.

Protocols often include timestamps or other proofs of freshness. They may also include sequence numbers for ordering messages. At a lower level, practical protocols

also include key identifiers (so that the recipient of an encrypted message knows which key to try), message padding, and facilities for message compression, for instance.

Furthermore, protocols often rely on trusted third parties. These trusted third parties may function as certification authorities, as trusted time servers, and in other roles. Trust is not absolute, nor always appropriate—so protocols often aim to eliminate trusted third parties or to limit the trust placed in them.

2.3 Authentication Protocols

In systems based on the access-control model of security [58], authorization relies on authentication, and protocols that establish communication channels with authenticity and confidentiality properties are often called authentication protocols. There are many such protocols. They typically involve two principals (hosts, users, or services) that wish to communicate, and some trusted third parties. In particular, the two principals may be a client and a server, and the purpose of the channel may be to convey requests and responses between them.

Despite these commonalities, there are also a number of differences across authentication protocols; no single authentication protocol will be suitable for all systems. For performance, designers consider communication, storage, and cryptographic costs, and sometimes trade between them. The choice of cryptographic algorithms is influenced by these cost considerations, and also by matters of convenience and law. In addition, systems rely on synchronized clocks to different extents.

At a higher level, no single authentication protocol will be suitable for all purposes. Protocols vary in their assumptions, in particular with respect to trusted third parties. They also vary in their objectives:

- Some protocols achieve mutual authentication; others achieve only one-way authentication, and in some cases guarantee the anonymity of one of the parties (typically the client).
- Data secrecy is sometimes optional.
- A few protocols include protection against denial-of-service attacks. This protection aims to ensure that protocol participants cannot be easily burdened with many costly cryptographic operations and other expensive work.
- Going beyond the basic security properties, some protocols aim to ensure non-repudiation (so participants cannot later deny some or all of their actions), for instance. A few protocols aim to support plausible deniability, which is roughly the opposite of non-repudiation.

3 An Example

We describe a protocol due to Needham and Schroeder as an example. The protocol is the first from their seminal paper on authentication [74]; it relies on a symmetric cryptosystem. Throughout, we refer to this protocol as the Needham-Schroeder protocol, because of its importance and because we do not consider other protocols by the same authors.

The Needham-Schroeder protocol is one of the classics in this field. It has served as the basis for the Kerberos system [56, 68] and much other subsequent work. Many of its ingredients occur in other protocols, including many recent ones. Recent protocols, however, typically have more moving parts—more modes, options, and layers.

3.1 Model

Needham and Schroeder set out the following informal model:

> We assume that an intruder can interpose a computer in all communication paths, and thus can alter or copy parts of messages, replay messages, or emit false material.
> We also assume that each principal has a secure environment in which to compute, such as is provided by a personal computer or would be by a secure shared operating system.

The first assumption is common across the field, and is probably even more reasonable now than it was when it was first formulated. The second assumption is also common, but unfortunately it is often somewhat questionable because of the widespread software failures that viruses and worms exploit.

3.2 The Protocol

In this protocol, A and B are two principals that wish to establish a secure communication session. An authentication server S is a trusted intermediary.

Initially the principals A and B share K_{AS} and K_{BS} with S, respectively. The goal of the protocol is to establish a session key K_{AB} for A and B.

In the course of the protocol, A and B invent the nonces N_A and N_B, respectively. Nonces are quantities generated for the purpose of being fresh. In particular, A can reason that any message that includes N_A was manufactured after N_A's invention; so A can conclude that any such message belongs in the current protocol session, and is not a replay from a previous session. The use of nonces dispenses with the requirement of a single network clock; nonces are still prevalent today, in protocols such as SSL and IKE [46, 53].

Figure 1 depicts the message exchange. Here, we write $\{X\}_K$ for an encryption of the plaintext X under the key K, and X, Y for the concatenation of X and Y (with markers, as needed, in order to avoid ambiguities).

Only A contacts the server S, in Message 1. This message includes N_A. Upon receipt of this message, S generates K_{AB}, which becomes the session key between A and B. In Message 2, S provides this session key to A, under K_{AS}. This message includes A's nonce, as a proof of freshness. Message 2 also includes a certificate (or a "ticket", in Kerberos parlance) under K_{BS} that conveys the session key and A's identity to B. Message 3 transmits this certificate from A to B. After decrypting Message 3 and obtaining the session key, B carries out a handshake with A, in Messages 4 and 5. The use of $N_B - 1$ in Message 5 is somewhat arbitrary; almost any function of N_B would do as long as B can distinguish this message from its own Message 4.

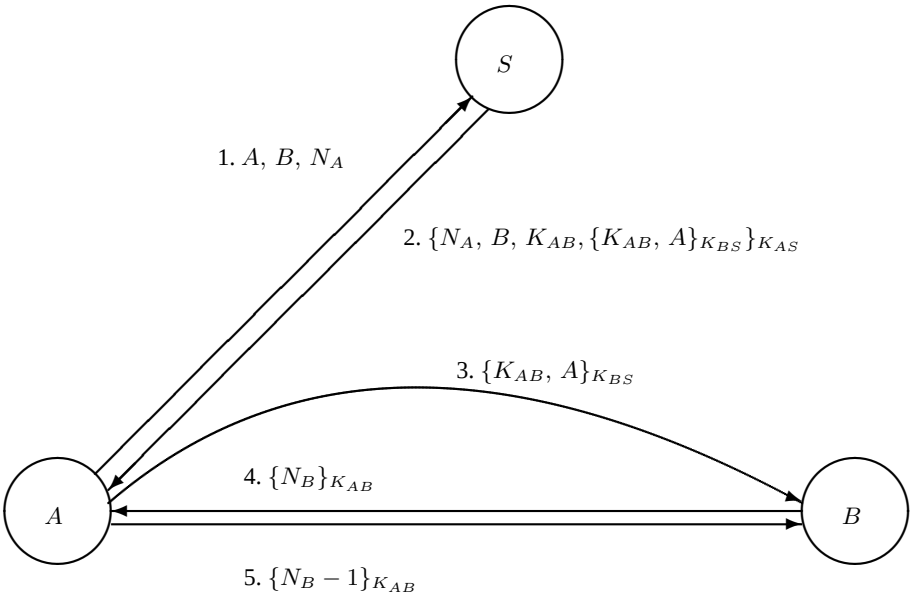

Fig. 1. The Needham-Schroeder protocol.

3.3 A Limitation

As Denning and Sacco observed [41], this protocol has a serious limitation:

- Suppose that an attacker has a log of the messages exchanged during a protocol run.
- Suppose further that, long after a run, the attacker may discover the session key K_{AB} somehow—for instance, through a long brute-force attack or as a result of the careless exposure of old key material.
- The attacker may then replay Message 3 to B. Unless B remembers K_{AB} or has some external indication of the attack, B is not able to distinguish the replay from a legitimate, new instance of Message 3.
- The attacker may then conduct a handshake with B. Although B uses a fresh nonce for this handshake, the attacker is able to produce a corresponding response because it knows K_{AB}.
- Subsequently, the attacker may continue to communicate with B under K_{AB}, impersonating A.

In order to address this limitation, one may try to make K_{AB} strong, and to change K_{BS} often, thus limiting the window of vulnerability. One may however prefer to use an improved protocol, in which B and S interact directly (as Needham and Schroeder suggested [75]), or in which messages include timestamps (as Denning and Sacco proposed, and as done in Kerberos).

As this example illustrates, most security protocols have subtleties and flaws. Many of these have to do with cryptography, but many of these do not have to do with the

details of cryptographic algorithms. For design, implementation, and analysis, a fairly abstract view of cryptography is often practical.

4 Principles of Protocol Design

While protocol design often proceeds informally, it need not be entirely driven by trial and error. Some general principles can guide the creation and understanding of protocols (e.g., [9, 14, 15, 79]). Such principles serve to simplify protocols and to avoid many mistakes. They also serve to simplify informal reasoning about protocols and their formal analysis [28, 76].

In this section we explain some of these principles, with reference to our example.

4.1 Explicit Messages

In the early logics of authentication, an informal process of idealization turned loose protocol narrations into formulas that expressed the perceived intended meanings of messages [37]. Over time, it was noticed that many attacks were identified in the course of this informal process—even more than during the later formal proofs. More broadly, it was noticed that many attacks appeared because of gaps between the actual contents of messages and their intended meanings. This realization and much experience led to the following principle [9]:

> Every message should say what it means: the interpretation of the message should depend only on its content.

In other words, the meaning of the message should not depend on implicit information that is presumed clear from context. Such presumptions are often unreliable in the presence of attackers that do not play by the rules.

As an important special case of this principle, it follows:

> If the identity of a principal is important for the meaning of a message, it is prudent to mention the principal's name explicitly in the message.

For instance, Message 2 of the Needham-Schroeder protocol consists of the ciphertext $\{N_A,\, B,\, K_{AB}, \{K_{AB},\, A\}_{K_{BS}}\}_{K_{AS}}$. The first three fields N_A, B, K_{AB} of the plaintext message are intended as a statement to A that K_{AB} is a good key for a session with B sometime after the generation of N_A. The name A is implicit, and can be deduced from K_{AS}. Making it explicit would not be costly, and it would lead to a more robust protocol—allowing, for instance, the possibility that the key K_{AS} would be a key for a node with multiple users (A_1, A_2, ...). On the other hand, the name B is and must be explicit. Omitting it enables an attack, as follows.

- Suppose that an attacker C intercepts Message 1, replaces B with C, and sends the modified message to S.
- In response, Message 2 includes $\{K_{AC},\, A\}_{K_{CS}}$, where K_{CS} is known to C, rather than $\{K_{AB},\, A\}_{K_{BS}}$. However, A cannot detect this substitution: A can check its nonce N_A, and obtains K_{AC} and this certificate, but the certificate is opaque to A. The subscript C in K_{AC} is merely a meta-notation; nothing in the key itself indicates that it is shared with C rather than with B.

– Suppose further that C intercepts Message 3 in which A forwards $\{K_{AC},\ A\}_{K_{CS}}$ to B. Then C obtains K_{AC} and can conduct a handshake with A, in B's place.
– Subsequently, C may continue to communicate with A under K_{AC}, impersonating B.

Many successful attacks against published protocols resemble this one, and stem from the omission of some names.

Similarly, Message 3 of the Needham-Schroeder protocol consists of the ciphertext $\{K_{AB},\ A\}_{K_{BS}}$. To B, this message should mean that K_{AB} is a good key for a session with A. Again, one of the names (B in this case) is implicit, while the other (A) is explicit and is needed in order to thwart an attack. (The attack is left as an easy exercise for the reader.) Remarkably, the meaning of this message does not specify at which time K_{AB} is a good key. While the handshake has something to do with timeliness, the exact significance of $\{N_B\}_{K_{AB}}$ and $\{N_B - 1\}_{K_{AB}}$ is a little unclear. Denning and Sacco exploited these shortcomings in their attack.

Often, the meanings of messages pertain to the goodness of keys. As Needham noted, years later, much progress can be made without further elaboration on what is a good key [73]:

> The statement that a key was "good" for certain communication bundles up all sorts of useful notions—that it was made by a careful agent, had not been scattered about, had sufficient variety, and so forth.

Still, if several kinds of good keys are possible (from different cryptosystems, or with different parameters), then messages should be explicit on which kind is intended.

4.2 Explicit Design

Cryptography is a powerful tool, but a proper understanding of its guarantees, nuances, and limitations is required for its effective use in protocols. Accordingly, the next principle concerns the use of cryptography, rather than the specifics of particular cryptographic algorithms [9].

> Be clear as to why encryption is being done.
> Encryption is not synonymous with security.

In protocols, encryption and other cryptographic functions are used for a variety of purposes.

– Encryption is sometimes used for confidentiality. For example, in Message 2, encryption protects the secrecy of K_{AB}.
– Encryption is sometimes used in order to guarantee authenticity. For example, A may reason that Message 2 is an authentic message from S because of the encryption.
– Encryption sometime serves for proving the presence of a principal or the possession of a secret. Message 5 exemplifies this use.

– Encryption may also serve for binding together the parts of a message. In Message 2, the double encryption may be said to serve this purpose. However, in this example, rewriting the message to $\{N_A,\ B,\ K_{AB}\}_{K_{AS}}, \{K_{AB},\ A\}_{K_{BS}}$ would work just as well. Double encryption is not double security—and indeed sometimes it is a source of confusion and insecurity, as in the Woo-Lam protocol [9, 81, 82].
– Encryption is sometimes used in random-number generation, in defining MACs, and in other cryptographic tasks. It is generally best to leave those uses for the lower-level constructions of cryptographic primitives, outside the scope of protocol design.

While the principle above refers to encryption, it also applies to other cryptographic functions. More generally, it is desirable that not only messages but also the protocol design be explicit [14]:

> Robust security is about explicitness; one must be explicit about any properties which can be used to attack a public key primitive, such as multiplicative homomorphism, as well as the usual security properties such as naming, typing, freshness, the starting assumptions and what one is trying to achieve.

5 Analysis

The development of methods for describing and analyzing security protocols seems to have started in the early 1980s (e.g., [30, 40, 42, 55, 66, 83]). The field matured considerably in the 1990s. Some of the methods rely on rigorous but informal frameworks, sometimes supporting sophisticated complexity-theoretic definitions and arguments (e.g., [19, 30, 47, 83]). Others rely on formalisms specially tailored for this task (e.g., [37, 80]). Yet others are based on temporal logics, process algebras such as CSP and the pi calculus, and other standard formalisms, sometimes in the context of various theorem-proving tools, such as Isabelle (e.g., [8, 52, 61, 64, 76, 78]). The next section presents the applied pi calculus as an example of this line of work.

Overall, the use of these methods has increased our confidence in some protocols. It has also resulted in the discovery of many protocol limitations and flaws, and in a better understanding of how to design secure protocols.

Many of these methods describe a protocol as a program, written in a programming notation, or as the corresponding set of executions. In addition to the expected principals, this model of the protocol should include an attacker. The attacker has various standard capabilities:

– it may participate in some protocol runs;
– it may know certain data in advance;
– it may intercept messages on some or all communication paths;
– it may inject any messages that it can produce.

The last of these is the most problematic: in order to obtain a realistic model, we should consider non-deterministic attackers, which may for example produce keys and nonces, but without such luck that they always guess the keys on which the security of the protocol depends.

One approach to this problem consists in defining the attacker as some sort of probabilistic program that is subject to complexity bounds. For instance, the attacker may be a probabilistic polynomial-time Turing machine. Such a machine is not able to explore an exponentially large space of possible values for secret keys. This approach can be quite successful in providing a detailed, convincing model of the attacker. Unfortunately, it can be relatively hard to use.

Going back to early work on decision procedures by Dolev and Yao [42], formal methods adopt a simpler solution. They arrange that non-deterministic choice of a key or a nonce always yields a fresh value (much like object allocation in object-oriented languages always returns a fresh address). In this respect, keys and nonces are not ordinary bitstrings. Accordingly, cryptographic operations are treated formally (that is, symbolically). Some assumptions commonly underly these formal methods. For instance, for symmetric encryption, we often find the following assumptions:

- Given K, anyone can compute $\{M\}_K$ from M.
- Conversely, given K, anyone can compute M from $\{M\}_K$.
- $\{M\}_K$ cannot be produced by anyone who does not know M and K.
- M cannot be derived from $\{M\}_K$ by anyone who does not know K (and K cannot be derived from $\{M\}_K$).
- An attempt to decrypt $\{M\}_K$ with an incorrect key K' will result in an evident failure.

Here, M, K, and $\{M\}_K$ represent formal expressions. The first assumption says that anyone with the expressions K and M can obtain the expression $\{M\}_K$; the operation applied is a symbolic abstraction of encryption, rather than a concrete encryption operation on bitstrings. Similarly, the second assumption corresponds to a symbolic abstraction of decryption, and the fifth assumption to a related symbolic check. The third and the fourth assumptions are reflected in the absence of any operations for encrypting or decrypting without the corresponding key.

Despite their somewhat simplistic treatment of cryptography, formal methods are often quite effective, in part because, as noted above, a fairly abstract view of cryptography often suffices in the design, implementation, and analysis of protocols. Formal methods enable relatively simple reasoning, and also benefit from substantial work on proof methods and from extensive tool support.

The simplistic treatment of cryptography does imply inaccuracies, possibly mistakes. The separation of keys and nonces from ordinary data implies that attackers cannot do arbitrary manipulations on keys and nonces. For instance, attackers may not be allowed to do bitwise shifts on keys, if that is not represented as a symbolic operation somehow. Thus, attacks that rely on shifts are excluded by the model, rather than by proofs.

A recent research effort aims to bridge the gap between complexity-theoretic methods and formal methods. It aims to provide rigorous justifications for abstract treatments of cryptography, while still enabling relatively easy formal proofs. For instance, a formal treatment of encryption is sound with respect to a lower-level computational model based on complexity-theoretic assumptions [10]. The formal treatment is simple but fairly typical, with symbolic cryptographic operations. In the computational model, on the other hand, keys and all other cryptographic data are bitstrings, and

adversaries have access to the full low-level vocabulary of algorithms on bitstrings. Despite these additional capabilities of the adversaries, the secrecy assertions that can be proved formally are also valid in the lower-level model, not absolutely but with high probability and against adversaries of reasonable computational power. Further research in this area addresses richer classes of systems and additional cryptographic functions (e.g., [16, 39, 59, 60, 67]). Further research also considers how to do automatic proofs in a computational model, starting from formal protocol descriptions but with semantics and proof principles from the complexity-theoretic literature (e.g., [26, 29]).

6 The Applied Pi Calculus

This section introduces the applied pi calculus [6], focusing on its syntax and its informal semantics. Section 7 describes ProVerif, a tool for the applied pi calculus; Section 8 gives an example of the use of the applied pi calculus.

6.1 Security Protocols in the Pi Calculus

The pi calculus is a minimal language for describing systems of processes that communicate on named channels, with facilities for dynamic creation of new channels [69, 70]. We use it here without defining it formally; some definitions appear below, in the context of the applied pi calculus. As usual, we write $\overline{c}\langle\ldots\rangle$ for a message emission and $c(\ldots)$ for a message reception, "." for sequential prefixing, "|" for parallel composition, and "ν" for name restriction. Equations like $A = \ldots$, $B = \ldots$, and $P = \ldots$ are definitions outside the pi calculus: the operator "=" is not part of the calculus itself.

At an abstract level, the pi calculus is sufficient for describing a wide range of systems, including security protocols. For instance, we may describe an abstract version of a trivial one-message protocol as follows:

$$\begin{aligned} A &= \overline{c}\langle V\rangle \\ B &= c(x).\overline{d}\langle\rangle \\ P &= (\nu c)(A \mid B) \end{aligned}$$

Here, A is a process that sends the message V on the channel c, and B is a process that receives a message on the channel c (with x as the argument variable to be bound to the message), then signals completion by sending an empty message on the channel d. Finally, P is the entire protocol, which consists of the parallel composition of A and B with a restriction on the channel c so that only A and B can access c.

The attacker, left implicit in the definitions of this example, is the context. It may be instantiated to an arbitrary expression Q of the pi calculus, and put in parallel with P, as in $P \mid Q$.

This process representation of the protocol has properties that we may interpret as security properties. In particular, in any context, P is equivalent to a variant P' that sends V' in place of V, for any other message V'. Indeed, P and P' are so trivial that they are equivalent to $\overline{d}\langle\rangle$. Thinking of the context as an attacker, we may say that this property expresses the secrecy of the message V from the attacker.

In more complicated examples, the security properties are less obvious, but they can still be formulated and established (or refuted) using the standard notations and proof techniques of the pi calculus. In particular, the formulations rely on universal quantification over all possible attackers, which are treated as contexts in the pi calculus. This treatment of attackers is both convenient and generally useful.

6.2 The Applied Pi Calculus

As in the small example above, the pi-calculus representations of protocols often model secure channels as primitive, without showing their possible cryptographic implementations. In practice, the channel c of the example may be implemented using a public channel plus a key K shared by A and B. Sending on c requires encryption under K, and receiving on c requires decryption with K. Additional precautions are necessary, for instance in order to prevent replay attacks. None of this implementation detail is exposed in the pi-calculus definitions.

Moreover, even with the abstraction from keys to channels, some protocols are hard to express. The separation of encryption from communication (an important aspect of the work of Needham and Schroeder) can be particularly problematic. For instance, Message 2 of the Needham-Schroeder protocol, from S to A, includes $\{K_{AB},\ A\}_{K_{BS}}$, to be forwarded to B. This message component might be modeled as a direct message from S to B on a secure channel—but such a model seems rather indirect, and might not be sound.

One approach to addressing this difficulty consists in developing encodings of encryption in the pi calculus [8, 17]. While this approach may be both viable and interesting, it amounts to a substantial detour.

Another approach to addressing this difficulty relies on extensions of the pi calculus with formal cryptographic operations, such as the spi calculus [8] and the applied pi calculus. The applied pi calculus is essentially the pi calculus plus function symbols that can be used for expressing data structures and cryptographic operations. The spi calculus can be seen as a fragment that focuses on a particular choice of function symbols. In both cases, the function symbols enable finer protocol descriptions. These descriptions may show how a secure channel is implemented with encryption, or how one key is computed from another key. Next we introduce the syntax and the informal semantics of the applied pi calculus.

We start with a sort of variables (such as x and y) and a sort of names (such as n). We use meta-variables u and v to range over both names and variables. We also start with a set of function symbols, such as f, $\mathsf{encrypt}$, and pair. These function symbols have arities and types, which we generally omit in this presentation. In addition to arities and types, the function symbols come with an equational theory (that is, with an equivalence relation on terms with certain closure properties). For instance, for binary function symbols senc and sdec, we may have the usual equation:

$$\mathsf{sdec}(\mathsf{senc}(x, y), y) = x$$

If in addition we have a binary function symbol scheck and a constant symbol ok, we may have the additional equation:

$$\mathsf{scheck}(\mathsf{senc}(x, y), y) = \mathsf{ok}$$

Intuitively, senc and sdec stand for symmetric encryption and decryption, while scheck provides the possibility of checking that a ciphertext is under a given symmetric key.

The set of terms is defined by the grammar:

$U, V ::=$	terms
$c, d, n, s, K, N, \ldots$	name
$x, y, K, \ldots$	variable
$f(U_1, \ldots, U_l)$	function application

where f ranges over the function symbols and $U_1, \ldots, U_l$ match the arity and type of f. Terms are intended to represent messages and other data items manipulated in protocols.

The set of processes is defined by the grammar:

$P, Q, R ::=$	processes
nil	null process
$P \mid Q$	parallel composition
$!P$	replication
$(\nu n)P$	name restriction ("new")
$\mathit{if}\ U = V\ \mathit{then}\ P\ \mathit{else}\ Q$	conditional
$u(x_1, \ldots, x_n).P$	message input
$\overline{u}\langle V_1, \ldots, V_n\rangle.P$	message output

Informally, the semantics of these processes is as follows:

- The null process nil does nothing.
- $P \mid Q$ is the parallel composition of P and Q.
- The replication $!P$ behaves as an infinite number of copies of P running in parallel.
- The process $(\nu n)P$ generates a new name n then behaves as P. The name n is bound, and subject to renaming.
  The use of ν is not limited to generating new channel names. We often use ν more broadly, as a generator of unguessable values. In some cases, those values may serve as nonces or as keys. In others, those values may serve as seeds, and various transformations may be applied for deriving keys from seeds.
- The conditional construct $\mathit{if}\ U = V\ \mathit{then}\ P\ \mathit{else}\ Q$ is standard. Here, $U = V$ represents equality in the equational theory, not strict syntactic identity. We abbreviate it $\mathit{if}\ U = V\ \mathit{then}\ P$ when Q is nil.
- The input process $u(x_1, \ldots, x_n).P$ is ready to input a message with n components from channel u, then to run P with the actual message components replaced for the formal parameters x_1, …, x_n. We may omit P when it is nil. The variables $x_1, \ldots, x_n$ are bound, and subject to renaming.
- The output process $\overline{u}\langle V_1, \ldots, V_n\rangle.P$ is ready to output a message with n components V_1, …, V_n on channel u, then to run P. Again, we may omit P when it is nil.

Processes are intended to represent the components of a protocol, but they may also represent attackers, users, or other entities that interact with the protocol.

As an abbreviation, we may also write $let\ x = U\ in\ P$. It can be defined as $(\nu c)(\overline{c}\langle U\rangle \mid c(x).P)$, where c is a name that does not occur in U or in P.

As these definitions indicate, the applied pi calculus is rather abstract. It allows us to omit many details of cryptography and communication. On the other hand, both cryptography and communication are represented in the applied pi calculus. We can describe every message, under what circumstances it is sent, how it is checked upon receipt, and what actions it triggers.

Research on the spi calculus and the applied pi calculus includes the development of formal semantics, the study of equivalences and type systems, the invention of decision procedures for particular problems, the definition of logics, other work on proof techniques and tools, and various applications (e.g., [2, 5, 18, 34–36, 43, 44, 49, 50, 54]). Research on related formalisms touches on many of these topics as well (e.g., [12, 13, 38, 45, 77]). We discuss only a fraction of this work in the present notes, and refer the reader to the research literature for further material on these topics.

7 ProVerif

A variety of methods for protocol analysis rely at least in part on tool support. They are effective on abstract but detailed models of important protocols. Many of them employ elaborate proof techniques—some general, some specific to this area.

Since the work of Dolev and Yao, there has been much research on special decision procedures. In recent years, these have been most successful for finite-state systems (e.g., [18]). Since the mid 1990s, general-purpose model-checking techniques have also been applied in this area (e.g., [62, 71]). Again, they are usually most effective for finite-state systems. There has also been research on proofs with semi-automatic proof assistants (e.g., [33, 76]). These proofs can require a fair amount of expert human guidance. On the other hand, they can produce sophisticated theorems and attacks, even for infinite-state systems.

Several other approaches rely on programming-language techniques, such as typing, control-flow analysis, and abstract interpretation (e.g., [2, 31, 72]). These techniques are often incomplete but useful in examples and (relatively) easy to use. It turns out that some of these techniques are equivalent, at least in theory [2, 32]. We give a brief description of ProVerif [23–25, 27], as an important example of this line of work.

ProVerif is an automatic checker for the applied pi calculus. It features a somewhat modified input syntax, in which function symbols are categorized as constructors and destructors. Pairing and encryption are typical examples of constructors, while projection operations and decryption are examples of destructors.

Internally, ProVerif translates from the applied pi calculus to Horn clauses, and thus represents protocols as logic programs. For example, if a process sends the name A on channel c when it receives the name B on channel d, then the Horn clauses that represent the protocol will imply

$$mess(d, B) \rightarrow mess(c, A)$$

where $mess$ is a predicate that indicates the possible presence of a message on a channel. Some of the Horn clauses deal with communication and with cryptography (not

specifically to a protocol). For example, we may have:

$$attacker(x) \wedge attacker(y) \rightarrow mess(x, y)$$
$$attacker(x) \wedge attacker(y) \rightarrow attacker(\mathsf{senc}(x, y))$$

where $attacker$ is a predicates that characterizes the knowledge of an attacker.

ProVerif then applies automated analysis techniques based on resolution to these Horn clauses. It contains proof methods for certain classes of properties. These include secrecy and authenticity properties. In particular, the secrecy of a name s may be formulated in terms of whether or not $attacker(s)$ is provable.

ProVerif has been effective on a wide range of examples. For instance, it can treat the Needham-Schroeder protocol without much difficulty, using definitions similar to those presented in Section 8. More advanced examples include a protocol for certified email [3], the JFK protocol [11] for keying for IP security [4], some password-based protocols [27], some electronic-voting protocols [57], and several web-services protocols [20, 63]. ProVerif seems to be fairly accessible to new users. Remarkably, it has also served as a powerful basis for sophisticated tools for analyzing web-services protocols [22].

ProVerif proofs typically take seconds or minutes, though longer proofs are possible too. ProVerif guarantees termination only in certain cases [28]. Manual arguments are sometimes combined with automatic proofs.

8 An Example, Revisited

As an example, we write the Needham-Schroeder protocol in the applied pi calculus.

An analysis of this example may be done by hand, using a variety of proof techniques for the applied pi calculus that go beyond the scope of these notes. An analysis may also be done automatically with ProVerif, as mentioned above.

8.1 Preliminaries

We assume that e is a public channel on which all principals may communicate. Therefore, we do not restrict the scope of e with the ν operator. We do not represent the details of addressing and routing. In our formulation of the code, it is possible for a principal to receive a message intended for some other principal, and for the processing to get stuck. It is straightforward to do better. We choose this simplistic model because the details of addressing are mostly orthogonal to the primary security concerns in this protocol. In other protocols, the details of addressing may be more important, for instance if one is interested in hiding the identities of the principals that communicate, in order to obtain privacy guarantees (e.g., [7]).

We use the following function symbols:

- We use constant symbols A, B, ... for principal names.
- We also use the function symbols introduced above for symmetric cryptography (senc, sdec, scheck, and ok).

– We use two unary function symbols that we write in postfix notation, as -1 and $+1$, with the equation $(x - 1) + 1 = x$.
  While this equation may not seem surprising, it is worth noting that it is not essential to writing the processes that represent the protocol. We introduce it because we wish to emphasize that anyone (including an attacker) can invert the -1 function. Without this equation, -1 might appear to be a one-way function, so one might wrongly expect that it would be impossible to recover N_A from $N_A - 1$. Similarly we could add other equations, such as $(x + 1) - 1 = x$. We return to the subject of choosing equations in Section 8.4.
– We also assume tupling and the corresponding projection operations. We write $(U_1, \ldots, U_n)$ for the tuple of $U_1, \ldots, U_n$, for any n, and write p_i for the projection function that retrieves U_i, for $i = 1..n$, with the equation $\mathsf{p}_i((x_1, \ldots, x_n)) = x_i$.
– Finally, we introduce a binary function symbol skeygen. We use skeygen to map a master key and a principal name to a symmetric key. Relying on this mapping, the server S needs to remember only a master key K_S, and can recover K_{AS} by computing $\mathsf{skeygen}(K_S, A)$ and K_{BS} by computing $\mathsf{skeygen}(K_S, B)$.
  Thus, we model a practical, modern strategy for reducing storage requirements at S. An alternative set of definitions might encode a table of shared keys at S.

8.2 A First Version

As an initial attempt, we may model the messages in the protocol rather directly. We write a process for each of A, B, and S, then combine them.

The code for A includes a top-level definition of K_{AS} (formally introduced as a variable, not a name). We do not model more realistic details of how A may obtain K_{AS}. We write the code for A in terms of auxiliary processes A_1, A_2, ... Basically, A_i represents A at Message i of a protocol execution. For instance, A_1 generates N_A, then sends A, B, N_A on e, then proceeds to A_2. In turn, A_2 receives a message x, checks that it is a ciphertext under the expected key, decrypts it, extracts four components from the plaintext, and checks that N_A is the first component and B the second, then proceeds to A_3; a failure in any of the verifications causes the processing to stop. Each of these auxiliary processes may have free names and variables bound in previous processes; for instance N_A is bound in A_1 and used in A_2.

$$
\begin{aligned}
A &= \mathit{let}\ K_{AS} = \mathsf{skeygen}(K_S, A)\ \mathit{in}\ A_1 \\
A_1 &= (\nu N_A)\overline{e}\langle A, B, N_A\rangle.A_2 \\
A_2 &= e(x).\mathit{if}\ \mathsf{scheck}(x, K_{AS}) = \mathsf{ok}\ \mathit{then} \\
&\qquad \mathit{let}\ x' = \mathsf{sdec}(x, K_{AS})\ \mathit{in} \\
&\qquad \mathit{let}\ x_1 = \mathsf{p}_1(x')\ \mathit{in} \\
&\qquad \mathit{let}\ x_2 = \mathsf{p}_2(x')\ \mathit{in} \\
&\qquad \mathit{let}\ x_3 = \mathsf{p}_3(x')\ \mathit{in} \\
&\qquad \mathit{let}\ x_4 = \mathsf{p}_4(x')\ \mathit{in} \\
&\qquad \mathit{if}\ x_1 = N_A\ \mathit{then} \\
&\qquad \mathit{if}\ x_2 = B\ \mathit{then}\ A_3 \\
A_3 &= \overline{e}\langle x_4\rangle.A_4 \\
A_4 &= e(x_5).\mathit{if}\ \mathsf{scheck}(x_5, x_3) = \mathsf{ok}\ \mathit{then}\ A_5 \\
A_5 &= \overline{e}\langle\mathsf{senc}((\mathsf{sdec}(x_5, x_3) - 1), x_3)\rangle
\end{aligned}
$$

Similarly, we write the code for S as follows:

$$\begin{aligned} S &= S_1 \\ S_1 &= e(x_1, x_2, x_3).S_2 \\ S_2 &= (\nu K) let\ x' = \mathsf{senc}((K, x_1), \mathsf{skeygen}(K_S, x_2))\ in \\ &\quad\ \overline{e}\langle \mathsf{senc}((x_3, x_2, K, x'), \mathsf{skeygen}(K_S, x_1))\rangle \end{aligned}$$

Here the key K stands for the new symmetric key for communication between clients (named K_{AB} above for clients A and B).

Finally, we write the code for B as follows:

$$\begin{aligned} B &= let\ K_{BS} = \mathsf{skeygen}(K_S, B)\ in\ B_3 \\ B_3 &= e(x).if\ \mathsf{scheck}(x, K_{BS}) = \mathsf{ok}\ then \\ &\quad\ let\ x' = \mathsf{sdec}(x, K_{BS})\ in \\ &\quad\ let\ x_1 = \mathsf{p}_1(x')\ in \\ &\quad\ let\ x_2 = \mathsf{p}_2(x')\ in\ B_4 \\ B_4 &= (\nu N_B)\overline{e}\langle \mathsf{senc}(N_B, x_1)\rangle.B_5 \\ B_5 &= e(y).if\ \mathsf{scheck}(y, x_1) = \mathsf{ok}\ then \\ &\quad\ if\ \mathsf{sdec}(y, x_1) - 1 = N_B - 1\ then\ nil \end{aligned}$$

We assemble the pieces so as to represent a system with A, B, and S:

$$P = (\nu K_S)(A \mid B \mid S)$$

8.3 A Second Version

The first version of the code is not entirely satisfactory in several respects.

- A appears to initiate a session with B spontaneously, and communication stops entirely after a shared key is established. For instance, B checks the last message, but stops independently of whether the check succeeds.
  A more complete model of the protocol would show that A initiates a session because of some event. This event may for example come from a process R_A that represents an application that uses the protocol at A. Upon completion of a successful exchange, the resulting key may be provided to the application. (Alternatively, the protocol could include its own layer for encrypted data communications, like SSL's record layer.) Similarly, upon completion of a successful exchange, the resulting key and the identity of the other endpoint could be passed to a process R_B at B, which may check the identity against an access-control policy. Thereafter, R_A and R_B may use the session key; they should never use the master key K_S.
- If the possibility of session-key compromise is important, as indicated by Denning and Sacco, then it should be modeled. For instance, upon completion of a successful exchange, the session key may be broadcast. An analysis of the protocol without such an addition would not detect the possibility of an attack that relies on the compromise of old session keys.
  In a model with user processes R_A and R_B, we may simply consider the possibility that one of these processes leaks old session keys.

– A should not be limited to initiating one session, and the identity of the principals A and B should not be fixed. Rather, each principal may engage in the protocol in the role of A or B, or even in both roles simultaneously, multiple times.
Therefore, the code for these roles should use, as a parameter, the claimed name of the principal that is running the code. In addition, the code should be replicated.

We arrive at the following variant of our definitions for A:

$$\begin{aligned}
A(x_A) &= (\nu c)(\nu d)(R_A \mid \mathit{let}\ K_{AS} = \mathsf{skeygen}(K_S, x_A)\ \mathit{in}\ !c(x_B).A_1) \\
A_1 &= (\nu N_A)\overline{e}\langle x_A, x_B, N_A\rangle.A_2 \\
A_2 &= \text{ as above, except for the last line, which becomes} \\
&\qquad \mathit{if}\ x_2 = x_B\ \mathit{then}\ A_3 \\
A_3 &= \text{ as above} \\
A_4 &= \text{ as above} \\
A_5 &= \overline{e}\langle \mathsf{senc}((\mathsf{sdec}(x_5, x_3) - 1), x_3)\rangle.\overline{d}\langle x_3, x_B\rangle
\end{aligned}$$

Here the variables x_A and x_B represent A's and B's names, respectively. Channels c and d are for communication between R_A and the rest of the code. Channel c conveys the identity of the other endpoint; channel d returns this identity and a session key; R_A may then use the session key, and perhaps leak it. A replication indicates that an unbounded number of sessions may be initiated.

Similarly, we revise the code for B as follows:

$$\begin{aligned}
B(x_B) &= (\nu d)(R_B \mid \mathit{let}\ K_{BS} = \mathsf{skeygen}(K_S, x_B)\ \mathit{in}\ !B_3) \\
B_3 &= \text{ as above} \\
B_4 &= \text{ as above} \\
B_5 &= e(y).\mathit{if}\ \mathsf{scheck}(y, x_1) = \mathsf{ok}\ \mathit{then} \\
&\qquad \mathit{if}\ \mathsf{sdec}(y, x_1) - 1 = N_B - 1\ \mathit{then}\ \overline{d}\langle x_1, x_2\rangle
\end{aligned}$$

As in the code for A, channel d conveys the session key and the identity of the other endpoint. Again, a replication indicates that an unbounded number of sessions may be initiated.

In S, only an extra replication is needed:

$$\begin{aligned}
S &= !S_1 \\
S_1 &= \text{ as above} \\
S_2 &= \text{ as above}
\end{aligned}$$

Suppose that we wish to represent a system with client principals named C_1, ..., C_n, all of them able to play the roles of A and B, and all of them with the same application code for each for the roles. The corresponding assembly is:

$$P = (\nu K_S)(A(C_1) \mid B(C_1) \mid \ldots \mid A(C_n) \mid B(C_n) \mid S)$$

Many variants and elaborations are possible. For instance, some of the checks may safely be removed from the code. Since the applied pi calculus is essentially a programming language, protocol models are enormously malleable. However, complex models are rarely profitable—the point of diminishing returns is reached fairly quickly in the analysis of most protocols.

8.4 Discussion

As in the pi calculus, scoping can be the basis of security properties for processes in the applied pi calculus. Moreover, attackers can be treated as contexts for processes. In our example, the scoping on K_S reflects that it cannot be used by attackers. Principals other than S use K_S only as prescribed in their code, which is given explicitly as part of the process P above.

On the other hand, the added expressiveness of the applied pi calculus enables writing detailed examples, such as this one. Not only we can represent cryptographic operations, but we need not commit to a particular cryptosystem: we can introduce new function symbols and equations as needed. The awareness of such extensibility is far from new: it appears already in Merritt's dissertation [66, page 60]. This extensibility can be a cause of concerns about soundness. Indeed, we may want to have a method for deciding whether a given set of rules captures "enough" properties of an underlying cryptosystem. At present, the most attractive approach to this problem consists in developing complexity-theoretic foundations for formal methods.

The applied pi calculus also gives rise to the possibility that a process may reveal a term that contains a fresh name s without revealing s itself. For instance, in our example, the process reveals an encryption of a session key, by sending this encryption on the public channel e, without necessarily disclosing the session key. This possibility does not arise in the pure pi calculus, where each name is either completely private to a process or completely known to its context. Technically, this possibility is a significant source of complications in reasoning about security in the applied pi calculus. These complications should not be too surprising, however: they reflect the difficulty of reasoning about security protocols.

9 Outlook

The development of new security protocols remains active. As mentioned in Section 3, recent protocols typically have many moving parts—many modes, options, and layers. Their complexity can be a source of serious concerns. Moreover, from time to time, security protocols are used in new contexts, in which their assumptions may not hold exactly. We may therefore conjecture that understanding how to design and analyze security protocols will remain important in the coming years. What new research will be necessary and most fruitful remains open to debate.

The applied pi calculus and its relatives are idealized programming languages. As formal analysis matures, it becomes applicable to more practical programming languages, at least for protocol code written in a stylized manner [21, 48, 51]. For such code, it is possible to translate to the applied pi calculus—more specifically to the dialect understood by ProVerif—and to obtain automatic proofs. We may expect that these stylistic requirements will be relaxed over time. We may also expect that general-purpose static analysis techniques (not specifically developed for security) will be helpful in this progress. Moreover, in light of some of the research described in Section 5, we may expect to obtain not only formal but also complexity-theoretic security results. With this further development, formalisms may ultimately be externally visible neither

in protocol descriptions (which would be in ordinary programming languages) nor in security guarantees.

Acknowledgments

These notes are largely based on joint work with Bruno Blanchet, Mike Burrows, Cédric Fournet, Andy Gordon, and Roger Needham. Bruno Blanchet, Cédric Fournet, and Andy Gordon commented on a draft of these notes. I am grateful to all of them.

I am also grateful to the organizers of the 2006 International School on Foundations of Security Analysis and Design for inviting me to lecture and then for the encouragement to write these notes.

This work was partly supported by the National Science Foundation under Grants CCR-0208800 and CCF-0524078.

References

1. Martín Abadi. Security protocols: Principles and calculi. Lectures at 6th International School on Foundations of Security Analysis and Design, September 2006. Slides at `http://www.sti.uniurb.it/events/fosad06/papers/Abadi-fosad06.pdf`.
2. Martín Abadi and Bruno Blanchet. Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM*, 52(1):102–146, 2005.
3. Martín Abadi and Bruno Blanchet. Computer-assisted verification of a protocol for certified email. *Science of Computer Programming*, 58(1–2):3–27, October 2005.
4. Martín Abadi, Bruno Blanchet, and Cédric Fournet. Just Fast Keying in the pi calculus. *ACM Transactions on Information and System Security*, 2007. To appear.
5. Martín Abadi and Véronique Cortier. Deciding knowledge in security protocols under equational theories. *Theoretical Computer Science*, 367(1–2):2–32, November 2006.
6. Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115, January 2001.
7. Martín Abadi and Cédric Fournet. Private authentication. *Theoretical Computer Science*, 322(3):427–476, September 2004.
8. Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, January 1999. An extended version appeared as Digital Equipment Corporation Systems Research Center report No. 149, January 1998.
9. Martín Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, January 1996.
10. Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (The computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.
11. William Aiello, Steven M. Bellovin, Matt Blaze, Ran Canetti, John Ioannidis, Angelos D. Keromytis, and Omer Reingold. Just Fast Keying: Key agreement in a hostile internet. *ACM Transactions on Information and System Security*, 7(2):242–273, May 2004.
12. Roberto Amadio and Denis Lugiez. On the reachability problem in cryptographic protocols. In *CONCUR 2000: Concurrency Theory*, volume 1877 of *LNCS*, pages 380–395. Springer-Verlag, August 2000.
13. Roberto Amadio and Sanjiva Prasad. The game of the name in cryptographic tables. In *Advances in Computing Science - ASIAN'99*, volume 1742 of *LNCS*, pages 15–27. Springer-Verlag, December 1999.

14. Ross Anderson and Roger Needham. Robustness principles for public key protocols. In *Proceedings of Crypto '95*, volume 963 of *LNCS*, pages 236–247. Springer-Verlag, 1995.
15. Tuomas Aura. Strategies against replay attacks. In *10th IEEE Computer Security Foundations Workshop*, pages 59–68, 1997.
16. Michael Backes, Birgit Pfitzmann, and Michael Waidner. A composable cryptographic library with nested operations. In *10th ACM conference on Computer and Communications security (CCS'03)*, pages 220–230, October 2003.
17. Michael Baldamus, Joachim Parrow, and Björn Victor. Spi calculus translated to pi-calculus preserving may-tests. In *19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*, pages 22–31, July 2004.
18. Mathieu Baudet. *Sécurité des protocoles cryptographiques: aspects logiques et calculatoires*. PhD thesis, Ecole Normale Supérieure de Cachan, 2007.
19. Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In *Advances in Cryptology—CRYPTO '94*, volume 773 of *LNCS*, pages 232–249. Springer-Verlag, 1993.
20. Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. Verifying policy-based security for web services. In *ACM Conference on Computer and Communications Security (CCS'04)*, pages 268–277, October 2004.
21. Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. Verified reference implementations of WS-security protocols. In *Web Services and Formal Methods, Third International Workshop, WS-FM 2006*, volume 4184 of *LNCS*, pages 88–106. Springer-Verlag, 2006.
22. Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Riccardo Pucella. TulaFale: A security tool for web services. In *Formal Methods for Components and Objects (FMCO 2003)*, volume 3188 of *LNCS*, pages 197–222. Springer-Verlag, November 2003.
23. Bruno Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *14th IEEE Computer Security Foundations Workshop*, pages 82–96, June 2001.
24. Bruno Blanchet. From secrecy to authenticity in security protocols. In *Static Analysis, 9th International Symposium (SAS'02)*, volume 2477 of *LNCS*, pages 342–359. Springer-Verlag, September 2002.
25. Bruno Blanchet. Automatic proof of strong secrecy for security protocols. In *2004 IEEE Symposium on Security and Privacy*, pages 86–100, May 2004.
26. Bruno Blanchet. A computationally sound mechanized prover for security protocols. In *2006 IEEE Symposium on Security and Privacy*, pages 140–154, May 2006.
27. Bruno Blanchet, Martín Abadi, and Cédric Fournet. Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 2007. To appear.
28. Bruno Blanchet and Andreas Podelski. Verification of cryptographic protocols: Tagging enforces termination. In *Foundations of Software Science and Computation Structures (FoSSaCS'03)*, volume 2620 of *LNCS*, pages 136–152. Springer-Verlag, April 2003.
29. Bruno Blanchet and David Pointcheval. Automated security proofs with sequences of games. In *CRYPTO'06*, volume 4117 of *Lecture Notes on Computer Science*, pages 537–554. Springer Verlag, August 2006.
30. Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudo random bits. In *23rd Annual Symposium on Foundations of Computer Science (FOCS 82)*, pages 112–117, 1982.
31. C. Bodei, P. Degano, F. Nielson, and H. Nielson. Flow logic for Dolev-Yao secrecy in cryptographic processes. *Future Generation Computer Systems*, 18(6):747–756, 2002.
32. Chiara Bodei. *Security Issues in Process Calculi*. PhD thesis, Università di Pisa, January 2000.

33. Dominique Bolignano. Towards a mechanization of cryptographic protocol verification. In *9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 131–142. Springer-Verlag, 1997.
34. Michele Boreale, Rocco De Nicola, and Rosario Pugliese. Proof techniques for cryptographic processes. *SIAM J. Comput.*, 31(3):947–986, 2001.
35. Johannes Borgström, Sébastien Briais, and Uwe Nestmann. Symbolic bisimulation in the spi calculus. In *CONCUR 2004: Concurrency Theory*, volume 3170 of *LNCS*, pages 161–176. Springer-Verlag, August 2004.
36. Johannes Borgström and Uwe Nestmann. On bisimulations for the spi calculus. In *Algebraic Methodology and Software Technology: 9th International Conference, AMAST 2002*, volume 2422 of *LNCS*, pages 287–303. Springer-Verlag, September 2002.
37. Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989. A preliminary version appeared as Digital Equipment Corporation Systems Research Center report No. 39, February 1989.
38. Anupam Datta, Ante Derek, John C. Mitchell, and Dusko Pavlovic. A derivation system and compositional logic for security protocols. *Journal of Computer Security*, 13(3):423–482, 2005.
39. Anupam Datta, Ante Derek, John C. Mitchell, and Arnab Roy. Protocol composition logic (PCL). *Electronic Notes in Theoretical Computer Science*, 172(1):311–358, April 2007.
40. Richard A. DeMillo, Nancy A. Lynch, and Michael Merritt. Cryptographic protocols. In *14th Annual ACM Symposium on Theory of Computing*, pages 383–400, 1982.
41. Dorothy E. Denning and Giovanni Maria Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(7):533–535, August 1981.
42. Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, March 1983.
43. Luca Durante, Riccardo Sisto, and Adriano Valenzano. A state-exploration technique for spi-calculus testing-equivalence verification. In *Formal Techniques for Distributed System Development, FORTE/PSTV*, volume 183 of *IFIP Conference Proceedings*, pages 155–170. Kluwer, October 2000.
44. Luca Durante, Riccardo Sisto, and Adriano Valenzano. Automatic testing equivalence verification of spi calculus specifications. *ACM Transactions on Software Engineering and Methodology*, 12(2):222–284, April 2003.
45. Riccardo Focardi and Roberto Gorrieri. The compositional security checker: A tool for the verification of information flow security properties. *IEEE Transactions on Software Engineering*, 23(9):550–571, September 1997.
46. Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL protocol: Version 3.0. `http://www.mozilla.org/projects/security/pki/nss/ssl/draft302.txt`, November 1996.
47. Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28:270–299, April 1984.
48. Andrew D. Gordon. Provable implementations of security protocols. In *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 345–346, 2006.
49. Andrew D. Gordon and Alan Jeffrey. Authenticity by typing for security protocols. In *14th IEEE Computer Security Foundations Workshop*, pages 145–159, June 2001.
50. Andrew D. Gordon and Alan Jeffrey. Types and effects for asymmetric cryptographic protocols. In *15th IEEE Computer Security Foundations Workshop*, pages 77–91, June 2002.
51. Jean Goubault-Larrecq and Fabrice Parrennes. Cryptographic protocol analysis on real C code. In *6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *LNCS*, pages 363–379. Springer-Verlag, January 2005.

52. James W. Gray, III, Kin Fai Epsilon Ip, and King-Shan Lui. Provable security for cryptographic protocols—exact analysis and engineering applications. In *10th IEEE Computer Security Foundations Workshop*, pages 45–58, 1997.
53. D. Harkins and D. Carrel. RFC 2409: The Internet Key Exchange (IKE). `http://www.ietf.org/rfc/rfc2409.txt`, November 1998.
54. Hans Hüttel. Deciding framed bisimilarity. In *4th International Workshop on Verification of Infinite-State Systems (INFINITY'02)*, pages 1–20, August 2002.
55. Richard A. Kemmerer, Catherine Meadows, and Jonathan K. Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7(2):79–130, Spring 1994.
56. J. Kohl and C. Neuman. RFC 1510: The Kerberos network authentication service (v5). `ftp://ftp.isi.edu/in-notes/rfc1510.txt`, September 1993.
57. Steve Kremer and Mark D. Ryan. Analysis of an electronic voting protocol in the applied pi calculus. In *Programming Languages and Systems: 14th European Symposium on Programming, ESOP 2005*, volume 3444 of *LNCS*, pages 186–200. Springer-Verlag, April 2005.
58. Butler W. Lampson. Protection. In *5th Princeton Conference on Information Sciences and Systems*, pages 437–443, 1971.
59. Peeter Laud. Symmetric encryption in automatic analyses for confidentiality against active adversaries. In *2004 IEEE Symposium on Security and Privacy*, pages 71–85, May 2004.
60. Peeter Laud. Secrecy types for a simulatable cryptographic library. In *12th ACM Conference on Computer and Communications Security (CCS'05)*, pages 26–35. ACM, November 2005.
61. P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *5th ACM Conference on Computer and Communications Security (CCS'98)*, pages 112–121, 1998.
62. Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *LNCS*, pages 147–166. Springer-Verlag, 1996.
63. Kevin D. Lux, Michael J. May, Nayan L. Bhattad, and Carl A. Gunter. WSEmail: Secure internet messaging based on web services. In *ICWS '05: Proceedings of the IEEE International Conference on Web Services*, pages 75–82, 2005.
64. Nancy Lynch. I/O automaton models and proofs for shared-key communication systems. In *12th IEEE Computer Security Foundations Workshop*, pages 14–29, 1999.
65. Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.
66. Michael J. Merritt. *Cryptographic Protocols*. PhD thesis, Georgia Institute of Technology, February 1983.
67. Daniele Micciancio and Bogdan Warinschi. Soundness of formal encryption in the presence of active adversaries. In *Theory of Cryptography Conference (TCC'04)*, volume 2951 of *LNCS*, pages 133–151. Springer-Verlag, February 2004.
68. S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer. Kerberos authentication and authorization system, Project Athena technical plan, section E.2.1. Technical report, MIT, July 1987.
69. Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.
70. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100:1–40 and 41–77, September 1992.
71. John C. Mitchell, Vitaly Shmatikov, and Ulrich Stern. Finite-state analysis of SSL 3.0. In *7th USENIX Security Symposium*, pages 201–216, January 1998.
72. David Monniaux. Abstracting cryptographic protocols with tree automata. *Science of Computer Programming*, 47(2–3):177–202, 2003.
73. R. M. Needham. Logic and over-simplification. In *13th Annual IEEE Symposium on Logic in Computer Science*, pages 2–3, 1998.

74. Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.
75. Roger M. Needham and Michael D. Schroeder. Authentication revisited. *Operating Systems Review*, 21(1):7, 1987.
76. Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.
77. Ajith Ramanathan, John Mitchell, Andre Scedrov, and Vanessa Teague. Probabilistic bisimulation and equivalence for security analysis of network protocols. In *FOSSACS 2004 - Foundations of Software Science and Computation Structures*, volume 2987 of *LNCS*, pages 468–483. Springer-Verlag, March 2004.
78. Steve Schneider. Security properties and CSP. In *1996 IEEE Symposium on Security and Privacy*, pages 174–187, 1996.
79. Paul Syverson. Limitations on design principles for public key protocols. In *1996 IEEE Symposium on Security and Privacy*, pages 62–73, 1996.
80. F. Javier Thayer Fábrega, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Why is a security protocol correct? In *1998 IEEE Symposium on Security and Privacy*, pages 160–171, May 1998.
81. Thomas Y. C. Woo and Simon S. Lam. Authentication for distributed systems. *Computer*, 25(1):39–52, January 1992.
82. Thomas Y. C. Woo and Simon S. Lam. A lesson on authentication protocol design. *Operating Systems Review*, 28(3):24–37, 1994.
83. Andrew C. Yao. Theory and applications of trapdoor functions. In *23rd Annual Symposium on Foundations of Computer Science (FOCS 82)*, pages 80–91, 1982.