Models and Proofs of Protocol Security: A Progress Report

Martín Abadi[1,2], Bruno Blanchet[3,4,5], and Hubert Comon-Lundh[5,6,7]

[1] Microsoft Research, Silicon Valley
[2] University of California, Santa Cruz
[3] CNRS
[4] École Normale Supérieure
[5] INRIA
[6] École Normale Supérieure de Cachan
[7] Research Center for Information Security, Advanced Industrial Science and Technology

Abstract. This paper discusses progress in the verification of security protocols. Focusing on a small, classic example, it stresses the use of program-like representations of protocols, and their automatic analysis in symbolic and computational models.

1 Introduction

As computer security has become a broad, rich field, rigorous models have been developed for many policies and mechanisms. Sometimes these models have been the subject of formal proofs, even automated ones. The goal of this paper is to discuss some of the progress in this direction and some of the problems that remain.

The paper focuses on the study of security protocols, a large, mature, and active area. It aims to offer an introduction and a partial perspective on this area, rather than a comprehensive survey. We explain notations, results, and tools informally, through the description of a basic example: a variant of the classic Wide-mouthed-frog protocol [25]. For this example, we consider specifications and automated proofs in two formalisms. We refer the reader to the research literature for presentations of other formalisms and for precise definitions and theorems, and to a recent tutorial [1] for additional background.

Current research in this area addresses at least three challenges:

1. the treatment of realistic, practical protocols;
2. the analysis of actual implementation code;
3. extending the analysis to refined models, in particular computational models with complexity-theoretic hypotheses on cryptographic functions.

With regard to (1), protocol analysis appears to be catching up with protocol development. In the last few years there have been increasingly thorough analyses of practical protocols. While these analyses remain laborious and difficult, the sophistication and power of the techniques and tools for protocol analysis seem to have grown faster than the complexity of practical protocols. For instance, in the last dozen years, the

understanding and formal analysis of SSL and its descendant TLS [30] has progressed considerably while this protocol has neither changed substantially nor been replaced (e.g., [12, 35, 42, 44]). In this paper we do not discuss (1) further, although we recognize its importance.

Progress on (2) is more recent and still more tentative, but quite encouraging [32, 34]. Moreover, we believe that much further progress is possible using static analyses and type systems, including ones that are not specific to protocol security. This paper concerns (2) in that it deals with protocols written in little programming languages, namely a dialect of the pi calculus [41] and a special-purpose language for writing cryptographic games. It concerns (2) also in that it relies on tools (ProVerif [15, 16, 18] and CryptoVerif [17, 21]) that can be applied to protocols written in a general-purpose programming language such as F♯ (a dialect of ML) [12, 14, 32].

As for (3), models and the corresponding proofs of security can concern several different levels of abstraction. For instance, at a high level, they may deal with secure communication channels as primitive. At a lower level, they may show how these channels are implemented in terms of cryptographic functions, while treating those as "black boxes". An even lower-level model would describe, in detail, how the cryptographic algorithms transform bitstrings. This lower-level model is however not necessarily the final one: we could also take into account such characteristics as timing and power consumption, which some clever attacks may exploit. In this paper we focus on the relation between "black-box" cryptography, in which cryptographic operations are symbolic, and "computational" cryptography, in which these operations are regarded as computations on bitstrings subject to complexity-theoretic assumptions.

The next section introduces our example informally. Section 3 shows how to code it in a dialect of the pi calculus and how to treat it with the tool ProVerif, symbolically. Section 4 gives a computational counterpart to this symbolic analysis via fairly general soundness results that map symbolic guarantees to computational guarantees (e.g., [7, 26, 40]). As an alternative, Section 5 treats the protocol directly in the computational model, with the tool CryptoVerif. Section 6 concludes.

2 An Example, Informally: The Wide-mouthed-frog Protocol

The Wide-mouthed-frog (WMF) protocol is a classic, simple method for establishing a secure channel via an authentication server. Mike Burrows originally invented it in order to show that two messages suffice for this task, in the 1980s. It became popular as an example in papers on protocol analysis.

The protocol enables two principals A and B to establish a shared session key K_{AB}. They rely on the help of an authentication server S with which they share keys K_{AS} and K_{BS}, respectively. Informally, the protocol goes roughly as follows:

- First, A generates the session key K_{AB}, and sends A, $\{T_A, B, K_{AB}\}_{K_{AS}}$ to S. Here T_A represents a timestamp, and the braces indicate encryption. It is assumed that clocks are synchronized, and that the encryption not only guarantees secrecy but protects the message from tampering.
- The server S can decrypt this message and check its timeliness. It then sends $\{T_S, A, K_{AB}\}_{K_{BS}}$ to B, where T_S is also a timestamp.

– Finally, B can decrypt this message, check its timeliness, and obtain K_{AB}.

The principals A and B trust that S does not divulge K_{AB} nor use it for its own purposes. They also trust S in other ways—for instance, to check timestamps properly. Defining such trust relations precisely has been one of the important goals of work in this area. Because of this trust, A and B can treat K_{AB} as a shared key. Afterwards, A and B may exchange messages directly under K_{AB}.

This simple example brings up a number of issues. In particular, we may ask what exactly is assumed of timestamps? of cryptographic operations? For instance, it is noteworthy that the protocol relies on A to generate a session key. While this may be acceptable, it is a non-trivial assumption that A can invent good shared secrets; in many other protocols, this important task is left for servers.

Formal analyses of the protocol address these and other questions, with various degrees of explicitness. While early analyses emphasized clock synchronization and A's generation of K_{AB}, those aspects of the protocol seem to be less central, or at least more implicit, in later work. This shift should not be too surprising. As Roger Needham has argued, the assumptions and objectives of security protocols are not uniform, and they have changed over the years [43]. Our analysis, below, focuses on other questions, and in particular on the required properties of cryptographic operations.

3 The WMF Protocol in the Pi Calculus

Specifying the WMF protocol or another protocol can be, to some extent, a simple matter of programming. For each role in the protocol (A, B, or S), one writes code that models the actions of a principal that plays this role. We need not write code for the adversary, which we treat as the environment, and over which we typically have a universal quantification. Similarly, we do not write code for principals that pretend to play a role but do not actually follow the protocol, since those principals can be regarded as part of the adversary.

Note that principals and roles are distinct. Indeed, a principal may play multiple roles, for instance being the initiator A in one session and the interlocutor B in a concurrent session. A role is basically a program, while a principal is a host that may run this program, as well as other programs.

The programs can be written in a variety of ways. We have often used process calculi, and in particular two extensions of the pi calculus: the spi calculus and the applied pi calculus [5, 6].

– The basic pi calculus offers facilities for communication on named channels, for parallel composition, and for generating fresh names, which may represent fresh cryptographic keys. For example, $(\nu k).\overline{c}\langle k\rangle) \mid c(x).\overline{d}\langle x\rangle$ is a process that generates a fresh name k and then sends it on the channel c, in parallel with a process that receives a message on c, with formal name x, then forwards it on d. In this small example, one may think of c and d as public channels on which an attacker may also communicate, for instance intercepting k. More generally, public channels are often represented by free names, not bound by ν.

– The extensions of the pi calculus include both data structures and symbolic representations of cryptographic functions. Tupling, encryption, hashing, signatures, and many of their variants can be accommodated. For instance, classically, $\{M\}_k$ may represent the shared-key encryption of the message M under the key k.

This approach has been followed in modeling many protocols (e.g., [3, 4, 9, 13, 18, 19, 37, 39]). Techniques from the programming-language literature, such as typing, have been employed for proofs, sometimes with substantial extensions or variations; special-purpose techniques have also been developed and exploited, as in the tool ProVerif on which we rely below (e.g., [2, 10, 23, 24, 28, 33, 36]). Research on related formalisms includes many similar themes and methods (e.g., [8, 22, 27, 29, 38, 45]).

Over the last decade, this approach to modeling and proving has evolved and matured considerably. Most noticeably, proof techniques have progressed in their power and sophistication. Partly because of this progress, the specifics of modeling protocols has changed as well.

We use the WMF protocol to illustrate this point. The original paper on the spi calculus [6] contains a description of the WMF protocol (with nonce handshakes rather than timestamps). Below, we give a new description of this protocol. The two descriptions differ on many small but often interesting points. In particular, the new description models probabilistic encryption [31], in which the encryption function has a third parameter that serves for randomizing ciphertexts: $\{M\}^r_k$ represents the encryption of M under k with random component r. This random component ensures that an attacker cannot recognize when two different ciphertexts have the same underlying plaintext. The new description is also crafted so as to be within the realm of application of ProVerif, CryptoVerif, and the general soundness results that map symbolic guarantees to computational guarantees.

The WMF Protocol in the Pi Calculus. We represent principal names by parameters like a and b. The role of A may be executed by any principal a, and the role of B by any principal b. We write k_{as} and k_{bs} for the respective keys shared with the server.

The code for the role of A may be given three parameters: an identity a, an identity b, and a key k_{as}. This code first generates a fresh process id *pid*. Since a may run concurrently several copies of the same program, possibly with the same partner, *pid* is useful in distinguishing the different copies. Next, the code generates a new key k, called K_{AB} in the informal description of Section 2. The code then communicates on a public channel c. It sends a triple that contains *pid*, a, and A. (This message and similar ones below do not appear in the informal description of the protocol but are helpful for enabling the application of computational-soundness results.) It also sends a pair that contains a and the ciphertext $\{\langle c_0, b, k\rangle\}^r_{k_{as}}$. Here r is a freshly generated name, and c_0 is a constant that represents the message type. A distinct constant c_1 will tag the message from S to B. The two tags c_0 and c_1, although rudimentary, serve for avoiding confusion between messages and suffice for the properties that we establish.

$$P_A(a, b, k_{as}) \stackrel{\text{def}}{=} (\nu \mathit{pid})(\nu k).\overline{c}\langle\langle \mathit{pid}, a, A\rangle\rangle.(\nu r).\overline{c}\langle\langle a, \{\langle c_0, b, k\rangle\}^r_{k_{as}}\rangle\rangle$$

Note that the messages do not specify destinations. Destinations could be included in message headers, but an attacker could change them anyway, so they have no value from the point of view of security.

As a variant, which we adopt, we may wish to quantify over any possible partner b, letting the environment (that is, the attacker) choose b, thus:

$$P_A(a, k_{as}) \stackrel{\text{def}}{=} (\nu \mathit{pid})(\nu k).\overline{c}\langle\langle \mathit{pid}, a, A\rangle\rangle.$$
$$c(x).\mathit{if}\ \pi_1(x) = \mathit{pid}\ \mathit{then}\ \mathit{let}\ b = \pi_2(x)\ \mathit{in}\ (\nu r).\overline{c}\langle\langle a, \{\langle c_0, b, k\rangle\}^r_{k_{as}}\rangle\rangle$$
$$\mathit{else}\ \mathbf{0}$$

Here a receives a message x and performs some computations and tests on x. Specifically, a tries to retrieve the first component of a (supposed) pair using the projection function π_1, then examines the first component to check that the message is intended for this instance. If the projection fails, then the equality test fails as well. If the equality test fails, then the execution stops. (An alternative could be to restart the program or to wait for another message.) In case of success, on the other hand, b is bound to the second component ($\pi_2(x)$) of the message. Otherwise, the execution stops; **0** is the null process.

In this presentation we take some liberties—all the i's will be dotted for the ProVerif version of the code, which we describe below. In particular, we omit the axioms for encryption and decryption. We also use a shorthand for pattern matching: we write inputs of the form $c(t)$ when t is a term that can supposedly be decomposed by the principal that receives the message. Such matching tests can be desugared to conditionals in a standard way; variables that occur in the term t are parts of the messages that are not checked and they are bound with a *let* construction. With this notation, we can rewrite P_A as follows:

$$P_A(a, k_{as}) \stackrel{\text{def}}{=} (\nu \mathit{pid})(\nu k).\overline{c}\langle\langle \mathit{pid}, a, A\rangle\rangle.c(\langle \mathit{pid}, x\rangle).(\nu r).\overline{c}\langle\langle a, \{\langle c_0, x, k\rangle\}^r_{k_{as}}\rangle\rangle$$

Similarly, we specify the process P_B:

$$P_B(a, b, k_{bs}, m) \stackrel{\text{def}}{=} (\nu \mathit{pid}).\overline{c}\langle\langle \mathit{pid}, b, B\rangle\rangle.\ c(\langle \mathit{pid}, x\rangle).$$
$$\mathit{let}\ \langle c_1, a, y\rangle = \mathit{decrypt}(x, k_{bs})\ \mathit{in}\ (\nu r).\overline{c}\langle\{m\}^r_y\rangle$$

Here m is an arbitrary message that is supposed to remain secret. According to this code, b sends the secret only to a. However, we may want to enable b to interact with any other principal, sending them an appropriate secret, or nothing at all. In order to model this possibility, we simply show another version of the program in which the final payload is not sent:

$$P^1_B(b, k_{bs}) \stackrel{\text{def}}{=} (\nu \mathit{pid}).\overline{c}\langle\langle \mathit{pid}, b, B\rangle\rangle.c(\langle \mathit{pid}, x\rangle).\mathit{let}\ \langle c_1, z, y\rangle = \mathit{decrypt}(x, k_{bs})\ \mathit{in}\ \mathbf{0}$$

Finally, we specify the process P_S:

$$P_S(a, b, k_{as}, k_{bs}) \stackrel{\text{def}}{=} (\nu \mathit{pid}).\overline{c}\langle\langle \mathit{pid}, S\rangle\rangle.c(\langle \mathit{pid}, a, x\rangle).$$
$$\mathit{let}\ \langle c_0, b, y\rangle = \mathit{decrypt}(x, k_{as})\ \mathit{in}\ (\nu r).\overline{c}\langle\{\langle c_1, a, y\rangle\}^r_{k_{bs}}\rangle$$

We also consider two variants of P_S in which one of the protocol participants A and B is compromised.

$$\begin{aligned}
P_S^1(a,b,k_{bs}) &\stackrel{\text{def}}{=} (\nu \mathit{pid}).\bar{c}\langle\langle \mathit{pid}, S\rangle\rangle.c(\langle \mathit{pid}, k\rangle).c(\langle \mathit{pid}, z, x\rangle).\mathit{if}\ z \neq a \wedge z \neq b\ \mathit{then} \\
&\quad \mathit{let}\ \langle c_0, b, y\rangle = \mathit{decrypt}(x,k)\ \mathit{in}\ (\nu r).\bar{c}\langle\{\langle c_1, z, y\rangle\}^r_{k_{bs}}\rangle \\
P_S^2(a,b,k_{as}) &\stackrel{\text{def}}{=} (\nu \mathit{pid}).\bar{c}\langle\langle \mathit{pid}, S\rangle\rangle.c(\langle \mathit{pid}, k\rangle).c(\langle \mathit{pid}, a, x\rangle). \\
&\quad \mathit{let}\ \langle c_0, z, y\rangle = \mathit{decrypt}(x,k_{as})\ \mathit{in} \\
&\quad \mathit{if}\ z \neq a \wedge z \neq b\ \mathit{then}\ (\nu r).\bar{c}\langle\{\langle c_1, a, y\rangle\}^r_k\rangle
\end{aligned}$$

We represent a corrupted principal by letting S get its key from the environment. The case in which both A and B are compromised is less interesting, because in that case S can be simulated by the environment entirely. (Similarly, we do not specify corrupted versions of P_A or P_B, because they can be simulated by the environment.)

We assemble these definitions, letting a, b, and the server run any number of copies of their respective programs (for simplicity with a single parameter m):

$$\begin{aligned}
P(m) \stackrel{\text{def}}{=}\ & (\nu k_{as})(\nu k_{bs}).((!P_A(a,k_{as})) \mid (!P_A(b,k_{bs})) \mid \\
& (!P_B(a,b,k_{bs},m)) \mid (!P_B(b,a,k_{as},m)) \mid (!P_B(a,a,k_{as},m)) \mid (!P_B(b,b,k_{bs},m)) \mid \\
& (!P_B^1(b,k_{bs})) \mid (!P_B^1(a,k_{as})) \mid \\
& (!P_S(a,b,k_{as},k_{bs})) \mid (!P_S(b,a,k_{bs},k_{as})) \mid \\
& (!P_S(a,a,k_{as},k_{as})) \mid (!P_S(b,b,k_{bs},k_{bs})) \mid \\
& (!P_S^1(a,b,k_{bs})) \mid (!P_S^1(b,a,k_{as})) \mid (!P_S^2(a,b,k_{as})) \mid (!P_S^2(b,a,k_{bs})))
\end{aligned}$$

Here ! is the replication operator, so $!P$ behaves like an unbounded number of copies of P in parallel; formally, $!P \equiv P \mid !P$. The names k_{as} and k_{bs} are bound. This binding manifests an important feature of the process calculus: such a construction hides the names, which are not visible outside their scope. The process P therefore expresses that k_{as} and k_{bs} are not a priori known outside, unless they are leaked on a public channel.

The process calculus and ProVerif also allow more compact and more convenient representations of P, as well as many variants and elaborations. We rely on the definitions above partly because we wish to match the conditions of the computational-soundness results. For instance, we avoid the use of functions that link keys to principal names (which are common in ProVerif models, but which appear to be computationally unsound), and also the use of private channels (which may be permitted by ongoing work on computational soundness). As research in this area progresses further, we anticipate that those results will be increasingly flexible and general.

The WMF protocol has several standard security properties. In particular, it preserves the secrecy of the payload m. Formally, this secrecy can be expressed as an observational equivalence: $P(m) \sim P(m')$, for all m and m'. It holds even in the case where m and m' are not atomic names, and it precludes even the leaking of partial information about m and m'. For these reasons, this property is sometimes called “strong secrecy”.

As we show below, ProVerif offers one particularly effective method for establishing such security properties. There are others, sometimes relying in part on techniques from the pi calculus (as in [6], for instance).

The WMF Protocol in ProVerif. The WMF protocol can be programmed much as above in the input language of ProVerif. In this language, the encryption $\{m\}_k^r$ is written $\mathsf{encrypt}(m, k, r)$. Encryption and decryption are declared in ProVerif by:

$\mathbf{fun}\ \mathsf{encrypt}/3.$
$\mathbf{reduc}\ \mathsf{decrypt}(\mathsf{encrypt}(x, y, r), y) = x.$

which introduces a function symbol encrypt of arity 3 and a function symbol decrypt defined by a rewrite rule $\mathsf{decrypt}(\mathsf{encrypt}(x, y, r), y) \to x$, which means that decryption of a ciphertext with the correct key yields the plaintext. Furthermore, we add a function symbol keyeq that allows the adversary to test equality between keys of two ciphertexts:

$\mathbf{reduc}\ \mathsf{keyeq}(\mathsf{encrypt}(x, y, r), \mathsf{encrypt}(x', y, r')) = \mathsf{true}.$

This function symbol models that the encryption scheme is not key-concealing (so the computational-soundness result of Section 4 can be applied without assuming that encryption is key-concealing).

At the level of processes, the input language of ProVerif is an ASCII syntax for the applied pi calculus. For example, the process $P_A(a, k_{as})$ is coded:

$\mathbf{let}\ processAa =$
$\quad \mathbf{new}\ pid;\ \mathbf{out}(\mathsf{c}, (pid, \mathsf{a}, \mathsf{A}));\ \mathbf{in}(\mathsf{c}, (= pid, xb));$
$\quad \mathbf{new}\ Kab;\ \mathbf{new}\ r;\ \mathbf{out}(\mathsf{c}, (\mathsf{a}, \mathsf{encrypt}((\mathsf{c0}, xb, Kab), Kas, r))).$

The language uses **new** for ν, $\mathbf{out}(c, m)$ for $\overline{c}\langle m\rangle$, and $\mathbf{in}(c, m)$ for $c(m)$. The syntax of patterns is made more explicit, by adding an equality sign (as in $= pid$, for example) when making a comparison with a known value. Other minor departures from the definition of $P_A(a, k_{as})$ above are changes in the identifiers.

The other processes that represent the WMF protocol are coded in a similar way in ProVerif. We therefore omit their ProVerif versions.

ProVerif can establish the security property claimed above, using the technique described in [16]. The proof is fully automatic. For a large class of protocols in which messages are tagged, ProVerif is guaranteed to terminate [20]; our example does not quite fit in this class (in particular, because of the use of inequality tests), but ProVerif does terminate nonetheless. ProVerif can similarly establish security properties of many more complex protocols.

4 Computational Soundness

While formal analysis of protocols has traditionally provided only formal results, like those stated in Section 3, the exact status of those results can be unclear. Do they entail any actual guarantees, or is formal analysis valuable only as a means of identifying assumptions, explaining protocols, and sometimes finding mistakes?

One approach to addressing such questions is to try to map the formal results to a more concrete model via a general theorem. Such a theorem should enable us to leverage high-level notations, proof techniques, and proof tools for obtaining guarantees for large families of protocols. In this section we discuss this approach and how it applies to the WMF protocol.

In the more concrete model, cryptographic operations work on bitstrings, not expressions, and are subject to standard complexity-theoretic assumptions. Names are interpreted as bitstrings—more precisely, ensembles of probability distributions on bitstrings, parameterized by a security parameter η. This interpretation is extended to a mapping from symbolic expressions to bitstrings. The adversary may perform any computation on bitstrings, and not only the basic expected cryptographic operations; the adversary is however constrained to run in (probabilistic) polynomial time with respect to η.

Computational-soundness theorems translate symbolic guarantees to computational guarantees. In particular, a recent computational-soundness theorem [26], on which we rely, roughly says that the symbolic equivalence of two processes implies their computational indistinguishability. In other words, the distinguishing capabilities of a computational attacker are not stronger than the distinguishing capabilities of the symbolic attacker, whose range of operations is much more limited. Of course, these theorems also indicate assumptions, in particular hypotheses on cryptographic operations. Unexpected but necessary hypotheses sometimes surface when one proves soundness theorems.

Assumptions. Specifically, the theorem of [26] requires some assumptions on the encryption scheme:

- IND-CPA security (the standard semantic guarantee for secrecy [31], also called "type-3 security" [7]), and
- INT-CTXT security (an authentication guarantee [11]).

It also requires that:

- the attacker can create a key only using the key-generation algorithm;
- there are no encryption cycles: there is an ordering $<$ on private keys such that, if $k < k'$, then k may appear in the plaintext of a ciphertext encrypted under k', but not the converse;
- finally, it is possible to compute a symbolic representation of any bitstring—this is a "parsing assumption".

The assumptions are far from trivial: IND-CPA is standard, but INT-CTXT is strong, and the inability to create keys without following the key-generation algorithms is quite unusual. These three properties are however necessary for the soundness theorem: if one of these three hypotheses fails, we can find protocols that appear secure symbolically but that are not secure computationally, under some encryption schemes. In fact, under some encryption schemes that do not satisfy INT-CTXT, there are computational attacks on the WMF protocol in particular.

Encryption cycles have attracted a great deal of attention in recent years, in part because of computational-soundness theorems, but they are of independent interest. Their exact status remains a well-known open question.

The parsing assumption is probably not necessary, but eases the proofs.

Application to WMF. Since $P(m) \sim P(m')$ has been proved using ProVerif, we should get some computational indistinguishability guarantee, thanks to the theorem of [26] discussed above. That theorem pertains to a symbolic equivalence relation $\sim_s$ that distinguishes slightly more processes than $\sim$ and ProVerif. For instance, $\sim_s$ distinguishes two symbolic messages whose computational interpretations have distinct lengths, while $\sim$ may not.

This discrepancy illustrates that further work is needed for establishing a perfect match between models. Moreover, the soundness theorems remain hard to establish and they do not yet cover all useful cryptographic primitives, nor all sensible styles for writing protocol code.

The discrepancy might be resolved by refining $\sim$ by introducing functions that, given a ciphertext, reveal the length, structure, or other properties of the underlying plaintext. Such functions could also be incorporated in ProVerif analyses.

For our specific example, more simply, we may require that encryption conceal the length of payloads, and we can weaken $\sim_s$ accordingly. This approach is acceptable for the WMF protocol since its messages can be assumed to have a constant length. In this case, $\sim$ and $\sim_s$ coincide, so the ProVerif verification actually establishes $P(m) \sim_s P(m')$. Moreover, we have proved manually the absence of encryption cycles so, for implementations that satisfy the other assumptions of the computational-soundness theorem, we obtain the desired computational indistinguishability guarantee.

5 The WMF Protocol in CryptoVerif

In this section, we study the WMF protocol using CryptoVerif. In contrast to the approach of Section 4, CryptoVerif works directly in the computational model, and provides proofs by sequences of games, like those constructed manually by cryptographers. In these proofs, one starts from an initial game that represents the protocol under study. This game is then transformed either by relying on security assumptions on cryptographic primitives or by syntactic transformations. These transformations are such that the difference of probability of success of an attack in consecutive games is negligible. The final game is such that the desired security property is obvious from the form of the game. One can then conclude that the security property also holds in the initial game.

The WMF Protocol in CryptoVerif. In order to automate this technique, the games are formalized in a process calculus, as we illustrate on the WMF protocol. Throughout this section, we refer to a and b as honest principals, and we focus on them in writing code. The adversary can play the role of dishonest principals.

The following process P_A models the role of A:

$$P_A = !^N c_2(xA : host, xB : host);\ \mathbf{if}\ xA = \mathsf{a} \vee xA = \mathsf{b}\ \mathbf{then}$$
$$\quad \mathbf{let}\ KAs = (\mathbf{if}\ xA = \mathsf{a}\ \mathbf{then}\ Kas\ \mathbf{else}\ Kbs)\ \mathbf{in}$$
$$\quad \mathbf{new}\ rKab : keyseed;\ \mathbf{let}\ Kab : key = \mathsf{kgen}(rKab)\ \mathbf{in}$$
$$\quad \mathbf{new}\ r : seed;\ \overline{c_3}\langle xA, \mathsf{encrypt}(\mathsf{concat}(\mathsf{c0}, xB, Kab), KAs, r)\rangle$$

The process P_A starts with a replication bounded by N, which is assumed to be polynomial in the security parameter: at most N copies of A can be run. Two host names

are then received on channel c_2: xA is the name of the host playing the role of A, xB is the name of its interlocutor; xA is required to equal a or b. Then KAs is defined as the key of xA. The protocol proper starts at this point: P_A chooses a fresh key Kab to be shared between xA and xB by generating a random seed $rKab$ ($\mathbf{new}\ rKab : keyseed$) and applying the key-generation algorithm kgen. Next, P_A forms the first message, and sends it on channel c_3. The function concat builds the plaintext to be encrypted by concatenating its arguments (a tag, a host name, and a key). After the output on c_3, control returns to the adversary.

Variables are typed. These types simply represent sets of bitstrings, and have no security meaning. They are still necessary in the computational model, in particular when generating random numbers: the random numbers can be drawn from various sets (keys, random seeds, nonces, ...).

The messages are each sent or received on a distinct channel c_j. Furthermore, the replication $!^N$ implicitly defines an index $i \in [1, N]$, and the channel names c_j are in fact abbreviations for $c_j[i]$, so that a distinct channel is used in each copy of the process. Thus, the adversary knows exactly to which process it is talking. Using distinct channel names and replication indices replaces the process identifiers (pid) of the model of Section 3.

The following process P_B represents the role of B:

$$\begin{array}{l} P_B = !^N c_8(xB : host);\ \mathbf{if}\ xB = \mathsf{a} \vee xB = \mathsf{b}\ \mathbf{then} \\ \quad \mathbf{let}\ KBs = (\mathbf{if}\ xB = \mathsf{a}\ \mathbf{then}\ Kas\ \mathbf{else}\ Kbs)\ \mathbf{in}\ \overline{c_9}\langle\rangle; \\ \quad c_{10}(x : bitstring);\ \mathbf{let}\ \mathsf{injbot}(\mathsf{concat}(= \mathsf{c1}, xA, kab)) = \mathsf{decrypt}(x, KBs)\ \mathbf{in} \\ \quad \mathbf{if}\ xA = \mathsf{a} \vee xA = \mathsf{b}\ \mathbf{then} \\ \quad \mathbf{new}\ r : seed;\ \overline{c_{11}}\langle \mathsf{encrypt}(\mathsf{pad}(mpayload), kab, r)\rangle \end{array}$$

Similarly to P_A, the process P_B is replicated, and expects as first message its own identity xB; xB is required to equal a or b and KBs is its key. Then a message (normally from the server) is received on channel c_{10}, and P_B decrypts this message. The decryption can succeed or fail. When it succeeds, it returns a normal bitstring; when it fails, it returns $\perp$. The function injbot is the natural injection from bitstrings to bitstrings union $\perp$, so that when $\mathsf{injbot}(y) = \mathsf{decrypt}(x, KBs)$, the decryption succeeded and its value is y. Next, when the interlocutor xA of xB is honest, the process P_B encrypts the payload $mpayload$ under the shared key kab and sends the ciphertext on channel c_{11}. (The function pad is only a type conversion function, which converts payloads to plaintexts; it leaves the bitstrings unchanged.)

The process P_K is a key-registration process:

$$\begin{array}{l} P_K = !^{N_2} c_{12}(h : host, k : key); \\ \quad \mathbf{let}\ Khs : key = \mathbf{if}\ h = \mathsf{a}\ \mathbf{then}\ Kas\ \mathbf{else\ if}\ h = \mathsf{b}\ \mathbf{then}\ Kbs\ \mathbf{else}\ k \end{array}$$

All variables defined under replications in CryptoVerif are implicitly arrays indexed by the replication index. So, here, P_K stands for:

$$\begin{array}{l} P_K = !^{i \leq N_2} c_{12}[i](h[i] : host, k[i] : key); \\ \quad \mathbf{let}\ Khs[i] : key = \mathbf{if}\ h[i] = \mathsf{a}\ \mathbf{then}\ Kas\ \mathbf{else\ if}\ h[i] = \mathsf{b}\ \mathbf{then}\ Kbs\ \mathbf{else}\ k[i] \end{array}$$

In order to register a key k_1 for host h_1, the adversary sends a pair (h_1, k_1) on channel $c_{12}[i]$ for some i. The host name h_1 is stored in $h[i]$ while the key k_1 is stored in $Khs[i]$,

except when h_1 is a or b; in this case, the key Kas or Kbs respectively is stored instead, so that the only keys that can be registered for a and b are Kas and Kbs respectively. In order to retrieve the key for host h', one can then look for an index u' such that $h[u'] = h'$; the key for h' is $Khs[u']$. This is done by the construct $\mathbf{find}\ u' \leq N_2\ \mathbf{suchthat\ defined}(Khs[u'], h[u']) \wedge h' = h[u']\ \mathbf{then} \ldots Khs[u'] \ldots$ used below.

The role of the server is specified by the process P_S:

$$
\begin{array}{l}
P_S = !^N c_6(xA : host, x : bitstring);\\
\quad \mathbf{find}\ uA \leq N_2\ \mathbf{suchthat\ defined}(Khs[uA], h[uA]) \wedge xA = h[uA]\ \mathbf{then}\\
\quad \mathbf{let}\ KAs = Khs[uA]\ \mathbf{in}\\
\quad \mathbf{let}\ \mathsf{injbot}(\mathsf{concat}(= \mathsf{c0}, xB, kab)) = \mathsf{decrypt}(x, KAs)\ \mathbf{in}\\
\quad \mathbf{find}\ uB \leq N_2\ \mathbf{suchthat\ defined}(Khs[uB], h[uB]) \wedge xB = h[uB]\ \mathbf{then}\\
\quad \mathbf{let}\ KBs = Khs[uB]\ \mathbf{in}\\
\quad \mathbf{new}\ r : seed;\ \overline{c_7}\langle \mathsf{encrypt}(\mathsf{concat}(\mathsf{c1}, xA, kab), KBs, r)\rangle
\end{array}
$$

The first message of the protocol is received on channel c_6. The variable KAs is set to the key of xA. Then the server decrypts the message with KAs, sets KBs to the key of xB, and finally outputs the second message of the protocol on channel c_7.

The following process P receives two payloads $m0$ and $m1$, chooses a bit $switch$, and sets the payload $mpayload$ to be encrypted by P_B to either $m0$ or $m1$ depending on the value of $switch$. (We will show that the adversary cannot distinguish $switch$ from a fresh random bit, so it cannot distinguish whether the encrypted payload is $m0$ or $m1$.) Next, P generates the keys Kas and Kbs for a and b respectively, using the key-generation algorithm kgen; then it launches processes for the various roles of the protocol and for key registration:

$$
\begin{array}{l}
P = c_{13}(m0 : payload, m1 : payload);\\
\quad \mathbf{new}\ switch : bool; \mathbf{let}\ mpayload : payload = \mathsf{test}(switch, m0, m1)\ \mathbf{in}\\
\quad \mathbf{new}\ rKas : keyseed; \mathbf{let}\ Kas : key = \mathsf{kgen}(rKas)\ \mathbf{in}\\
\quad \mathbf{new}\ rKbs : keyseed;\ \mathbf{let}\ Kbs : key = \mathsf{kgen}(rKbs)\ \mathbf{in}\ \overline{c_{14}}\langle\rangle; (P_A \mid P_B \mid P_S \mid P_K)
\end{array}
$$

Here test is defined by $\mathsf{test}(\mathsf{true}, m0, m1) = m0$ and $\mathsf{test}(\mathsf{false}, m0, m1) = m1$.

Assumptions. In addition to these processes, the CryptoVerif model also specifies several hypotheses:

- The encryption scheme is IND-CPA and INT-CTXT.
- The function concat returns bitstrings of constant length. Moreover, concat is injective, and it is possible to compute x, y, z from $\mathsf{concat}(x, y, z)$ in polynomial time.
- All payloads have the same length.

We do not assume that the attacker can create a key only using the key-generation algorithm. This contrasts with the assumptions of Section 4, which apply to a large class of protocols, including protocols for which there would be computational attacks without this assumption. Neither do we assume the absence of encryption cycles; however, the success of the game transformation sequence shows that there is a key hierarchy. Finally, we do not have any parsing assumption.

Analysis. With the model presented above, CryptoVerif is not able to complete the proof of the desired properties. Manual inspection of the games computed by CryptoVerif shows that, in P_A, it fails to distinguish automatically the cases in which the key Kab is generated for an honest interlocutor a or b from the cases in which it is generated for a dishonest interlocutor. The code can easily be modified to make this distinction from the start, simply by adding the test "**if** $xB = \mathsf{b} \vee xB = \mathsf{a}$ **then**" just before the generation of $rKab$ and duplicating the rest of the process P_A. With this modification, the proof of secrecy of $switch$ succeeds automatically. That is, the adversary cannot distinguish $switch$ from a fresh random bit, so it cannot tell whether the encrypted payload is $m0$ or $m1$.

Additionally, CryptoVerif can also show secrecy properties of the key exchanged between A and B, after removal of the payload message. (We do not present the corresponding process for brevity.) More precisely, CryptoVerif shows that the keys Kab chosen by P_A when xA and xB are honest principals are secret, that is, indistinguishable from fresh independent random keys. However, CryptoVerif cannot show the secrecy of the keys kab received by P_B when xA and xB are honest principals. This failure is not due to a limitation of CryptoVerif, but to an attack: by replaying messages in the protocol, the adversary can force several sessions of B to use the same key kab. Hence, those keys kab may not be independent. CryptoVerif still establishes what we call "one-session secrecy", that is, that each key kab (for xA and xB honest) is indistinguishable from a fresh random key.

The Sequence of Games (Summary). In order to establish the secrecy of $switch$, CryptoVerif successively reduces the original game to simpler games, using the security assumptions. In a first step, it performs syntactic transformations to make explicit all usages of the key Kbs and to replace it with its value $\mathsf{kgen}(rKbs)$. The obtained game is then transformed using the INT-CTXT assumption: CryptoVerif replaces every decryption of a message M under Kbs with a look-up that searches for M among all ciphertexts built by encryption under Kbs. If the ciphertext M is found, the look-up returns the corresponding plaintext; otherwise, decryption fails and the look-up returns $\perp$. If the attacker wins the game before this transformation, then either it wins the new game or, at some point, it has been able to forge an encryption under Kbs. In the latter case, it would break INT-CTXT. Then, CryptoVerif replaces any plaintext M that is encrypted under Kbs with $Z(M)$, a bitstring of the same length as M but consisting only of zeroes. This time, if the attacker wins the game before this transformation, then either it wins the new game or it wins an IND-CPA game.

CryptoVerif performs similar transformations for the key Kas.

At this stage, the key Kab no longer occurs as a plaintext. CryptoVerif now applies the same transformations as above, for this key, and finally replaces all payloads $mpayload$ encrypted under Kab with the same plaintext $Z(mpayload)$. The final game is trivial: it cannot be won by an attacker.

6 Conclusion

Model refinements such as those that we discuss in this paper, while numerous and varied, should not be fundamentally surprising. After all, reasoning about software

and hardware correctness often employs similar refinements. Furthermore, in any area, models and the corresponding proofs may be incomplete and inaccurate.

Security, however, is different in at least one important respect: an adversary may be doing its best to undermine the validity of the models. This specificity increases the importance of understanding refinements, and the interest of the corresponding theory. Within this domain, we believe that the transition from symbolic to computational models is particularly worthwhile. It can serve for strengthening the foundations of formal analysis, for enabling proofs, and also for indicating implicit hypotheses and subtle flaws.

It remains open to debate whether computational results should be obtained directly, with a tool such as CryptoVerif, or indirectly from symbolic proofs via soundness theorems. Soundness theorems often require more hypotheses: there are situations in which a computational proof can be obtained using CryptoVerif, while the hypotheses of soundness theorems are not met. However, when the hypotheses are satisfied, a symbolic proof suffices, and is generally easier to obtain, often automatically.

At present, both avenues still present challenges. ProVerif, CryptoVerif, and the soundness theorems all still have important limitations. These imply, for instance, that one should be careful in writing protocol specifications—not all equivalent formulations are equally easy to handle. Despite these limitations, as this paper illustrates, the progress to date is substantial.

Acknowledgments

We are grateful to our coauthors Véronique Cortier, Cédric Fournet, Andy Gordon, David Pointcheval, and Phil Rogaway. They are responsible for a good part of the ideas, techniques, and results presented in this paper. This work was partly supported by the ANR project FormaCrypt.

References

1. Martín Abadi. Security protocols: Principles and calculi. In Alessandro Aldini and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design IV, FOSAD 2006/2007 Tutorial Lectures*, volume 4677 of *LNCS*, pages 1–23. Springer-Verlag, 2007.
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*, 10(3):1–59, 2007.
5. Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *28th ACM Symposium on Principles of Programming Languages*, pages 104–115, 2001.
6. 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.
7. Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (The computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.

8. 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, 1999.
9. Michael Backes, Matteo Maffei, and Dominique Unruh. Zero-knowledge in the applied pi-calculus and automated verification of the direct anonymous attestation protocol. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 202–215, 2008.
10. Mathieu Baudet. *Sécurité des protocoles cryptographiques: aspects logiques et calculatoires*. PhD thesis, Ecole Normale Supérieure de Cachan, 2007.
11. Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Journal of Cryptology*, 21(4):469–491, 2008.
12. Karthikeyan Bhargavan, Cédric Fournet, Ricardo Corin, and Eugen Zalinescu. Cryptographically verified implementations for TLS. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 459–468, 2008.
13. Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. Verifying policy-based security for web services. In *ACM Conference on Computer and Communications Security*, pages 268–277, 2004.
14. 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.
15. Bruno Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *14th IEEE Computer Security Foundations Workshop*, pages 82–96, 2001.
16. Bruno Blanchet. Automatic proof of strong secrecy for security protocols. In *2004 IEEE Symposium on Security and Privacy*, pages 86–100, 2004.
17. Bruno Blanchet. A computationally sound mechanized prover for security protocols. In *2006 IEEE Symposium on Security and Privacy*, pages 140–154, 2006.
18. Bruno Blanchet, Martín Abadi, and Cédric Fournet. Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 75(1), February-March 2008.
19. Bruno Blanchet and Avik Chaudhuri. Automated formal analysis of a protocol for secure file sharing on untrusted storage. In *2008 IEEE Symposium on Security and Privacy*, pages 417–431, 2008.
20. Bruno Blanchet and Andreas Podelski. Verification of cryptographic protocols: Tagging enforces termination. In *Foundations of Software Science and Computation Structures*, volume 2620 of *LNCS*, pages 136–152. Springer-Verlag, 2003.
21. Bruno Blanchet and David Pointcheval. Automated security proofs with sequences of games. In *CRYPTO'06*, volume 4117 of *LNCS*, pages 537–554. Springer Verlag, 2006.
22. Chiara Bodei. *Security Issues in Process Calculi*. PhD thesis, Università di Pisa, January 2000.
23. Michele Boreale, Rocco De Nicola, and Rosario Pugliese. Proof techniques for cryptographic processes. *SIAM Journal on Computing*, 31(3):947–986, 2001.
24. 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, 2004.
25. 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.
26. Hubert Comon-Lundh and Véronique Cortier. Computational soundness of observational equivalence. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 109–118, 2008.

27. 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.
28. 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.
29. 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.
30. 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.
31. Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28:270–299, April 1984.
32. Andrew D. Gordon. Provable implementations of security protocols. In *21st Annual IEEE Symposium on Logic in Computer Science*, pages 345–346, 2006.
33. Andrew D. Gordon and Alan Jeffrey. Types and effects for asymmetric cryptographic protocols. In *15th IEEE Computer Security Foundations Workshop*, pages 77–91, 2002.
34. Jean Goubault-Larrecq and Fabrice Parrennes. Cryptographic protocol analysis on real C code. In *6th International Conference on Verification, Model Checking and Abstract Interpretation*, volume 3385 of *LNCS*, pages 363–379. Springer-Verlag, 2005.
35. Changhua He, Mukund Sundararajan, Anupam Datta, Ante Derek, and John C. Mitchell. A modular correctness proof of IEEE 802.11i and TLS. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 2–15, 2005.
36. Hans Hüttel. Deciding framed bisimilarity. In *4th International Workshop on Verification of Infinite-State Systems*, pages 1–20, 2002.
37. 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*, volume 3444 of *LNCS*, pages 186–200. Springer-Verlag, 2005.
38. 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.
39. 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.
40. Daniele Micciancio and Bogdan Warinschi. Soundness of formal encryption in the presence of active adversaries. In *Theory of Cryptography Conference*, volume 2951 of *LNCS*, pages 133–151. Springer-Verlag, 2004.
41. 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.
42. Paul Morrissey, Nigel P. Smart, and Bogdan Warinschi. A modular security analysis of the TLS handshake protocol. In Josef Pieprzyk, editor, *ASIACRYPT*, volume 5350 of *LNCS*, pages 55–73. Springer-Verlag, 2008.
43. Roger M. Needham. The changing environment for security protocols. *IEEE Network*, 11(3):12–15, May/June 1997.
44. Lawrence C. Paulson. Inductive analysis of the internet protocol TLS. *ACM Transactions on Information and System Security*, 2(3):332–351, 1999.
45. Ajith Ramanathan, John Mitchell, Andre Scedrov, and Vanessa Teague. Probabilistic bisimulation and equivalence for security analysis of network protocols. In *Foundations of Software Science and Computation Structures*, volume 2987 of *LNCS*, pages 468–483. Springer-Verlag, 2004.