# From Computationally-Proved Protocol Specifications to Implementations and Application to SSH*

David Cadé and Bruno Blanchet
INRIA Paris-Rocquencourt, Paris, France
{david.cade,bruno.blanchet}@inria.fr

**Abstract**

This paper presents a novel technique for obtaining implementations of security protocols, proved secure in the computational model. We formally specify the protocol to prove, we prove this specification using the computationally-sound protocol verifier CryptoVerif, and we automatically translate it into an implementation in OCaml using a new compiler that we have implemented. We applied this approach to the SSH Transport Layer protocol: we proved the authentication of the server and the secrecy of the session keys in this protocol and verified that the generated implementation successfully interacts with OpenSSH. We explain these proofs, as well as an extension of CryptoVerif needed for the proof of secrecy of the session keys. The secrecy of messages sent over the SSH tunnel cannot be proved due to known weaknesses in SSH with CBC-mode encryption.

## 1 Introduction

The verification of security protocols is an important research area since the 1990s: the design of security protocols is notoriously error-prone, and security errors cannot be detected by testing since they appear only in the presence of a malicious adversary. There are basically two models for protocol verification. The symbolic model, also called Dolev-Yao model, represents messages as terms in a term algebra, and the attacker can only create terms in this algebra. The computational model represents messages as bitstrings, and the attacker is a polynomial-time probabilistic Turing machine. This model is more complex than the symbolic one, but models reality much more faithfully. Furthermore, even if the protocol specification is proved secure in such a model, errors can appear at the implementation level: errors may occur in implementation details left unspecified in the specification, or the specification may not be correctly implemented. It is therefore important to make sure that the implementation is secure, and not only the specification. Hence our goal is to obtain protocol implementations proved secure in the computational model.

There are two ways of obtaining a secure protocol implementation. First, one can take an existing implementation, analyze it to obtain a specification of the implemented protocol, and then prove that this specification is secure. Second, one can prove secure the protocol specification we want to implement, and then generate an implementation from it. We chose the second way for multiple reasons. First, being sure that the protocol specification—the foundation on which the implementation is built—is correct before trying to implement it seems to be better than the opposite. Second, generating protocol implementations is relatively easier than analyzing them; analyzing existing protocol implementations not written with verification in mind is especially difficult, and very few methods can do that (see related work below).

Hence we start from a protocol specification. We prove it secure in the computational model using CryptoVerif [14, 13, 15], a protocol verifier in the computational model that can prove authentication and secrecy properties. The generated proofs are sequences of games, like the manual proofs written by cryptographers: the first game is the protocol specification, two consecutive games are distinguishable

*This paper is an extended version of the work originally presented at the 6th International Conference on Availability, Reliability and Security (ARES'12), Prague, Czech Republic, August 2012 [16].

only with negligible probability, and the desired security property is obviously true in the last game. So, the desired property is true in the initial game with overwhelming probability. CryptoVerif also provides a formula expressing the probability of success of an attack against the protocol, as a function of the probability of breaking each primitive. The input file we give to this tool contains functional assumptions (e.g., the decryption of the ciphertext with the correct key yields the plaintext) and security assumptions (e.g., the encryption is IND-CPA) on the cryptographic primitives, a specification of the protocol to verify in a probabilistic process calculus, and the security properties to prove.

We wrote a compiler that translates a CryptoVerif protocol specification into an implementation in OCaml (`http://caml.inria.fr`). We chose this language as target for our compiler because it has a clean semantics and is memory safe, which helps prove the compiler correct. OCaml is a functional language, which facilitates the compilation because the CryptoVerif specification uses oracles that can easily be translated into functions. A cryptographic library is available for OCaml, Cryptokit (`http://forge.ocamlcore.org/projects/cryptokit/`). Our approach could obviously be adapted to other target languages, such as Java or C, if desired. We added annotations to the CryptoVerif input language in order to specify implementation details. These annotations specify how to divide a protocol into several executable programs. For example, most key exchange protocols are divided into key generation, client, and server. The annotations also specify how each cryptographic primitive is implemented. The implementation of these primitives must satisfy the assumptions made in the specification.

We proved in [17] that this compiler preserves security. More precisely, we proved that, under assumptions that formalize the assumptions mentioned at the end of Section 4, when an adversary has probability $p$ of breaking a security property in the generated code, then there also exists an adversary that breaks this property with the same probability in the specification from which the generated code comes. The main difficulty in this proof is that many technical details of the CryptoVerif and OCaml semantics need to be taken into account.

We applied our approach to the SSH Transport Layer Protocol. We crafted a CryptoVerif specification of this protocol. We then used CryptoVerif to obtain automatically a proof of the authentication of the server to the client, and manually a proof of secrecy of the generated session keys. This proof required us to develop a new extension of CryptoVerif to be able to distinguish cases depending on the order of definition of variables. We then applied our compiler to this specification, and verified that the obtained implementation successfully interoperates with OpenSSH.

**Related Work** Several tools already use the approach of generating an implementation from a specification: AGVI [32] first generates a protocol from security requirements, proves its correctness using the protocol verifier Athena, then compiles the protocol into Java. $\chi$-spaces [25] provide a domain-specific language for specifying protocols, which can be interpreted or compiled to Java. Spi2Java [31, 29] translates spi-calculus protocols into Java implementations; the soundness of this translation is proved in [29]. The protocols can also be verified using the automatic protocol verifier ProVerif. Spi2Java has been applied to the key exchange part of the SSH Transport Layer Protocol [28]. The JavaSPI framework [4] is a variant of Spi2Java in which the modeling language is also Java itself, instead of the spi calculus. All these approaches differ from our work in that they verify protocols in the symbolic model, while we verify them in the more realistic computational model.

Other approaches analyze implementations instead of generating them. Many of these approaches do not provide computational security guarantees. The tool CSur [21] analyzes protocols written in C by translating them into Horn clauses, given as input to the $\mathscr{H}_1$ prover. Similarly, JavaSec [22] translates Java programs into first-order logic formulas, given as input to the first-order theorem prover e-SETHEO. Poll and Schubert [30] verified an implementation of SSH in Java using ESC/Java2: ESC/Java2 verifies that the implementation does not raise exceptions, and follows a specification of SSH by a finite automaton, but does not prove security properties. ASPIER [18] uses software model-checking to verify C implementations of protocols, assuming the size of messages and the number of sessions are bounded. This tool has been used to verify the main loop of OpenSSL 3. Dupressoir et al. [19] use the general-purpose C verifier VCC to prove both memory safety and security properties of protocols.

The tool FS2PV [12] translates protocols written in a subset of the functional language F# into the input language of ProVerif, to prove them in the symbolic model. This technique was applied to the protocol TLS [10]. Similarly, Elijah [26] translates Java programs into LySa protocol specifications, which can be verified by the LySatool. Aizatulin et al. [1] use symbolic execution in order to extract ProVerif models from pre-existing protocol implementations in C. This technique currently analyzes a single execution path of the protocol, so it is limited to protocols without branching. Together with ASPIER [18], it is one of the rare methods that can analyze implementations not written specifically for verification. The tools F7 and F$^\star$ [9, 11, 33] use a dependent type system in order to prove security properties of protocols implemented in F#, in the symbolic model. This approach scales well to large implementations but requires type annotations, which facilitate automatic verification.

In contrast, the following approaches provide computational security guarantees. Similarly to FS2PV, the tool FS2CV (`http://msr-inria.inria.fr/projects/sec/fs2cv/`) translates a subset of F# to the input language of CryptoVerif, which can then provide a proof of the protocol in the computational model. This tool has been applied to a very small subset of the TLS protocol [10]. The F7 approach has also been extended to the computational model [20], but still requires type annotations to help the proof. [1] provides computational security guarantees by applying the computational soundness result of [5]: this result shows that, if a trace property (such as authentication) holds in the symbolic model, then it also holds in the computational model, provided the protocol uses only cryptographic primitives in a certain set (e.g. IND-CCA public-key encryption) and satisfies certain soundness conditions. The idea of using a computational soundness result could also be applied to other techniques that prove protocols in the symbolic model. However, as mentioned above, this restricts the class of protocols that can be considered. To overcome this limitation, the authors of [1] have recently extended their approach to generate a CryptoVerif model [2], thus getting proofs directly in the computational model, still with the limitation to a single execution path. Our work nicely complements these approaches by allowing one to generate implementations instead of analyzing them.

**Outline** Section 2 is a general presentation of our approach. Section 3 describes the specification language used by our compiler and Section 4 details how this language is compiled into OCaml. Finally, Section 5 presents the application of this compiler to the SSH protocol. This paper is an extended version of the conference paper [16]. We add several examples and the explanation of the treatment of the **insert** and **get** constructs that deal with key tables. We also add more details on our model of SSH; we explain how CryptoVerif proves its security properties and we describe an extension of CryptoVerif that we had to implement in order to achieve the proof of secrecy of the session keys in SSH. Our compiler, our model and our implementation of the SSH Transport Layer protocol are available as part of the CryptoVerif distribution at `http://cryptoverif.inria.fr`.

## 2 Overview of the Approach

Figure 1 presents an overview of our approach to obtain a proved implementation of a cryptographic protocol. We proceed in two steps.

First, we write a CryptoVerif specification of this protocol. This specification contains a representation of the protocol in a process calculus described in the next section, and a list of security assumptions on the cryptographic primitives, for example, encryption is IND-CPA. We prove that this specification

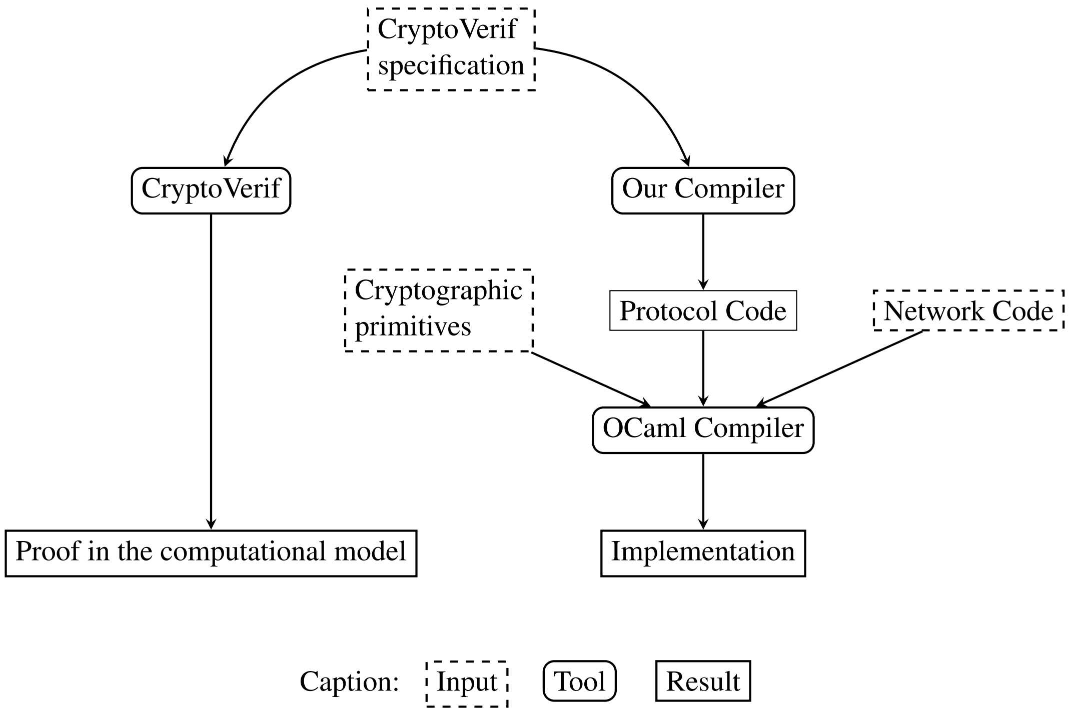

Figure 1: Overview of the approach

guarantees the desired security properties (e.g. secrecy, authentication, ...) in the computational model by using the CryptoVerif tool.

Second, the compiler we developed transforms the specification into protocol code. To build the implementation, we furthermore need to write:

- the code corresponding to the exchange of messages across the network, which uses the results given by the functions in the protocol code. This code can be considered as a part of the adversary, and so it is not required to prove this part of the code.

- the code corresponding to the cryptographic primitives. This part is used by the protocol code, and thus we must prove manually that the primitives satisfy the security assumptions we made in the specification file.

We then use the OCaml compiler on these parts to obtain an implementation of the protocol. Therefore, from a single protocol specification, we obtain both a proof that the protocol is secure in the computational model and an executable implementation of the protocol.

## 3 The Specification Language

CryptoVerif uses a process calculus in order to represent the protocol to prove and the intermediate games of the proof. We survey this calculus, explaining the extensions we have implemented and the annotations we have added to allow automatic compilation into an implementation.

| | |
|---|---|
| $M, N ::=$ | terms |
| $x$ | variable |
| $f(M_1, \dots, M_m)$ | function application |
| | |
| $Q ::=$ | oracle declarations |
| $0$ | nil |
| $Q \mid Q'$ | parallel composition |
| **foreach** $i \leq n$ **do** $Q$ | replication $n$ times |
| $O(x_1 : T_1, \dots, x_k : T_k) := P$ | oracle declaration |
| | |
| $P ::=$ | oracle body |
| **return**$(M_1, \dots, M_k); Q$ | return |
| **end** | end |
| $x \xleftarrow{R} T; P$ | random number |
| $x \leftarrow M; P$ | assignment |
| **if** $M$ **then** $P$ **else** $P'$ | conditional |
| **event** $e(M_1, \dots, M_l); P$ | event |
| **insert** $Tbl(M_1, \dots, M_k); P$ | insert in table |
| **get** $Tbl(x_1 : T_1, \dots, x_k : T_k)$ **suchthat** $M$ **in** $P$ **else** $P'$ | get from table |

Figure 2: Protocol representation language

## 3.1 Protocol Representation Language

The protocol is represented in the language of Figure 2. This language uses types denoted by $T$, which are subsets of $bitstring_\bot = bitstring \cup \{\bot\}$ where $bitstring$ is the set of all bitstrings and $\bot$ is a special symbol (used for example to represent the failure of a decryption). Some types are predefined: $bool = \{\text{true}, \text{false}\}$, where false is 0 and true is 1; $bitstring$; and $bitstring_\bot$.

The language also uses function symbols $f$. Each function symbol comes with a type declaration $f : T_1 \times \ldots \times T_m \to T$, and represents an efficiently computable, deterministic function that maps each tuple in $T_1 \times \ldots \times T_m$ to an element of $T$. Particular functions are predefined, and some of them use the infix notation: $M = N$ for the equality test, $M \neq N$ for the inequality test, $M \vee N$ for the boolean or, $M \wedge N$ for the boolean and, $\neg M$ for the boolean negation.

In this language, terms represent computations on bitstrings. The term $x$ evaluates to the content of the variable $x$. We use $x, y, z, u$ as variable names. The function application $f(M_1, \dots, M_m)$ returns the result of applying the function $f$ to $M_1, \dots, M_m$.

This language distinguishes oracle declarations and oracle bodies. An oracle declaration provides some oracles, which can be called by the adversary, while an oracle body specifies the computations to perform upon oracle call, and returns the result of the oracle. The oracle declaration 0 is empty: it declares no oracle at all. The oracle declaration $Q \mid Q'$ is a parallel composition: it simultaneously provides the oracles declared in $Q$ and those in $Q'$. These oracles can be called in any order by the adversary. The oracle declaration **foreach** $i \leq n$ **do** $Q$ provides $n$ copies of the oracles declared in $Q$, indexed by $i \in [1, n]$, where $n$ is a parameter (an unspecified integer). This parameter is used by CryptoVerif to express the maximum probability of breaking the protocol, which typically depends on the number of calls to the various oracles. Finally, the oracle declaration $O(x_1 : T_1, \dots, x_k : T_k) := P$ declares the oracle $O$, taking arguments $x_1, \dots, x_k$ of types $T_1, \dots, T_k$ respectively. The result of this oracle is computed by the oracle body $P$.

The oracle body $x \xleftarrow{R} T; P$ chooses a new random number uniformly in $T$, stores it in $x$, and executes $P$. Function symbols represent deterministic functions, so all random numbers must be chosen by $x \xleftarrow{R} T$. Using deterministic functions facilitates the proofs of protocols in CryptoVerif by making automatic syntactic manipulations easier: we can duplicate a term without changing its value. The assignment $x \leftarrow M; P$ stores the value of $M$ in $x$ and executes $P$. The test **if** $M$ **then** $P$ **else** $P'$ executes $P$ when $M$ evaluates to true and $P'$ otherwise. The construct **event** $e(M_1, \dots, M_l); P$ executes the event $e(M_1, \dots, M_l)$, then runs $P$. This event records that a certain program point has been reached with certain values of $M_1, \dots, M_l$, but otherwise does not affect the execution of the system. (Events only serve in specifying authentication properties [13].) The construct **return**$(M_1, \dots, M_k); Q$ returns the result $M_1, \dots, M_k$ of the oracle. Additionally, it makes available the oracles defined in $Q$; these oracles can then be called by the adversary. The construct **end** terminates the oracle with an error, yielding control to the adversary.

The constructs **insert** and **get** handle tables, used for instance to store the keys of the protocol participants. A table can be represented as a list of tuples; **insert** $Tbl(M_1, \dots, M_k); P$ inserts the element $(M_1, \dots, M_k)$ in the table $Tbl$; **get** $Tbl(x_1 : T_1, \dots, x_k : T_k)$ **suchthat** $M$ **in** $P$ **else** $P'$ tries to retrieve an element $(x_1, \dots, x_k)$ in the table $Tbl$ such that $M$ is true. When such an element is found, it executes $P$ with $x_1, \dots, x_k$ bound to that element. (When several such elements are found, one of them is chosen randomly with uniform probability. We cannot for instance take the first element found because the game transformations made by CryptoVerif may reorder the elements. For these transformations to preserve the behavior of the game, the distribution of the chosen element must be invariant by reordering.) When no such element is found, $P'$ is executed.

The original CryptoVerif language does not include **insert** and **get**. Instead, it considers all variables as arrays, and offers a construct for looking up values in arrays, **find**. The constructs **insert** and **get** are intuitively easier to understand, closer to the constructs used by cryptographers, and much easier to implement. However, arrays and **find** are very helpful for the automatic proofs performed by CryptoVerif, as explained in [14]. Therefore, in order to implement **insert** and **get**, we first transform them into arrays and **find**, so that CryptoVerif can run as before after this transformation.

The **find** construct has the following syntax:

$$\textbf{find } (\bigoplus_{j=1}^{m} u_{j1} \leq n_{j1}, \dots, u_{jm_j} \leq n_{jm_j} \textbf{ suchthat defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j \textbf{ then } P_j) \textbf{ else } P$$

This construct finds indices $u_{j1}, \dots, u_{jm_j}$ such that $M_{j1}, \dots, M_{jl_j}$ are defined and $M_j$ is true. If such indices are found, it runs $P_j$. If no such indices can be found for any $j$, it runs $P$. More formally, this construct computes the set $S$ of elements $j, a_1, \dots, a_{m_j}$ where $a_1, \dots, a_{m_j}$ are replication indices such that all terms $M_{jk}$ are defined and $M_j$ evaluates to true, after replacing each $u_{jk}$ with $a_k$. If the set $S$ is empty, no instance of the replication indices could satisfy the conditions and so we continue with $P$. Otherwise, we choose randomly an element in $S$, and if the chosen element is $j_0, a_1, \dots, a_{m_{j_0}}$ we instantiate the variables $u_{j_0 1}, \dots, u_{j_0 m_{j_0}}$ to $a_1, \dots, a_{m_{j_0}}$ and continue with $P_{j_0}$.

The transformation of **insert** and **get** into **find** proceeds by storing the inserted list elements in fresh array variables, and looking up in these arrays instead of performing **get**. More precisely, when **insert** $Tbl(M_1, \dots, M_k); P$ is under the replications **foreach** $i_1 \leq n_1$ **do** ... **foreach** $i_l \leq n_l$ **do**, it is transformed into

$$y_1[i_1, \dots, i_l] \leftarrow M_1; \dots; y_k[i_1, \dots, i_l] \leftarrow M_k; P$$

where $y_1, \dots, y_k$ are fresh array variables, and we add $(y_1, \dots, y_k; i_1 \leq n_1, \dots, i_l \leq n_l)$ in a set $S'$, to remember them. The construct **get** $Tbl(x_1 : T_1, \dots, x_k : T_k)$ **suchthat** $M$ **in** $P$ **else** $P'$ is then transformed into

$$\textbf{find} \left( \bigoplus_{(y_1,\dots,y_k;i_1\le n_1,\dots,i_l\le n_l)\in S'} \begin{array}{l} u_1 \le n_1,\dots,u_l \le n_l \textbf{ suchthat} \\ \textbf{defined}(y_1[\widetilde{u}],\dots,y_k[\widetilde{u}]) \wedge M\{y_1[\widetilde{u}]/x_1,\dots,y_k[\widetilde{u}]/x_k\} \\ \textbf{then } x_1 \leftarrow y_1[\widetilde{u}];\dots;\ x_k \leftarrow y_k[\widetilde{u}];\ P \end{array} \right)$$
$$\textbf{else } P'$$

where $\widetilde{u}$ stands for $u_1,\dots,u_l$. This construct looks in all arrays used for translating insertion in table *Tbl*, for indices $\widetilde{u}$ such that $y_1[\widetilde{u}],\dots,y_k[\widetilde{u}]$ are defined, that is, an element has been inserted at indices $\widetilde{u}$, and $M\{y_1[\widetilde{u}]/x_1,\dots,y_k[\widetilde{u}]/x_k\}$ is true, that is, that element satisfies $M$. When it finds such an element, it stores it in $x_1,\dots x_k$, and runs $P$. (When it finds several elements, one of them is chosen randomly with uniform probability.) When it finds no element, it executes $P'$.

CryptoVerif also offers a pattern-matching construct. A function $f : T_1 \times \dots \times T_m \to T$ that can be used for pattern-matching is declared with the attribute `compos`. This attribute means that $f$ is injective and that its inverses are efficiently computable, that is, there exist efficiently computable functions $f_j^{-1} : T \to T_j$ $(1 \le j \le m)$ such that $f_j^{-1}(f(x_1,\dots,x_m)) = x_j$. We can then define the pattern-matching construct **let** $f(x_1,\dots,x_m) = M$ **in** $P$ **else** $P'$ as an abbreviation for $y \leftarrow M; x_1 \leftarrow f_1^{-1}(y);\dots;x_m \leftarrow f_m^{-1}(y);$ **if** $f(x_1,\dots,x_m) = y$ **then** $P$ **else** $P'$. This construct tries to extract the values of $x_1,\dots,x_n$ such that $f(x_1,\dots,x_n) = M$, and runs $P$ when this extraction succeeds, and $P'$ when it fails. Also, we define the construct **let** $(=M) = M'$ **in** $P$ **else** $P'$ as **if** $M = M'$ **then** $P$ **else** $P'$. We generalize this construct to **let** $pat = M$ **in** $P$ **else** $P'$ where $pat$ is built from `compos` functions, variable names, and equality to terms $=M$.

**Example 1.** *Let $f : T_1 \times T_2 \to T$ be a function with the* `compos` *attribute. Let $f_1^{-1} : T \to T_1$ and $f_2^{-1} : T \to T_2$ be efficiently computable inverses of $f$.*

*The oracle body* **let** $f(=M,x) = M'$ **in** $P$ **else** $P'$ *is an abbreviation of:*

$$y \leftarrow M'; x_1 \leftarrow f_1^{-1}(y); x \leftarrow f_2^{-1}(y); \textbf{if } f(x_1,x) = y \wedge M = x_1 \textbf{ then } P \textbf{ else } P'.$$

*If there exists a value $x$ such that $f(M,x) = M'$, it runs $P$ with that value of $x$, else it runs $P'$.*

An **else** branch of **if**, **get**, or **let** may be omitted when it is **else end**. Similarly, **end** may be omitted after a random choice, an assignment, an event, or a table insertion. A trailing 0 after a return may also be omitted.

The original CryptoVerif language appears in two versions, using channels [14, 13] or oracles [15]. We use the version with oracles in this paper, because it is closer to OCaml code. (Oracles resemble functions.) Our compiler also works on the version with channels. This language uses a simple type system to check that bitstrings are of the appropriate type; this type system and the formal semantics of this language are detailed in [14], for the version with channels. Additional constructs exist in this language for calling oracles and for hiding oracles so that they cannot be called by the adversary. These constructs are not necessary for encoding the protocol itself, so we omit them here.

**Example 2.** *Let us consider a simple protocol in which the first participant $A$ generates a nonce $x$, and sends it to the second participant $B$ encrypted under the shared secret key $K_{ab}$: $A \to B : \{x\}_{K_{ab}}$. This protocol can be modeled in CryptoVerif as follows:*

$$\begin{array}{l} \mathrm{Ostart}() := rK_{ab} \xleftarrow{R} \mathit{keyseed};\ K_{ab} \leftarrow \mathrm{kgen}(rK_{ab}); \textbf{return}(); \\ \quad (\textbf{foreach } i_1 \le N \textbf{ do } P_A \mid \textbf{foreach } i_2 \le N \textbf{ do } P_B) \end{array}$$

$$P_A = \mathrm{OA}() := x \xleftarrow{R} \mathit{nonce};\ s \xleftarrow{R} \mathit{seed}; \textbf{return}(\mathrm{enc}(x, K_{ab}, s))$$

$$P_B = \mathrm{OB}(m : \mathit{bitstring}) := \textbf{let } \mathrm{injbot}(r') = \mathrm{dec}(m, K_{ab}) \textbf{ in return}()$$

*The only oracle callable at the beginning is* Ostart*, which generates a symmetric encryption key $K_{ab}$ by generating a random seed $rK_{ab}$ and using the key generation algorithm* kgen *on it. It returns nothing. The key $K_{ab}$ is available to the following oracles in the process, but is not given to the adversary. After having called* Ostart*, one can call $N$ times the oracles* OA *and* OB*. In the oracle* OA*, we generate a nonce $x$, a seed for the encryption $s$, and return the encryption of $x$ under the key $K_{ab}$ with the random seed $s$. The oracle* OB *takes as argument $m$, which should be the message returned by the oracle* OA*. It decrypts the message under the symmetric key $K_{ab}$. A decrypted message is of type $bitstring_\perp$: it can be a bitstring or the $\perp$ value, which means that decryption failed. The function* injbot *is the injection that takes a nonce value and returns its value in $bitstring_\perp$, which is different from $\perp$. When decryption succeeds, the oracle* OB *stores in $r'$ the result of the decryption, and returns normally. Otherwise, it terminates with* **end** *(implicit in the omitted* **else** *branch of* **let***).*

## 3.2 Annotations for Implementation

The protocol specification language also includes annotations to specify which parts of the protocol will be compiled into which OCaml modules, and which OCaml types, functions, and files correspond to the CryptoVerif types, functions, and key tables. These annotations are simply ignored when CryptoVerif proves the protocol.

A protocol typically includes several parts of code run by different participants, for instance a client and a server. These parts of code will be included in different programs, so we split the protocol into multiple roles *role* that will be translated into different OCaml modules. The boundaries of roles are marked as follows. The annotation $role\,[x_1 > \text{"}filex_1\text{"}, \dots, x_n > \text{"}filex_n\text{"}, y_1 < \text{"}filey_1\text{"}, \dots, y_m < \text{"}filey_m\text{"}]\,\{$ indicates the beginning of the role *role*. It should be placed just above an oracle declaration $Q$. The indication $x_i > \text{"}filex_i\text{"}$ means that the variable $x_i$ will be stored in file $filex_i$ when it is defined. The variable $x_i$ can then be used in other roles defined after the end of *role*; these roles will read it automatically from the file $filex_i$. The indication $y_i < \text{"}filey_i\text{"}$ means that the role *role* will read at initialization the value of the variable $y_i$ from the files $filey_i$. The variable $y_i$ must be free in *role* (i.e. it is defined before the beginning of *role*). A declaration $x > \text{"}filex\text{"}$ in a role $role'$ above *role* implicitly implies $x < \text{"}filex\text{"}$ in *role* when *role* uses $x$: $x$ is written to *filex* in $role'$ and read in *role*. All variables free in role *role* must be declared as being read from a file in *role*, either explicitly or implicitly as mentioned above. All variables read from or written to a file must be defined under no replication. (Otherwise, several copies of the variable would have to be stored in the file.) Storing variables in files is useful for variables that are communicated across roles, for example long-term keys that are set in a key generation program and later used by the client and/or server programs. The closing brace $\}$ indicates the end of the current role. It must be placed just after a **return** statement.

**Example 3.** *Let us annotate the process we have seen in Example 2.*

$$\begin{aligned}
&role_{Keygen}[K_{ab} > \text{"}keyfile\text{"}]\{\mathrm{Ostart}() := \dots \mathbf{return}()\};\\
&\qquad(\mathbf{foreach}\ i_1 \le N\ \mathbf{do}\ P_A \mid \mathbf{foreach}\ i_2 \le N\ \mathbf{do}\ P_B)\\
&P_A = role_A\{\mathrm{OA}() := \dots\\
&P_B = role_B\{\mathrm{OB}(m : bitstring) := \dots
\end{aligned}$$

*We divide the process into three roles. First, the key generation role is represented by $role_{Keygen}$, containing just the oracle* Ostart*. We store the value of $K_{ab}$ in the file keyfile, in order to be able to read the value of the key in the other roles. The role $role_A$, which contains the oracle* OA*, corresponds to the role of A, and the role $role_B$, which contains the oracle* OB*, corresponds to the role of B. For these two roles, there is no need to write the closing brace $\}$ because there is nothing after them.*

The correspondence between CryptoVerif and OCaml types, functions, and tables is specified by declarations in the input file. These declarations associate to each CryptoVerif type $T$:

- its corresponding OCaml type $\mathbb{G}_{\mathbf{T}}(T)$.
- the serialization function $\mathbb{G}_{\mathbf{ser}}(T)$ of type $\mathbb{G}_{\mathbf{T}}(T) \rightarrow$ **string**, which converts an element of type $\mathbb{G}_{\mathbf{T}}(T)$ to a bitstring, and the deserialization function $\mathbb{G}_{\mathbf{deser}}(T)$ of type **string** $\rightarrow \mathbb{G}_{\mathbf{T}}(T)$, which performs the inverse operation. These functions serve for writing values to files and for reading them. When deserialization fails, it must raise the exception **Bad_file**; this exception is raised only when a file has been corrupted.
- the predicate function $\mathbb{G}_{\mathbf{pred}}(T)$ of type $\mathbb{G}_{\mathbf{T}}(T) \rightarrow$ **bool**, which returns whether an OCaml element of type $\mathbb{G}_{\mathbf{T}}(T)$ belongs to type $T$ or not. Indeed, the CryptoVerif values of type $T$ may correspond only to a subset of the OCaml values of type $\mathbb{G}_{\mathbf{T}}(T)$.
- the random number generation function $\mathbb{G}_{\mathbf{random}}(T)$, of type **unit** $\rightarrow \mathbb{G}_{\mathbf{T}}(T)$, which returns a random element uniformly chosen in type $T$.

They also associate to each table *Tbl* the name $\mathbb{G}_{\mathbf{table}}(\mathit{Tbl})$ of the file that contains that table, and to each CryptoVerif function $f$ of type $T_1 \times \ldots \times T_m \rightarrow T$ the corresponding OCaml function $\mathbb{G}_{\mathbf{f}}(f)$ of type $\mathbb{G}_{\mathbf{T}}(T_1) \rightarrow \ldots \rightarrow \mathbb{G}_{\mathbf{T}}(T_m) \rightarrow \mathbb{G}_{\mathbf{T}}(T)$.

These correspondences are specified in the specification by the following **implementation** declarations in the CryptoVerif input file:

- **implementation type** $T = \mathbb{G}_{\mathbf{T}}(T)$ [*options*] sets the OCaml type corresponding to the CryptoVerif type $T$. The possible options are:
  - *serial* $= \mathbb{G}_{\mathbf{ser}}(T), \mathbb{G}_{\mathbf{deser}}(T)$ sets the serialization and deserialization functions for type $T$.
  - *pred* $= \mathbb{G}_{\mathbf{pred}}(T)$ sets the predicate function.
  - *random* $= \mathbb{G}_{\mathbf{random}}(T)$ sets the random number generation function.
- **implementation type** $T = n$ [*options*] sets the length of the type $T$. The type $T$ is then represented by a bitstring of length $n$. If $n$ is a multiple of 8, then $T$ will be represented by a **string**: $\mathbb{G}_{\mathbf{T}}(T) =$ **string**; if $n = 1$, then $T$ will be represented by a boolean: $\mathbb{G}_{\mathbf{T}}(T) = \mathit{bool}$. Otherwise, an error occurs. The only allowed option is *serial*, which allows one to override the default serialization and deserialization functions to choose a different representation of the bitstring. This sets the functions $\mathbb{G}_{\mathbf{random}}(T)$ and $\mathbb{G}_{\mathbf{pred}}(T)$ to correct values.
- **implementation table** $\mathit{Tbl} = \mathbb{G}_{\mathbf{table}}(\mathit{Tbl})$ sets the file in which the table *Tbl* is written.
- **implementation fun** $f = \mathbb{G}_{\mathbf{f}}(f)$ [*options*] sets the translation of the function $f$. If $f$ has the `compos` attribute, that is to say that $f$ is injective, this declaration can take the option *inverse* $= \mathbb{G}_{\mathbf{f^{-1}}}(f)$, which declares $\mathbb{G}_{\mathbf{f^{-1}}}(f)$ as the inverse function. If $f$ is of type $T_1 \times \ldots \times T_m \rightarrow T$, this function must be of type $\mathbb{G}_{\mathbf{T}}(T) \rightarrow \mathbb{G}_{\mathbf{T}}(T_1) \times \ldots \times \mathbb{G}_{\mathbf{T}}(T_m)$. $\mathbb{G}_{\mathbf{f^{-1}}}(f)\ x$ must return a tuple $(x_1, \ldots, x_m)$ such that $\mathbb{G}_{\mathbf{f}}(f)\ x_1\ \ldots\ x_m = x$. If there is no such element, $\mathbb{G}_{\mathbf{f^{-1}}}(f)$ must raise **Match_fail**. The function $\mathbb{G}_{\mathbf{f^{-1}}}(f)$ is used for translating the pattern-matching construct into OCaml; for simplicity, we do not detail this translation.
- **implementation const** $f = \mathbb{G}_{\mathbf{f}}(f)$ sets the implementation of the function $f$ that has no arguments to the OCaml constant $\mathbb{G}_{\mathbf{f}}(f)$.

**Example 4.** *For instance, the declaration*

$$\textbf{implementation type}\ \mathit{nonce} = \texttt{"}\textbf{string}\texttt{"}\ [\mathit{serial} = \texttt{"id"}, \texttt{"deserial 64"}; \mathit{pred} = \texttt{"sizep 64"}; \mathit{random} = \texttt{"rand\_string 8"}].$$

*means that the CryptoVerif type nonce is implemented by the OCaml type* **string***, with serialization function identity, deserialization function* `deserial 64` *which is the identity for strings of 8 bytes (64 bits) and raises* **Match_fail** *for other strings, predicate function* `sizep 64` *which returns true for strings of 8 bytes and false for other strings, and random number generation function* `rand_string 8` *which returns random strings of 8 bytes. In other words, nonces are bitstrings of 64 bits, which we can abbreviate by*

$$\textbf{implementation type}\ \mathit{nonce} = 64.$$

*The declaration*

$$\textbf{implementation fun}\ \mathrm{kgen} = \texttt{"kgen"}.$$

*means that the CryptoVerif function* kgen *is implemented by the OCaml function* `kgen`.

A trick can be used to provide, for the same function $f$, both an OCaml implementation and a CryptoVerif definition of $f$ from other functions. Indeed, CryptoVerif allows one to define $f$ as a macro: **letfun** $f(x_1 : T_1, \dots, x_m : T_m) = M$. Specifying an OCaml implementation for these macros is optional. When the OCaml implementation is not specified, our compiler generates code according to the **letfun** macro. When the OCaml implementation is specified, it is used when generating the OCaml code, while the CryptoVerif macro defined by **letfun** is used for proving the protocol. This feature can be used, for instance, to define probabilistic functions: the OCaml implementation generates the random choices inside the function, while the CryptoVerif definition by **letfun** first makes the random choices, then calls a deterministic function.

**Example 5.** *We can define an encryption function that generates the random seed internally as follows:*

$$\textbf{letfun}\ \mathrm{renc}(x : \mathit{bitstring}, k : \mathit{key}) = s \xleftarrow{R} \mathit{seed};\ \mathrm{enc}(x, k, s).$$

*where* enc *is a deterministic encryption function that takes the random seed as argument. We can give an OCaml implementation for* renc *by*

$$\textbf{implementation fun}\ \mathrm{renc} = \texttt{"renc"}.$$

*Obviously, this OCaml function must also choose the random seed for encryption internally.*

## 4 The Translation into OCaml

Our compiler automatically translates a specification written in the CryptoVerif language into OCaml. Let us describe this translation.

The annotations of Section 3.2 split the CryptoVerif code into multiple parts corresponding to different roles. Our compiler translates each of these roles *role* into an OCaml module $\mu_{role}$. For each role *role*, let $Q(role)$ be the oracle declaration located between $role[\dots]\{$ and the following closing braces $\}$. $Q(role)$ is the CryptoVerif code for the role *role*. Our compiler translates the oracles of $Q(role)$ into OCaml functions. More precisely, the implementation of the module $\mu_{role}$ consists of the **init** function, which reads the values of the variables required by the oracles in $Q(role)$ from the files, and returns the functions corresponding to the oracles declared by $Q(role)$. Functions corresponding to the oracles declared after a **return** in $Q(role)$ are not returned by **init**, but will be returned by that **return**, like continuations. Hence, the available functions correspond exactly to the oracles that can be called.

**Example 6.** *Suppose that the role role is defined by*

$$role\ [\ldots]\ \{O_1(\ldots) := \ldots; \textbf{return}(M_1); O_2(\ldots) := \ldots; \textbf{return}(M_2)\}$$

*Then the generated OCaml module $\mu_{role}$ provides a function* **init** *that returns a function that implements oracle $O_1$. When this function is called, it returns both the result of the oracle $O_1$ (the value of $M_1$) and the function that implements oracle $O_2$. That function just returns the result of $O_2$, that is, the value of $M_2$.*

This translation requires us to restrict the process when an oracle has several **return** statements: all these **return** statements must return data of the same type and oracles of the same name and type. We can work around this restriction as follows: when an oracle is missing at some **return** statements, we add a dummy oracle that ends immediately. As usual in functional languages, functions are represented by closures that contain a pointer to the code of the function and an environment that contains the free variables of the function. We rely on the OCaml type system to guarantee that the environment of closures is not accessed by the rest of the code, and in particular not sent directly to the adversary. The rest of this section details how the function **init** is generated.

For simplicity, we rename the variables in the CryptoVerif code in order to have a unique name for each variable. CryptoVerif already does this internally. Let $\mathbb{G}_{\mathbf{var}}$ be an injective function taking a CryptoVerif variable name, and returning an OCaml variable name. Let us also denote by $T_M$ the type of a CryptoVerif term $M$.

The function $\mathbb{G}_{\mathbf{M}}$ transforms a term $M$ into an OCaml term, in the obvious way:

$$\begin{aligned}&\mathbb{G}_{\mathbf{M}}(x) = \mathbb{G}_{\mathbf{var}}(x)\\ &\mathbb{G}_{\mathbf{M}}(f(M_1,\ldots,M_m)) = \mathbb{G}_{\mathbf{f}}(f)\ (\mathbb{G}_{\mathbf{M}}(M_1))\ \ldots\ (\mathbb{G}_{\mathbf{M}}(M_m))\end{aligned}$$

The function oracles takes an oracle declaration $Q$ and returns a list containing the oracles declared in $Q$. For each oracle, it also returns a boolean that is true when the oracle is defined under **foreach** (so can be called several times), and false otherwise. This function is defined as follows, where we denote by $\varepsilon$ the empty list:

$$\begin{aligned}&\text{oracles}(0) = \varepsilon\\ &\text{oracles}(Q_1 \mid Q_2) = \text{oracles}(Q_1), \text{oracles}(Q_2)\\ &\text{oracles}(\textbf{foreach}\ i \le n\ \textbf{do}\ Q) = (Q_1, \text{true}), \ldots, (Q_k, \text{true})\ \text{when}\\ &\qquad\qquad (Q_1, b_1), \ldots, (Q_k, b_k) = \text{oracles}(Q)\ \text{for some}\ b_1, \ldots, b_k\\ &\text{oracles}(O(x_1, \ldots, x_k) := P) = (O(x_1, \ldots, x_k) := P, \text{false})\end{aligned}$$

This function is used in the generation of the **init** function in order to determine the oracles we can call at the beginning of the role, and in the translation of the **return** statement to determine which closures to give back to the caller.

In Figure 3, we define the function $\mathbb{G}$ that translates an oracle body into an OCaml term, as explained below.

As mentioned in Section 3.2, a role is declared with variables read from and written to files. Let **write_file** be an OCaml function of type **string** $\rightarrow$ **string** $\rightarrow$ **unit** that takes a file name and the contents to write and writes the contents to the file, and **read_file** a function of type **string** $\rightarrow$ **string** that takes a file name and returns its contents. We define a function $\mathbb{G}_{\mathbf{file}}$ that writes a variable to a file when needed: $\mathbb{G}_{\mathbf{file}}(x) = \textbf{write\_file}\ f\ (\mathbb{G}_{\mathbf{ser}}(T_x)\ \mathbb{G}_{\mathbf{var}}(x))$ when variable $x$ is written to file $f$ in role *role*, that is, *role* is annotated with $x > f$, and $\mathbb{G}_{\mathbf{file}}(x) = ()$ when $x$ is not written to a file.

$$\mathbb{G}(x \xleftarrow{R} T; P) = \textbf{let } \mathbb{G}_{\textbf{var}}(x) = \mathbb{G}_{\textbf{random}}(T)\ ()\ \textbf{in } \mathbb{G}_{\textbf{file}}(x);\ \mathbb{G}(P)$$

$$\mathbb{G}(x \leftarrow M; P) = \textbf{let } \mathbb{G}_{\textbf{var}}(x) = \mathbb{G}_{\mathbf{M}}(M)\ \textbf{in } \mathbb{G}_{\textbf{file}}(x);\ \mathbb{G}(P)$$

$$\mathbb{G}(\textbf{if } M \textbf{ then } P \textbf{ else } P') = \textbf{if } \mathbb{G}_{\mathbf{M}}(M) \textbf{ then } \mathbb{G}(P) \textbf{ else } \mathbb{G}(P')$$

$$\mathbb{G}(\textbf{event } e(M_1, \ldots, M_k); P) = \mathbb{G}(P)$$

$$\begin{aligned}\mathbb{G}(\textbf{return}(N_1, \ldots, N_k); Q) = {} & (\mathbb{G}_{\mathbf{O}}(Q_1, b_1), \ldots, \mathbb{G}_{\mathbf{O}}(Q_l, b_l), \mathbb{G}_{\mathbf{M}}(N_1), \ldots, \mathbb{G}_{\mathbf{M}}(N_k)) \\ & \text{when oracles}(Q) = (Q_1, b_1), \ldots, (Q_l, b_l)\end{aligned}$$

$$\mathbb{G}(\textbf{end}) = \textbf{raise Match\_fail}$$

$$\begin{aligned}&\mathbb{G}(\textbf{insert } \mathit{Tbl}(M_1, \ldots, M_k); P) = \\ &\quad \textbf{add\_to\_table}\ (\mathbb{G}_{\textbf{table}}(\mathit{Tbl})\ \mathbb{G}_{\textbf{ser}}(T_{M_1})\ \mathbb{G}_{\mathbf{M}}(M_1), \ldots, \mathbb{G}_{\textbf{ser}}(T_{M_k})\ \mathbb{G}_{\mathbf{M}}(M_k));\ \mathbb{G}(P)\end{aligned}$$

$$\begin{aligned}&\mathbb{G}_{\textbf{filter}}((x_1, \ldots, x_k), M) = \\ &\quad (\textbf{function } [\mathbb{G}_{\textbf{var}}(x_1); \ldots; \mathbb{G}_{\textbf{var}}(x_k)] \rightarrow \\ &\qquad \textbf{let } \mathbb{G}_{\textbf{var}}(x_1) = \mathbb{G}_{\textbf{deser}}(T_{x_1})\ \mathbb{G}_{\textbf{var}}(x_1)\ \textbf{in } \ldots\ \textbf{let } \mathbb{G}_{\textbf{var}}(x_k) = \mathbb{G}_{\textbf{deser}}(T_{x_k})\ \mathbb{G}_{\textbf{var}}(x_k)\ \textbf{in} \\ &\qquad \textbf{if } \mathbb{G}_{\mathbf{M}}(M) \textbf{ then } (\mathbb{G}_{\textbf{var}}(x_1), \ldots, \mathbb{G}_{\textbf{var}}(x_k)) \\ &\qquad \textbf{else raise Match\_fail} \\ &\quad \mid \_ \rightarrow \textbf{raise Bad\_file})\end{aligned}$$

$$\begin{aligned}&\mathbb{G}(\textbf{get } \mathit{Tbl}(x_1, \ldots, x_k) \textbf{ suchthat } M \textbf{ in } P \textbf{ else } P') = \\ &\quad \textbf{let } \mathit{list} = \textbf{read\_table}\ \mathbb{G}_{\textbf{table}}(\mathit{Tbl})\ \mathbb{G}_{\textbf{filter}}((x_1, \ldots, x_k), M)\ \textbf{in} \\ &\quad \textbf{if } \mathit{list} = [\,] \textbf{ then } \mathbb{G}(P') \\ &\quad \textbf{else let } (\mathbb{G}_{\textbf{var}}(x_1), \ldots, \mathbb{G}_{\textbf{var}}(x_k)) = \textbf{random}_l\ \mathit{list}\ \textbf{in} \\ &\qquad (\mathbb{G}_{\textbf{file}}(x_1); \ldots; \mathbb{G}_{\textbf{file}}(x_k); \mathbb{G}(P))\end{aligned}$$

Figure 3: Translation function $\mathbb{G}$ of an oracle body in OCaml

We translate $x \xleftarrow{R} T; P$ by binding the variable $\mathbb{G}_{\textbf{var}}(x)$ to a random value in the type $T$, then writing its contents to the appropriate file if required, and finally continuing on the translation of the rest of the process $P$. We translate $x \leftarrow M; P$ in the same way, but we bind $\mathbb{G}_{\textbf{var}}(x)$ to the result of $\mathbb{G}_{\mathbf{M}}(M)$, which is the translation of the CryptoVerif term $M$ into OCaml. The translation of the **if** construct is straightforward. We simply ignore events in the translation, since they do not affect the execution of the system.

We translate the **return** statement into an OCaml tuple containing the closures of the oracles that become callable after that **return** (computed by the oracles function), and the translation of the terms $N_1, \ldots, N_k$. (The function $\mathbb{G}_{\mathbf{O}}$ is defined in Figure 4 and explained below.) **end** is translated into an exception because we need to stop the execution of the oracle here, and one must be able to distinguish whether we terminated on a **return** or on an **end** statement.

We translate the **insert** construct by simply adding to the appropriate file the serialization of the translation of arguments of **insert**. This translation uses the function **add_to_table** of type **string** $\rightarrow$ **string list** $\rightarrow$ **unit**, which takes a table file and a list of strings that represents an element of the table *Tbl*, and adds this element to the file. To translate a **get** construct, we use a function $\mathbb{G}_{\textbf{filter}}((x_1, \ldots, x_k), M)$ that takes an element of the table, returns its deserialization if it satisfies $M$, and raises **Match_fail** otherwise. We also use a function **read_table** of type **string** $\rightarrow$ (**string list** $\rightarrow$ **'a**) $\rightarrow$ **'a list** such that **read_table** $f_{\mathit{Tbl}}$ *filter* reads the table file $f_{\mathit{Tbl}}$ and returns the list of values *filter e* for all elements $e$ of the table such that *filter e* does not raise **Match_fail**. Therefore, by **read_table** $\mathbb{G}_{\textbf{table}}(\mathit{Tbl})\ \mathbb{G}_{\textbf{filter}}((x_1, \ldots, x_k)$

```
$\mathbb{G}_{\mathbf{O}}(O(x_1 : T_1, \dots, x_k : T_k) := P, \text{false}) =$
  **let** *token* = **ref** true **in**
    **function** $(\mathbb{G}_{\mathbf{var}}(x_1), \dots, \mathbb{G}_{\mathbf{var}}(x_k)) \to$
      **if** $(!token)$ && $(\mathbb{G}_{\mathbf{pred}}(T_1)\ \mathbb{G}_{\mathbf{var}}(x_1))$ && ... && $(\mathbb{G}_{\mathbf{pred}}(T_k)\ \mathbb{G}_{\mathbf{var}}(x_k))$ **then**
        $(token := \text{false};$
         $\mathbb{G}(P))$
      **else raise Bad_call**

$\mathbb{G}_{\mathbf{O}}(O(x_1 : T_1, \dots, x_k : T_k) := P, \text{true}) =$
  **function** $(\mathbb{G}_{\mathbf{var}}(x_1), \dots, \mathbb{G}_{\mathbf{var}}(x_k)) \to$
    **if** $(\mathbb{G}_{\mathbf{pred}}(T_1)\ \mathbb{G}_{\mathbf{var}}(x_1))$ && ... && $(\mathbb{G}_{\mathbf{pred}}(T_k)\ \mathbb{G}_{\mathbf{var}}(x_k))$ **then**
      $\mathbb{G}(P)$
    **else raise Bad_call**
```

Figure 4: Translation of an oracle

Let $x_1 < f_1, \dots, x_m < f_m$ be the annotations of role *role* that indicate variables read from files (explicit or implicit because of an annotation $x_i > f_i$ in a role above *role* when $x_i$ is defined above *role* and used in *role*).
Let $\text{oracles}(Q(role)) = (Q_1, b_1), \dots, (Q_k, b_k)$.

```
**let** *token* = **ref** true
**let init** = **function** () $\to$
    **if** $(!token)$ **then**
        $(token := \text{false};$
         **let** $\mathbb{G}_{\mathbf{var}}(x_1) = \mathbb{G}_{\mathbf{deser}}(T_{x_1})$ (**read_file** $f_1$) **in** ...
         **let** $\mathbb{G}_{\mathbf{var}}(x_m) = \mathbb{G}_{\mathbf{deser}}(T_{x_m})$ (**read_file** $f_m$) **in**
             $(\mathbb{G}_{\mathbf{O}}(Q_1, b_1), \dots, \mathbb{G}_{\mathbf{O}}(Q_k, b_k)))$
    **else raise Bad_call**
```

Figure 5: The **init** function for the module $\mu_{role}$

$, M)$, we collect all elements of the table that satisfy the term $M$. If there is no such element, we continue with the translation of the process $P'$. If there are such elements, we choose one of them randomly, we bind the variables $(\mathbb{G}_{\mathbf{var}}(x_1), \dots, \mathbb{G}_{\mathbf{var}}(x_k))$ accordingly and add them to their respective files if necessary, and finally we continue with the translation of the process $P$.

An oracle $O(x_1, \dots, x_n) := P$ is transformed into a closure by the function $\mathbb{G}_{\mathbf{O}}$ shown in Figure 4. The implementation differs depending on whether the oracle is under replication or not. If the oracle is not under replication, it must be callable at most once, so we create a new boolean reference that we store in *token*: *token* is true if and only if the oracle can still be executed. We initialize *token* to true. When we execute the oracle, we set *token* to false, to prevent other executions. The function also checks that its arguments are correct elements of their type by using the function $\mathbb{G}_{\mathbf{pred}}$, and then proceeds to execute the translation of the oracle body $P$. If the arguments are not correct elements of their type, or if the oracle is not under replication and has already been called, then it raises the exception **Bad_call** without executing the translation of $P$.

The implementation of the module $\mu_{role}$ consists in the **init** function presented in Figure 5. It begins

```
       let init = function () →
         if (!token) then
           (token := false;
            let $\mathbb{G}_{\mathbf{var}}(K_{ab}) = \mathbb{G}_{\mathbf{deser}}(key)$ (read_file "keyfile") in
     ⎧      let token = ref true in function() →
     ⎪        if (!token) then
     ⎪          (token := false;
OA   ⎨           let $\mathbb{G}_{\mathbf{var}}(x) = \mathbb{G}_{\mathbf{random}}(nonce)$ () in
     ⎪           let $\mathbb{G}_{\mathbf{var}}(s) = \mathbb{G}_{\mathbf{random}}(seed)$ in
     ⎪           ($\mathbb{G}_{\mathbf{f}}$(enc) ($\mathbb{G}_{\mathbf{var}}(x), \mathbb{G}_{\mathbf{var}}(K_{ab}), \mathbb{G}_{\mathbf{var}}(s)$)))
     ⎩        else raise Bad_call)
         else raise Bad_call
```

Figure 6: The module $\mu_{role_A}$

by reading all the required files, and then returns closures for all oracles that are callable at the beginning of the module. So, by calling this **init** function, the user gets access to the oracles present in the module. The **init** function can be called only once, as guaranteed by the boolean *token*.

**Example 7.** *The role $role_A$ whose process is:*

$$\mathrm{OA}() := x \xleftarrow{R} nonce;\ s \xleftarrow{R} seed;\ \mathbf{return}(\mathrm{enc}(x, K_{ab}, s))$$

*and reads the shared key $K_{ab}$ from the file keyfile is translated in the OCaml module $\mu_{role_A}$ of Figure 6.*

*To use this module, the file keyfile must already have been generated. The network code calls the* **init** *function to get a closure of the oracle OA. Then it can call this closure with argument* () *to get back the encryption of the nonce x. The following OCaml code calls the* **init** *function and then calls the closure, and stores the encryption of x in r:*

**let** *r* = **init** () ()

*The network code is then responsible for sending this encryption to B.*

To make sure that this implementation behaves as expected, the network code, which is manually written and calls this implementation, must satisfy certain constraints. This code must not use unsafe OCaml functions (such as Obj.magic or marshalling/unmarshalling with different types) to bypass the typesystem (in particular to access the environment of closures). We also require that this code does not mutate the values received from or passed to functions generated by CryptoVerif. This can be guaranteed by using unmutable types, with the above requirement that the typesystem is not bypassed. However, OCaml typically uses **string** for cryptographic functions and for network input/output, and the type **string** is mutable in OCaml. For simplicity and efficiency, the generated code uses the type **string**, with the no-mutation requirement above. We also require that all data structures manipulated by the generated code are non-circular. This is necessary because we use the OCaml structural equality to compare values, and this equality may not terminate in the presence of circular data structures. This can be guaranteed by requiring that all OCaml types corresponding to CryptoVerif types are non-recursive. We also require that the network code does not fork after obtaining but before calling an oracle that can be called only once (because it is not under a replication in the CryptoVerif specification). Indeed, forking at this point would allow the oracle to be called several times. In general, forking occurs only at the very beginning of the protocol, when the server starts a new session, so this requirement should be easily fulfilled. These requirements could be verified by program analysis.

Finally, we require that the programs are executed in the order specified by the CryptoVerif specification. For instance, in general, the key generation programs must be executed before the client and the server. We also require that several programs that insert elements in the same table are not run concurrently, to avoid conflicting writes. This requirement could be enforced using locks, but in practice, it is generally obtained for free if the programs are run in the intended order. We also require that the files used by the generated code are not read or written by other software, as this could obviously break security.

## 5 An Application: SSH

This section applies our work to an implementation of the Secure Shell (SSH) protocol. We first recall the protocol, then present our model, the proofs of the security properties, and the generated implementation.

### 5.1 Description of the Protocol

The SSH protocol is a protocol that permits a client to contact a server and run an application on it securely. When a session is established, the client and the server are authenticated and data runs through a secure channel to ensure its privacy and integrity.

SSH (version 2.0) is divided in three parts [34]. The SSH Transport Layer Protocol [36] authenticates the server to the client and establishes a secure tunnel for the other parts. This secure tunnel is implemented using encryption and MAC (message authentication code), with keys chosen by a Diffie-Hellman key exchange. The tunnel aims to guarantee the privacy and integrity of the data going through. The SSH Authentication Protocol [35] authenticates the client. The SSH Connection Protocol [37] multiplexes multiple channels through the tunnel.

We concentrated our efforts on the Transport Layer part. In Figure 7, we present an overview of this part. The key exchange part consists of four groups of messages:

1. The client and the server send their identification string, which specifies the version of SSH they use.

2. Then the server sends to the client the lists of the cryptographic algorithms for key exchange, signature, encryption, MAC, and compression it can use in order of preference, and the client sends the list of cryptographic algorithms it supports. Based on this information, the protocol chooses which algorithms to use. Our implementation uses diffie-hellman-group14-sha1, RSA signature, AES128-CBC, HMAC-SHA1, and no compression as algorithms, respectively. SSH specifies other algorithms as well. Most of them would be very easy to include in our implementation; still, the additional counter modes encryptions specified in [8] raise an additional difficulty as discussed below in Section 5.5.

3. Then the actual key exchange takes place. The key exchange messages depend on the chosen key exchange algorithm. The algorithm we use relies on a group defined in [23]. Let $p$ be a large prime and $g$ be a generator of a subgroup of $\mathbb{Z}_p^\star$.

   First, the client chooses a random exponent $x$ and sends to the server $e = g^x \bmod p$.

   Then the server chooses a random exponent $y$ and computes $f = g^y \bmod p$, the shared key $K = e^y \bmod p$, and the SHA1 hash $H$ of the messages previously sent by the client and the server, the server public host key $pk_s$, $f$, and $K$. It then signs this hash with its private host key $sk_S$. Let $s = \mathrm{sign}(H, sk_S)$ be this signature. It finally sends back $pk_s$, $f$, and $s$.

Key exchange:

$$
\begin{array}{l|rcl}
 & \text{Client } C & \text{Message} & \text{Server } S \\
\hline
1. & & \xrightarrow{id_C=\text{SSH-2.0-version}_C} & \\
 & & \xleftarrow{id_S=\text{SSH-2.0-version}_S} & \\
2. & & \xrightarrow{\text{KEX\_INIT},cookie_C,algos_C} & \\
 & & \xleftarrow{\text{KEX\_INIT},cookie_S,algos_S} & \\
3. & x \xleftarrow{R} [2,q-1], e = g^x & \xrightarrow{\text{KEXDH\_INIT},e} & y \xleftarrow{R} [1,q-1], f = g^y \\
 & K = f^x & \xleftarrow{\text{KEXDH\_REPLY},pk_S,f,\text{sign}(H,sk_S)} & K = e^y \\
4. & pk_S, \text{sign}(H, sk_S) \text{ ok?} & \xrightarrow{\text{NEWKEYS}} & \\
 & & \xleftarrow{\text{NEWKEYS}} & 
\end{array}
$$

$$\text{where } H = \text{SHA1}(id_C, id_S, cookie_C, algos_C, cookie_S, algos_S, pk_S, e, f, K)$$

Tunnel keys:

$$
\begin{aligned}
sessionid &= H \\
IV_C &= \text{SHA1}(K, H, \text{"A"}, sessionid) \\
IV_S &= \text{SHA1}(K, H, \text{"B"}, sessionid) \\
K_{\text{enc},C} &= \text{SHA1}(K, H, \text{"C"}, sessionid) \\
K_{\text{enc},S} &= \text{SHA1}(K, H, \text{"D"}, sessionid) \\
K_{\text{MAC},C} &= \text{SHA1}(K, H, \text{"E"}, sessionid) \\
K_{\text{MAC},S} &= \text{SHA1}(K, H, \text{"F"}, sessionid)
\end{aligned}
$$

Tunnel:

$$
\text{Client } C \quad
\begin{array}{c}
\xrightarrow{\text{enc}(K_{\text{enc},C},packet,IV_C),\text{MAC}(K_{\text{MAC},C},sequence\_number_C \| packet)} \\
\xleftarrow{\text{enc}(K_{\text{enc},S},packet,IV_S),\text{MAC}(K_{\text{MAC},S},sequence\_number_S \| packet)}
\end{array}
\quad \text{Server } S
$$

$$\text{where } packet = packet\_length \| padding\_length \| payload \| padding$$

Figure 7: Overview of the SSH Transport Layer Protocol

The client must then verify that $pk_s$ is indeed the key for the server it intended to reach, then compute the shared key $K = f^x \bmod p$, the hash $H$ in the same manner as the server, and then verify the signature.

4. When the client has verified the server's message, it sends a "new keys" message declaring that the key they agreed upon is to be used afterwards, and the server acknowledges this by also sending the same message.

   From the values of $H$ and $K$, SSH then generates two encryption keys (one for client to server messages, and one for server to client messages) $K_{\text{enc},C}$ and $K_{\text{enc},S}$, two initialization vectors (IVs) for the encryption $IV_C$ and $IV_S$, and two keys for MAC $K_{\text{MAC},C}$ and $K_{\text{MAC},S}$, by computing hashes of $H$, $K$, and different constants. The forthcoming messages in the SSH protocol will be encrypted and a MAC will be computed based on the clear message and on a sequence number that is incremented at each message.

Each message of the protocol, save the identification string messages, begins with five bytes indicating the size of the message (first four bytes) and the size of the random padding (one byte) present after the message, and is padded to a multiple of the block size of the encryption scheme (or 8, at the beginning when the encryption scheme is not chosen yet).

## 5.2 Our Model of SSH in CryptoVerif

We have modeled the SSH Transport Layer Protocol in the CryptoVerif specification language. In our model, the first role corresponds to the key generation. Its oracle generates the public/private key pair $pk_S, sk_S$ of the server, and returns the public part of the key to the adversary. After the execution of this role, one can execute $N$ times a client and $N$ times a server.

To illustrate our model of SSH, we give the server process in Figure 8. An adversary can give to the protocol malformed messages, so that the server and the client may have different values for the same variable. So, for a given variable $x$ of the protocol, we denote by $x_S$ the variable the server uses to hold $x$, and by $x_C$ the variable the client uses to hold $x$.

The protocol begins by exchanging the identification strings. Since this exchange requires no cryptography, it is not included in the CryptoVerif model but is done by the network code, part of the adversary. The identification strings $id_C$ and $id_S$ are given as argument to the first oracle that requires them; hence, on Line 5, the oracle key_exchange$_{2S}$ takes $id_{CS}$ and $id_{SS}$ as arguments.

Then in the protocol, we have the algorithms negotiation phase, that is done in the first oracle negotiation$_S$ on Line 1. It first generates a random cookie $cookie_{SS}$. The function concatm is an injective function that concatenates a message tag with a bitstring. All functions whose name begins with concat are injective functions that concatenate their arguments. On Line 3, we create the payload of the negotiation packet using these concatenation functions. Then, we pad the payload accordingly to the specification with pad to get a packet that we return on Line 4 to the adversary. This part cannot be done by the adversary, because we need to be sure that the cookie is randomly generated.

Then, the client sends in a similar KEX_INIT packet containing the algorithms the client supports and the cookie of the client. It then sends the first message of the key exchange KEXDH_INIT. The server must then send back the next message KEXDH_REPLY. This is done in the oracle key_exchange$_{2S}$ on Line 5. It takes $id_{CS}, id_{SS}$ as we said before, and the packets $m_1$ and $m_2$ corresponding to the KEX_INIT and KEXDH_INIT messages of the client. We first obtain the payload corresponding to the negotiation packet $m_1$ on Line 6 by using the function unpad. This function takes a packet and returns its payload if it is a valid packet and $\perp$ otherwise. Next, we verify on Line 7 that this payload is indeed a KEX_INIT message and we obtain the values of the client cookie $cookie_{CS}$ and the list of algorithms of the client $ns_{CS}$. Next, we verify that the algorithms are compatible with the algorithms of the server on Line 8. We deconstruct the KEXDH_INIT packet $m_2$ as above, we randomly generate $y_S$, and compute $f_S$, $K_S$ and $H_S$. The hash function hash takes a key $hk$ that represents the choice of the algorithm of the hash function. (This key is present in the cryptographic model, but not in the implementation.) On Line 12, we execute the event endS that is used in the proof of authentication (see Section 5.3). We sign the hash $H_S$ on Line 13, and return the KEXDH_REPLY packet on Line 14.

After verifying that this message is correct, the client sends back to the server a NEWKEYS packet. The oracle key_exchange$_{4S}$ on Line 15 takes this packet and also returns a NEWKEYS packet.

At this point, the client and the server have agreed upon the values $K$ and $H$ to create the tunnel IVs, encryption and MAC keys. The oracle get_keys$_S$ on Line 18 takes nothing, and computes these keys. The function gen$IV_C$ is defined as follows:

$$\textbf{letfun}\ \mathrm{gen}IV_{\mathrm{C}}(hk : hkey, K : G, H : hash, sid : hash) = \\ \mathrm{iv\_of\_hash}(\mathrm{hash}(hk, \mathrm{concat4}(K, H, \texttt{"}A\texttt{"}, sid))).$$

The function gen$IV_C$ generates the SHA1 hash as shown in Figure 7 (Tunnel keys), and truncates this

```
negotiations_S() :=
    $cookie_{SS} \stackrel{R}{\leftarrow} cookie$;
    $init_{SS} \leftarrow$ concatm(KEX_INIT, concat$_{\text{KEX\_INIT}}$($cookie_{SS}, negotiation\_string$));
    return(pad($init_{SS}$));
key_exchange$_{2S}$($id_{CS}$ : bitstring, $id_{SS}$ : bitstring, $m_1$ : bitstring, $m_2$ : bitstring) :=
    let injbot($init_{CS}$) = unpad($m_1$) in
    let concatm(=KEX_INIT, concat$_{\text{KEX\_INIT}}$($cookie_{CS}, ns_{CS}$)) = $init_{CS}$ in
    if (check_algorithms($ns_{CS}$)) then
    let injbot(concatm(=KEXDH_INIT, bitstring_of_G($e_S$))) = unpad($m_2$) in
    $y_S \stackrel{R}{\leftarrow} Z$; $f_S \leftarrow$ exp($g, y_S$); $K_S \leftarrow$ exp($e_S, y_S$);
    $H_S \leftarrow$ hash($hk$, concat8($id_{CS}, id_{SS}, init_{CS}, init_{SS}, pk_S, e_S, f_S, K_S$));
    event endS($id_{CS}, id_{SS}, init_{CS}, init_{SS}, pk_S, e_S, f_S, K_S, H_S$);
    $s_S \leftarrow$ sign(block_of_hash($H_S$), $sk_S$);
    return(pad(concatm(KEXDH_REPLY, concat$_{\text{KEXDH\_REPLY}}$($pk_S, f_S, s_S$))));
key_exchange$_{4S}$($m$ : bitstring) :=
    let injbot($nk_{CS}$) = unpad($m$) in let concatm(=NEWKEYS, =$null\_string$) = $nk_{CS}$ in
    return(pad(concatm(NEWKEYS, $null\_string$)));
get_keys_S() :=
    $IV_{CS} \leftarrow$ gen$IV_C$($hk, K_S, H_S, H_S$);            $IV_{SS} \leftarrow$ gen$IV_S$($hk, K_S, H_S, H_S$);
    $K_{enc,CS} \leftarrow$ gen$K_{enc,C}$($hk, K_S, H_S, H_S$);      $K_{enc,SS} \leftarrow$ gen$K_{enc,S}$($hk, K_S, H_S, H_S$);
    $K_{MAC,CS} \leftarrow$ gen$K_{MAC,C}$($hk, K_S, H_S, H_S$); $K_{MAC,SS} \leftarrow$ gen$K_{MAC,S}$($hk, K_S, H_S, H_S$);
    return($IV_{CS}, IV_{SS}, H_S$);
(foreach $j \leq N'$ do
    tunnel_send_S($payload$ : bitstring, $IV_S$ : IV, $sequence\_number_S$ : uint32) :=
        $packet \leftarrow$ pad($payload$);
        return(concatem(enc($packet, K_{enc,SS}, IV_S$),
                         mac(concatnm($sequence\_number_S, packet$), $K_{MAC,SS}$)))
| foreach $j \leq N'$ do
    tunnel_recv$_{1S}$($m$ : bitstring, $IV_C$ : IV) :=
        let injbot($m_1$) = dec($m, K_{enc,CS}, IV_C$) in
        return(get_size($m_1$));
    tunnel_recv$_{2S}$($m$ : bitstring, $IV_C$ : IV, $m'$ : mac, $sequence\_number_C$ : uint32) :=
        let injbot($m_2$) = dec($m, K_{enc,CS}, IV_C$) in
        let $packet$ = concat($m_1, m_2$) in
        if (check_mac(concatnm($sequence\_number_C, packet$), $K_{MAC,CS}, m'$)) then
        let injbot($payload$) = unpad($packet$) in
        return($payload$)).
```

Figure 8: The server role in the SSH model

hash by function iv_of_hash to obtain a valid IV. The other IV, and the encryption and MAC keys are computed in a similar way. Next, we return the IVs and the session identifier to the network code on Line 22. SSH with AES128-CBC (or other CBC mode encryptions) uses CBC mode [24, Section 7.2.2 (ii)] with chained IVs, that is, the IV for the next message is the last block of ciphertext. Since CryptoVerif does not allow maintaining a mutable state across several oracle invocations, we simply get the IV from the network code which keeps in memory the last block of ciphertext it saw. That is why we return the initial IVs to the network code. The session identifier is required in the next parts of the protocol that we implemented in the network code.

We model the SSH tunnel by oracles that get an encrypted packet from the network and return the clear payload to the application, and get a clear payload from the application and return the corresponding encrypted packet to the network code. After the return on Line 22, we can call the tunnel sending and receiving parts $N'$ times.

Sending a packet is implemented by the oracle $\mathrm{tunnel\_send}_S$ on Line 24 taking a payload, the current server to client IV, and the sequence number. We need to pass the sequence number as argument, since we cannot keep it in a state in CryptoVerif. We pad the payload, yielding a packet. We encrypt this packet, append the MAC of the sequence number and the packet, and return the obtained message on Line 26.

The packets after the key exchange are completely encrypted under the key derived from the key exchange, the first five bytes containing the size of the packet included. Therefore, an implementation must decrypt the first block of the packet to get its size, then input the rest of the packet, decrypt it, and then check that the MAC that follows in the stream is correct. So we implemented receiving a packet by two successive oracles: first, the oracle $\mathrm{tunnel\_recv}_{1S}$ on Line 29 that takes the first block of the packet and the current client to server IV, decrypts this block, and returns the size of the packet on Line 31. The network code can then input a packet of the required length, and call the second oracle $\mathrm{tunnel\_recv}_{2S}$ on Line 32 that takes the rest of the packet, its MAC, IV, and the sequence number corresponding to this message, checks the MAC and returns the decrypted payload if the MAC is correct.

## 5.3 Proof of Authentication of the Server

We have proved the authentication of the server in the computational model automatically by using CryptoVerif, assuming the RSA signature is UF-CMA (unforgeable under chosen message attacks) and the SHA1 hash function is collision-resistant. The authentication property shows that each session of the client $C$ with the server $S$ corresponds to a distinct session of the server $S$ with the client $C$, and that the client $C$ and the server $S$ share all protocol parameters: identification strings, algorithm lists, $pk_S$, $e$, $f$, $K$, and $H$.

More formally, we define the events:

$$\textbf{event } \mathit{endC}(\mathit{bitstring}, \mathit{bitstring}, \mathit{bitstring}, \mathit{bitstring}, \mathit{spkey}, G, G, G, \mathit{hash}).$$
$$\textbf{event } \mathit{endS}(\mathit{bitstring}, \mathit{bitstring}, \mathit{bitstring}, \mathit{bitstring}, \mathit{spkey}, G, G, G, \mathit{hash}).$$

where event *endC* occurs in the client just after he verifies the signature of the server, and event *endS* occurs in the server just after he computes $H_S$ (line 12 of Figure 8). The first four arguments of these events correspond to the messages exchanged in the session, two for the identification strings and two for the negotiation messages. The fifth argument corresponds to the public key of the server. The sixth and seventh arguments are the group elements $e$ and $f$. So the first seven messages correspond to the messages passed between the client and the server in a session until the end of the key exchange phase. The eighth argument corresponds to the shared key $K$ and the last argument corresponds to the hash $H$.

We ask CryptoVerif to prove the following properties:

$$\begin{aligned}&\forall vc : \mathit{bitstring}, vs : \mathit{bitstring}, ic : \mathit{bitstring}, is : \mathit{bitstring}, pk : \mathit{spkey}, x : G, y : G, k : G, h : \mathit{hash};\\ &\quad \mathbf{inj}\colon \mathit{endC}(vc, vs, ic, is, pk, x, y, k, h) \Longrightarrow \mathbf{inj}\colon \mathit{endS}(vc, vs, ic, is, pk, x, y, k, h)\,,\end{aligned} \tag{1}$$

$$\begin{aligned}&\forall vc : \mathit{bitstring}, vs : \mathit{bitstring}, ic : \mathit{bitstring}, is : \mathit{bitstring}, pk : \mathit{spkey}, x : G, y : G, k : G, h : \mathit{hash},\\ &\quad k' : G, h' : \mathit{hash};\\ &\quad \mathit{endC}(vc, vs, ic, is, pk, x, y, k, h) \wedge \mathit{endS}(vc, vs, ic, is, pk, x, y, k', h') \Longrightarrow k = k' \wedge h = h'\,.\end{aligned} \tag{2}$$

Property (1) means that each execution of event *endC* corresponds to a distinct execution of event *endS*, with the same arguments. (The indication **inj**: means that the correspondence is injective, that is, two executions of *endC* cannot correspond to the same execution of *endS*.) Property (2) means that, if events *endC* and *endS* are executed with the same first seven arguments, then their last two arguments are also the same, that is, if the client and server exchange the same public messages, then the key and the hash they compute are the same.

The proof found by CryptoVerif is the following:

1. CryptoVerif first simplifies the initial game. In particular, it transforms the **insert** and **get** constructs into **find** constructs.

2. After these transformations, it can prove Property (2), because, if the first seven arguments of the events *endC* and *endS* are equal, the eighth and ninth are computed in the same manner from the first seven, so they are equal.

3. Next, CryptoVerif replaces the secret and public keys of the server, $sk_S$ and $pk_S$, with their values, $\mathrm{sskgen}(r)$ and $\mathrm{spkgen}(r)$ respectively, where $r$ is a random number and sskgen and spkgen are the key generation functions for the signature scheme. This replacement allows CryptoVerif to apply the security assumption on the signature in the next step.

4. Next, CryptoVerif transforms the game by relying on the assumption that the signature scheme is UF-CMA. Indeed, by the UF-CMA property, up to negligible probability, the adversary cannot forge a signature, so the verification of the signature in the client, $\mathrm{check}(m_C, pk_{SC}, s_C)$ where $m_C = \mathrm{block\_of\_hash}(H_C)$ and $pk_{SC} = pk_S$, can succeed only if the message $m_C$ has been signed under $sk_S$. Moreover, the only signature under $sk_S$ occurs in the server (line 13 of Figure 8). CryptoVerif transforms this signature by first storing $\mathrm{block\_of\_hash}(H_S)$ in $m_S$, then computing $\mathrm{sign}(m_S, sk_S)$. It replaces the verification of the signature in the client, $\mathrm{check}(m_C, pk_{SC}, s_C)$, with a **find** that looks for a signature of $m_C$ under $sk_S$, that is, a **find** that looks for a session $u$ of the server such that $m_S[u]$ is defined and $m_S[u] = m_C$. (Recall that variables are implicitly arrays; $m_S[u]$ is the value of $m_S$ in session $u$ of the server.)

5. The obtained game is then simplified. In particular, the equality $m_S[u] = m_C$ above becomes $\mathrm{block\_of\_hash}(H_S[u]) = \mathrm{block\_of\_hash}(H_C)$, that is, $H_S[u] = H_C$ since block_of_hash is injective. Hence, this equality becomes $\mathrm{hash}(hk, \mathrm{concat8}(id_{CS}[u], id_{SS}[u], init_{CS}[u], init_{SS}[u], pk_S, e_S[u], f_S[u], K_S[u])) = \mathrm{hash}(hk, \mathrm{concat8}(id_{CC}, id_{SC}, init_{CC}, init_{SC}, pk_{SC}, e_C, f_C, K_C))$. Since hash is collision-resistant and concat8 is injective, this equality becomes $id_{CS}[u] = id_{CC} \wedge id_{SS}[u] = id_{SC} \wedge init_{CS}[u] = init_{CC} \wedge init_{SS}[u] = init_{SC} \wedge pk_S = pk_{SC} \wedge e_S[u] = e_C \wedge f_S[u] = f_C \wedge K_S[u] = K_C$.

   CryptoVerif can then prove Property (1). In the initial game, the event *endC* is located after the signature verification in the client. Therefore, when the client executes event $\mathit{endC}(id_{CC}, id_{SC}, init_{CC}, init_{SC}, pk_{SC}, e_C, f_C, K_C, H_C)$, the **find** that replaces signature verification succeeds, so $m_S[u]$ is defined, which implies that the server has executed event $\mathit{endS}(id_{CS}[u], id_{SS}[u], init_{CS}[u], init_{SS}[u], pk_S,$

$e_S[u], f_S[u], K_S[u], H_S[u])$ located above the definition of $m_S$, and the condition $m_S[u] = m_C$ simplified above holds, so the arguments of these events are equal. Moreover, two distinct executions of *endC* have distinct arguments $e_C$ up to negligible probability (because $e_C = g^x$ for a random exponent $x$), so they correspond to two distinct executions of *endS*, which proves injectivity.

## 5.4 Proof of Secrecy of the Session Keys

We have also proved the secrecy of the session keys obtained by key exchange (the encryption keys, MAC keys, and initialization vectors for encryption), that is, an adversary has a negligible probability of distinguishing these keys from random numbers, assuming the group used by the key exchange satisfies the CDH (Computational Diffie-Hellman) assumption, the SHA1 hash function is a random oracle, and the RSA signature is UF-CMA. This proof is performed on a protocol that stops just after key exchange, because the cryptographic secrecy of the keys is broken as soon as they are used by the protocol. Moreover, we prove secrecy for the keys computed by the client; the keys of the server are not always secret, because the server may also execute sessions with the adversary. The proof is performed by CryptoVerif with manual guidance of the user. It also required an extension of CryptoVerif, so that it can perform case distinctions depending on the order of definitions of variables. This extension will also be useful to prove other cryptographic protocols with CryptoVerif. We explain this extension when it is used in the proof (Step 10 below).

In our proof of secrecy of the session keys, we also prove the authentication property again assuming SHA1 is a random oracle (which implies collision resistance). With the random oracle model, we need to provide the adversary with a hash oracle $O_H$, so that it can compute hashes. This oracle $O_H$ takes as argument a bitstring $h$ and returns its hash:

$$O_H(h : bitstring) := \textbf{return}(\mathrm{hash}(hk, h))\,.$$

We provided CryptoVerif with proof indications to help the tool prove this property, as follows:

1. CryptoVerif first simplifies the initial game automatically. In particular, it transforms the **insert** and **get** constructs into **find** constructs.

2. By the command **success**, we ask CryptoVerif to try to prove the desired security properties. It manages to prove Property (2), as in Section 5.3. The other properties cannot be proved yet.

3. The hash function hash is used with two kinds of arguments. It is used to compute the hash $H$ with the concatenation by concat8 of eight arguments corresponding to the messages in the current session, and it is also used to compute the generated keys with argument $\mathrm{concat4}(K, H, c, H)$ for several constants $c$. To simplify the game obtained after applying the random oracle assumption (Step 4), we distinguish in the hash oracle $O_H$ these two uses of the hash function.

   By command

   $$\textbf{insert}\ 350\ \textbf{let}\ \mathrm{concat8}(a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8) = h\ \textbf{in}$$

   we add a **let** at the beginning of the oracle $O_H$. (The occurrence 350 corresponds to the beginning of $O_H$. Occurrence numbers for each program point in the game can be shown by command **show_game occ**.) Then $O_H$ becomes **let** $\mathrm{concat8}(a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8) = h$ **in** **return**$(\mathrm{hash}(hk, h))$ **else return**$(\mathrm{hash}(hk, h))$. We also insert **let** $\mathrm{concat4}(b_1, b_2, b_3, b_4) = h$ **in** in the **else** branch of the previous **let**. So the oracle is now split in three, where the first part is for concat8 arguments, the second part for concat4 arguments, and the last part for other arguments.

4. By command **crypto** rom(hash), we apply the random oracle assumption on the function hash. CryptoVerif transforms each call to the function hash into a lookup in the previous calls to the function hash: if the same hash has already been computed, we return the same result; otherwise, we return a fresh random hash.

5. By command **crypto** uf_cma(signr), we apply the UF-CMA transformation after replacing $sk_S$ and $pk_S$ with their values, as in Section 5.3, and we simplify the obtained game. (The function signr is the deterministic signature function, which takes random coins as argument. The function sign is defined by **letfun** as in Example 5. It generates random coins and calls signr with these coins.)

6. By command **success**, CryptoVerif proves Property (1). Instead of using collision resistance as it did in Section 5.3, it relies on the negligible probability of collisions between fresh random hashes.

7. In order to prove the secrecy of the keys generated by the client, we want to transform the game using the CDH assumption. This transformation basically transforms equality tests of the form $M = \exp(g, \text{mult}(x,y))$ into false when the only usages of the random exponents $x$ and $y$ are for computing $\exp(g,x)$, $\exp(g,y)$ and equality tests of the form $M = \exp(g, \text{mult}(x,y))$. Indeed, the CDH assumption says that one has a negligible probability of computing $M$ such that $M = \exp(g, \text{mult}(x,y))$ knowing $\exp(g,x)$, $\exp(g,y)$ for random exponents $x$ and $y$. In order to use this transformation, we need to eliminate as many usages of $x$ and $y$ as possible.

   The game contains a process of the following form in the client:

$$\begin{array}{l} K_C \leftarrow \exp(f_C, x_C); \\ \textbf{find } w' \leq N \textbf{ suchthat defined}(id_{CS}[w'], \dots, f_S[w']) \wedge \\ \qquad id_{CC} = id_{CS}[w'] \wedge \dots \wedge f_C = f_S[w'] \textbf{ then} \\ \quad \dots IV_{CC} \leftarrow \dots \\ \oplus\ w'' \leq N_H \textbf{ suchthat defined}(a_1[w''], \dots, a_8[w'']) \wedge \\ \qquad id_{CC} = a_1[w''] \wedge \dots \wedge K_C = a_8[w''] \textbf{ then } \dots \textbf{ else } \dots \end{array} \tag{3}$$

   This **find** tests whether there exists a session of the server indexed by $w'$ that has exactly the same messages as the current session of the client, and if this is the case, it generates the session keys. In the other **then** branch and in the **else** branch of this **find**, we do not generate the keys.

   We use an **insert** command to add the following **find**:

$$\begin{array}{l} \textbf{find } w \leq N \textbf{ suchthat defined}(id_{CS}[w], \dots, f_S[w]) \wedge \\ \qquad id_{CC} = id_{CS}[w] \wedge \dots \wedge f_C = f_S[w] \textbf{ then} \end{array} \tag{4}$$

   above the definition of $K_C$ in (3). The rest of the code following the **find** (4) is duplicated in the **then** and **else** branches of that **find**.

   In subsequent simplifications (Step 9 below), in the **find** (3) that occurs in the **else** branch of (4), the first **then** branch is removed, because when we take the **else** branch of (4), no $w$ satisfying the condition can be found, so also no $w''$ satisfying the same condition, hence we never take the first **then** branch of (3). The **find** (3) that occurs in the **then** branch of (4) is transformed into its first **then** branch, because when we take the **then** branch of (4), (3) always finds a $w'$ equal to $w$. As a result, the usage of $K_C$ in $K_C = a_8[w'']$ disappears when the condition of (4) holds. All other usages of $K_C$ when this condition holds are of the form $M = \exp(g, \text{mult}(x,y))$, so they can be handled by the CDH assumption.

8. To continue helping CryptoVerif remove usages of $x$ and $y$, we distinguish cases depending on whether the server runs a session with the honest client or with the adversary. We use an **insert** command to insert the **find**:

$$\textbf{find } v \leq N \textbf{ suchthat defined}(e_C[v]) \wedge e_C[v] = e_S \textbf{ then} \tag{5}$$

before the creation of the shared Diffie-Hellman key $K_S$ in the server (middle of line 10 in Figure 8). This allows us to distinguish the case in which the group element $e_S$ of the server comes from the client (**then** branch) from the case in which $e_S$ comes from the adversary (**else** branch).

9. By command **simplify**, CryptoVerif simplifies the game. In particular, it renames the variable $K_S$ into two variables, $K_{S1}$ for the variable $K_S$ defined in the **then** branch of the **find** (5) and $K_{S2}$ for the one defined in the **else** branch of this **find**.

10. By command **crypto** cdh(exp), we transform the game using the CDH assumption. CryptoVerif automatically performs some preparatory steps before actually using CDH, and simplifies the game after applying CDH.

    Our extension is applied in the simplification that follows the application of CDH. Let us first explain it. After applying CDH, we arrive at a game of the following form:

```
**foreach** $i \leq N$ **do** $\dots$ key_exchange$_{2S}(\dots) := \dots$
    **find** $v \leq N$ **suchthat defined**$(e_C[v]) \wedge e_C[v] = e_S$ **then** $\dots$        (comes from (5))
    **else** $K_{S2} \leftarrow \exp(e_S, y_S); \dots$
| **foreach** $j \leq N$ **do** $\dots$ key_exchange$_{1C}(\dots) := \dots$
    $x_C \xleftarrow{R} Z; e_C \leftarrow \exp(g, x_C); \dots$
    **find** $w \leq n$ **suchthat defined**$(e_S[w], \dots) \wedge \dots \wedge (e_C = e_S[w])$ **then**
      $\dots$                                                          (comes from (4))
      **if defined**$(K_{S2}[w])$ **then** $\dots$ **else** $P$
```

    We want to prove that the condition **defined**$(K_{S2}[w])$ of the last test cannot be satisfied. Assuming that we reach the last test and $K_{S2}[w]$ is defined, we have taken the **else** branch of the **find** in key_exchange$_{2S}$ in the run of index $w$, so at the definition of $K_{S2}[w]$, we have that, for all $v$, **defined**$(e_C[v]) \wedge e_C[v] = e_S[w]$ does not hold. We have two cases:

    - If the variable $e_C[j]$ (the value of $e_C$ with the current index $j$, also denoted $e_C$) is defined before $K_{S2}[w]$, then at the definition of $K_{S2}[w]$, $e_C[j]$ was defined and for all $v$, **defined**$(e_C[v]) \wedge e_C[v] = e_S[w]$ does not hold. Taking $v = j$, **defined**$(e_C[j]) \wedge e_C[j] = e_S[w]$ does not hold, so $e_C[j] \neq e_S[w]$: the condition of the last **find** construct is false in this case, so we cannot reach the last test.
    - Otherwise, the variable $e_C[j]$ is defined after $K_{S2}[w]$, so $x_C[j]$ is defined after $e_S[w]$. Since $x_C[j]$ is chosen randomly after $e_S[w]$, it is independent of $e_S[w]$, so $e_C[j]$ is a random element chosen uniformly in $G$ independent of $e_S[w]$. Therefore, the probability that $e_C[j] = e_S[w]$ is $1/|G|$. We eliminate this collision, which happens with negligible probability, so that we also have $e_C[j] \neq e_S[w]$, so we also cannot reach the last test.

    This is a contradiction, so we cannot reach the last test with $K_{S2}[w]$ defined, hence we can replace this test with its **else** branch $P$, taking into account the collision probability $1/|G|$ in the probability of success of an attack. In other words, when the client makes a successful run with the server (the signature verification succeeds, so the **find** (4) succeeds), then the server has also used an element $e_S$ coming from the client, so we have taken the **then** branch of the **find** (5), so $K_{S2}$ is not defined.

Basically, the transformations we did previously allow us to distinguish cases depending on whether the exponents $x$ and $y$ are used in sessions between the honest client and server, or they are used in sessions with the adversary. Furthermore, for exponents $x$ and $y$ used in sessions between the honest client and server, the only usages of $x$ and $y$ left after the previous transformations are of the form $\exp(g,x)$, $\exp(g,y)$, and $M = \exp(g,\text{mult}(x,y))$. By the CDH assumption, these equality tests can then be replaced with false.

In particular, in the initial game, IVs (and session keys) are computed by formulas such as $IV_{CC} \leftarrow \text{iv\_of\_hash}(\text{hash}(hk,\text{concat4}(K_C,H_C,\text{"A"},H_C)))$. By the random oracle model, the call to hash is replaced with a **find** that compares $\text{concat4}(K_C,H_C,\text{"A"},H_C)$ to the arguments of the previous hash queries (Step 4). Due to the previous simplifications, it just compares $\text{concat4}(K_C,H_C,\text{"A"},H_C)$ to the hash queries $\text{concat4}(b_1,b_2,b_3,b_4)$ made by the adversary in oracle $O_H$. In case the same arguments are found, we compute the IV by truncating the result returned by the previous call to $O_H$, so in this case, the adversary would have the IV. Otherwise, we generate a fresh IV by returning $\text{iv\_of\_hash}(r)$ for a random $r$. Importantly, the former case is removed by the CDH assumption, since the comparison $b_1 = K_C$ is of the form $M = \exp(g,\text{mult}(x,y))$, so it is false. Hence, in fact, we always generate a fresh IV by returning $\text{iv\_of\_hash}(r)$ for a random $r$.

11. The function iv_of_hash truncates its input to the size of its output. Hence, if the argument of iv_of_hash is uniformly distributed, then so is its result. We give that information to CryptoVerif by adding the transformation hash_to_iv_random that transforms an assignment $x \leftarrow \text{iv\_of\_hash}(r)$ when $r$ is a fresh randomly generated value into $x \overset{R}{\leftarrow} IV$. By command **crypto** hash_to_iv_random, we apply this transformation: we replace the creation of $IV_{CC}$ outlined above with the generation of a random value in $IV$.

    At this point, by command **success**, CryptoVerif is able to see that $IV_{CC}$ is generated randomly and never used, and concludes that the secrecy of $IV_{CC}$ is guaranteed.

    The secrecy can be proved for the other IV and keys by just repeating this last step for each one of them (possibly using the truncation functions for MAC or encryption keys instead of iv_of_hash).

## 5.5 About the Secrecy of Messages Sent in the Tunnel

In our model, we cannot prove the secrecy of messages sent in the tunnel. This point is actually related to known weaknesses in SSH with CBC mode encryption (which is still the only required encryption mode) [7, 3]. CBC mode encryption with chained IVs is not IND-CPA (indistinguishable under chosen plaintext attacks [6]), and this insecurity also applies to SSH [7]. This problem appears clearly when we try to do the proof. Because CryptoVerif does not allow encryption and decryption to generate random values internally or to maintain an internal state, even the interface of encryption in SSH differs from the one of IND-CPA encryption: in SSH, encryption receives a non-random IV while IND-CPA encryption receives random coins, and decryption receives an IV while IND-CPA decryption does not. Moreover, the oracle that decrypts the first block of a packet to get its length leaks the first four bytes of every packet. In fact, because of properties of CBC mode, using this oracle, one can compute the first four bytes of the cleartext of any ciphertext block [3, Section 3.2]. This problem is actually related to a real attack against some SSH implementations [3]: in practice, the length field is not immediately obtained by the adversary, but can be determined by sending messages block by block until one gets a reply, leading to the leakage of the cleartext. Such problems would be likely to remain unnoticed with an analysis of SSH in the symbolic model; that is why it is important to prove the protocol in the computational model.

In order to get a security proof, we could use counter mode encryption as specified in [8] instead of CBC mode encryption, by relying on its recent formalization in [27]. That would probably require extensions of CryptoVerif to keep a mutable counter internally. More generally, the main limitations of our approach come from limitations of CryptoVerif: it currently cannot handle mutable state, and may also be unable to prove some protocols secure even if they can be encoded. Additionally, it would also be interesting to formalize the SSH authentication and connection protocols.

## 5.6 Implementation

In order to implement the SSH Transport Layer Protocol, we wrote the network code and the cryptographic primitives. The cryptographic primitives are for the most part an interface to Cryptokit. Some specific algorithm encapsulations used by SSH had to be implemented. Message building and parsing are also implemented as if they were cryptographic primitives, with a basic specification of their properties: in particular, parsing is the inverse of message building. The network code sends and receives messages from the network, and also does some basic non-cryptographic manipulations (for instance, it sends the identification string directly).

We have verified that our client and server correctly interoperate with OpenSSH. This shows that our implementation respects the message format and contents of SSH, and that it is a working implementation. However, we have omitted a few details of the SSH specification for simplicity: key re-exchange, IGNORE and DISCONNECT messages are not implemented yet. Since our compiler preserves security as shown in [17], our implementation also satisfies the authentication of the server and the secrecy of session keys shown on the specification in Sections 5.3 and 5.4 (assuming the cryptographic primitives are correctly implemented). In order to give an idea on the amount of code this work represents, the CryptoVerif specification amounts to 331 lines of code, and we generate from it 531 lines of OCaml, split among multiple files. The manually written code representing the primitives and the authentication and connection protocols amount to 1124 lines.

The throughput of our implementation when tunneling random data is about 30 MB/s, whereas OpenSSH using the same algorithms as our implementation (those described in Section 5.1) ramps up to 90 MB/s on a Dual Core 3.2 GHz. It is slower because our generated code and the cryptographic primitives in Cryptokit are both slower than their OpenSSH equivalents, but it is still usable. We believe that the main reason for this slower speed is that our implementation allocates and copies strings when building messages instead of using a single buffer that would be modified in place. It would theoretically be possible to implement an optimizing compiler that would avoid string copies as much as possible, but the generated code would then be more difficult to relate to its CryptoVerif specification, and the compiler and its proof would be more complicated. The time required by our implementation to do an handshake (tunnel establishment and user authentication) varies widely depending on how we implement random number generation: much time may be spent waiting for entropy in the random number generator. OpenSSH uses the random number generator `arc4random` which uses an ARC4 pseudo-random generator regularly seeded with entropy gathered by the kernel, to reduce this waiting time to a minimum. However, the Cryptokit library does not provide access to `arc4random`, so one needs to seed a pseudo-random generator with new entropy at each run of SSH. This entropy can be taken from `/dev/random`, which waits until the kernel gathered enough entropy, so this is secure but slow, or from `/dev/urandom`, which does not wait, so this is fast, but may not be secure in case there is not enough entropy available. One could obviously extend Cryptokit to have access to `arc4random`. Ignoring the waiting time in the random number generator, the handshake takes about 20 ms, both in our implementation and in OpenSSH, with an RSA key of 2048 bits and a Diffie-Hellman modulus of 2048 bits. (This is the user plus system time, measured on an average of 100 runs. To eliminate network delays, all these measures have been performed with the client and the server running on the same machine.)

## 6 Conclusion

We presented a compiler that translates an annotated CryptoVerif specification into an OCaml implementation. Thanks to this compiler and to CryptoVerif, we can, from a single specification of the protocol, both prove security properties of the protocol by CryptoVerif and get a runnable implementation of the protocol using our compiler. We proved in [17] that this compiler preserves security, so the generated implementation also satisfies the security properties proved on the protocol specification. We applied our work to the SSH Transport Layer Protocol: we proved the authentication of the server and the secrecy of the session keys, and we generated an implementation of the protocol that could interact with an existing implementation of SSH, namely OpenSSH. This work was also an opportunity to extend CryptoVerif so that it can make case distinctions depending on the order of definitions of variables; this extension was necessary in order to prove the secrecy of the session keys in SSH.

Our generated implementations do not include countermeasures against side-channel attacks. It would be interesting to add such countermeasures, or even to have tools to detect certain side-channel attacks or prove their absence. This is however long-term future work.

**Acknowledgments** This work was partly supported by the ANR project ProSe (decision ANR 2010-VERS-004). It was partly done while the authors were at École Normale Supérieure, Paris.

## References

[1] Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. Extracting and verifying cryptographic models from C protocol code by symbolic execution. In *Proc. of the 18th ACM Conference on Computer and Communications Security (CCS'11), Chicago, IL, USA*, pages 331–340. ACM, October 2011.

[2] Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. Computational verification of C protocol implementations by symbolic execution. In *Proc. of the 19th ACM Conference on Computer and Communications Security (CCS'12), Raleigh, NC, USA*, pages 712–723. ACM, October 2012.

[3] Martin R. Albrecht, Kenny G. Paterson, and Gaven J. Watson. Plaintext recovery attacks against SSH. In *Proc. of the 30th IEEE Symposium on Security and Privacy, Oakland, CA, USA*, pages 16–26. IEEE, May 2009.

[4] Matteo Avalle, Alfredo Pironti, Riccardo Sisto, and Davide Pozza. The JavaSPI framework for security protocol implementation. In *Proc. of the 6th International Conference on Availability, Reliability and Security (ARES'11), Vienna, Austria*, pages 746–751. IEEE, August 2011.

[5] Michael Backes, Dennis Hofheinz, and Dominique Unruh. CoSP: A general framework for computational soundness proofs. In *Proc. of the 16th ACM Conference on Computer and Communications Security (CCS'09), Chicago, IL, USA*, pages 66–78. ACM, November 2009.

[6] Mihir Bellare, Anand Desai, E. Jokipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption. In *Proc. of the 38th Annual Symposium on Foundations of Computer Science (FOCS'97), Miami Beach, Florida*, pages 394–403. IEEE, October 1997.

[7] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. Authenticated encryption in SSH: Provably fixing the SSH binary packet protocol. In *Proc. of the 9th ACM conference on Computer and communications security (CCS'02), Washington, DC, USA*, pages 1–11. ACM, November 2002.

[8] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. The secure shell (SSH) transport layer encryption modes, January 2006. `http://www.ietf.org/rfc/rfc4344.txt`.

[9] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andy Gordon, and Sergio Maffeis. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(2), 2011.

[10] Karthikeyan Bhargavan, Ricardo Corin, Cédric Fournet, and Eugen Zălinescu. Cryptographically verified implementations for TLS. In *Proc. of the 15th ACM Conference on Computer and Communications Security (CCS'08), Alexandria, VA, USA*, pages 459–468. ACM, October 2008.

[11] Karthikeyan Bhargavan, Cédric Fournet, and Andrew Gordon. Modular verification of security protocol code by typing. In *Proc. of the 37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'10), Madrid, Spain*, pages 445–456. ACM, January 2010.

[12] Karthikeyan Bhargavan, Cédric Fournet, Andrew Gordon, and Stephen Tse. Verified interoperable implementations of security protocols. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(1), 2008.

[13] Bruno Blanchet. Computationally sound mechanized proofs of correspondence assertions. In *Proc. of the 20th IEEE Computer Security Symposium (CSF'07), Venice, Italy*, pages 97–111. IEEE, July 2007.

[14] Bruno Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207, October-December 2008.

[15] Bruno Blanchet and David Pointcheval. Automated security proofs with sequences of games. In *Proc. of the 26th Annual International Cryptology Conference (CRYPTO'06), Santa Barbara, California, USA, LNCS*, volume 4117, pages 537–554. Springer-Verlag, August 2006.

[16] David Cadé and Bruno Blanchet. From computationally-proved protocol specifications to implementations. In *Proc. of the 7th International Conference on Availability, Reliability and Security (ARES'12), Prague, Chzech Republic*, pages 65–74. IEEE, August 2012.

[17] David Cadé and Bruno Blanchet. Proved generation of implementations from computationally secure protocol specifications. In *Proc. of the 2nd Conference on Principles of Security and Trust (POST 2013), Rome, Italy, LNCS*, volume 7796, pages 63–82. Springer-Verlag, March 2013.

[18] Sagar Chaki and Anupam Datta. ASPIER: An automated framework for verifying security protocol implementations. In *Proc. of the 22nd IEEE Computer Security Foundations Symposium (CSF'09), Port Jefferson, NY, USA*, pages 172–185. IEEE, July 2009.

[19] François Dupressoir, Andrew D. Gordon, Jan Jürjens, and David A. Naumann. Guiding a general-purpose C verifier to prove cryptographic protocols. In *Proc. of the 24th IEEE Computer Security Foundations Symposium (CSF'11), Cernay-la-Ville, France*, pages 3–17. IEEE, June 2011.

[20] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. Modular code-based cryptographic verification. In *Proc. of the 18th ACM Conference on Computer and Communications Security (CCS'11), Chicago, IL, USA*, pages 341–350. ACM, October 2011.

[21] Jean Goubault-Larrecq and Fabrice Parrennes. Cryptographic protocol analysis on real C code. In *Proc. of the 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05), Paris, France, LNCS*, volume 3385, pages 363–379. Springer-Verlag, January 2005.

[22] Jan Jürjens. Security analysis of crypto-based Java programs using automated theorem provers. In *Proc. of the 21th IEEE/ACM International Conference on Automated Software Engineering (ASE'06), Tokyo, Japan*, pages 167–176. IEEE, September 2006.

[23] Tero Kivinen and Mika Kojo. RFC 3526: More modular exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE), May 2003. `http://www.ietf.org/rfc/rfc3526.txt`.

[24] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[25] Giuseppe Milicia. $\chi$-spaces: Programming security protocols. In *Proc. of the 14th Nordic Workshop on Programming Theory (NWPT'02), Tallinn, Estonia*, November 2002.

[26] Nicholas O'Shea. Using Elyjah to analyse Java implementations of cryptographic protocols. In *Proc. of the Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (FCS-ARSPA-WITS'08), Pittsburgh, PA, USA*, 2008.

[27] Kenneth G. Paterson and Gaven J. Watson. Plaintext-dependent decryption: A formal security treatment of SSH-CTR. In *Proc. of Eurocrypt 2010, French Riviera*, volume 6110, pages 345–361. Springer-Verlag, May-June 2010. Full version available at `http://eprint.iacr.org/2010/095`.

[28] Alfredo Pironti and Riccardo Sisto. An experiment in interoperable cryptographic protocol implementation using automatic code generation. In *Proc. of the IEEE Symposium on Computers and Communications*

*(ISCC'07), Aveiro, Portugal*, pages 839–844. IEEE, July 2007.

[29] Alfredo Pironti and Riccardo Sisto. Provably correct Java implementations of spi calculus security protocols specifications. *Computers and Security*, 29(3):302–314, May 2010.

[30] Erik Poll and Aleksy Schubert. Verifying an implementation of SSH. In *Proc. of the 17th Annual Workshop on Information Technologies (WITS'07), Braga, Portugal*, March 2007.

[31] Davide Pozza, Riccardo Sisto, and Luca Durante. Spi2Java: Automatic cryptographic protocol Java code generation from spi calculus. In *Proc. of the Advanced Information Networking and Applications, 2004 (AINA'04), Fukuoka, Japan*, volume 1, pages 400–405. IEEE, March 2004.

[32] Dawn Song, Adrian Perrig, and Doantam Phan. AGVI—Automatic Generation, Verification, and Implementation of security protocols. In *Proc. of the 13th Conference on Computer Aided Verification (CAV'01), Paris, France, LNCS*, volume 2102, pages 241–245. Springer-Verlag, July 2001.

[33] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bharagavan, and Jean Yang. Secure distributed programming with value-dependent types. In *Proc. of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP'11), Tokyo, Japan*, pages 266–278. ACM, September 2011.

[34] Tatu Ylönen. RFC 4251: The Secure Shell (SSH) Protocol Architecture, January 2006. `http://www.ietf.org/rfc/rfc4251.txt`.

[35] Tatu Ylönen. RFC 4252: The Secure Shell (SSH) Authentication Protocol, January 2006. `http://www.ietf.org/rfc/rfc4252.txt`.

[36] Tatu Ylönen. RFC 4253: The Secure Shell (SSH) Transport Layer Protocol, January 2006. `http://www.ietf.org/rfc/rfc4253.txt`.

[37] Tatu Ylönen. RFC 4254: The Secure Shell (SSH) Connection Protocol, January 2006. `http://www.ietf.org/rfc/rfc4254.txt`.