HAL
open science

From Computationally-proved Protocol Specifications to Implementations

David Cadé, Bruno Blanchet

To cite this version:

David Cadé, Bruno Blanchet. From Computationally-proved Protocol Specifications to Implementations. 7th International Conference on Availability, Reliability and Security (AReS 2012), 2012, Prague, Czech Republic. pp.65–74. hal-00863382

HAL Id: hal-00863382

https://inria.hal.science/hal-00863382v1

Submitted on 18 Sep 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

From Computationally-proved Protocol Specifications to Implementations

David Cadé and Bruno Blanchet
École Normale Supérieure, CNRS, INRIA, Paris, France
{cade,blanchet}@di.ens.fr

***Abstract*—This paper presents a novel framework for proving specifications of security protocols in the computational model and generating runnable implementations from such proved specifications. We rely on the computationally-sound protocol verifier CryptoVerif for proving the specification, and we have implemented a compiler that translates a CryptoVerif specification into an implementation in OCaml. We have applied this compiler 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. The secrecy of messages sent over the SSH tunnel cannot be proved due to known weaknesses in SSH with CBC-mode encryption.**

I. Introduction

The verification of security protocols is an important research area since the 1990s: the design of security protocols is notoriously error-prone, and errors can have serious consequences. In order to verify protocols, two main models have been considered: the symbolic model and the computational model. The symbolic model represents messages as terms in a term algebra. The attacker can only create terms in this algebra, and so it can only use the cryptographic primitives defined in it. The computational model is the model used by cryptographers, in which messages are bitstrings and attackers are polynomial-time probabilistic Turing machines. Proofs in the latter model are more difficult than in the former, but yield a much more precise analysis of the protocol. However, proving specifications of protocols in such models is not sufficient. Even if the specification is correct, an implementation of the protocol may be insecure, because of errors in implementation details left unspecified at the specification level, or because the specification has not been 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 secure in the computational model.

There are two ways of obtaining a secure implementation: write an implementation of a protocol, analyze it to extract a protocol specification and then prove this specification, or in the other way around, write a specification, prove it correct, and then generate an implementation from it. We chose the latter way for two reasons. First, we believe that starting by designing a protocol, formalizing it, proving it secure formally, and only after that implementing it, is a better methodology than starting from the implementation. Second, generating protocol implementations is also relatively easier than analyzing them; analyzing existing protocol implementations not written for verification is especially difficult, and very few methods can do that (see related work below).

Therefore, we start from a formal specification of the protocol. In order to prove the specified protocol secure in the computational model, we rely on the automatic protocol verifier CryptoVerif [1–3]. This verifier can prove secrecy and authentication properties. The generated proofs are proofs by sequences of games, like the manual proofs written by cryptographers. The games are formalized in a probabilistic polynomial-time process calculus.

In order to generate protocol implementations, we wrote a compiler that takes a CryptoVerif specification and returns an implementation in OCaml (http://caml.inria.fr). We chose this language for several reasons, starting with the fact that it is memory safe and has a clean semantics, which is useful to prove the correctness of the compiler. OCaml is a functional language, which also facilitates the compilation because the CryptoVerif specification uses oracles that can be immediately translated into functions. A cryptographic library is also available for OCaml, Cryptokit (http://forge.ocamlcore.org/projects/cryptokit/). It would obviously be possible to adapt our approach to other target languages, such as Java or C, if desired. We believe that adding a new target language to our compiler would be much easier than writing an analyzer for a new language, which is also an interest of the approach that generates implementations. To implement the compiler, we had to enrich the CryptoVerif specification with annotations that specify details of the implementation of the protocol. There are two kinds of annotations. First, the annotations that specify how to divide the protocol in different parts corresponding to different roles, for example, key generation, server, and client. Second, the annotations that specify how to implement the various cryptographic primitives and types. In order to obtain strong guarantees that the code generated by this compiler is correct, we should prove the correctness of the compiler. This proof is still in progress.

To show the applicability of our approach, we crafted a CryptoVerif specification of the SSH Transport Layer protocol, and used our compiler to generate the corresponding implementation. We proved the authentication of the server and the secrecy of the session keys using CryptoVerif, 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 [4] first generates a protocol from security requirements, proves its correctness using the protocol verifier Athena, then compiles the protocol into Java. χ-spaces [5] provide a domain-specific language for specifying protocols, which can be interpreted or compiled to Java. Spi2Java [6, 7] translates spi-calculus protocols into Java implementations; the soundness of this translation is proved in [7]. 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 [8]. The JavaSPI framework [9] 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 [10] analyzes protocols written in C by translating them into Horn clauses, given as input to the $\mathcal{H}_1$ prover. Similarly, JavaSec [11] translates Java programs into first-order logic formulas, given as input to the first-order theorem prover e-SETHEO. Poll and Schubert [12] 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 [13] 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. [14] use the general-purpose C verifier VCC to prove both memory safety and security properties of protocols.

The tool FS2PV [15] 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 [16]. Similarly, Elijah [17] translates Java programs into LySa protocol specifications, which can be verified by the LySatool. Aizatulin et al. [18] 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 [13], it is one of the rare methods that can analyze implementations not written specifically for verification. The tools F7 and F* [19–21] 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 [16]. The F7 approach has also been extended to the computational model [22], but still requires type annotations to help the proof. [18] provides computational security guarantees by applying the computational soundness result of [23]: 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 [18] have recently extended their approach to generate a CryptoVerif model [24], 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.

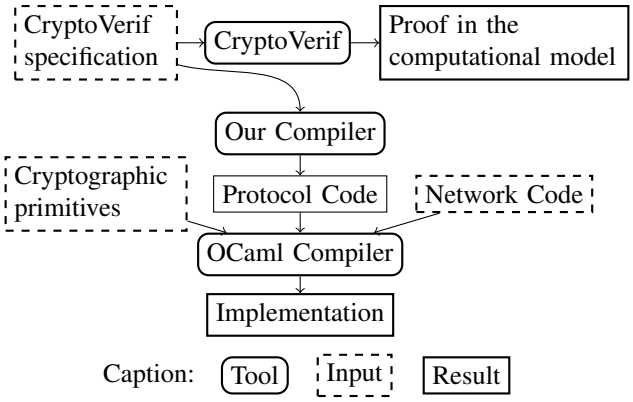

Figure 1. Overview of the approach

Outline: Section II is a general presentation of our approach. Section III describes the specification language used by our compiler and Section IV details how this language is compiled into OCaml. Finally, Section V presents the application of this compiler to the SSH protocol. Our compiler and our model and implementation of the SSH Transport Layer protocol are available as part of the CryptoVerif distribution at http://www.cryptoverif.ens.fr.

II. 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 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.

III. 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.

A. Protocol Representation Language

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

The language also uses function symbols f. Each function symbol comes with a type declaration $f : T_1 \times \ldots \times T_m \rightarrow T$, 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 (both taking two values of the same type T and returning a value of type $bool$), $M \vee N$ for the boolean or, $M \wedge N$ for the boolean and, $\neg M$ for the boolean negation (taking and returning values of type $bool$).

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

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

Figure 2. Protocol representation language

$f(M_1, \ldots, M_m)$ returns the result of applying the function f to $M_1, \ldots, M_m$.

This language distinguishes oracle declarations and oracle bodies. The oracle declarations provide some oracles, which can be called by the adversary, while the 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 $\mathbf{foreach}\ i \leq n\ \mathbf{do}\ Q$ 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 $O(x_1 : T_1, \ldots, x_k : T_k) := P$ declares the oracle O, taking arguments $x_1, \ldots, x_k$ of types $T_1, \ldots, T_k$ respectively. The result of this oracle is computed by the oracle body P.

The oracle body $x \stackrel{R}{\leftarrow} T; P$ 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 $x \stackrel{R}{\leftarrow} T$. 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 $x : T \leftarrow M; P$ stores the value of M (which must be in T) in x and executes P. The test $\mathbf{if}\ M\ \mathbf{then}\ P\ \mathbf{else}\ P'$ executes P when M evaluates to true and P' otherwise. The construct $\mathbf{event}\ e(M_1, \ldots, M_l); P$ executes the event

$e(M_1, \ldots, M_l)$, then runs P. This event records that a certain program point has been reached with certain values of $M_1, \ldots, M_l$, but otherwise does not affect the execution of the system. (Events only serve in specifying authentication properties [2].) The construct $\mathbf{return}(M_1, \ldots, M_k); Q$ returns the result $M_1, \ldots, M_k$ 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; $\mathbf{insert}\ Tbl(M_1, \ldots, M_k); P$ inserts the element $M_1, \ldots, M_k$ in the table Tbl; $\mathbf{get}\ Tbl(x_1 : T_1, \ldots, x_k : T_k)\ \mathbf{suchthat}\ M\ \mathbf{in}\ P\ \mathbf{else}\ P'$ tries to retrieve an element $(x_1, \ldots, x_n)$ in the table Tbl such that M is true. When such an element is found, it executes P with $x_1, \ldots, x_n$ 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 [1]. 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 transformation proceeds by storing the inserted list elements in fresh array variables, and looking up in these arrays instead of performing **get**.

CryptoVerif also offers a pattern-matching construct. A function $f : T_1 \times \ldots \times T_m \to T$ 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 $f_j^{-1} : T \to T_j$ $(1 \leq j \leq m)$ such that $f_j^{-1}(f(x_1, \ldots, x_m)) = x_j$. We can then define the pattern-matching construct $\mathbf{let}\ f(x_1, \ldots, x_m) = M\ \mathbf{in}\ P\ \mathbf{else}\ Q$ as an abbreviation for $y : T \leftarrow M; x_1 : T_1 \leftarrow f_1^{-1}(y); \ldots; x_m : T_m \leftarrow f_m^{-1}(y); \mathbf{if}\ f(x_1, \ldots, x_m) = y\ \mathbf{then}\ P\ \mathbf{else}\ Q$. This construct tries to extract the values of $x_1, \ldots, x_n$ such that $f(x_1, \ldots, x_n) = M$, and runs P when this extraction succeeds, and Q when it fails. We generalize this construct to $\mathbf{let}\ N = M\ \mathbf{in}\ P\ \mathbf{else}\ Q$ where N is built from `compos` functions and variables.

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. Types can be omitted in assignments.

The original CryptoVerif language appears in two versions, using channels [1, 2] or oracles [3]. 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 [1], 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 1 *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:*

$$\begin{aligned}
&\mathrm{Ostart}() := rK_{ab} \stackrel{R}{\leftarrow} \mathit{keyseed}; K_{ab} \leftarrow \mathrm{kgen}(rK_{ab});\\
&\qquad \mathbf{return}(); (\mathbf{foreach}\ i_1 \leq N\ \mathbf{do}\ P_A\\
&\qquad\qquad \mid \mathbf{foreach}\ i_2 \leq N\ \mathbf{do}\ P_B)\\
&P_A = \mathrm{OA}() := x \stackrel{R}{\leftarrow} \mathit{nonce}; s \stackrel{R}{\leftarrow} \mathit{seed};\\
&\qquad \mathbf{return}(\mathrm{enc}(x, K_{ab}, s))\\
&P_B = \mathrm{OB}(m : \mathit{bitstring}) :=\\
&\qquad \mathbf{let}\ \mathrm{injbot}(r' : \mathit{nonce}) = \mathrm{dec}(m, K_{ab})\ \mathbf{in}\\
&\qquad \mathbf{return}()
\end{aligned}$$

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 $\mathit{bitstring}_\perp$: 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 $\mathit{bitstring}_\perp$, 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***).*

B. 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 them into several OCaml modules. The boundaries of OCaml modules are marked as follows. The annotation $\mu\,[x_1 > \texttt{"}\mathit{filex}_1\texttt{"}, \ldots, x_n > \texttt{"}\mathit{filex}_n\texttt{"}, y_1 < \texttt{"}\mathit{filey}_1\texttt{"}, \ldots, y_m < \texttt{"}\mathit{filey}_m\texttt{"}]\,\{$ indicates the beginning of the OCaml module μ. It should be placed just above an oracle declaration Q. The indication $x_i > \texttt{"}\mathit{filex}_i\texttt{"}$ 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 modules defined after the end of μ; these modules will read it automatically from the file filex_i. The indication $y_i < \texttt{"}\mathit{filey}_i\texttt{"}$ means that the module μ will read at initialization the value of the variable y_i from the files filey_i. The variable y_i must be free in μ (i.e. it is defined before the beginning of μ). A declaration $x > \texttt{"}\mathit{filex}\texttt{"}$ in a module μ' above μ implicitly implies $x < \texttt{"}\mathit{filex}\texttt{"}$ in μ when μ uses x: x is written to filex in μ' and read in μ. All variables free in module μ must be declared as being read from a file in μ, 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 OCaml modules, 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 OCaml module. It must be placed just after a $\mathbf{return}$ statement.

Example 2 *Let us annotate the process we have seen in Example 1.*

$$\begin{array}{l}\mu_{\mathit{Keygen}}[K_{ab} > \texttt{"}\mathit{keyfile}\texttt{"}]\{\mathrm{Ostart}() := \ldots \mathbf{return}()\};\\ \quad(\mathbf{foreach}\ i_1 \leq N\ \mathbf{do}\ P_A \mid \mathbf{foreach}\ i_2 \leq N\ \mathbf{do}\ P_B)\\ P_A = \mu_A\{\mathrm{OA}() := \ldots\\ P_B = \mu_B\{\mathrm{OB}(m : \mathit{bitstring}) := \ldots\end{array}$$

We divide the process into three parts. First, the key generation part is represented by the module μ_{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 parts of the process. The module μ_A, which contains the oracle* OA*, corresponds to the role of A, and the module μ_B, which contains the oracle* OB*, corresponds to the role of B. For these two modules, 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 $\mathbb{G}_{\mathbf{T}}(T) \to \mathbf{string}$, 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 $\mathbf{string} \to \mathbb{G}_{\mathbf{T}}(T)$, 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 $\mathbf{Bad_file}$; this exception is raised only when a file has been corrupted.
- the predicate function $\mathbb{G}_{\mathbf{pred}}(T)$ of type $\mathbb{G}_{\mathbf{T}}(T) \to \mathbf{bool}$, 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 $\mathbf{unit} \to \mathbb{G}_{\mathbf{T}}(T)$, which returns a random element uniformly chosen in type T.

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

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: $\mathbf{letfun}\ f(x_1 : T_1, \ldots, x_n : T_n) = M$. Specifying an OCaml implementation for these macros is optional. When the OCaml implementation is not specified, our compiler generates code according to the $\mathbf{letfun}$ macro. When the OCaml implementation is specified, it is used when generating the OCaml code, while the CryptoVerif macro defined by $\mathbf{letfun}$ 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 $\mathbf{letfun}$ first makes the random choices, then calls a deterministic function.

IV. The Translation into OCaml

Our compiler automatically translates the CryptoVerif language into OCaml. Let us describe this translation.

The annotations of Section III-B split the CryptoVerif code into multiple parts corresponding to different OCaml modules. For each module μ, let Q be the oracle declaration that follows $\mu\,[\ldots]\,\{$. Let Q_0 be obtained by removing code that follows closing braces $\}$ in Q. Q_0 is the CryptoVerif code for module μ. Our compiler translates the oracles of Q_0 into OCaml functions. More precisely, the implementation of the module μ consists of the $\mathbf{init}$ function, which reads the values of the variables required by the oracles in Q_0 from the files, and returns the functions corresponding to the oracles declared by Q_0. Functions corresponding to the oracles declared after a $\mathbf{return}$ in Q_0 are not returned by $\mathbf{init}$, but will be returned by that $\mathbf{return}$, like continuations.

Hence, the available functions correspond exactly to the oracles that can be called. 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:

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

The function oracles takes an oracle declaration Q and returns a set 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:

$$\text{oracles}(0) = \emptyset$$
$$\text{oracles}(Q_1 \mid Q_2) = \text{oracles}(Q_1) \cup \text{oracles}(Q_2)$$
$$\text{oracles}(\mathbf{foreach}\ i \leq n\ \mathbf{do}\ Q) = \{(Q', \text{true}) \mid (Q', b) \in \text{oracles}(Q) \text{ for some } b\}$$
$$\text{oracles}(O(x_1, \ldots, x_k) := P) = \{(O(x_1, \ldots, x_k) := P, \text{false})\}$$

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 module, 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 III-B, a module 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: $\mathbb{G}_{\mathbf{file}}(x) = \mathbf{write_file}\ f\ (\mathbb{G}_{\mathbf{ser}}(T_x)\ \mathbb{G}_{\mathbf{var}}(x))$ when variable x is written to file f in module μ, that is, μ is annotated with $x > f$, and $\mathbb{G}_{\mathbf{file}}(x) = ()$ when x is not written to a file.

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

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

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

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

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

$$\mathbb{G}(\mathbf{end}) = \mathbf{raise\ Match_fail}$$

$$\mathbb{G}(\mathbf{insert}\ Tbl(M_1, \ldots, M_k); P) = \mathbf{add_to_table}\ \mathbb{G}_{\mathbf{table}}(Tbl)\ (\mathbb{G}_{\mathbf{ser}}(T_{M_1})\ \mathbb{G}_{\mathbf{M}}(M_1), \ldots, \mathbb{G}_{\mathbf{ser}}(T_{M_k})\ \mathbb{G}_{\mathbf{M}}(M_k)); \mathbb{G}(P)$$

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

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

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

We translate $x \stackrel{R}{\leftarrow} T; P$ by binding the variable $\mathbb{G}_{\mathbf{var}}(x)$ 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 $\mathbb{G}_{\mathbf{var}}(x)$ 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

$$
\begin{array}{l}
\mathbb{G}_{\mathbf{O}}(O(x_1 : T_1, \ldots, x_k : T_k) := P, \text{false}) = \\
\quad (\textbf{let } token = \textbf{ref } \text{true } \textbf{in} \\
\quad\quad \textbf{function } (\mathbb{G}_{\mathbf{var}}(x_1), \ldots, \mathbb{G}_{\mathbf{var}}(x_k)) \rightarrow \\
\quad\quad\quad \textbf{if } (!token) \ \&\& \ (\mathbb{G}_{\mathbf{pred}}(T_1)\ \mathbb{G}_{\mathbf{var}}(x_1)) \ \&\& \\
\quad\quad\quad\quad \ldots \ \&\& \ (\mathbb{G}_{\mathbf{pred}}(T_k)\ \mathbb{G}_{\mathbf{var}}(x_k)) \textbf{ then} \\
\quad\quad\quad\quad (token := \text{false};\ \mathbb{G}(P)) \\
\quad\quad\quad \textbf{else raise Bad_call})
\end{array}
$$

$$
\begin{array}{l}
\mathbb{G}_{\mathbf{O}}(O(x_1 : T_1, \ldots, x_k : T_k) := P, \text{true}) = \\
\quad (\textbf{function } (\mathbb{G}_{\mathbf{var}}(x_1), \ldots, \mathbb{G}_{\mathbf{var}}(x_k)) \rightarrow \\
\quad\quad \textbf{if } (\mathbb{G}_{\mathbf{pred}}(T_1)\ \mathbb{G}_{\mathbf{var}}(x_1)) \ \&\& \ \ldots \ \&\& \\
\quad\quad\quad (\mathbb{G}_{\mathbf{pred}}(T_k)\ \mathbb{G}_{\mathbf{var}}(x_k)) \textbf{ then } \mathbb{G}(P) \\
\quad\quad \textbf{else raise Bad_call})
\end{array}
$$

Figure 4. Translation of an oracle

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

$$
\begin{array}{l}
\textbf{let } token = \textbf{ref } \text{true} \\
\textbf{let init} = \textbf{function } () \rightarrow \\
\quad \textbf{if } (!token) \textbf{ then} \\
\quad\quad (token := \text{false}; \\
\quad\quad\ \textbf{let } \mathbb{G}_{\mathbf{var}}(x_1) = \mathbb{G}_{\mathbf{deser}}(T_{x_1})\ (\textbf{read_file } f_1) \textbf{ in } \ldots \\
\quad\quad\ \textbf{let } \mathbb{G}_{\mathbf{var}}(x_m) = \mathbb{G}_{\mathbf{deser}}(T_{x_m})\ (\textbf{read_file } f_m) \textbf{ in} \\
\quad\quad\quad (\mathbb{G}_{\mathbf{O}}(Q_1, b_1), \ldots, \mathbb{G}_{\mathbf{O}}(Q_k, b_k))) \\
\quad \textbf{else raise Bad_call}
\end{array}
$$

Figure 5. The **init** function for the module μ

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 $\mathbb{G}_{\mathbf{filter}}((x_1, \ldots, x_k), M)$ 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** $\rightarrow$ $'\mathbf{a}$) $\rightarrow$ $'\mathbf{a}$ **list** such that **read_table** f_{Tbl} *filter* reads the table file f_{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** $\mathbb{G}_{\mathbf{table}}(Tbl)\ \mathbb{G}_{\mathbf{filter}}((x_1, \ldots, x_k), M)$, 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 $(\mathbb{G}_{\mathbf{var}}(x_1), \ldots, \mathbb{G}_{\mathbf{var}}(x_k))$ accordingly and add them to their respective files if necessary, and finally we continue with the translation of the process P.

An oracle $O(x_1, \ldots, x_n) := P$ 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 μ consists in the **init** function presented in Figure 5. It begins 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 $token$.

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.

V. 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 results.

A. 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 [25]. The SSH Transport Layer Protocol 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 authenticates the client. The SSH Connection Protocol multiplexes multiple channels through the tunnel.

We concentrated our efforts on the Transport Layer 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 [26] raise an additional difficulty as discussed below.
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 [27]. 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.
   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 key" 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), two initialization vectors (IVs) for the encryption, and two keys for MAC, 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 counter 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 (four first 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 is not chosen yet).

B. Our application

We have modeled the SSH Transport Layer Protocol in the CryptoVerif specification language. We have then 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.

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 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. This 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.

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. 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. Its throughput when tunnelling random data is about 30 MB/s, whereas OpenSSH 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.

A few tricks were needed in order to get this implementation to work. We model the SSH tunnel by oracles that get an encrypted packet from the network and return the clear packet to the application, and get a clear packet from the application and return the encrypted packet to the network code. SSH with AES128-CBC (or other CBC mode encryptions) uses CBC mode [28, 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.

Moreover, the messages after the key exchange are completely encrypted under the key derived from the key exchange, the five first bytes containing the size of the message included. Therefore, an implementation must decrypt the first block of the message to get its size, then input the rest of the message, decrypt it, and then check that the MAC that follows in the stream is correct. So we implemented reading a message by two successive oracles: first, an oracle that takes the first packet of the message, and returns the size of the message (so that the network code can input a message of the required length), then an oracle that takes the rest of the message and its MAC, checks the MAC and returns the decrypted message if the MAC is correct.

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) [29, 30]. CBC mode encryption with chained IVs is not IND-CPA (indistinguishable under chosen plaintext attacks [31]), and this insecurity also applies to SSH [29]. 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 [30, Section 3.2]. This problem is actually related to a real attack against some SSH implementations [30]: 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 [26] instead of CBC mode encryption, by relying on its recent formalization in [32]. 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.

VI. 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 have 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.

To make sure that the implementations generated by our compiler are secure, we need to prove that if the specification satisfies a certain security property, then the generated implementation also satisfies it. The proof relates the traces of the CryptoVerif specification and those of the generated OCaml implementation. It is still in progress.

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).

References

[1] B. Blanchet, "A computationally sound mechanized prover for security protocols," *IEEE Transactions on Dependable and Secure Computing*, vol. 5, no. 4, pp. 193–207, 2008.

[2] ——, "Computationally sound mechanized proofs of correspondence assertions," in *CSF'07*. IEEE, 2007, pp. 97–111.

[3] B. Blanchet and D. Pointcheval, "Automated security proofs with sequences of games," in *CRYPTO'06*, ser. LNCS, vol. 4117. Springer, 2006, pp. 537–554.

[4] D. Song, A. Perrig, and D. Phan, "AGVI—Automatic Generation, Verification, and Implementation of security protocols," in *CAV'01*, ser. LNCS, vol. 2102. Springer, 2001, pp. 241–245.

[5] G. Milicia, "χ-spaces: Programming security protocols," in *NWPT'02*, 2002.

[6] D. Pozza, R. Sisto, and L. Durante, "Spi2Java: Automatic cryptographic protocol Java code generation from spi calculus," in *AINA'04*, vol. 1. IEEE, 2004, pp. 400–405.

[7] A. Pironti and R. Sisto, "Provably correct Java implementations of spi calculus security protocols specifications," *Computers and Security*, vol. 29, no. 3, pp. 302–314, 2010.

[8] ——, "An experiment in interoperable cryptographic protocol implementation using automatic code generation," in *ISCC'07*. IEEE, 2007, pp. 839–844.

[9] M. Avalle, A. Pironti, R. Sisto, and D. Pozza, "The JavaSPI framework for security protocol implementation," in *ARES'11*. IEEE, 2011, pp. 746–751.

[10] J. Goubault-Larrecq and F. Parrennes, "Cryptographic protocol analysis on real C code," in *VMCAI'05*, ser. LNCS, vol. 3385. Springer, 2005, pp. 363–379.

[11] J. Jürjens, "Security analysis of crypto-based Java programs using automated theorem provers," in *ASE'06*. IEEE, 2006, pp. 167–176.

[12] E. Poll and A. Schubert, "Verifying an implementation of SSH," in *WITS'07*, 2007.

[13] S. Chaki and A. Datta, "ASPIER: An automated framework for verifying security protocol implementations," in *CSF'09*. IEEE, 2009, pp. 172–185.

[14] F. Dupressoir, A. D. Gordon, J. Jürjens, and D. A. Naumann, "Guiding a general-purpose C verifier to prove cryptographic protocols," in *CSF'11*. IEEE, 2011, pp. 3–17.

[15] K. Bhargavan, C. Fournet, A. Gordon, and S. Tse, "Verified interoperable implementations of security protocols," *ACM TOPLAS*, vol. 31, no. 1, 2008.

[16] K. Bhargavan, R. Corin, C. Fournet, and E. Zălinescu, "Cryptographically verified implementations for TLS," in *CCS'08*. ACM, 2008, pp. 459–468.

[17] N. O'Shea, "Using Elyjah to analyse Java implementations of cryptographic protocols," in *FCS-ARSPA-WITS'08*, 2008.

[18] M. Aizatulin, A. D. Gordon, and J. Jürjens, "Extracting and verifying cryptographic models from C protocol code by symbolic execution," in *CCS'11*. ACM, 2011, pp. 331–340.

[19] J. Bengtson, K. Bhargavan, C. Fournet, A. Gordon, and S. Maffeis, "Refinement types for secure implementations," *ACM TOPLAS*, vol. 33, no. 2, 2011.

[20] K. Bhargavan, C. Fournet, and A. Gordon, "Modular verification of security protocol code by typing," in *POPL'10*. ACM, 2010, pp. 445–456.

[21] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bharagavan, and J. Yang, "Secure distributed programming with value-dependent types," in *ICFP'11*. ACM, 2011, pp. 266–278.

[22] C. Fournet, M. Kohlweiss, and P.-Y. Strub, "Modular code-based cryptographic verification," in *CCS'11*. ACM, 2011, pp. 341–350.

[23] M. Backes, D. Hofheinz, and D. Unruh, "CoSP: A general framework for computational soundness proofs," in *CCS'09*. ACM, 2009, pp. 66–78.

[24] M. Aizatulin, A. D. Gordon, and J. Jürjens, "Computational verification of C protocol implementations by symbolic execution," unpublished manuscript available at http://users.mct.open.ac.uk/ma4962/files/computational2012-full.pdf.

[25] T. Ylönen, "RFC 4251–4254: The Secure Shell (SSH) Protocol," 2006, http://www.ietf.org/rfc/rfc425[1–4].txt.

[26] M. Bellare, T. Kohno, and C. Namprempre, "The secure shell (SSH) transport layer encryption modes," 2006, http://www.ietf.org/rfc/rfc4344.txt.

[27] T. Kivinen and M. Kojo, "RFC 3526: More modular exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)," 2003, http://www.ietf.org/rfc/rfc3526.txt.

[28] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996.

[29] M. Bellare, T. Kohno, and C. Namprempre, "Authenticated encryption in SSH: Provably fixing the SSH binary packet protocol," in *CCS'02*. ACM, 2002, pp. 1–11.

[30] M. R. Albrecht, K. G. Paterson, and G. J. Watson, "Plaintext recovery attacks against SSH," in *IEEE Symposium on Security and Privacy*. IEEE, 2009, pp. 16–26.

[31] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway, "A concrete security treatment of symmetric encryption," in *FOCS'97*. IEEE, 1997, pp. 394–403.

[32] K. G. Paterson and G. J. Watson, "Plaintext-dependent decryption: A formal security treatment of SSH-CTR," in *Eurocrypt 2010*, ser. LNCS, vol. 6110. Springer, 2010, pp. 345–361, full version available at http://eprint.iacr.org/2010/095.