Automatically Verified Mechanized Proof of One-Encryption Key Exchange

Bruno Blanchet
INRIA, École Normale Supérieure, CNRS
Paris, France
blanchet@di.ens.fr

***Abstract*—We present a mechanized proof of the password-based protocol One-Encryption Key Exchange (OEKE) using the computationally-sound protocol prover CryptoVerif. OEKE is a non-trivial protocol, and thus mechanizing its proof provides additional confidence that it is correct. This case study was also an opportunity to implement several important extensions of CryptoVerif, useful for proving many other protocols. We have indeed extended CryptoVerif to support the computational Diffie-Hellman assumption. We have also added support for proofs that rely on Shoup's lemma and additional game transformations. In particular, it is now possible to insert case distinctions manually and to merge cases that no longer need to be distinguished. Eventually, some improvements have been added on the computation of the probability bounds for attacks, providing better reductions. In particular, we improve over the standard computation of probabilities when Shoup's lemma is used, which allows us to improve the bound given in a previous manual proof of OEKE, and to show that the adversary can test at most one password per session of the protocol. In this paper, we present these extensions, with their application to the proof of OEKE. All steps of the proof, both automatic and manually guided, are verified by CryptoVerif.**

***Keywords*-Automatic proofs, Formal methods, Provable security, Protocols, Password-based authentication**

I. Introduction

Since the beginning of public-key cryptography, more and more complex security notions have been defined, with protocols getting also more intricate. Initially, a long time without attack was a good argument in favor of the security of a scheme. But some schemes took a long time before being broken. A famous example is the Chor-Rivest cryptosystem [1], [2], which took more than 10 years to be totally broken [3]. Nowadays, the lack of attacks is no longer considered as a security validation, and provable security is a requirement for any new proposal.

The basic idea of provable security consists in reducing a well-known hard problem to an attack, in the complexity theory framework. Such a reduction guarantees that an efficient adversary against the cryptosystem could be converted into an efficient algorithm against the hard problem. First security proofs were essentially theoretical, providing polynomial reductions only. But "exact security" [4] or "concrete security" [5] asked for more efficient reductions.

Unfortunately, a security result should be considered with care. As explained above, it consists of a theorem which states that under a precise intractability assumption a specific security model (goals and means of the adversary) is satisfied. The reduction constitutes the proof of the theorem. Weaknesses can appear at several steps: the intractability assumption can be too strong, or even wrong; the security model might not correspond to the expected security level; the reduction may not be tight; and the proof can be erroneous. Because of more and more complex security models and proofs, most of them are never (double)-checked.

A famous example is the OAEP construction [6] that has been proven to achieve chosen-ciphertext security. But because of ambiguous security models in the early 90s, there was no real difference between the so-called IND-CCA1 and IND-CCA2 security levels. As a consequence, the proof was believed to achieve the IND-CCA2 level, until Shoup [7] exhibited a counter-example. Fortunately, a complete proof for IND-CCA2 has quickly been provided [8]. A machine-checked proof has later been provided [9].

As suggested by Halevi [10], computers could help in verifying proofs. This paper follows this path, with computationally-sound computer-aided proof and verification of cryptographic protocols.

Related Work: Various methods have been proposed for reaching Halevi's goal. Following the seminal paper by Abadi and Rogaway [11], many results show the soundness of the Dolev-Yao model with respect to the computational model, which makes it possible to use Dolev-Yao provers in order to prove protocols in the computational model (see, e.g., [12], [13], [14], [15], [16] and the survey [17]). However, these results have limitations, in particular in terms of allowed cryptographic primitives (they must satisfy strong security properties so that they correspond to Dolev-Yao style primitives), and they require some restrictions on protocols (such as the absence of key cycles). A tool [18] was developed based on [12] to obtain computational proofs using the formal verifier AVISPA, for protocols that rely on public-key encryption and signatures.

Several frameworks exist for formalizing proofs of protocols in the computational model. Backes, Pfitzmann ,and Waidner [19], [20] designed an abstract cryptographic library

and showed its soundness with respect to computational primitives, under arbitrary active attacks. This framework has been used for a computationally-sound machine-checked proof of the Needham-Schroeder-Lowe protocol [21], [22]. Canetti [23] introduced the notion of universal composability. With Herzog [24], they show how a Dolev-Yao-style symbolic analysis can be used to prove security properties of protocols within the framework of universal composability, for a restricted class of protocols using public-key encryption as only cryptographic primitive. Then, they use the automatic Dolev-Yao verification tool ProVerif [25] for verifying protocols in this framework. Process calculi have been designed for representing cryptographic games, such as the probabilistic polynomial-time calculus of [26] and the cryptographic lambda-calculus of [27]. Logics have also been designed for proving security protocols in the computational model, such as the computational variant of PCL (Protocol Composition Logic) [28], [29] and CIL (Computational Indistinguishability Logic) [30]. Canetti *et al.* [31] use the framework of time-bounded task-PIOAs (Probabilistic Input/Output Automata) to prove security protocols in the computational model. This framework makes it possible to combine probabilistic and non-deterministic behaviors. These frameworks can be used to prove security properties of protocols in the computational sense, but except for [24] which relies on a Dolev-Yao prover, they have not been automated up to now, as far as we know.

Several techniques have been used for directly mechanizing proofs in the computational model. Type systems [32], [33], [34], [35] provide computational security guarantees. For instance, [32] handles shared-key and public-key encryption, with an unbounded number of sessions, by relying on the Backes-Pfitzmann-Waidner library. A type inference algorithm is given in [36]. In another line of research, a specialized Hoare logic was designed for proving asymmetric encryption schemes in the random oracle model [37], [38].

The tool CertiCrypt [39], [40], [41], [42], [9] enables the machine-checked construction and verification of cryptographic proofs by sequences of games [43], [44]. It relies on the general-purpose proof assistant Coq, which is widely believed to be correct. EasyCrypt [45] generates CertiCrypt proofs from proof sketches that formally represent the sequence of games and hints, which makes the tool easier to use. Nowak *et al.* [46], [47], [48] follow a similar idea by providing Coq proofs for several cryptographic primitives.

Independently, we have built the tool CryptoVerif [49] to help cryptographers, not only for the verification, but also by generating the proofs by sequences of games [43], [44], automatically or with little user interaction. The games are formalized in a probabilistic polynomial-time process calculus. CryptoVerif provides a generic method for specifying security properties of many cryptographic primitives. It proves secrecy and authentication properties. It also provides a bound on the probability of success of an attack. It has already been used to prove several cryptographic protocols, and also primitives [50]. This tool extends considerably early work by Laud [51], [52] which was limited either to passive adversaries or to a single session of the protocol. More recently, Tšahhirov and Laud [53], [54] developed a tool similar to CryptoVerif but that represents games by dependency graphs. It handles public-key and shared-key encryption and proves secrecy properties; it does not provide bounds on the probability of success of an attack.

Contributions: In this paper, we use the tool CryptoVerif in order to prove the password-based key exchange protocol One-Encryption Key-Exchange (OEKE) [55], a variant of Encrypted Key Exchange (EKE) [56]. This is a non-trivial case study, since EKE was not proved correct before 2003, 10 years after its publication. This mechanized proof provides additional confidence that the protocol OEKE is secure. More precisely, we have shown that OEKE guarantees the secrecy of the session key and the authentication of the client to the server. The proof combines manually-guided and automatic steps, as detailed in Section IV. With the manual proof indications included in the CryptoVerif input file, the runtime of CryptoVerif version 1.14 for this proof was 3 s on an Intel Core i5 2.67 GHz (4 cores).

This case study was also an opportunity for implementing several extensions of CryptoVerif, useful for proving many other protocols. Here are these extensions:

- CryptoVerif's specification mechanism for assumptions on primitives did not support the computational Diffie-Hellman (CDH) assumption, needed for proving OEKE and many important protocols. We have extended it to support CDH (Section III-D). This extension also allowed us to prove a signed Diffie-Hellman protocol, in a fully automatic way.
- We have extended CryptoVerif to be able to apply Shoup's lemma [43], by introducing events and later bounding their probability. We improve over the standard computation of probabilities, for applications of Shoup's lemma, by avoiding to count several times probabilities that in fact correspond to the same runs. This allows us to obtain better probability bounds than [55] and to show that the adversary can test at most one password per session of the client or the server, which is the optimal result. This improvement applies both to CryptoVerif proofs and to manual proofs, and it is not specific to the OEKE protocol (Section IV-A).
- Additional game transformations were also needed for manually introducing case distinctions or for merging cases. We have implemented these transformations (Sections IV-A and IV-C).
- Password-based protocols require a careful computation of the probability of an attack, since one aims to compute how many passwords the adversary can test by interacting with the protocol. We have improved CryptoVerif in this respect (Section IV-D).

Outline: We recall the protocol OEKE in the next section. Section III presents the CryptoVerif model of the protocol, and Section IV presents its proof. We conclude in Section V. The appendices give background on CryptoVerif and additional details. The tool CryptoVerif and the input and output files can be found at http://www.cryptoverif.ens.fr/OEKE/.

Notations: $|S|$ denotes the cardinal of the set S. $\#O$ denotes the number of calls to oracle O.

II. The OEKE Protocol

Password-authenticated key exchange protocols allow two parties that share a low-entropy common secret (a password) to agree on a common high-entropy secret key thereafter used with symmetric primitives, such as symmetric encryption for privacy and message authentication codes for authentication. The goal of such a protocol is to guarantee the secrecy of the resulting common key between the two participating players. Furthermore, the protocol should succeed if and only if the two players actually share the same password, which guarantees the identity of the partner to both of them. Because of the low-entropy, an active adversary will succeed in impersonating a party to the other one with non-negligible probability by successive password guesses. Such an on-line dictionary attack is unavoidable. However, one should guarantee that this is the best attack: one active attack allows the adversary to test and thus eliminate at most one password, and not more. Namely, passive attacks should not (computationally) leak any information about the password. One definitely wants to prevent off-line dictionary attacks, where after a few active attacks and possibly many passive ones the collected information is enough to eliminate many passwords, and thus accelerate impersonation from the on-line dictionary attack.

The first password-authenticated key exchange protocol has been proposed by Bellovin and Merritt [56], the Encrypted Key Exchange (EKE). This is basically a Diffie-Hellman key exchange where the two flows are encrypted with a symmetric encryption scheme, using the password as secret key. Several variants have thereafter been proposed, such as AuthA [57]. The One-Encryption Key Exchange protocol (OEKE) studied in [55] is the particular variant where the second flow only is encrypted under the password, and the first player proves his knowledge of the password with an additional key confirmation flow. Figure 1 provides a description of this OEKE protocol, which guarantees client authentication and key secrecy, under the assumptions that $\mathcal{H}_0$ and $\mathcal{H}_1$ are random oracles, that $\mathcal{E}$ and $\mathcal{D}$ are respectively the encryption and decryption of an ideal cipher, and that $\mathbb{G}$ is a finite group of prime order q, with generator g, in which the computational Diffie-Hellman problem is hard (see the definition in Section III-D), as proven in [55]. If the password pw is chosen among a finite dictionary $passwd$ of size N equipped with the uniform distribution, their proof shows that the probability for any adversary,

Client U		Server S
pw		pw
$\mathsf{accept} \leftarrow \mathsf{false}$		$\mathsf{accept} \leftarrow \mathsf{false}$
$\mathsf{terminate} \leftarrow \mathsf{false}$		$\mathsf{terminate} \leftarrow \mathsf{false}$
$x \xleftarrow{R} [1, q-1]$		$y \xleftarrow{R} [1, q-1]$
$X \leftarrow g^x$	$\xrightarrow{U, X}$	$Y \leftarrow g^y$
$Y \leftarrow \mathcal{D}_{pw}(Y^\star)$	$\xleftarrow{S, Y^\star}$	$Y^\star \leftarrow \mathcal{E}_{pw}(Y)$
$K_U \leftarrow Y^x$		$K_S \leftarrow X^y$
$M_U \leftarrow U\|S\|X\|Y\|K_U$		
$\mathit{Auth} \leftarrow \mathcal{H}_1(M_U)$		
$sk_U \leftarrow \mathcal{H}_0(M_U)$		
$\mathsf{accept} \leftarrow \mathsf{true}$	$\xrightarrow{\mathit{Auth}}$	$M_S \leftarrow U\|S\|X\|Y\|K_S$
		$\mathit{Auth} \stackrel{?}{=} \mathcal{H}_1(M_S)$
		if true, $\mathsf{accept} \leftarrow \mathsf{true}$
		$sk_S \leftarrow \mathcal{H}_0(M_S)$
$\mathsf{terminate} \leftarrow \mathsf{true}$		$\mathsf{terminate} \leftarrow \mathsf{true}$

Figure 1. An execution of OEKE, run by client U and server S. The session key is $sk = \mathcal{H}_0(U\|S\|X\|Y\|Y^x) = \mathcal{H}_0(U\|S\|X\|Y\|X^y)$.

within time t, and with less than N_U sessions with a client, N_S sessions with a server (active attacks) and N_P passive eavesdroppings (passive attacks), and, asking q_h hash-queries and q_e encryption/decryption queries, to make a server instance accept with no terminating client partner is bounded by

$$\frac{N_U + 2N_S}{N} + 3q_h \times \mathsf{Succ}^{\mathsf{cdh}}_{\mathbb{G}}(t') + p_{coll}$$
$$\text{with } p_{coll} = \frac{(2q_e + 2N_U + 3N_S + 3N_P)^2}{2(q-1)} + \frac{q_h^2 + 4N_s}{2^{\ell_1+1}}$$

where ℓ_1 is the length of the output of $\mathcal{H}_1$ and $t' \leq t + (N_U + N_s + N_P + q_e + 1) \cdot \tau_{\mathsf{exp}}$, with τ_{exp} denoting the computation time for an exponentiation in $\mathbb{G}$.[1] Furthermore, $\mathsf{Succ}^{\mathsf{cdh}}_{\mathbb{G}}(t)$ denotes the maximal success probability an adversary can gain within time t against the computational Diffie-Hellman problem in $\mathbb{G}$. Similarly, no adversary can distinguish the session key from a random key with advantage greater than

$$\frac{2N_U + 4N_S}{N} + 8q_h \times \mathsf{Succ}^{\mathsf{cdh}}_{\mathbb{G}}(t') + 2p_{coll}.$$

The proofs basically show that the unique way for the adversary to gain something (against both client authentication and secrecy of the session key) is to correctly guess the password, by either sending a $Y^\star$ that is really an encryption under the correct password, or using the correct password to

[1] In [55], they use as parameter the number q_s of interactions with the parties, instead of the numbers of sessions N_U and N_S. It is straightforward to recompute the probabilities to use N_S and N_U instead, and this yields a more precise evaluation.

decrypt $Y^\star$ and compute the authenticator *Auth*. One could hope to prove that the former event, denoted Encrypt, is bounded by NU/N and the latter event, denoted Auth, is bounded by NS/N. But, because of the way probabilities are computed when one uses Shoup's lemma [43], some factor appears to the $(NU + NS)/N$ main term.

III. Modeling OEKE in CryptoVerif

In this section, we present the model of the protocol given as input to CryptoVerif. We first recall some basic ideas behind CryptoVerif, and then present the model itself: the security assumptions on the primitives, the model of the protocol, and the security properties that we want to prove. The complete CryptoVerif model, and the reusable library that provides the definitions of cryptographic primitives, can be found at http://www.cryptoverif.ens.fr/OEKE/.

A. Review of CryptoVerif

CryptoVerif builds proofs by sequences of games [43], [44]. It starts from the initial game given as input, which represents the protocol to prove in interaction with an adversary. Then, it transforms this game step by step using a set of predefined game transformations, such that each game is indistinguishable from the previous one.

More formally, a game G interacts with an adversary represented by a *context* C, and we denote by $C[G]$ the combination of C and G. During execution, $C[G]$ may execute events, collected in a sequence $\mathcal{E}$, and finally returns a result a, either a bitstring or the special value abort when the game has been aborted. These events and result can be used to distinguish games, so we introduce an additional algorithm, a *distinguisher* D that takes as input the sequence of events $\mathcal{E}$ and the result a, and returns true or false. An example of distinguisher is D_e defined by $D_e(\mathcal{E}, a) = \mathsf{true}$ if and only if $e \in \mathcal{E}$: this distinguisher detects the execution of event e. We will denote the distinguisher D_e simply by e. More generally, distinguishers can detect various properties of the sequence of events $\mathcal{E}$ executed by the game and of its result a. We denote by $D \vee D'$, $D \wedge D'$, and $\neg D$ the distinguishers such that $(D \vee D')(\mathcal{E}, a) = D(\mathcal{E}, a) \vee D'(\mathcal{E}, a)$, $(D \wedge D')(\mathcal{E}, a) = D(\mathcal{E}, a) \wedge D'(\mathcal{E}, a)$, and $(\neg D)(\mathcal{E}, a) = \neg D(\mathcal{E}, a)$, where $\vee$ is the logical disjunction, $\wedge$ the logical conjunction, and $\neg$ the logical negation. We denote by $\Pr[C[G] : D]$ the probability that $C[G]$ executes a sequence of events $\mathcal{E}$ and returns a result a, such that $D(\mathcal{E}, a) = \mathsf{true}$.

A context C is acceptable for G with public variables V when it can read directly the variables of G that are in V, and it makes no other access to variables of G. (This is more formally defined in Appendix A.) We define indistinguishability as an equivalence $G \approx_p^V G'$:

Definition 1 (Indistinguishability) We write $G \approx_p^V G'$ when, for all contexts C acceptable for G and G' with public variables V and all distinguishers D that run in time at most t_D, $|\Pr[C[G] : D] - \Pr[C[G'] : D]| \leq p(C, t_D)$.

This definition formalizes that the probability that algorithms C and D distinguish the games G and G' is at most $p(C, t_D)$. The probability p typically depends on the runtime of C and D, but may also depend on other parameters, such as the number of queries to each oracle made by C. That is why p takes as arguments the whole algorithm C and the runtime of D. When V is empty, we write $G \approx_p G'$. Therefore, we obtain a sequence of indistinguishable games $G_0 \approx_{p_1}^V G_1 \approx_{p_2}^V G_2 \ldots G_{n-1} \approx_{p_n}^V G_n$, which implies $G_0 \approx_{p_1+\ldots+p_n}^V G_n$. In the last game G_n, the desired security property is proved by direct inspection of the game, without using any computational assumption. For example, to bound the probability that an event e is executed, event e does not occur at all in the last game, so $\Pr[C[G_n] : e] = 0$, hence the probability of executing e in the initial game is $\Pr[C[G_0] : e] \leq (p_1 + \cdots + p_n)(C, e)$.

The game transformations used by CryptoVerif can be split into two categories:

- syntactic transformations, which are used by CryptoVerif to simplify games and to prepare cryptographic transformations. These transformations do not rely on any security assumption on primitives.
- cryptographic transformations, which rely on a security assumption on a primitive. These security assumptions are themselves formalized as indistinguishability properties $L \approx_p R$, which are given as input to CryptoVerif and need to be proved manually. They are proved once for each primitive and can then be reused in many protocols. We present such equivalences for the primitives used in OEKE below.
  CryptoVerif uses these equivalences to perform proofs by reduction automatically. It detects that a game G can be written as a context C that calls the oracles of L, that is, $G \approx_0^V C[L]$ by purely syntactic transformations, and builds a game G' such that $C[R] \approx_0^V G'$ by purely syntactic transformations. C is the simulator usually defined for reductions. From $L \approx_p R$, we can infer that $C[L] \approx_{p'}^V C[R]$ where V is a subset of the variables of C and $p'(C', t_D) = p(C'[C[]], t_D)$. Indeed, if C' is the adversary against $C[L] \approx_{p'}^V C[R]$, the adversary against $L \approx_p R$ is $C'[C[]]$. Therefore, $G \approx_{p'}^V G'$ and CryptoVerif can transform G into G'.

B. The Random Oracle Model

The random oracle model was introduced in [58] to model hash functions. It was encoded in CryptoVerif in [50]. We improve this model by using the equivalence $L_1 \approx_{\#\mathsf{Oeq}/|hashoutput|} R_1$ where L_1 and R_1 are defined in Figure 2. This model is not specific to OEKE. The hash function hash takes as input a key of type key and the bitstring to hash of type $hashinput$ and returns a result of

$L_1 =$ **foreach** $ih \leq nh$ **do** $k \stackrel{R}{\leftarrow} key$;
(**foreach** $i \leq n$ **do**
$\mathsf{OH}(x : hashinput) :=$ **return**$(\mathsf{hash}(k, x))$ |
foreach $ieq \leq neq$ **do**
$\mathsf{Oeq}(x' : hashinput, r' : hashoutput) :=$
return$(r' = \mathsf{hash}(k, x'))$)

$R_1 =$ **foreach** $ih \leq nh$ **do**
(**foreach** $i \leq n$ **do** $\mathsf{OH}(x : hashinput) :=$
find[**unique**] $u \leq n$ **suchthat**
defined$(x[u], r[u]) \wedge x = x[u]$
then return$(r[u])$
else $r \stackrel{R}{\leftarrow} hashoutput$; **return**$(r)$ |
foreach $ieq \leq neq$ **do**
$\mathsf{Oeq}(x' : hashinput, r' : hashoutput) :=$
find[**unique**] $u \leq n$ **suchthat**
defined$(x[u], r[u]) \wedge x' = x[u]$
then return$(r' = r[u])$
else return(false))

Figure 2. Random oracle model

type $hashoutput$. The key models the choice of the hash function. The key must be chosen once and for all at the beginning of the game for each hash function, and the game must include a hash oracle, which allows the adversary to compute hashes. For each hash function indexed by $ih \leq nh$, the games L_1 and R_1 define two oracles, OH and Oeq:

- In L_1, $\mathsf{OH}(x)$ returns the image of x by $\mathsf{hash}(k, \cdot)$. This oracle can be called at most n times for each hash function, and its calls are indexed by $i \in [1, n]$, as defined by **foreach** $i \leq n$ **do**. We can replace this oracle with a random oracle, that is, an oracle that returns a fresh random number when it is called with a new argument, and the previously returned result when it is called with the same argument as in a previous call. Such a random oracle is implemented in R_1 as follows. Like all variables defined under **foreach** $i \leq n$, x is in fact an array indexed by i, so that $x[u]$ represents the value of x in the u-th call to OH. The **find** construct looks for an index u such that $x[u]$ and $r[u]$ are defined, and $x = x[u]$, that is, the current argument of OH is the same as the argument in the u-th call, and if we find one, then we return the result of the u-th call, $r[u]$. Otherwise, we return a fresh random number r.
- The oracle Oeq aims to optimize the treatment of comparisons with the result of the hash function, an operation that appears frequently. In L_1, the oracle $\mathsf{Oeq}(x', r')$ compares r' with $\mathsf{hash}(k, x')$. In R_1, this comparison is replaced with a lookup in previous calls to the hash function. If x' was already given as argument to $\mathsf{hash}(k, \cdot)$, in the u-th call ($x' = x[u]$), then $\mathsf{hash}(k, x')$ is $r[u]$, so we compare r' with $r[u]$. Otherwise, x' was never given as argument to $\mathsf{hash}(k, \cdot)$, so $\mathsf{hash}(k, x')$ is a fresh random number, and it is equal to r' with probability $1/|hashoutput|$. We eliminate this case in R_1, so the result of the comparison $r' = \mathsf{hash}(k, x')$ is replaced with false and the probability of distinguishing L_1 from R_1 is at most $\#\mathsf{Oeq}/|hashoutput|$, where $\#\mathsf{Oeq}$ denotes the total number of calls to Oeq.

We can notice that there exists at most one u that can satisfy the condition of **find** in OH in R_1. Indeed, suppose that $u_1 \neq u_2$ are such that $x[u_1], r[u_1], x[u_2], r[u_2]$ are defined and $x = x[u_1] = x[u_2]$. Suppose that the query OH with $i = u_1$ is called before OH with $i = u_2$. (The other case is symmetric.) Thus, when executing the query OH with $i = u_2$, $x[u_1]$ and $r[u_1]$ are defined and $x[u_2] = x[u_1]$, so the **find** succeeds with $u = u_1$, so $r[u_2]$ will not be defined (since r is defined only in the **else** branch of the **find**). Contradiction. Therefore, u is unique. Following a similar reasoning, u is also unique in Oeq in R_1. That is why the **find**s in R_1 are marked [**unique**]. Formally, the modifier [**unique**] means that, in case several choices satisfy the condition of **find**, an event NonUnique occurs and the game is aborted. As we have shown, the event NonUnique never occurs in R_1, so the modifier [**unique**] does not alter the equivalence $L_1 \approx_{\#\mathsf{Oeq}/|hashoutput|} R_1$. The modifier [**unique**] allows additional transformations of **find**, which are correct only when there never exist several choices that make the condition of the **find** succeed. These transformations are detailed in Appendix E-B.

The novelties with respect to [50] are the use of keyed hash functions, the oracle Oeq, and the modifier [**unique**]. We believe that using keyed hash functions leads to a better modeling of random oracles, for several reasons:

- In the random oracle model, the adversary cannot evaluate the hash function by himself, without calling the random oracle. With the key, this is natural, since the adversary does not have the key, whereas in the absence of key, this is counterintuitive: the adversary should be able to reproduce the algorithm of h.
- In the absence of key, the transformation of L_1 into R_1 above replaces a deterministic function h with a probabilistic one, since the results are chosen randomly in the right-hand side. The key removes this discrepancy: with the key, the hash oracle is also probabilistic in the left-hand side thanks to the choice of the key.
- The transformation of L_1 into R_1 above is correct only when it is applied to all occurrences of h simultaneously. In the absence of key, this has to be enforced by an additional constraint on the transformation. With the key, this is naturally enforced, since all occurrences of the key need to be encoded as calls to the oracles of L_1 for the transformation to be performed.
- Finally, keyed hash functions are used in the mod-

$L_2 =$ **foreach** $ick \leq nck$ **do** $ck \stackrel{R}{\leftarrow} cipherkey$;
(**foreach** $ie \leq ne$ **do** $\mathsf{Oenc}(me : blocksize, ke : key) :=$ **return**$(\mathsf{enc}(ck, me, ke))$ |
foreach $id \leq nd$ **do** $\mathsf{Odec}(md : blocksize, kd : key) :=$ **return**$(\mathsf{dec}(ck, md, kd))$)

$R_2 =$ **foreach** $ick \leq nck$ **do**
(**foreach** $ie \leq ne$ **do** $\mathsf{Oenc}(me : blocksize, ke : key) :=$
find[**unique**] $j \leq ne$ **suchthat defined**$(me[j], ke[j], re[j]) \wedge me = me[j] \wedge ke = ke[j]$ **then return**$(re[j])$
$\oplus$ $k \leq ne$ **suchthat defined**$(rd[k], md[k], kd[k]) \wedge me = rd[k] \wedge ke = kd[k]$ **then return**$(md[k])$
else $re \stackrel{R}{\leftarrow} blocksize$; **return**$(re)$ |
foreach $id \leq nd$ **do** $\mathsf{Odec}(md : blocksize, kd : key) :=$
find[**unique**] $j \leq ne$ **suchthat defined**$(me[j], ke[j], re[j]) \wedge md = re[j] \wedge kd = ke[j]$ **then return**$(me[j])$
$\oplus$ $k \leq nd$ **suchthat defined**$(rd[k], md[k], kd[k]) \wedge md = md[k] \wedge kd = kd[k]$ **then return**$(rd[k])$
else $rd \stackrel{R}{\leftarrow} blocksize$; **return**$(rd)$)

Figure 3. Ideal cipher model

eling of other assumptions on hash functions, such as collision resistance. By always using keyed hash functions, we can easily change the assumption on the hash function without changing its interface.

Designing CryptoVerif specifications of primitives requires some expertise. That is why the specifications for most common cryptographic primitives are grouped in a reusable library. Therefore, CryptoVerif users generally do not have to design such specifications.

C. The Ideal Cipher Model

The ideal cipher model [59] models block ciphers by saying that encryption and decryption are two random permutations, inverse of each other. This can be encoded in CryptoVerif similarly to the random oracle model: we replace encryption and decryption with lookups in previous encryption/decryption queries; if a previous query matches, we return the previous result; otherwise, we return a fresh random number. This is modeled by the equivalence $L_2 \approx_{p_2} R_2$ where L_2 and R_2 are defined in Figure 3 and $p_2 = (\#\mathsf{Oenc} + \#\mathsf{Odec})(\#\mathsf{Oenc} + \#\mathsf{Odec} - 1)/|blocksize|$. The encryption and decryption functions map bitstrings of type $blocksize$ to bitstrings of type $blocksize$; they take two keys as additional arguments: the standard encryption/decryption key of type key, but also a key of type $cipherkey$ that models the choice of the scheme itself (like the key of the hash function in Section III-B). The games L_2 and R_2 define two oracles Oenc and Odec, respectively the encryption and decryption oracles. In L_2, they call the encryption and decryption functions. In R_2, they are replaced with lookups in previous encryption/decryption queries. For instance, for oracle Oenc, we look for a previous encryption query of the same cleartext ($me = me[j]$) under the same key ($ke = ke[j]$) and, if we find one, we return the same ciphertext $re[j]$. We also look for a previous decryption query that has returned as cleartext the cleartext to encrypt ($me = rd[k]$) using the same key ($ke = kd[k]$) and, if we find one, we return the corresponding ciphertext $md[k]$. Otherwise, we return a fresh random ciphertext re. This definition does not yield random permutations, because the random choices of re and rd may collide with each other and with previous values of me and md. Let us consider a game R'_2 obtained from R_2 by excluding such collisions. By adapting the reasoning used for the random oracle model in Section III-B, we can show that, in R'_2, there never exist several choices of j/k that satisfy the conditions of the **find**s in Oenc and Odec, so these **find**s can be marked [**unique**] without modifying their behavior. The game L_2 is perfectly indistinguishable from R'_2, and R'_2 can be distinguished from R_2 with probability at most p_2 (the probability of the collisions excluded in R'_2), so the adversary can distinguish L_2 from R_2 with probability at most p_2.

D. The Computational Diffie-Hellman Assumption

A classical intractability assumption in asymmetric cryptography is the hardness of the Diffie-Hellman problem: let us be given a group $\mathbb{G}$ of prime order q, with a generator g, and two random elements $A = g^a$ and $B = g^b$ with $a, b \in [1, q-1]$, compute $\mathsf{CDH}_g(A, B) = g^{ab}$. The Computational Diffie-Hellman (CDH) assumption claims that for any polynomial-time adversary $\mathcal{A}$, $\mathsf{Succ}^{\mathsf{cdh}}_{\mathbb{G}}(\mathcal{A}) = \Pr[\mathcal{A}(\mathbb{G}, g, A, B) = \mathsf{CDH}_g(A, B)]$ is negligible. More generally, we note $\mathsf{Succ}^{\mathsf{cdh}}_{\mathbb{G}}(t)$ the maximal success probability for any adversary $\mathcal{A}$ within time t.

This assumption can be written in CryptoVerif as follows:

foreach $i \leq n$ **do** $a \stackrel{R}{\leftarrow} Z; b \stackrel{R}{\leftarrow} Z;$
$(\mathsf{OA}() := \mathsf{exp}(\mathsf{g}, a) \mid \mathsf{OB}() := \mathsf{exp}(\mathsf{g}, b) \mid$
foreach $i' \leq n'$ **do** $\mathsf{ODDH}(z : G) :=$
$z = \mathsf{exp}(\mathsf{g}, \mathsf{mult}(a, b)))$

$\approx_{\#\mathsf{ODDH} \times \mathsf{Succ}^{\mathsf{cdh}}_{\mathbb{G}}(t + (n + \#\mathsf{ODDH})\tau_{\mathsf{exp}})}$

foreach $i \leq n$ **do** $a \stackrel{R}{\leftarrow} Z; b \stackrel{R}{\leftarrow} Z;$

$L_3 =$ **foreach** $ia \leq na$ **do** $a \stackrel{R}{\leftarrow} Z; ($
$\quad \mathsf{OA}() := \mathbf{return}(\mathsf{exp}(\mathsf{g}, a)) \mid$
$\quad \mathsf{Oa}() := \mathbf{return}(a) \mid$
$\quad \mathbf{foreach}\ iaDDH \leq naDDH\ \mathbf{do}$
$\quad\quad \mathsf{ODDHa}(m : G, j \leq nb) :=$
$\quad\quad\quad \mathbf{return}(m = \mathsf{exp}(\mathsf{g}, \mathsf{mult}(b[j], a)))) \mid$
$\mathbf{foreach}\ ib \leq nb\ \mathbf{do}\ b \stackrel{R}{\leftarrow} Z; ($
$\quad \mathsf{OB}() := \mathbf{return}(\mathsf{exp}(\mathsf{g}, b)) \mid$
$\quad \mathsf{Ob}() := \mathbf{return}(b) \mid$
$\quad \mathbf{foreach}\ ibDDH \leq nbDDH\ \mathbf{do}$
$\quad\quad \mathsf{ODDHb}(m : G, j \leq na) :=$
$\quad\quad\quad \mathbf{return}(m = \mathsf{exp}(\mathsf{g}, \mathsf{mult}(a[j], b))))$

$R_3 =$ **foreach** $ia \leq na$ **do** $a \stackrel{R}{\leftarrow} Z; ($
$\quad \mathsf{OA}() := \mathbf{return}(\mathsf{exp}(\mathsf{g}, a)) \mid$
$\quad \mathsf{Oa}() := \mathbf{let}\ ka : bitstring = \mathsf{mark}\ \mathbf{in}\ \mathbf{return}(a) \mid$
$\quad \mathbf{foreach}\ iaDDH \leq naDDH\ \mathbf{do}$
$\quad\quad \mathsf{ODDHa}(m : G, j \leq nb) :=$
$\quad\quad\quad \mathbf{find}\ u \leq nb\ \mathbf{suchthat}\ \mathbf{defined}(kb[u], b[u])$
$\quad\quad\quad\quad \wedge b[j] = b[u]\ \mathbf{then}$
$\quad\quad\quad\quad \mathbf{return}(m = \mathsf{exp}(\mathsf{g}, \mathsf{mult}(b[j], a)))$
$\quad\quad\quad \mathbf{else\ if\ defined}(ka)\ \mathbf{then}$
$\quad\quad\quad\quad \mathbf{return}(m = \mathsf{exp}(\mathsf{g}, \mathsf{mult}(b[j], a)))$
$\quad\quad\quad \mathbf{else\ return}(\mathsf{false})) \mid$
$\mathbf{foreach}\ ib \leq nb\ \mathbf{do}\ b \stackrel{R}{\leftarrow} Z; ($
$\quad \mathsf{OB}() := \mathbf{return}(\mathsf{exp}(\mathsf{g}, b)) \mid$
$\quad \mathsf{Ob}() := \mathbf{let}\ kb : bitstring = \mathsf{mark}\ \mathbf{in}\ \mathbf{return}(b) \mid$
$\quad \mathbf{foreach}\ ibDDH \leq nbDDH\ \mathbf{do}$
$\quad\quad \mathsf{ODDHb}(m : G, j \leq na) :=$
$\quad\quad\quad$ (symmetric of ODDHa)

Figure 4. Computational Diffie-Hellman assumption

$(\mathsf{OA}() := \mathsf{exp}(\mathsf{g}, a) \mid \mathsf{OB}() := \mathsf{exp}(\mathsf{g}, b) \mid$
$\mathbf{foreach}\ i' \leq n'\ \mathbf{do}\ \mathsf{ODDH}(z : G) := \mathsf{false})$

The type Z represents $[1, q-1]$, that is, the group $\mathbb{Z}_q^*$; mult is the product in that group; G represents the group $\mathbb{G}$ without its neutral element; and exp is the exponentiation $G \times Z \to G$. These two games define three oracles: OA and OB return the exponentials g^a and g^b respectively, and the oracle ODDH checks whether its argument z is equal to g^{ab} in the left-hand side while it always returns false in the right-hand side. The adversary can distinguish these two games if and only if it can provide a z such that $z = g^{ab}$, that is, it breaks the CDH assumption. However, in CryptoVerif, this model requires that a and b be chosen one after the other under the same **foreach**: while this is true in some cryptographic schemes such as ElGamal, this is not true for most protocols: as in OEKE, a and b are chosen by different protocol participants that can each execute several sessions.

Therefore, we need a more general model, which is given by the indistinguishability between the two games presented in Figure 4. In these two games, one generates na exponents a, nb exponents b and the adversary (any context) has access to various oracles: OA and OB that return the group elements associated to a, resp. b; Oa and Ob that return the exponents a and b themselves; and Diffie-Hellman decisions oracles ODDHa and ODDHb that check whether the adversary correctly solved a Diffie-Hellman problem with the above generated elements. Basically, the difference between the two games is in the answers of the decision oracles: in the first game they answer correctly, while in the second game, they answer false if the adversary did not ask for any of the two exponents. Unless the adversary can break the Diffie-Hellman problem, and then ask correct Diffie-Hellman decision queries, the two executions are perfectly indistinguishable. In more detail, in R_3, the variable ka is defined if and only if the oracle Oa has been called and thus the exponent a has been asked by the adversary. All variables and oracles defined under **foreach** $ia \leq na$ are implicitly indexed by ia, so that $ka[ia]$ is defined if and only if $a[ia]$ has been asked by the adversary. The variable kb plays the same role for b. The oracle ODDHa computes the equality test $m = g^{a[ia]b[j]}$ when $b[j]$ has been asked by the adversary, i.e., $kb[j]$ is defined, or $a[ia]$ has been asked by the adversary, i.e. $ka[ia]$ (abbreviated ka) is defined. Otherwise, it returns false. The condition "$kb[j]$ is defined" is encoded as "$kb[u]$ is defined for some u such that $b[u] = b[j]$" (**defined**$(kb[u], b[u]) \wedge b[j] = b[u]$), because CryptoVerif allows to reference a variable $x[\tilde{u}]$ in **defined** conditions in the right-hand side of an equivalence only when its indices $\tilde{u}$ are a prefix of the indices looked up by **find**, so a reference to $kb[j]$ would not be allowed. We can refer to $b[j]$ without including it in a **defined** condition because it also occurs in the left-hand side of the equivalence, so CryptoVerif knows that it must be defined. That is why the condition **defined**$(kb[u], b[u]) \wedge b[j] = b[u]$ is accepted by CryptoVerif.

In Appendix B-A, we formally prove that $L_3 \approx_{p_3} R_3$, that is, no adversary can distinguish the two games L_3 and R_3, within time t, with advantage greater than

$$p_3 = (\#\mathsf{ODDHa} + \#\mathsf{ODDHb}) \times \max(1, 7.4\#\mathsf{Oa}) \times \max(1, 7.4\#\mathsf{Ob}) \times \mathsf{Succ}^{\mathsf{cdh}}_{\mathbb{G}}(t + (na + nb + \#\mathsf{ODDHa} + \#\mathsf{ODDHb})\tau_{\mathsf{exp}}).$$

The proof technique consists in guessing the two elements a and b that will be involved in the critical decisional Diffie-Hellman query (but with Coron's improvement [60]), and then to guess the critical query, hence the factor $\#\mathsf{ODDHa} + \#\mathsf{ODDHb}$.

For this equivalence to be supported by CryptoVerif, we had to implement two extensions:

- Oracles ODDHa and ODDHb take as argument an array index j, which was not supported.

- In typical usages of the CDH assumption in protocols, g^{ab} is often an argument of a hash function in the random oracle model. The transformation that comes from the random oracle model, presented in Section III-B, transforms $\mathsf{hash}(\ldots g^{ab} \ldots)$ into lookups that compare g^{ab} with previous arguments of hash. These comparisons $m = g^{ab}$, which occur in conditions of **find**, are themselves transformed into **find** using the CDH assumption. We therefore end up with a **find** inside the condition of a **find**, which was not supported.

In addition to the modeling of the CDH assumption itself, our model of Diffie-Hellman key agreements includes further properties, such as commutativity and injectivity of several functions. They are formally defined in Appendix B-B. We stress that our above modeling is not specific to the OEKE protocol. We have also used it to prove a signed Diffie-Hellman key exchange, and we believe that it can be used for proving many other protocols.

E. The Protocol Itself

If we consider a general configuration with several clients and servers, each client-server pair shares a different password, and there is no other secret shared initially. Therefore, different client-server pairs have no common secret, so we can encode a single client U and a single server S that wish to talk to each other; the other clients and servers, which may be corrupted, and the interactions of U and S with other clients and servers are included in the adversary. This model supports static corruptions; dynamic corruptions and forward secrecy properties are left for future work.

The protocol model first chooses random keys *hk0* and *hk1* to model the choice of the hash functions h0 (i.e. $\mathcal{H}_0$) and h1 (i.e. $\mathcal{H}_1$) respectively and a key ck to model the choice of the ideal cipher scheme. It also randomly chooses a password pw in the type $passwd$. Then, it makes available hash oracles for h0 and h1, encryption and decryption oracles, as well as oracles that represent the client and the server. As an example, we detail the code for the client:

foreach $iU \leq NU$ **do**
$\mathsf{OC1}() := x \stackrel{R}{\leftarrow} Z; X \leftarrow \mathsf{exp}(\mathsf{g}, x); \mathbf{return}(\mathsf{U}, X);$
$\mathsf{OC2}(= \mathsf{S}, Ystar\_u : G) := Y\_u \leftarrow \mathsf{dec}(ck, Ystar\_u, pw);$
$K\_u \leftarrow \mathsf{exp}(Y\_u, x);$
$auth\_u \leftarrow \mathsf{h1}(hk1, \mathsf{concat}(\mathsf{U}, \mathsf{S}, X, Y\_u, K\_u));$
$sk\_u : hash0 \leftarrow \mathsf{h0}(hk0, \mathsf{concat}(\mathsf{U}, \mathsf{S}, X, Y\_u, K\_u));$
return$(auth\_u)$

This code models NU sessions of the client, indexed by iU. Each session defines two oracles OC1 and OC2. OC1 takes no argument and returns the first message of the protocol U, X computed as specified in Figure 1. OC2 takes as argument the second message of the protocol $\mathsf{S}, Ystar\_u$ received by the client and returns the third one $auth\_u$. It also computes the shared key sk_u. In this code, $\mathsf{concat}(\mathsf{U}, \mathsf{S}, X, Y\_u, K\_u)$ is the concatenation $\mathsf{U}\|\mathsf{S}\|X\|Y\_u\|K\_u$. These oracles are implicitly indexed by iU, so that they can be written $\mathsf{OC1}[iU]$, $\mathsf{OC2}[iU]$. (This index is omitted in CryptoVerif code for readability.) The adversary can call the oracles with any index it likes in the order it likes, except that, obviously, $\mathsf{OC2}[iU]$ can be called only if $\mathsf{OC1}[iU]$ has been called before with the same iU. This gives the adversary full control over the network.

We represent NS sessions of the server in a similar way. The NU sessions of the client and the NS sessions of the server model active attacks. Additionally, we represent NP sessions of the protocol in which the adversary just eavesdrops messages without altering them. In order to represent such sessions, we simply compute and output their transcript. They model passive attacks. Since we are considering dictionary attacks against a password-authenticated key exchange protocol, it is important to distinguish passive sessions/attacks from active ones against the honest players.

F. Security Properties

Our goal is to prove that OEKE is a secure key exchange that provides unilateral (explicit) authentication. (OEKE guarantees client authentication but not server authentication.) To do that, we follow the ideas of [61, Section 7.2]: instead of proving semantic security of the key and authentication, we prove secrecy of the key on the client side and a slightly stronger authentication property. This technique avoids the burden of considering partnering when proving secrecy of the key and still implies authenticated key exchange [61, Proposition 4]: intuitively, authentication guarantees that a key of the server is also a key of a client. Authentication is modeled by correspondence properties [62] of the form "if some event occurs, then some other event occurred". There are still two differences with respect to [61]:

- [61] considers mutual authentication, while we consider unilateral authentication, so we remove the correspondence that guaranteed authentication of the server.
- In [61], each protocol participant may interact with honest participants (U and S here) but also with dishonest participants, and in the latter situation, the exchanged key is published when the participant accepts. As mentioned in Section III-E, in OEKE, we need not code explicitly for U and S interacting with other clients and servers, so the output of the exchanged key disappears.

Taking into account these points, we add events to record that the participants accept or terminate:

- **event** $\mathsf{acceptU}(\mathsf{U}, X, \mathsf{S}, Ystar\_u, auth\_u, sk\_u)$ when the client accepts (line "accept $\leftarrow$ true" of the client in Figure 1, that is, before the last line in the code of Section III-E).
- **event** $\mathsf{termS}(\mathsf{U}, X\_s, \mathsf{S}, Ystar, auth\_s, sk\_s)$ when the server terminates (line "terminate $\leftarrow$ true" of the server in Figure 1).

and we prove that the resulting process preserves the secrecy of sk_u and satisfies the correspondences

$$\textbf{inj-event}(\mathsf{termS}(U, X, S, Ystar, a, k)) \Rightarrow \textbf{inj-event}(\mathsf{acceptU}(U, X, S, Ystar, a, k)) \quad (1)$$

$$\textbf{event}(\mathsf{termS}(U, X, S, Ystar, a, k)) \wedge \textbf{event}(\mathsf{acceptU}(U, X, S, Ystar, a, k')) \Rightarrow k = k' \quad (2)$$

with public variables $\{sk\_u\}$. A variant of [61, Proposition 4] allows us to conclude one-way authenticated key exchange. Next, we define secrecy and correspondences.

Intuitively, the secrecy of sk_u means that the keys sk_u of all sessions of the client are indistinguishable from independent random keys. Formally, secrecy is defined as follows:

Definition 2 (Secrecy) Assume that the variable x of type T is defined in G under a single **foreach** $i \leq n$. The game G *preserves the secrecy of* x up to probability p when, for all contexts C acceptable for $G \mid R_x$ without public variables that do not contain S and $\overline{\mathsf{S}}$, $\Pr[C[G \mid R_x] : \mathsf{S}] - \Pr[C[G \mid R_x] : \overline{\mathsf{S}}] \leq p(C)$ where

$R_x = O_0() := b \xleftarrow{R} bool$; **return**;
(**foreach** $i' \leq n'$ **do** $O(u : [1, n]) :=$
if defined$(x[u])$ **then**
if b **then return**$(x[u])$ **else**
find $u' \leq n'$ **suchthat defined**$(y[u'], u[u']) \wedge u[u'] = u$ **then return**$(y[u'])$ **else**
$y \xleftarrow{R} T$; **return**(y)
$\mid O'(b' : bool) :=$ **if** $b = b'$ **then event** S; **abort** **else event** $\overline{\mathsf{S}}$; **abort**)

O_0, O, O', b, b', u, u', y, S, and $\overline{\mathsf{S}}$ do not occur in G.

We define the secrecy of x with the Real-or-Random model of [63]: in R_x, we choose a random bit b, and provide the oracle O that the adversary can use to perform several test queries on $x[u]$: if $b = 1$, the test query returns $x[u]$; if $b = 0$, it returns a random value y (the same value if the same query $x[u]$ is asked twice). Finally, the adversary should guess the bit b: it calls oracle O' with its guess b' and, if the guess is correct, then event S is executed, and otherwise, event $\overline{\mathsf{S}}$ is executed. The probability of getting some information on the secret is the difference between the probability of S and the probability of $\overline{\mathsf{S}}$. (When the game always runs oracle O', we have $\Pr[C[G \mid R_x] : \overline{\mathsf{S}}] = 1 - \Pr[C[G \mid R_x] : \mathsf{S}]$, so the advantage of the adversary is $\Pr[C[G \mid R_x] : \mathsf{S}] - \Pr[C[G \mid R_x] : \overline{\mathsf{S}}] = 2\Pr[C[G \mid R_x] : \mathsf{S}] - 1$, which is a more standard formula.) As shown in [63], the Real-or-Random model is stronger than the Find-Then-Guess model used in [55], which allows a single test query and several reveal queries. (Reveal queries always return the real $x[u]$.)

The correspondence (1) means that each execution of event $\mathsf{termS}(U, X, S, Ystar, a, k)$ corresponds to a distinct execution of event $\mathsf{acceptU}(U, X, S, Ystar, a, k)$; in other words, each session of the server that accepts with transcript $U, X, S, Ystar, a$ and shared key k corresponds to a distinct session of the client that accepts with the same transcript and same key. It corresponds to the authentication of the client. The keyword **inj-event** is used in CryptoVerif to require injective correspondences, that is, $\mathsf{acceptU}$ has been executed at least as many times as termS, and not only once. The correspondence (2) means that when events $\mathsf{termS}(U, X, S, Ystar, a, k)$ and $\mathsf{acceptU}(U, X, S, Ystar, a, k')$ have been executed, $k = k'$, that is, if a client and a server have the same transcript, then they share the same key. These correspondences are proved "with public variables $\{sk\_u\}$", that is, they hold even when the adversary is allowed to access sk_u directly. Formally, we write $\mathcal{E} \vdash \psi \Rightarrow \varphi$ when the sequence of events $\mathcal{E}$ satisfies the correspondence $\psi \Rightarrow \varphi$. (This is formally defined in [61].) For instance, $\mathcal{E} \vdash$ **inj-event**$(\mathsf{termS}(U, X, S, Ystar, a, k)) \Rightarrow$ **inj-event**$(\mathsf{acceptU}(U, X, S, Ystar, a, k))$ if and only if, for each event $\mathsf{acceptU}(\ldots)$ in $\mathcal{E}$, there is a distinct event $\mathsf{termS}(\ldots)$ in $\mathcal{E}$ with the same arguments as the event $\mathsf{acceptU}(\ldots)$.

Definition 3 (Correspondence) The game G *satisfies the correspondence* $\psi \Rightarrow \varphi$ with public variables V up to probability p if and only if, for all contexts C acceptable for G with public variables V that do not contain events, $\Pr[C[Q] : D] \leq p(C)$, where $D(\mathcal{E}, a) = (\mathcal{E} \not\vdash \psi \Rightarrow \varphi)$.

IV. Proving OEKE in CryptoVerif

In the previous section, we have presented the formalization of the protocol given as input to CryptoVerif. In this section, we explain how CryptoVerif proceeds with the proof. Some parts of the proof are automatic, some are guided by the user. The commands for guiding CryptoVerif can be given interactively, which allows one to see the current game and understand what should be done next, or in a `proof {...}` declaration in the CryptoVerif input file, so that CryptoVerif can then run on its own. The input file presented at http://www.cryptoverif.ens.fr/OEKE/ includes such a declaration. We stress that, even with manual guidance, all game transformations are verified by CryptoVerif, so that one cannot perform an incorrect proof.

A. Applying Shoup's Lemma

The first step of the proof is to introduce the events Auth and $\mathsf{Encrypt}$, which correspond to cases in which the adversary succeeds in testing a password and were also used in the manual proof of [55].

By Shoup's lemma [43], if G' is obtained from G by inserting an event e and modifying the code executed after e, the probability of distinguishing G' from G is bounded by the probability of executing e: for all contexts C acceptable for G and G' (with any public variables) and all distinguishers D, $|\Pr[C[G] : D] - \Pr[C[G'] : D]| \leq \Pr[C[G'] : e]$.

Hence, $\Pr[C[G] : D] \leq \Pr[C[G'] : e] + \Pr[C[G'] : D]$. We improve over this computation of probabilities by considering e and D simultaneously instead of making the sum of the two probabilities: $\Pr[C[G] : D] \leq \Pr[C[G'] : D \vee e]$.

Lemma 1 *Let C be a context acceptable for G and G' with public variables V.*

1) *If G' differs from G only when G' executes event e, then* $\Pr[C[G] : D] \leq \Pr[C[G'] : D \vee e]$.
2) *If G differs from G' only when G executes event* $\mathsf{NonUnique}$ *and $D = (D_0 \wedge \neg \mathsf{NonUnique}) \vee e_1 \vee \ldots \vee e_n$ where we abort just after executing events $e_1, \ldots, e_n$, then* $\Pr[C[G] : D] \leq \Pr[C[G'] : D]$.
3) *If $G \approx_p^V G'$, then* $\Pr[C[G] : D] \leq p(C, t_D) + \Pr[C[G'] : D]$.
4) $\Pr[C[G] : D \vee D'] \leq \Pr[C[G] : D] + \Pr[C[G] : D']$.

This lemma, and Lemma 2 below, are proved in Appendix C. In order to bound the probability that a distinguisher D_0 returns true for some game G_0, we consider any context C acceptable for G_0 with public variables V and that does not contain events, and bound $\Pr[C[G_0] : D_0 \wedge \neg \mathsf{NonUnique}]$ which is equal to $\Pr[C[G_0] : D_0]$ because no **find**[**unique**] occurs in the initial game. For each game transformation, we assume that the introduced variables are fresh, so that C remains acceptable for all games of the sequence. We can then apply Lemma 1 for each game transformation. Points 1, 2, and 3 of this lemma allow us to handle several events simultaneously, as long as the proof uses the same sequence of games to bound their probabilities. Point 2 is useful for transformations that rely on the uniqueness of the values that satisfy the conditions of **find**, detailed in Appendix E-B: these transformations preserve the behavior of the game when G does not execute event $\mathsf{NonUnique}$. The distinguisher D is always of the desired form $(D_0 \wedge \neg \mathsf{NonUnique}) \vee e_1 \vee \ldots \vee e_n$ because we start from $D_0 \wedge \neg \mathsf{NonUnique}$ and add events introduced by Shoup's lemma using point 1; we abort immediately after these events. When the proof uses different sequences of games to bound the probabilities of events, we use point 4 of the lemma to bound each probability separately and compute the sum. The standard computation of probabilities corresponds to always applying point 4.

For example, suppose that we want to bound the probability of event e_0 in G_0, G_1 differs from G_0 only when G_1 executes event e, $G_1 \approx_p G_2$, and G_2 executes neither e_0 nor e. Suppose for simplicity that no **find**[**unique**] occurs, so that $\mathsf{NonUnique}$ never occurs. Lemma 1 yields $\Pr[C[G_0] : e_0] \leq \Pr[C[G_1] : e_0 \vee e] \leq p(C, t_{e_0 \vee e}) + \Pr[C[G_2] : e_0 \vee e] = p(C, t_{e_0 \vee e})$. The standard computation of probabilities yields $\Pr[C[G_0] : e_0] \leq \Pr[C[G_1] : e_0] + \Pr[C[G_1] : e] \leq p(C, t_{e_0}) + p(C, t_e)$. The runtime t_D of D is essentially the same for e_0, e, and $e_0 \vee e$, so $\Pr[C[G_0] : e_0] \leq p(C, t_D)$ by Lemma 1, while $\Pr[C[G_0] : e_0] \leq 2p(C, t_D)$ by the standard computation, so we have gained a factor 2.

For secrecy, the advantage $\Pr[C[G \mid R_x] : \mathsf{S}] - \Pr[C[G \mid R_x] : \overline{\mathsf{S}}]$ introduces a factor 2 in the probability: if $G \approx_p^{\{x\}} G'$, then $\Pr[C[G \mid R_x] : \mathsf{S}] - \Pr[C[G \mid R_x] : \overline{\mathsf{S}}] \leq \mathbf{2}p(C[[\,] \mid R_x], t_{\mathsf{S}}) + (\Pr[C[G' \mid R_x] : \mathsf{S}] - \Pr[C[G' \mid R_x] : \overline{\mathsf{S}}])$, since $t_{\mathsf{S}} = t_{\overline{\mathsf{S}}}$. The next lemma avoids this factor 2 for probabilities of events:

Lemma 2 *Let C be a context acceptable for G and G' with public variables V. Let the distinguishers D, D' be disjunctions of events $e_1 \vee \ldots \vee e_n$ such that we abort just after executing each e_i. Let* $\mathsf{Adv}_G^{\mathsf{Secrecy}}(C, D) = \Pr[C[G \mid R_x] : \mathsf{S} \vee D] - \Pr[C[G \mid R_x] : \overline{\mathsf{S}} \vee \mathsf{NonUnique}]$.

1) *If G' differs from G only when G' executes event e and we abort just after executing e, then* $\mathsf{Adv}_G^{\mathsf{Secrecy}}(C, D) \leq \mathsf{Adv}_{G'}^{\mathsf{Secrecy}}(C, D \vee e)$.
2) *If G differs from G' only when G executes* $\mathsf{NonUnique}$*, then* $\mathsf{Adv}_G^{\mathsf{Secrecy}}(C, D) \leq \mathsf{Adv}_{G'}^{\mathsf{Secrecy}}(C, D)$.
3) *If $G \approx_p^V G'$, then* $\mathsf{Adv}_G^{\mathsf{Secrecy}}(C, D) \leq 2p(C[[\,] \mid R_x], t) + \mathsf{Adv}_{G'}^{\mathsf{Secrecy}}(C, D)$ *where* $t = \max(t_{\mathsf{S} \vee D}, t_{\overline{\mathsf{S}} \vee \mathsf{NonUnique}})$.
4) $\mathsf{Adv}_G^{\mathsf{Secrecy}}(C, D \vee D') \leq \mathsf{Adv}_G^{\mathsf{Secrecy}}(C, D) + \Pr[C[G \mid R_x] : D']$.
5) *If CryptoVerif proves the secrecy of x in game G, then* $\Pr[C[G \mid R_x] : \mathsf{S}] = \Pr[C[G \mid R_x] : \overline{\mathsf{S}}]$*, so* $\mathsf{Adv}_G^{\mathsf{Secrecy}}(C, D) \leq \Pr[C[G \mid R_x] : D]$.

In order to prove secrecy of x in the initial game G_0, we bound $\Pr[C[G_0 \mid R_x] : \mathsf{S}] - \Pr[C[G_0 \mid R_x] : \overline{\mathsf{S}}] = \mathsf{Adv}_{G_0}^{\mathsf{Secrecy}}(C, \mathsf{false})$, by applying Lemma 2 for each game transformation. When we apply points 4 and 5 of this lemma, we use bounds on the probabilities of events, $\Pr[C[G \mid R_x] : D']$ and $\Pr[C[G \mid R_x] : D]$ respectively, which can be established using Lemma 1. (They can be written $\Pr[C[G \mid R_x] : (\mathsf{false} \wedge \neg \mathsf{NonUnique}) \vee D]$, so they are of the form required by point 2 of Lemma 1.) These probabilities are not multiplied by 2, so we improve over the standard computation of probabilities for secrecy.

These improvements are implemented in CryptoVerif but also apply to manual proofs. For instance, by applying this result to the manual proof of OEKE [55], we obtain that the probability for any adversary to make a server instance accept with no terminating client partner is bounded by

$$\frac{N_U + N_S}{N} + q_h \mathsf{Succ}_{\mathbb{G}}^{\mathsf{cdh}}(t') + p'_{coll}$$

$$\text{with } p'_{coll} = \frac{(2q_e + 2N_U + 3N_S + 3N_P)^2}{2(q-1)} + \frac{q_h^2 + 2N_S}{2^{l_1+1}}$$

and that no adversary can distinguish the session key from a random key with advantage greater than

$$\frac{N_U + N_S}{N} + q_h \mathsf{Succ}_{\mathbb{G}}^{\mathsf{cdh}}(t') + 2p'_{coll}$$

with the notations of Section II. (The detailed computation is in Appendix D.) For both properties, the first term of the probability $\frac{N_U+N_S}{N}$ shows that the adversary can test at most one password for each interaction with the client or the server, which is the optimal result, while the standard evaluation of probabilities given in Section II yields $\frac{N_U+\mathbf{2}N_S}{N}$ for the first property and $\frac{\mathbf{2}N_U+\mathbf{4}N_S}{N}$ for the second one. Similar improvements could also be obtained for the AuthA protocol [55, Section 4.1] and for the forward secrecy property [55, Appendix D].

1) Inserting events: In order to introduce events, we have implemented a new game transformation in CryptoVerif: `insert_event` e o inserts **event** e; **abort** at program point o. The program point o is an integer, which can be determined using the command `show_game occ`: this command displays the current game with the corresponding label $\{o\}$ at each program point. The command `show_game occ` also allows one to inspect the game, for instance to know the names of fresh variables created by CryptoVerif during previous transformations. Program points and variable names may depend on the version of CryptoVerif; this paper uses CryptoVerif 1.14. CryptoVerif cannot guess where events should be introduced, so the command `insert_event` must be manually given to the tool.

We have also defined a command `insert` o ins which adds instruction ins at the program point o. The instruction ins can for instance be a test, in which case all branches of the test will be copies of the code that follows program point o (so that the semantics of the game is unchanged). It can also be an assignment or a random generation of a fresh variable. In all cases, CryptoVerif checks that this instruction preserves the semantics of the game, and rejects it with an error message if it does not.

2) Transformations for h1*:* At the beginning of the proof, we transform the game using the random oracle assumption for h1. This transformation helps us make a program point appear at which we will next insert an event. Before actually performing this transformation, we first introduce a case distinction that leads to a simpler game after applying the random oracle assumption:

- By command `insert 261` "**let** $\mathsf{concat}(x1, x2, x3, x4, x5) = h1x$ **in**", we introduce a **let** in the hash oracle for h1. As the result, this hash oracle becomes $\mathsf{OH1}(h1x : bitstring) :=$ **let** $\mathsf{concat}(x1, x2, x3, x4, x5) = h1x$ **in return**$(\mathsf{h1}(hk1, h1x))$ **else return**$(\mathsf{h1}(hk1, h1x))$: the meaning of this **let** construct is that, if $h1x$ is of the form $\mathsf{concat}(x1, x2, x3, x4, x5)$, then the **in** branch is taken with $x1, x2, x3, x4, x5$ bound to their value (which is uniquely determined because the length of the fields of the concatenation is fixed); otherwise, the **else** branch is taken. Thus, we distinguish cases depending on whether $h1x$ is of the form $\mathsf{concat}(\ldots)$ or not. In the next transformation, which applies the random oracle assumption to h1, we are going to replace calls to h1 with lookups in the previous queries to h1. All queries to h1 in the protocol have an argument of the form $\mathsf{concat}(x1', x2', x3', x4', x5')$. When comparing this query to a query in the hash oracle, the comparison $h1x = \mathsf{concat}(x1', x2', x3', x4', x5')$ can then be simplified as follows:
  - If $\mathsf{h1}(h1x)$ was computed in the **in** branch of the introduced **let**, the comparison becomes $\mathsf{concat}(x1, x2, x3, x4, x5) = \mathsf{concat}(x1', x2', x3', x4', x5')$, that is, $x1 = x1' \wedge \ldots \wedge x5 = x5'$.
  - If $\mathsf{h1}(h1x)$ was computed in the **else** branch of this **let**, the comparison becomes false, because $h1x$ cannot be of the form $\mathsf{concat}(\ldots)$, since the **in** branch would have been taken in that case.
- `crypto` $\mathsf{rom}(\mathsf{h1})$ applies the equivalence $L_1 \approx_{p_1} R_1$ of Figure 2, designated by rom for Random Oracle Model, to the hash function h1: it transforms calls to h1 into lookups in the previous queries to h1, as outlined in Section III-B.

3) Event Auth*:* Next, we introduce event Auth: This event corresponds to the case in which the group element X received by the server (denoted X_s) does not come from the client, the authenticator $Auth$ received by the server (denoted $auth\_s$) comes from a hash query by the adversary, and authentication still succeeds. To be able to introduce this event, we first make the program point appear, at which this event will be inserted:

- `insert 179` "**find** $j \leq NU$ **suchthat defined**$(X[j]) \wedge X[j] = X\_s$ **then**" inserts a test after receiving the authenticator in the server, to distinguish the case in which X_s comes from the client ($X\_s = X[j]$ for some j).
- `insert 341` "**find** $jh \leq qH1$ **suchthat defined**$(x1[jh], x2[jh], x3[jh], x4[jh], hash_1[jh]) \wedge (\mathsf{U} = x1[jh]) \wedge (\mathsf{S} = x2[jh]) \wedge (X\_s = x3[jh]) \wedge (Y = x4[jh]) \wedge (auth\_s = hash_1[jh])$ **then**" inserts a test, in the **else** branch of the previous one, to detect when authentication succeeds with an authenticator $auth\_s$ that comes from a hash query made by the adversary. The result of that hash query[2] is $hash_1[jh]$ and its arguments are $x1[jh], \ldots, x5[jh]$. We purposely do not test that the 5-th argument of the hash query is the expected one. This avoids computing an exponentiation $\mathsf{exp}(X\_s, y)$ where X_s comes from the adversary and y is a secret exponent, thus removing a query to Ob in the CDH equivalence.
- `insert_event Auth 384` inserts the event itself in the **then** branch of the previous test.
- Finally, `simplify` cleans up the obtained game. The **else** branch of the **find** jh inserted above is removed: in that branch, authentication always fails so the protocol executes nothing.

[2]In CryptoVerif 1.14, the variable $hash_1$ is in fact named $@11\_r\_134$. We have renamed it to $hash_1$ for readability.

4) Event Encrypt*:* Next, we introduce the event Encrypt: This event corresponds to the case in which the value $Y^\star$ received by the client (denoted $Ystar\_u$) comes from an encryption query of the adversary under the correct password. As above, we have to prepare this insertion:

- `crypto` icm(enc) applies the equivalence that represents the Ideal Cipher Model, designated by icm, to the encryption scheme enc: it replaces calls to encryption/decryption with lookups in previous queries, as outlined in Section III-C.
- `insert_event` Encrypt 633 inserts the event Encrypt when the lookup in previous encryption/decryption queries that comes from the decryption of $Ystar\_u$ succeeds with an encryption query of the adversary.

5) Transformations for h0*:* We proceed for h0 similarly to what we did for h1 at the beginning of the proof:

- `insert` 1251 "**let** concat($x01, x02, x03, x04, x05$) = $h0x$ **in**" distinguishes cases depending on whether the argument $h0x$ of the hash oracle for h0 is of the form concat(...) or not.
- `crypto` rom(h0) applies the random oracle assumption to h0 (Section III-B).

B. Automatic Steps

After distinguishing cases for h0 and h1 and introducing events, we can use the automatic proof strategy of CryptoVerif, by command `auto`. Basically, this strategy consists in applying all possible cryptographic transformations (coming from equivalences $L \approx_p R$) and simplifying the game after each such transformation. When the transformations fail, they advise syntactic transformations that could make them succeed; these transformations are executed and the cryptographic transformation is then retried [49, Section 5].

More precisely, in our case, CryptoVerif renames several variables and simplifies terms, in order to be able to apply the CDH assumption (Section III-D). Details are provided in Appendix F. After these transformations, no automatic step can be performed, so the automatic proof stops.

C. Reorganizing Random Number Generations

We end up in a situation in which random values for Y are generated, but are used only in comparisons with previous queries. We would like to delay or remove these random number generations. This situation occurs at three places:

- When Y_u (the value of Y in the client) is a fresh random group element, $auth\_u$ and K_u are also fresh random values, independently of the value of Y_u, so Y_u is used only in comparisons with previous encryption/decryption queries.
- The value of Y in the passive eavesdroppings, Yp, is a fresh random group element; the encryption $Y^\star$ of Y is thus also a fresh random group element by the ideal cipher model, and the hash queries return a random value independently of the value of Yp, so Yp is also used only in comparisons with previous encryption/decryption queries.
- The value of Y in the server is also a fresh random group element; it is used in the test that decides whether to execute event Auth and in comparisons with previous encryption/decryption queries.

We have implemented new game transformations in CryptoVerif, detailed in Appendix E, to handle this situation:

- `move array` X delays the generation of a random value X until the point at which it is first used.
- `merge_arrays` $x_{11} \ldots x_{1n}, \ldots, x_{m1} \ldots x_{mn}$ merges the variables $x_{j1}, \ldots, x_{jn}$ into a single variable x_{j1} for each $j \leq m$. Each variable x_{jk} must have a single definition. For each $j \leq n$, the variables $x_{j1}, \ldots, x_{jn}$ must have the same type and indices of the same type. They must not be defined for the same value of their indices (that is, x_{jk} and $x_{jk'}$ must be defined in different branches of **if** or **find** when $k \neq k'$), so that they can be merged into a single array.
- `merge_branches` merges branches of **if** and **find** when they execute the same code.

Using these transformations, we can eliminate the random number generations for Y as outlined at the beginning of this section. We consider the three generations of Y in turn. For each of these generations, we first apply `move array` to the corresponding variable, to delay its generation. For OEKE, this has the effect of generating it in the decryption oracle available to the adversary. So, in this oracle, we end up with two possibilities of generating a fresh result, the one that comes from the delayed generation of Y, say Y', and the one that corresponds to the situation in which the query is really a fresh decryption query, say Y_d. We would like to merge these two cases by `merge_branches`. However, `merge_branches` does not succeed directly: we first need to merge the two variables Y_d and Y' into a single variable by `merge_arrays` $Y_d\, Y'$, then we can apply `merge_branches`. In the case of the value of Y in the server, we additionally need to rewrite the condition that triggers the event Auth for `merge_branches` to succeed. This is done by a few manual commands, checked correct by CryptoVerif. In this process, the event Auth is renamed into Auth2. These steps are detailed in Appendix F.

D. The Final Computation of Probabilities

In the obtained game, the events Auth2 and Encrypt are guarded by the following conditions (variables have been renamed for readability):

(**foreach** $iU \leq NU$ **do** ...
  find[**unique**] $je \leq qE$ **suchthat defined**$(re[je], ke[je]) \wedge Ystar\_u = re[je] \wedge pw = ke[je]$ **then event** Encrypt ...)
| (**foreach** $iS \leq NS$ **do** ...
  find $jh' \leq qH1, jd \leq qD$ **suchthat defined**$(x1[jh'],$

$x2[jh'], x3[jh'], x4[jh'], hash_1[jh'], m[jd], kd[jd],$
$rd[jd]) \wedge m[jd] = Y\_star \wedge \mathsf{U} = x1[jh'] \wedge$
$\mathsf{S} = x2[jh'] \wedge X\_s = x3[jh'] \wedge rd[jd] = x4[jh'] \wedge$
$auth\_s = hash_1[jh'] \wedge kd[jd] = pw$ **then**
event Auth2 ...) | ...

So, in order to bound the probabilities of these events, we just have to eliminate collisions between the password pw and the encryption and decryption keys, $ke[je]$ and $kd[jd]$. This is done by the command `simplify coll_elim` pw. The collisions on pw are not eliminated automatically by CryptoVerif because the type $passwd$ of pw is declared with annotation **password**. This annotation allows manual elimination of collisions but prevents automatic elimination of collisions. For passwords, whose set is not very large, the automatic elimination of collisions would yield a too large probability bound.

We have to evaluate the probability of these collisions. A naive evaluation considers that one makes at most $NU \times qE$ comparisons $pw = ke[je]$ (there are NU sessions of the client and the condition of **find** is evaluated at most qE times) and similarly at most $NS \times qH1 \times qD$ comparisons $kd[jd] = pw$, which yields the probability $(NU \times qE + NS \times qH1 \times qD)/|passwd|$. A slightly more clever way is to notice that $pw = ke[je]$ contains as only index $je \leq qE$, so at most qE distinct comparisons are performed (there are at most qE distinct encryption keys), and similarly at most qD distinct comparisons $kd[jd] = pw$, which yields the probability $(qE + qD)/|passwd|$. This is not satisfactory yet, because the encryption and decryption queries can be performed by the adversary without interacting with the protocol, so qE and qD can be large. So we have extended CryptoVerif to improve this probability bound. We start from the most naive evaluation $NU \times qE$ and try to eliminate each factor. We can eliminate NU as shown above, but we can also eliminate qE: for each session of the client, $Ystar\_u$ is fixed; since $Ystar\_u = re[je]$, $re[je]$ is also fixed. By eliminating collisions on re, there is a unique je that can make the comparison $Ystar\_u = re[je]$ succeed, so a unique je for which the comparison $pw = ke[je]$ is performed. Similarly, the comparison $kd[jd] = pw$ is performed at most once for each session of the server. Thus we obtain the probability $(NU + NS)/|passwd|$. To know which factors we should preferably eliminate, we annotate qE and qD with **noninteractive**, which means that the adversary can perform the corresponding queries without interacting with the protocol, so qE and qD will typically be larger than other bounds. Therefore, the bound $(NU + NS)/|passwd|$ is better than $(qE + qD)/|passwd|$, so CryptoVerif returns the former.

CryptoVerif then concludes that the events Encrypt and Auth2 can be executed with probability at most $(NU + NS)/|passwd|$ in the last game. Finally, CryptoVerif shows that OEKE preserves the secrecy of sk_u up to probability

$$\frac{NS + NU}{|passwd|} + (2qH0 + 3qH1)\mathsf{Succ}^{\mathsf{cdh}}_{\mathbb{G}}(t') + 2p''_{coll}$$

and satisfies the correspondences (1) and (2) with public variables $\{sk\_u\}$ up to probability

$$\frac{NS + NU}{|passwd|} + (4qH0 + 6qH1)\mathsf{Succ}^{\mathsf{cdh}}_{\mathbb{G}}(t') + p''_{coll}$$

where $t' = t + (2qH0 + 3qH1 + qD + 2NU + 2NP + NS)\tau_{\mathsf{exp}}$ and the terms in p''_{coll} come from elimination of collisions between hashes and between group elements: $p''_{coll} \leq (NS + NU + qH1 \times NU + qH1^2)/|hash1| + (qD \times NU \times NP + NU^2 \times NP + qD \times NU \times NS + NU^2 \times NS + 2qH1 \times NP + 4qE \times NP + 4qE \times NS + 4NP^2 + 3NS^2 + 2.5qD^2 + 9NP \times NU + 9NU \times NS + 7NS \times qD + 6NP \times qD + 10NS \times NP + 12.5NU^2 + 2qD \times qE + qH1 \times NU + 2qH0 \times NU + 4NU \times qD + 3NU \times qE + 1.5qE^2 + 6NS + 10NU)/|G|$. The main term in this probability is $(NS + NU)/|passwd|$: the adversary can test at most one password per session with the client or the server (active attack), which is the best bound we can hope. In contrast, [55] yields a bound of at most 4 passwords per session. In Section IV-A, by applying our improvement of the computation of probabilities to the manual proof of [55], we obtained the same first term as CryptoVerif, and even better second and third terms. CryptoVerif obtains a second term larger than in Section IV-A because it counts several Diffie-Hellman queries, which in fact correspond to the same query, and because the CDH assumption does not benefit from the improvement of Lemma 2, points 4 and 5: the probability of breaking CDH is taken into account using Lemma 2, point 3, so it is multiplied by 2.

V. Conclusion

We have proved the security of OEKE using the tool CryptoVerif. This proof provides additional confidence that the protocol is correct. Moreover, we have improved the probability bound given in [55]: we have shown that the adversary can test at most one password per session with the client or with the server, which is the optimal result. OEKE is a non-trivial case study, which is interesting on its own. It was also an opportunity to implement many extensions to CryptoVerif, which will be useful for proving many other protocols. We have already used the model of CDH to prove a signed Diffie-Hellman protocol. We plan to apply these extensions to other protocols, such as IKEv2 or SSH, which also rely on Diffie-Hellman. Our improvement of the computation of probabilities is also of general interest, and applies to manual proofs as well as CryptoVerif proofs.

Acknowledgments: We thank David Pointcheval for his advice and help during this project. This work was partly supported by the ANR project ProSe (decision number ANR-2010-VERS-004-01).

REFERENCES

[1] B. Chor and R. L. Rivest, "A Knapsack type public key cryptosystem based on arithmetic in finite fields," in *CRYPTO'84*, ser. LNCS, vol. 196. Springer, 1985, pp. 54–65.

[2] H. W. Lenstra Jr., "On the Chor-Rivest knapsack cryptosystem," *Journal of Cryptology*, vol. 3, no. 3, pp. 149–155, 1991.

[3] S. Vaudenay, "Cryptanalysis of the Chor-Rivest cryptosystem," in *CRYPTO'98*, ser. LNCS, vol. 1462. Springer, 1998, pp. 243–256.

[4] M. Bellare and P. Rogaway, "The exact security of digital signatures: How to sign with RSA and Rabin," in *EUROCRYPT'96*, ser. LNCS, vol. 1070. Springer, 1996, pp. 399–416.

[5] K. Ohta and T. Okamoto, "On concrete security treatment of signatures derived from identification," in *CRYPTO'98*, ser. LNCS, vol. 1462. Springer, 1998, pp. 354–369.

[6] M. Bellare and P. Rogaway, "Optimal asymmetric encryption," in *EUROCRYPT'94*, ser. LNCS, vol. 950. Springer, 1994, pp. 92–111.

[7] V. Shoup, "OAEP reconsidered," *Journal of Cryptology*, vol. 15, no. 4, pp. 223–249, 2002.

[8] E. Fujisaki, T. Okamoto, D. Pointcheval, and J. Stern, "RSA-OAEP is secure under the RSA assumption," *Journal of Cryptology*, vol. 17, no. 2, pp. 81–104, 2004.

[9] G. Barthe, B. Grégoire, S. Z. Béguelin, and Y. Lakhnech, "Beyond provable security. Verifiable IND-CCA security of OAEP," in *CT-RSA 2011*, ser. LNCS, vol. 6558. Springer, 2011, pp. 180–196.

[10] S. Halevi, "A plausible approach to computer-aided cryptographic proofs," Cryptology ePrint Archive, Report 2005/181, 2005, http://eprint.iacr.org/.

[11] M. Abadi and P. Rogaway, "Reconciling two views of cryptography (the computational soundness of formal encryption)," *Journal of Cryptology*, vol. 15, no. 2, pp. 103–127, 2002.

[12] V. Cortier and B. Warinschi, "Computationally sound, automated proofs for security protocols," in *ESOP'05*, ser. LNCS, vol. 3444. Springer, 2005, pp. 157–171.

[13] R. Janvier, Y. Lakhnech, and L. Mazaré, "Completing the picture: Soundness of formal encryption in the presence of active adversaries," in *ESOP'05*, ser. LNCS, vol. 3444. Springer, 2005, pp. 172–185.

[14] H. Comon-Lundh and V. Cortier, "Computational soundness of observational equivalence," in *CCS'08*. ACM, 2008, pp. 109–118.

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

[16] V. Cortier and B. Warinschi, "A composable computational soundness notion," in *CCS'11*. ACM, 2011, pp. 63–74.

[17] V. Cortier, S. Kremer, and B. Warinschi, "A survey of symbolic methods in computational analysis of cryptographic systems," *Journal of Automated Reasoning*, vol. 46, no. 3–4, pp. 225–259, 2011.

[18] V. Cortier, H. Hördegen, and B. Warinschi, "Explicit randomness is not necessary when modeling probabilistic encryption," in *ICS 2006*, ser. ENTCS, vol. 186. Elsevier, 2006, pp. 49–65.

[19] M. Backes, B. Pfitzmann, and M. Waidner, "A composable cryptographic library with nested operations," in *CCS'03*. ACM, 2003, pp. 220–230.

[20] M. Backes and B. Pfitzmann, "Symmetric encryption in a simulatable Dolev-Yao style cryptographic library," in *CSFW'04*. IEEE, 2004, pp. 204–218.

[21] C. Sprenger, M. Backes, D. Basin, B. Pfitzmann, and M. Waidner, "Cryptographically sound theorem proving," in *CSFW'06*. IEEE, 2006, pp. 153–166.

[22] C. Sprenger and D. Basin, "Cryptographically-sound protocol-model abstractions," in *LICS'08*. IEEE, 2008, pp. 3–17.

[23] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *FOCS'01*. IEEE, 2001, pp. 136–145.

[24] R. Canetti and J. Herzog, "Universally composable symbolic analysis of cryptographic protocols (the case of encryption-based mutual authentication and key exchange)," Cryptology ePrint Archive, Report 2004/334, 2004, available at http://eprint.iacr.org/2004/334.

[25] B. Blanchet, "Automatic proof of strong secrecy for security protocols," in *IEEE Symposium on Security and Privacy*, 2004, pp. 86–100.

[26] J. C. Mitchell, A. Ramanathan, A. Scedrov, and V. Teague, "A probabilistic polynomial-time calculus for the analysis of cryptographic protocols," *Theoretical Computer Science*, vol. 353, no. 1–3, pp. 118–164, 2006.

[27] D. Nowak and Y. Zhang, "A calculus for game-based security proofs," in *ProvSec 2010*, ser. LNCS, vol. 6402. Springer, 2010, pp. 35–52.

[28] A. Datta, A. Derek, J. C. Mitchell, V. Shmatikov, and M. Turuani, "Probabilistic polynomial-time semantics for a protocol security logic," in *ICALP'05*, ser. LNCS, vol. 3580. Springer, 2005, pp. 16–29.

[29] A. Datta, A. Derek, J. C. Mitchell, and B. Warinschi, "Computationally sound compositional logic for key exchange protocols," in *CSFW'06*. IEEE, 2006, pp. 321–334.

[30] G. Barthe, M. Daubignard, B. Kapron, and Y. Lakhnech, "Computational indistinguishability logic," in *CCS'10*. ACM Press, 2010, pp. 375–386.

[31] R. Canetti, L. Cheung, D. Kaynar, M. Liskov, N. Linch, O. Pereira, and R. Segala, "Time-bounded task-PIOAs: A framework for analyzing security protocols," in *DISC'06*, ser. LNCS, vol. 4167. Springer, 2006, pp. 238–253.

[32] P. Laud, "Secrecy types for a simulatable cryptographic library," in *CCS'05*. ACM, 2005, pp. 26–35.

[33] P. Laud and V. Vene, "A type system for computationally secure information flow," in *FCT'05*, ser. LNCS, vol. 3623. Springer, 2005, pp. 365–377.

[34] G. Smith and R. Alpízar, "Secure information flow with random assignment and encryption," in *FMSE'06*, 2006, pp. 33–43.

[35] J. Courant, C. Ene, and Y. Lakhnech, "Computationally sound typing for non-interference: The case of deterministic encryption," in *FSTTCS'07*, ser. LNCS, vol. 4855. Springer, 2007, pp. 364–375.

[36] M. Backes and P. Laud, "Computationally sound secrecy proofs by mechanized flow analysis," in *CCS'06*. ACM, 2006, pp. 370–379.

[37] J. Courant, M. Daubignard, C. Ene, P. Lafourcade, and Y. Lakhnech, "Towards automated proofs for asymmetric encryption schemes in the random oracle model," in *CCS'08*. ACM, 2008, pp. 371–380.

[38] ——, "Automated proofs for asymmetric encryption," in *Concurrency, Compositionality, and Correctness*, ser. LNCS, vol. 5930. Springer, 2010, pp. 300–321.

[39] G. Barthe, B. Grégoire, and S. Zanella, "Formal certification of code-based cryptographic proofs," in *POPL'09*. ACM, 2009, pp. 90–101.

[40] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, "Formal certification of ElGamal encryption. A gentle introduction to CertiCrypt," in *FAST 2008*, ser. LNCS, vol. 5491. Springer, 2009, pp. 1–19.

[41] S. Z. Béguelin, B. Grégoire, G. Barthe, and F. Olmedo, "Formally certifying the security of digital signature schemes," in *IEEE Symposium on Security and Privacy*. IEEE, 2009, pp. 237–250.

[42] S. Z. Béguelin, G. Barthe, S. Heraud, B. Grégoire, and D. Hedin, "A machine-checked formalization of sigma-protocols," in *CSF'10*. IEEE, 2010, pp. 246–260.

[43] V. Shoup, "Sequences of games: a tool for taming complexity in security proofs," Cryptology ePrint Archive, Report 2004/332, 2004, available at http://eprint.iacr.org/2004/332.

[44] M. Bellare and P. Rogaway, "The security of triple encryption and a framework for code-based game-playing proofs," in *EUROCRYPT 2006*, ser. LNCS, vol. 4004. Springer, 2006, pp. 409–426.

[45] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, "Computer-aided security proofs for the working cryptographer," in *CRYPTO 2011*, 2011, to appear.

[46] D. Nowak, "A framework for game-based security proofs," in *ICICS 2007*, ser. LNCS, vol. 4861. Springer, 2007, pp. 319–333.

[47] ——, "On formal verification of arithmetic-based cryptographic primitives," in *ICISC 2008*, ser. LNCS, vol. 5461. Springer, 2008, pp. 368–382.

[48] R. Affeldt, D. Nowak, and K. Yamada, "Certifying assembly with formal cryptographic proofs: the case of BBS," in *AVoCS'09*, ser. Electronic Communications of the EASST, vol. 23, 2009.

[49] 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.

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

[51] P. Laud, "Handling encryption in an analysis for secure information flow," in *ESOP'03*, ser. LNCS, vol. 2618. Springer, 2003, pp. 159–173.

[52] ——, "Symmetric encryption in automatic analyses for confidentiality against active adversaries," in *IEEE Symposium on Security and Privacy*, 2004, pp. 71–85.

[53] I. Tšahhirov and P. Laud, "Application of dependency graphs to security protocol analysis," in *TGC'07*, ser. LNCS, vol. 4912. Springer, 2007, pp. 294–311.

[54] P. Laud and I. Tšahhirov, "A user interface for a game-based protocol verification tool," in *FAST2009*, ser. LNCS, vol. 5983. Springer, 2009, pp. 263–278.

[55] E. Bresson, O. Chevassut, and D. Pointcheval, "Security proofs for an efficient password-based key exchange," in *CCS'03*. ACM, 2003, pp. 241–250.

[56] S. M. Bellovin and M. Merritt, "Encrypted key exchange: Password-based protocols secure against dictionary attacks," in *IEEE Symposium on Security and Privacy*. IEEE, 1992, pp. 72–84.

[57] M. Bellare and P. Rogaway, "The AuthA protocol for password-based authenticated key exchange," Mar. 2000, contributions to IEEE P1363. Available from http://grouper.ieee.org/groups/1363/.

[58] ——, "Random oracles are practical: A paradigm for designing efficient protocols," in *CCS'93*. ACM, 1993, pp. 62–73.

[59] M. Bellare, D. Pointcheval, and P. Rogaway, "Authenticated key exchange secure against dictionary attacks," in *EUROCRYPT 2000*, ser. LNCS, vol. 1807. Springer, 2000, pp. 139–155.

[60] J.-S. Coron, "Security proof for partial-domain hash signature schemes," in *CRYPTO 2002*, ser. LNCS, vol. 2442. Springer, 2002, pp. 613–626.

[61] B. Blanchet, "Computationally sound mechanized proofs of correspondence assertions," in *CSF'07*. IEEE, 2007, pp. 97–111, extended version available as ePrint Report 2007/128, http://eprint.iacr.org/2007/128.

[62] T. Y. C. Woo and S. S. Lam, "A semantic model for authentication protocols," in *IEEE Symposium on Research in Security and Privacy*, 1993, pp. 178–194.

[63] M. Abdalla, P.-A. Fouque, and D. Pointcheval, "Password-based authenticated key exchange in the three-party setting," *IEE Proceedings Information Security*, vol. 153, no. 1, pp. 27–39, 2006.

[64] V. Shoup, "Lower bounds for discrete logarithms and related problems," in *EUROCRYPT'97*, ser. LNCS, vol. 1233. Springer, 1997, pp. 256–266.

APPENDIX A.
BASIC DEFINITIONS AND PROPERTIES

This appendix recalls and sometimes adapts the definitions of the basic concepts used by CryptoVerif. In CryptoVerif, games are represented in a process calculus. A similar calculus was presented in detail in [49], using channels instead of oracles and asymptotic security instead of exact security. The syntax of this calculus is recalled in Figure 5.

This calculus uses parameters, denoted by n, which represent integer values. This calculus also uses types, denoted by T, corresponding to subsets of $bitstring \cup \{\bot\}$ where $bitstring$ is the set of all bitstrings and $\bot$ is a special symbol. We say that a type is *large* when its cardinal is large enough so that we can harmlessly eliminate collisions between random values of this type. Particular types are predefined: $bool = \{\mathsf{true}, \mathsf{false}\}$, where false is 0 and true is 1; $bitstring$; $bitstring_\bot = bitstring \cup \{\bot\}$; $[1, n]$ where n is a parameter. (We consider integers as bitstrings without leading zeroes.)

The calculus 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 a function from m-tuples of bitstrings or $\bot$ in $T_1 \times \ldots \times T_m$ to a bitstring or $\bot$. 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 calculus, terms represent computations on bitstrings. The replication index i is an integer which serves in distinguishing different copies of a replicated process **foreach** $i \leq n$ **do** Q. (Replication indices are typically used as array indices.) The variable access $x[M_1, \ldots, M_m]$ returns the content of the cell of indices $M_1, \ldots, M_m$ of the m-dimensional array variable x. The function application $f(M_1, \ldots, M_m)$ returns the result of applying function f to $M_1, \ldots, M_m$.

The calculus distinguishes two categories of processes: oracle definitions Q consist of a set of definitions of oracles, while oracle bodies P describe the content of an oracle definition. Oracle bodies perform some computations and return a result. After returning the result, they may define new oracles. (An oracle definition Q follows the **return**$(N_1, \ldots, N_k)$ instruction.)

The nil oracle definition 0 defines no oracle; $Q \mid Q'$ is the parallel composition of Q and Q': it makes available both oracles defined in Q and in Q'; **foreach** $i \leq n$ **do** Q represents n copies of Q in parallel, each with a different value of $i \in [1, n]$. The construct **newOracle** $O; Q$ hides oracle O outside Q: oracle O can be called only inside Q. The oracle definition $O[\widetilde{i}](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k) := P$ defines an oracle O, taking arguments $x_1, \ldots, x_k$ of types

$M, N ::=$	terms
i	replication index
$x[M_1, \ldots, M_m]$	variable access
$f(M_1, \ldots, M_m)$	function application
$FC ::=$	find condition
M	term
$x[\widetilde{i}] : T \leftarrow M; FC$	assignment
if defined$(M_1, \ldots, M_l) \wedge M$ **then** FC **else** FC'	conditional
find$[unique?]$ $(\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}$ **suchthat defined**$(M_{j1}, \ldots, M_{jl_j}) \wedge FC_j$ **then** FC'_j) **else** FC	array lookup
$Q ::=$	oracle definitions
0	nil
$Q \mid Q'$	parallel composition
foreach $i \leq n$ **do** Q	n parallel copies
newOracle $O; Q$	restriction for oracles
$O[\widetilde{i}](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k) := P$	oracle definition
$P ::=$	oracle body
$x[\widetilde{i}] \stackrel{R}{\leftarrow} T; P$	random choice
$x[\widetilde{i}] : T \leftarrow M; P$	assignment
if defined$(M_1, \ldots, M_l) \wedge M$ **then** P **else** P'	conditional
find$[unique?]$ $(\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}$ **suchthat defined**$(M_{j1}, \ldots, M_{jl_j}) \wedge FC_j$ **then** P_j) **else** P	array lookup
event $e(M_1, \ldots, M_m); P$	event
$(x_1[\widetilde{i}] : T_1, \ldots, x_{k'}[\widetilde{i}] : T_{k'}) \leftarrow O[M_1, \ldots, M_l](N_1, \ldots, N_k); P$ **else** P'	oracle call
return$(N_1, \ldots, N_k); Q$	return
end	end
abort	abort
$C ::=$	contexts
$[\,]$	hole
$C \mid Q$	parallel composition
$Q \mid C$	parallel composition
newOracle $O; C$	restriction for oracles

Figure 5. Syntax of the process calculus

$T_1, \ldots, T_k$ respectively, and computed as described in oracle body P, where $\widetilde{i}$ denotes a tuple $i_1, \ldots, i_m$.

The random choice $x[\widetilde{i}] \stackrel{R}{\leftarrow} T; P$ chooses a new random number uniformly in T, stores it in $x[\widetilde{i}]$, and executes P. Function symbols represent deterministic functions, so all random numbers must be chosen by $x[\widetilde{i}] \stackrel{R}{\leftarrow} T$. The assignment $x[\widetilde{i}] : T \leftarrow M; P$ stores the bitstring value of M (which must be in T) in $x[\widetilde{i}]$ and executes P.

Next, we explain the array lookup **find** $(\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}$ **suchthat defined**$(M_{j1}, \ldots, M_{jl_j}) \wedge FC_j$ **then** $P_j)$ **else** P. The order and array indices on tuples are taken component-wise, so for instance, $u_{j1}[\widetilde{i}] \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}$ can be abbreviated $\widetilde{u_j}[\widetilde{i}] \leq \widetilde{n_j}$. A simple example is the following: **find** $u \leq n$ **suchthat defined**$(x[u]) \wedge x[u] = a$ **then** P' **else** P tries to find an index u such that $x[u]$ is defined and $x[u] = a$, and when such a u is found, it executes P' with that value of u; otherwise, it executes P. In other words, this **find** construct looks for the value a in the array x, and when a is found, it stores in u an index such that $x[u] = a$. More generally, **find** $u_1[\widetilde{i}] \leq n_1, \ldots, u_m[\widetilde{i}] \leq n_m$ **suchthat defined**$(M_1, \ldots, M_l) \wedge FC$ **then** P' **else** P tries to find values of $u_1, \ldots, u_m$ for which $M_1, \ldots, M_l$ are defined and FC is true. In case of success, it executes P'. In case of failure, it executes P. This is further generalized to m branches: **find** $(\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}$ **suchthat defined**$(M_{j1}, \ldots, M_{jl_j}) \wedge FC_j$ **then** $P_j)$ **else** P tries to find a branch j in $[1, m]$ such that there are values of $u_{j1}, \ldots, u_{jm_j}$ for which $M_{j1}, \ldots, M_{jl_j}$ are defined and FC_j is true. In case of success, it executes P_j. In case of failure for all branches, it executes P. More formally, it evaluates the conditions **defined**$(M_{j1}, \ldots, M_{jl_j}) \wedge FC_j$ for each j and each value of $u_{j1}[\widetilde{i}], \ldots, u_{jm_j}[\widetilde{i}]$ in $[1, n_{j1}] \times \ldots \times [1, n_{jm_j}]$. If none of these conditions is true, it executes P. Otherwise, it chooses randomly with (almost) uniform probability one j and one value of $u_{j1}[\widetilde{i}], \ldots, u_{jm_j}[\widetilde{i}]$ such that the corresponding condition is true, and executes P_j. (When the number of possibilities is not a power of 2, a probabilistic bounded-time Turing machine cannot choose these values exactly with uniform probability, but it can choose them with a probability distribution as close as we wish to uniform.) Optionally, one may add a [**unique**] modifier to **find**, represented in Figure 5 by [*unique*?]. When this modifier is present and there are several values of $j, u_{j1}[\widetilde{i}], \ldots, u_{jm_j}[\widetilde{i}]$ such that the corresponding condition is true, we execute the event NonUnique and abort the game. The **find** conditions FC can themselves contain not only terms but also assignments, conditionals, and array lookups.

The process **if defined**$(M_1, \ldots, M_l) \wedge M$ **then** P **else** P' is syntactic sugar for **find suchthat defined**$(M_1, \ldots, M_l) \wedge M$ **then** P **else** P'.

The construct **event** $e(M_1, \ldots, M_m); P$ executes the event $e(M_1, \ldots, M_m)$, then executes P. Events just record that a certain program point has been reached, with certain values of the arguments of the event. They do not influence the execution of the rest of the process.

The oracle call $(x_1[\widetilde{i}] : T_1, \ldots, x_{k'}[\widetilde{i}] : T_{k'}) \leftarrow O[M_1, \ldots, M_l](N_1, \ldots, N_k); P$ **else** P' calls oracle $O[M_1, \ldots, M_l]$ with arguments $N_1, \ldots, N_k$. When this oracle returns a result by **return**$(N'_1, \ldots, N'_{k'})$, this result is stored in $x_1[\widetilde{i}], \ldots, x_{k'}[\widetilde{i}]$ and the process executes P. When the oracle $O[M_1, \ldots, M_l]$ terminates by **end**, the process executes P'. (Returning a result by **return** corresponds to the normal termination of the oracle O, while terminating with **end** corresponds to abnormal termination.) Finally, the instruction **abort** aborts the game: the whole game terminates immediately and returns the special value abort.

To lighten notations, $\wedge$ true and **defined**() $\wedge$ may be omitted in conditions of **if** and **find**. Moreover, **else end**, a trailing 0, or a trailing **end** may be omitted. Types may be omitted when they can be inferred.

The *current replication indices* at a certain program point in a process are $i_1, \ldots, i_m$ when the considered program point is under **foreach** $i_1 \leq n_1$ **do** ... **foreach** $i_m \leq n_m$ **do** . We abbreviate $x[i_1, \ldots, i_m]$ by x when $i_1, \ldots, i_m$ are the current replication indices, but it should be kept in mind that this is only an abbreviation. Similarly, an oracle definition $O[i_1, \ldots, i_m](\ldots) := P$ under **foreach** $i_1 \leq n_1 \ldots$ **foreach** $i_m \leq n_m$ is abbreviated $O(\ldots) := P$. Variables and oracles defined under **foreach** must be indexed by the current replication indices: for example **foreach** $i_1 \leq n_1$ **do** ... **foreach** $i_m \leq n_m$ **do** ... $x[i_1, \ldots, i_m] : T \leftarrow M; \ldots$

We require some *well-formedness invariants* to guarantee that several definitions of the same oracle cannot be simultaneously available, that bitstrings are of their expected type, and that arrays are used properly (that each cell of an array is assigned at most once during execution, and that variables are accessed only after being initialized). Formally, we require the following invariants:

Invariant 1 (Single definition for oracles) The process Q_0 satisfies Invariant 1 if and only if

1) in every definition of $O[i_1, \ldots, i_m]$ in Q_0, the indices $i_1, \ldots, i_m$ of O are the current replication indices at that definition, and
2) two different definitions of the same oracle O in Q_0 are in different branches of a **find** (or **if**).

Invariant 1 guarantees that each oracle is defined at most once for each value of its indices. (Indeed, item 2 shows that only one definition of each oracle can be available for given indices in each trace.)

Invariant 2 (Single definition for variables) The process Q_0 satisfies Invariant 2 if and only if

1) in every definition of $x[i_1, \ldots, i_m]$ in Q_0, the indices $i_1, \ldots, i_m$ of x are the current replication indices at that definition, and
2) two different definitions of the same variable x in Q_0 are in different branches of a **find** (or **if**).

Similarly to the previous invariant, Invariant 2 guarantees that each variable is assigned at most once for each value of its indices.

Invariant 3 (Defined variables) The process Q_0 satisfies Invariant 3 if and only if every occurrence of a variable access $x[M_1, \ldots, M_m]$ in Q_0 is either

- syntactically under the definition of $x[M_1, \ldots, M_m]$ (in which case $M_1, \ldots, M_m$ are in fact the current replication indices at the definition of x);
- or in a **defined** condition in a **find** construct;
- or in FC'_j or P_j in a process of the form **find** $(\bigoplus_{j=1}^{m''} \widetilde{u_j}[\tilde{i}] \leq \widetilde{n_j}$ **suchthat defined**$(M'_{j1}, \ldots, M'_{jl_j}) \wedge FC'_j$ **then** $P_j)$ **else** P where for some $k \leq l_j$, $x[M_1, \ldots, M_m]$ is a subterm of M'_{jk};
- or in FC'_j or FC_j in a **find** condition of the form **find** $(\bigoplus_{j=1}^{m''} \widetilde{u_j}[\tilde{i}] \leq \widetilde{n_j}$ **suchthat defined**$(M'_{j1}, \ldots, M'_{jl_j}) \wedge FC'_j$ **then** $FC_j)$ **else** FC where for some $k \leq l_j$, $x[M_1, \ldots, M_m]$ is a subterm of M'_{jk}.

Invariant 3 guarantees that variables can be accessed only when they have been initialized. It checks that the definition of the variable access is either in scope (first item) or checked by the **defined** condition of a **find** (last two items). A variable access that does not correspond to the first item of Invariant 3 is called an *array access*. We furthermore require the following invariant.

Invariant 4 (Variables defined in find conditions) The process Q_0 satisfies Invariant 4 if and only if the variables defined in conditions of **find** have no array accesses.

This invariant did not appear in previous versions of the calculus because conditions of **find** were restricted to be terms.

We use a type system (see [49, Appendix A]) to check that bitstrings of the proper type are passed to each function and that array indices are used correctly.

Invariant 5 (Typing) The process Q_0 satisfies Invariant 5 if and only if it is well-typed.

We require the adversary to be well-typed. This requirement does not restrict its computing power, because it can always define type-cast functions $f : T \to T'$ to bypass the type system. Similarly, the type system does not restrict the class of protocols that we consider, since the protocol may contain type-cast functions. The type system just makes explicit which set of bitstrings may appear at each point of the protocol.

These invariants are checked by the prover for the initial game and preserved by all game transformations. We suppose that all games satisfy these invariants.

We use a context C to represent an adversary. A context is a process with a hole. In this paper, we consider only evaluation contexts, generated by the grammar given at the bottom of Figure 5. A context C is put around a process Q by $C[Q]$. This construct means that Q is put in parallel with some other process Q' contained in C, possibly hiding some oracles defined in Q, so that, when considering $C'[C[Q]]$, C' cannot call these oracles.

A context C is said to be *acceptable* for Q with public variables V if and only if the common variables of C and Q are in V, and $C[Q]$ satisfies the well-formedness invariants.

We name $Ostart$ the oracle called to start the experiment. We denote by $\Pr[Q : D]$ the probability that, for some sequence of events $\mathcal{E}$ and bitstring a, when oracle $Ostart()$ is called, Q executes exactly the sequence $\mathcal{E}$, in the same order, and returns the result a, such that the algorithm $D(\mathcal{E}, a)$ returns true.

We denote by $\mathrm{var}(Q)$ (resp. $\mathrm{var}(C)$) the set of variables of the process Q (resp. context C).

The following lemma is a straightforward consequence of Definition 1:

Lemma 3 1) $Q \approx_0^V Q$.
2) $\approx_p^V$ *is symmetric.*
3) *If* $Q \approx_p^V Q'$ *and* $Q' \approx_{p'}^V Q''$, *then* $Q \approx_{p+p'}^V Q''$.
4) *If* $Q \approx_p^V Q'$ *and* C *is a context acceptable for* Q *and* Q' *with public variables* V, *then* $C[Q] \approx_{p'}^{V'} C[Q']$, *where* $p'(C', t_D) = p(C'[C[\,]], t_D)$ *and* $V' \subseteq V \cup \mathrm{var}(C)$.

APPENDIX B.
THE COMPUTATIONAL DIFFIE-HELLMAN ASSUMPTION

A. Proof of the Reduction

First, let us rephrase the two games:

G1: the adversary is allowed to query

- for group elements, via oracles OA and OB: OA$[i]()$[3] provides a g^{a_i} element for a random a_i (at most na queries), and OB$[j]()$ provides a g^{b_j} element for a random b_j (at most nb queries);
- for discrete logarithms, via oracles Oa and Ob: Oa$[i]()$ outputs a_i (at most $\#\mathsf{Oa} \leq na$ queries), Ob$[j]()$ outputs b_j (at most $\#\mathsf{Ob} \leq nb$ queries);
- for Diffie-Hellman decisions, via ODDHa and ODDHb oracles: ODDHa$[i](m, j)$ checks whether $m = g^{a_i b_j}$ (at most naDDH queries for each a, at most $\#$ODDHa queries total),

[3]The argument $[i]$ between brackets is implicit in the CryptoVerif syntax and corresponds to the replication index.

$\mathsf{ODDHb}[j](m,i)$ checks whether $m = g^{a_i b_j}$ (at most nbDDH queries for each b, at most $\#\mathsf{ODDHb}$ queries total). We can thus combine them into $\mathsf{DDH}(m,a_i,b_j)$ queries (at most $q_{\mathsf{ddh}} = \#\mathsf{ODDHa} + \#\mathsf{ODDHb} \leq na \cdot nb$ queries) which check whether $m = g^{a_i b_j}$.

G2: the adversary is allowed to query

- OA and OB oracles, that answer as above;
- Oa and Ob oracles, that answer as above;
- for $\mathsf{DDH}(m,a_i,b_j)$, Diffie-Hellman decisions, oracle. But in this game, the correct answer is given if either a_i or b_j has been asked *before* for an Oa or Ob query. Otherwise, the answer is 'false'.

We thus insist on the fact that the 2 games differ on $\mathsf{DDH}(m,a_i,b_j)$ Diffie-Hellman decisions queries, if neither a_i nor b_j has been asked *before* for an Oa or Ob query. In the first game, the answer is the correct one; in the second game, the answer is always 'false'.

Let us be given a CDH tuple $(X = g^x, Y = g^y)$ for which we want to compute $Z = g^{xy}$. And we provide a simulator $\mathcal{A}$ for these games:

- For the query $\mathsf{OA}[i]$, one chooses a random bit γ_i with bias p_a, and a random scalar $\alpha_i \stackrel{R}{\leftarrow} \mathbb{Z}_p$, and sets $A_i = X^{\gamma_i} g^{\alpha_i}$. This makes $a_i = \alpha_i + \gamma_i x$, and thus $a_i = \alpha_i$ if $\gamma_i = 0$;
- For the query $\mathsf{OB}[j]$, one chooses a random bit δ_j with bias p_b, and a random scalar $\beta_j \stackrel{R}{\leftarrow} \mathbb{Z}_q$, and sets $B_j = Y^{\delta_j} g^{\beta_j}$. This makes $b_j = \beta_j + \delta_j x$, and thus $b_j = \beta_j$ if $\delta_j = 0$;
- For the query $\mathsf{Oa}[i]$, with probability $1-p_a$, $\gamma_i = 0$, and then the correct answer α_i can be sent. However, with probability p_a on the current a, the simulation fails;
- For the query $\mathsf{Ob}[j]$, with probability $1-p_b$, $\delta_j = 0$, and then the correct answer β_j can be sent. However, with probability p_b on the current b, the simulation fails.

Since there are at most $\#\mathsf{Oa}$ Oa queries and $\#\mathsf{Ob}$ Ob queries, with probability at least $(1-p_a)^{\#\mathsf{Oa}}(1-p_b)^{\#\mathsf{Ob}}$, the simulation does not fail, and is perfectly indistinguishable from the real oracles OA, OB, Oa, and Ob. Let us now consider the DDH Diffie-Hellman Decision queries, and the simulation of the answers when no failure happens during Oa and Ob simulation: for any $\mathsf{DDH}(m,a_i,b_j)$,

- if one of the a_i or b_j has been asked an Oa or Ob-query (and did not lead to a failure), which means that either $\gamma_i = 0$ or $\delta_j = 0$, then one can either test whether $m = (g^{b_j})^{\alpha_i}$ or not, or whether $m = (g^{a_i})^{\beta_j}$ or not, and provide the correct answer. This leads to a perfect simulation of the DDH oracle;
- otherwise, one can safely answer 'false', which leads to a perfect simulation of the DDH oracle in the second game. However, it differs from the first game oracle if $m = g^{a_i b_j}$.

As a consequence, the two games differ if for one DDH query, $m = g^{a_i b_j}$ but neither a_i or b_j has been asked for an Oa or Ob query. In this case, with probability $p_a p_b$, both $\gamma_i = 1$ and $\delta_j = 1$:

$$m = Z \times X^{\beta_j} Y^{\alpha_i} g^{\alpha_i \beta_j}.$$

If the two above games differ with probability ε, then such a critical DDH query happens with probability ε, since our simulation is perfectly indistinguishable from the second game and such a critical query is the unique event that makes the two games different.

Let us randomly choose k between 1 and q_{ddh}, and bet that the k-th DDH query is the first critical one, which is true with probability $\varepsilon/q_{\mathsf{ddh}}$. Furthermore, with probability $(1-p_a)^{\#\mathsf{Oa}}(1-p_b)^{\#\mathsf{Ob}} p_a p_b$ the simulation did not fail and the critical DDH query leads to the expected Z value, by computing $Z = m/X^{\beta_j} Y^{\alpha_i} g^{\alpha_i \beta_j}$. This means that our simulator $\mathcal{A}$ achieves $\mathsf{Succ}^{\mathsf{cdh}}_{\mathbb{G}}(\mathcal{A}) \geq (1-p_a)^{\#\mathsf{Oa}}(1-p_b)^{\#\mathsf{Ob}} p_a p_b \varepsilon / q_{\mathsf{ddh}}$, and is bounded by $\mathsf{Succ}^{\mathsf{cdh}}_{\mathbb{G}}(t + (na + nb + q_{\mathsf{ddh}})\tau_{\mathsf{exp}})$:

$$\varepsilon \leq \frac{q_{\mathsf{ddh}} \times \mathsf{Succ}^{\mathsf{cdh}}_{\mathbb{G}}(t + (na+nb+q_{\mathsf{ddh}})\tau_{\mathsf{exp}})}{(1-p_a)^{\#\mathsf{Oa}}(1-p_b)^{\#\mathsf{Ob}} p_a p_b}$$

Note that, two cases appear for the function $x \mapsto x(1-x)^n$:

- if $n = 0$, then the maximum is 1, for $x = 1$;
- if $n \geq 1$, then the maximum is greater than $e^{-2}/n \geq 1/7.4n$, for $x = 1/(n+1)$.

By choosing $p_a = 1/(\#\mathsf{Oa}+1)$ and $p_b = 1/(\#\mathsf{Ob}+1)$ in the latter case, we get $1/(1-p_a)^{\#\mathsf{Oa}} p_a \leq \max(1, 7.4\#\mathsf{Oa})$ and $1/(1-p_b)^{\#\mathsf{Ob}} p_b \leq \max(1, 7.4\#\mathsf{Ob})$, so

$$\begin{aligned}\varepsilon \leq\ & (\#\mathsf{ODDHa} + \#\mathsf{ODDHb}) \times \\ & \max(1, 7.4\#\mathsf{Oa}) \times \max(1, 7.4\#\mathsf{Ob}) \times \\ & \mathsf{Succ}^{\mathsf{cdh}}_{\mathbb{G}}(t + (na + nb + \#\mathsf{ODDHa} + \#\mathsf{ODDHb})\tau_{\mathsf{exp}}).\end{aligned}$$

One could use Shoup's amplification technique [64] to eliminate the factor $\#\mathsf{ODDHa} + \#\mathsf{ODDHb}$, but at a computational cost: one runs the attack twice with different randomizations of the CDH instance, and looks for a collision in the two executions: if the attack succeeded twice, we have a collision on the correct answer; if the attack fails once, the collision probability is negligible.

B. Additional Modeling

In the CryptoVerif implementation, the equivalence $L_3 \approx_{p_3} R_3$ additionally contains three proof strategy indications, which we have omitted in Figure 4:

- In R_3, we use the symbol exp' instead of exp, although the two symbols represent the same function on bitstrings. This technique avoids infinite loops: if we used exp in R_3, R_3 would be an instance of L_3, so the transformation of L_3 into R_3 could be applied again on R_3, leading to an infinite loop. By using

exp', we prevent applying the transformation again on occurrences that have already been transformed.

- In L_3, the oracles Oa and Ob are marked with the integer "[3]". CryptoVerif tries to use oracles with the lowest mark first. (No mark means [0].) Here, the goal is to make sure that g^a is obtained by calling OA and not by calling Oa to obtain a and then computing g^a (and similarly for g^{ab} obtained by calling ODDHa or ODDHb rather than Oa and Ob). Indeed, if Oa is called, then the CDH assumption on that a can no longer be applied. Therefore, CryptoVerif should use OA, OB, ODDHa, or ODDHb rather than Oa or Ob where possible, hence we give Oa and Ob a higher mark than the other oracles.
- In L_3, the oracle ODDHa is marked $[useful\_change]$. This prevents the application of the transformation of L_3 into R_3 when the initial game can be encoded without calling ODDHa. Indeed, the transformation has a useful effect only when ODDHa or ODDHb are called and, by symmetry, we can require that ODDHa is called.

Moreover, we use the following properties. The multiplication mult is commutative and we have the following equalities:

$$\forall a : G, \forall x : Z, \forall y : Z, \\ \mathsf{exp}(\mathsf{exp}(a, x), y) = \mathsf{exp}(a, \mathsf{mult}(x, y)) \quad (3)$$

$$\forall x : Z, \forall y : Z, (\mathsf{exp}(\mathsf{g}, x) = \mathsf{exp}(\mathsf{g}, y)) = (x = y) \quad (4)$$

$$\forall x : Z, \forall y : Z, (\mathsf{exp}'(\mathsf{g}, x) = \mathsf{exp}'(\mathsf{g}, y)) = (x = y) \quad (5)$$

$$\forall x : Z, \forall y : Z, \forall y' : Z, \\ (\mathsf{mult}(x, y) = \mathsf{mult}(x, y')) = (y = y') \quad (6)$$

The commutativity of mult combined with (3) shows that $(g^a)^b = g^{ab} = g^{ba} = (g^b)^a$, the standard equality that shows that the client and the server compute the same key in the Diffie-Hellman key exchange. Equations (4) and (5) express the injectivity of exp and exp' respectively. They hold because g is a generator of the group $\mathbb{G}$ of order q and $x, y \in [1, q-1]$. Equation (6) is obtained by dividing the equality $xy = xy'$ by x in the group $\mathbb{Z}_q^*$.

The statement

$$\begin{array}{l} \textbf{collision } x1 \stackrel{R}{\leftarrow} Z; x2 \stackrel{R}{\leftarrow} Z; x3 \stackrel{R}{\leftarrow} Z; x4 \stackrel{R}{\leftarrow} Z; \\ \quad \textbf{return}(\mathsf{mult}(x1, x2) = \mathsf{mult}(x3, x4)) \\ \quad\quad \approx_{1/|Z|} \textbf{return}(\mathsf{false}). \end{array} \quad (7)$$

means that, when $x1$, $x2$, $x3$, $x4$ are uniformly randomly and independently chosen in Z, except with probability $1/|Z|$, one can replace $\mathsf{mult}(x1, x2) = \mathsf{mult}(x3, x4)$ with false. Indeed, $\mathsf{mult}(x1, x2) = \mathsf{mult}(x3, x4)$ if and only if $x1 \times x2/x3 = x4$ so the probability of $\mathsf{mult}(x1, x2) = \mathsf{mult}(x3, x4)$ is the probability of choosing an $x4$ in Z equal to a given $x1 \times x2/x3$, that is, $1/|Z|$. The formulas (4), (5), (6), and (7) allow CryptoVerif to simplify equalities between exponentials.

If we choose uniformly x in Z and compute g^x, the result is a uniformly distributed group element, so we have the equivalence

$$\begin{array}{l} \quad \textbf{foreach } i \leq n \textbf{ do } x \stackrel{R}{\leftarrow} Z; \mathsf{OX}() := \textbf{return}(\mathsf{exp}(\mathsf{g}, x)) \\ \approx_0 \textbf{foreach } i \leq n \textbf{ do } X \stackrel{R}{\leftarrow} G; \mathsf{OX}() := \textbf{return}(X). \end{array} \quad (8)$$

We have a similar equivalence for exp'. Although equivalences are symmetric, CryptoVerif always applies them from left to right, replacing the code of oracles in the left-hand side with the corresponding code in the right-hand side. For this reason, we state the symmetric equivalence explicitly:

$$\begin{array}{l} \textbf{foreach } i \leq n \textbf{ do } X \stackrel{R}{\leftarrow} G; \mathsf{OX}() := \textbf{return}(X) \\ \approx_0 [manual] \\ \textbf{foreach } i \leq n \textbf{ do } x \stackrel{R}{\leftarrow} Z; \mathsf{OX}() := \textbf{return}(\mathsf{exp}(\mathsf{g}, x)). \end{array} \quad (9)$$

However, this equivalence is applied only manually, as indicated by $[manual]$; otherwise, it would yield an infinite loop by applying alternatively (8) and (9). We use the following more restricted form for automatic proofs

$$\begin{array}{l} \quad \textbf{foreach } i \leq n \textbf{ do } X \stackrel{R}{\leftarrow} G; \\ \quad\quad (\mathsf{OX}() := \textbf{return}(X) \mid \\ \quad\quad\quad \textbf{foreach } i' \leq n' \textbf{ do } \mathsf{OXm}(m : Z) \\ \quad\quad\quad\quad [useful\_change] := \textbf{return}(\mathsf{exp}(X, m))) \\ \approx_0 \textbf{foreach } i \leq n \textbf{ do } x \stackrel{R}{\leftarrow} Z; \\ \quad\quad (\mathsf{OX}() := \textbf{return}(\mathsf{exp}(\mathsf{g}, x)) \mid \\ \quad\quad\quad \textbf{foreach } i' \leq n' \textbf{ do } \mathsf{OXm}(m : Z) := \\ \quad\quad\quad\quad \textbf{return}(\mathsf{exp}(\mathsf{g}, \mathsf{mult}(x, m)))). \end{array} \quad (10)$$

which can be applied only when X is used as argument of exp.

Appendix C. Proofs of Lemmas 1 and 2

Proof of Lemma 1: For Property 1, if $C[G']$ does not execute e and D returns false, then $C[G]$ behaves like $C[G']$ since $C[G]$ and $C[G']$ differ only when e is executed, so D also returns false on the execution of $C[G]$. Hence $\Pr[C[G'] : \neg(D \vee e)] \leq \Pr[C[G] : \neg D]$. Property 1 follows.

For Property 2, if $C[G]$ satisfies $D = (D_0 \wedge \neg\mathsf{NonUnique}) \vee e_1 \vee \ldots \vee e_n$, then $C[G]$ does not execute NonUnique: this is clear by definition when $C[G]$ satisfies $D_0 \wedge \neg\mathsf{NonUnique}$; when it satisfies $e_1 \vee \ldots \vee e_n$, $C[G]$ also does not execute NonUnique because one aborts immediately after the events $e_1, \ldots, e_n$ and after NonUnique, so these events are pairwise incompatible. Hence, G' behaves as G when $C[G]$ satisfies D. Hence $\Pr[C[G] : D] \leq \Pr[C[G'] : D]$.

Property 3 is an immediate consequence of Definition 1. Property 4 is obvious. ∎

Proof of Lemma 2: Property 1: By Lemma 1 (Property 1), $\Pr[C[G \mid R_x] : \mathsf{S} \vee D] \leq \Pr[C[G' \mid R_x] : \mathsf{S} \vee D \vee e]$. Moreover, if $C[G' \mid R_x]$ executes $\overline{\mathsf{S}}$ or NonUnique, $C[G' \mid R_x]$ does not execute e (since we abort immediately after $\overline{\mathsf{S}}$, NonUnique, and e), so $C[G \mid R_x]$ behaves like $C[G' \mid R_x]$, thus $C[G \mid R_x]$ also executes $\overline{\mathsf{S}}$ or NonUnique. Therefore, $\Pr[C[G' \mid R_x] : \overline{\mathsf{S}} \vee \mathsf{NonUnique}] \leq \Pr[C[G \mid R_x] : \overline{\mathsf{S}} \vee \mathsf{NonUnique}]$. Property 1 follows.

Property 2: We have

$$\begin{aligned}&\Pr[C[G \mid R_x] : \neg(\overline{\mathsf{S}} \vee \mathsf{NonUnique})]\\ &\quad \leq \Pr[C[G' \mid R_x] : \neg(\overline{\mathsf{S}} \vee \mathsf{NonUnique})]\end{aligned}$$

since, when G does not execute $\overline{\mathsf{S}}$ nor NonUnique, a fortiori, it does not execute NonUnique, so G' behaves as G. Therefore,

$$\begin{aligned}&-\Pr[C[G \mid R_x] : \overline{\mathsf{S}} \vee \mathsf{NonUnique}]\\ &\quad \leq -\Pr[C[G' \mid R_x] : \overline{\mathsf{S}} \vee \mathsf{NonUnique}]\end{aligned}$$

Moreover,

$$\Pr[C[G \mid R_x] : \mathsf{S} \vee D] \leq \Pr[C[G' \mid R_x] : \mathsf{S} \vee D]$$

since, when G executes S or an event in D, it does not execute NonUnique (because one aborts immediately after S, the events in D, and NonUnique), so G' behaves as G. Property 2 follows.

Property 3: Let $t = \max(t_{\mathsf{S} \vee D}, t_{\overline{\mathsf{S}} \vee \mathsf{NonUnique}})$. By definition of indistinguishability,

$$\begin{aligned}&\Pr[C[G \mid R_x] : \mathsf{S} \vee D]\\ &\quad \leq p(C[[\,] \mid R_x], t) + \Pr[C[G' \mid R_x] : \mathsf{S} \vee D]\end{aligned}$$

and

$$\begin{aligned}&\Pr[C[G' \mid R_x] : \overline{\mathsf{S}} \vee \mathsf{NonUnique}]\\ &\quad \leq p(C[[\,] \mid R_x], t) + \Pr[C[G \mid R_x] : \overline{\mathsf{S}} \vee \mathsf{NonUnique}]\end{aligned}$$

So

$$\mathsf{Adv}_G^{\mathsf{Secrecy}}(C, D) \leq 2p(C[[\,] \mid R_x], t) + \mathsf{Adv}_{G'}^{\mathsf{Secrecy}}(C, D)$$

Property 4 is obvious.

Property 5: When CryptoVerif proves the secrecy of x in game G, it shows that only a certain set of variables depends on value of x, but the output messages and the control-flow do not depend on x. Hence an execution of $C[G \mid R_x]$ that calls oracle O' (defined in Definition 2) in which $b = 1$ sends the same messages and has the same value of b', as the executions with the same random choices except that $b = 0$, $y[i]$ has the value of $x[u[i]]$, and x takes any value. The execution with $b = 1$ executes S if and only if the executions with $b = 0$ execute $\overline{\mathsf{S}}$.

Therefore, $\Pr[C[G \mid R_x] : \mathsf{S}] = \Pr[C[G \mid R_x] : \overline{\mathsf{S}}]$. Since we abort immediately after each event S, $e_1, \ldots, e_n$, S is incompatible with $e_1, \ldots, e_n$, so

$$\begin{aligned}&\Pr[C[G \mid R_x] : \mathsf{S} \vee D]\\ &\quad = \Pr[C[G \mid R_x] : \mathsf{S}] + \Pr[C[G \mid R_x] : D]\end{aligned}$$

Hence

$$\begin{aligned}&\mathsf{Adv}_G^{\mathsf{Secrecy}}(C, D)\\ &\quad = \Pr[C[G \mid R_x] : \mathsf{S}] + \Pr[C[G \mid R_x] : D]\\ &\qquad - \Pr[C[G \mid R_x] : \overline{\mathsf{S}} \vee \mathsf{NonUnique}]\\ &\quad \leq \Pr[C[G \mid R_x] : D]\end{aligned}$$

∎

Appendix D.
Improved Computation of Probabilities for the Manual Proof of OEKE

To illustrate the use of our improved computation of probabilities of Section IV-A, we apply it to the manual proof of OEKE [55]. We just recall the structure of the proof and the computation of probabilities, and refer the reader to [55] for details of the proof. Let us consider the proof of semantic security [55, Section 3.2]. The proof starts from a game G_0 that represents the OEKE protocol, in which we define a test-query that returns either the session key or a random value, depending on the value of a bit b, an event S executed when the adversary guesses b correctly, and an event $\overline{\mathsf{S}}$ executed when the adversary guesses the wrong value of b. The probability that the adversary C guesses b correctly in G_0 is $\Pr[C[G_0] : \mathsf{S}]$, the probability that it guesses the wrong value of b is $\Pr[C[G_0] : \overline{\mathsf{S}}]$ and the advantage of the adversary C in distinguishing the session key from a random key is $\mathsf{Adv}_{\mathsf{oeke}}^{\mathsf{ake}}(C) = \Pr[C[G_0] : \mathsf{S}] - \Pr[C[G_0] : \overline{\mathsf{S}}]$.

The proof then proceeds as follows. The game G_0 is transformed into games G_1, G_2, G_3, by eliminating collisions, such that

$$G_0 \approx_{\frac{q_{\mathcal{E}}^2}{2(q-1)}} G_1 \approx_{\frac{q_S q_{\mathcal{E}}}{q-1}} G_2 \approx_{\frac{2q_{\mathcal{E}}^2 + q_S^2}{2(q-1)} + \frac{q_h^2}{2^{l_1+1}}} G_3$$

where q_S is the number of involved server instances, $q_{\mathcal{E}}$ is the number of encryption/decryption queries, q_h is the number of hash queries, l_1 is the length of the output of $\mathcal{H}_1$, q is the order of $\mathbb{G}$.

Then G_3 is transformed into G_4 by inserting event Encrypt. Game G_4 is transformed into G_5 by excluding traces in which a correct authenticator is guessed, so that $G_4 \approx_{\frac{N_S}{2^{l_1}}} G_5$ where N_S is the number of sessions of the server S interacting with the adversary. Game G_5 is transformed into G_6 by inserting event Auth′, and G_6 is transformed into G_7 by inserting event AskH. Finally, one evaluates the probability of the various events in game G_7:

$$\Pr[C[G_7] : \mathsf{Encrypt}] \leq \frac{N_U}{N}$$

$$\Pr[C[G_7] : \mathsf{Auth}'] \leq \frac{N_S}{N}$$
$$\Pr[C[G_7] : \mathsf{AskH}] \leq q_h \mathsf{Succ}^{\mathsf{cdh}}_{\mathbb{G}}(t')$$
$$\Pr[C[G_7] : \mathsf{S}] = \Pr[C[G_7] : \overline{\mathsf{S}}]$$

where the password in chosen in a dictionary of size N, N_U is the number of sessions of the client U interacting with the adversary, N_S is the number of sessions of the server S interacting with the adversary, N_P is the number of sessions between the client U and server S that the adversary passively eavesdrops, $t' \leq t_C + (N_U + N_S + N_P + q_e + 1)\tau_{\mathbb{G}}$, with q_e denoting the number encryption/decryption queries asked by the adversary and $\tau_{\mathbb{G}}$ denoting the computation time for an exponentiation in $\mathbb{G}$.

From this proof, we can bound the advantage $\mathsf{Adv}^{\mathsf{ake}}_{\mathsf{oeke}}(C)$ in G_0. Let $\mathsf{Adv}^{\mathsf{ake}}_{G}(C, D) = \Pr[C[G] : \mathsf{S} \vee D] - \Pr[C[G] : \overline{\mathsf{S}}]$ as in Lemma 2. (Here, G already includes the test queries, so we need not compose with R_x in parallel; the event NonUnique never occurs, so we omit it.) By Lemma 2,

$$\begin{aligned}
&\mathsf{Adv}^{\mathsf{ake}}_{\mathsf{oeke}}(C) = \mathsf{Adv}^{\mathsf{ake}}_{G_0}(C, \mathsf{false}) \\
&\quad \leq 2p_{coll0} + \mathsf{Adv}^{\mathsf{ake}}_{G_3}(C, \mathsf{false}) \qquad \text{(Lemma 2, Point 3)} \\
&\text{where } p_{coll0} = \frac{q_{\mathcal{E}}^2}{2(q-1)} + \frac{q_S q_{\mathcal{E}}}{q-1} + \frac{2q_{\mathcal{E}}^2 + q_S^2}{2(q-1)} + \frac{q_h^2}{2^{l_1+1}} \\
&\mathsf{Adv}^{\mathsf{ake}}_{\mathsf{oeke}}(C) \\
&\quad \leq 2p_{coll0} + \mathsf{Adv}^{\mathsf{ake}}_{G_4}(C, \mathsf{Encrypt}) \qquad \text{(Point 1)} \\
&\quad \leq 2p_{coll0} + \frac{2N_S}{2^{l_1}} + \mathsf{Adv}^{\mathsf{ake}}_{G_5}(C, \mathsf{Encrypt}) \qquad \text{(Point 3)} \\
&\quad \leq 2p_{coll0} + \frac{2N_S}{2^{l_1}} + \mathsf{Adv}^{\mathsf{ake}}_{G_6}(C, \mathsf{Encrypt} \vee \mathsf{Auth}') \\
&\qquad\qquad \text{(Point 1)} \\
&\quad \leq 2p_{coll0} + \frac{2N_S}{2^{l_1}} + \mathsf{Adv}^{\mathsf{ake}}_{G_7}(C, \mathsf{Encrypt} \vee \mathsf{Auth}' \vee \mathsf{AskH}) \\
&\qquad\qquad \text{(Point 1)} \\
&\quad \leq 2p_{coll0} + \frac{2N_S}{2^{l_1}} + \Pr[C[G_7] : \mathsf{Encrypt} \vee \mathsf{Auth}' \vee \mathsf{AskH}] \\
&\qquad\qquad \text{(Point 5)} \\
&\quad \leq 2p_{coll0} + \frac{2N_S}{2^{l_1}} + \frac{N_U}{N} + \frac{N_S}{N} + q_h \mathsf{Succ}^{\mathsf{cdh}}_{\mathbb{G}}(t')
\end{aligned}$$

Moreover,

$$\begin{aligned}
p_{coll0} &= \frac{q_{\mathcal{E}}^2 + 2q_S q_{\mathcal{E}} + 2q_{\mathcal{E}}^2 + q_S^2}{2(q-1)} + \frac{q_h^2}{2^{l_1+1}} \\
&\leq \frac{(2q_{\mathcal{E}} + q_S)^2}{2(q-1)} + \frac{q_h^2}{2^{l_1+1}} \\
&\leq \frac{(2q_e + 2N_U + 3N_S + 3N_P)^2}{2(q-1)} + \frac{q_h^2}{2^{l_1+1}}
\end{aligned}$$

since $q_{\mathcal{E}} \leq q_e + N_U + N_S + N_p$ and $q_S \leq N_S + N_P$. So

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{ake}}_{\mathsf{oeke}}(C) \leq{}& \frac{N_U + N_S}{N} + q_h \mathsf{Succ}^{\mathsf{cdh}}_{\mathbb{G}}(t') + \\
&\frac{(2q_e + 2N_U + 3N_S + 3N_P)^2}{q-1} + \frac{q_h^2 + 2N_S}{2^{l_1}}
\end{aligned}$$

Similarly, for unilateral authentication [55, Section 3.3], we use an event Auth executed when the adversary submits an authenticator accepted by the server and built by the adversary itself, so the probability for an adversary C to make a server instance accept with no terminating client partner is

$$\mathsf{Adv}^{\mathsf{c-auth}}_{\mathsf{oeke}}(C) = \Pr[C[G_0] : \mathsf{Auth}]$$

We obtain similarly by Lemma 1

$$\begin{aligned}
\Pr[C[G_0] : \mathsf{Auth}] \leq{}& p_{coll0} + \frac{N_S}{2^{l_1}} + \\
&\Pr[C[G_7] : \mathsf{Auth} \vee \mathsf{Encrypt} \vee \mathsf{Auth}' \vee \mathsf{AskH}]
\end{aligned}$$

Since G_7 never executes event Auth,

$$\Pr\left[C[G_7] : \begin{array}{l}\mathsf{Auth} \vee \mathsf{Encrypt} \\ \vee\, \mathsf{Auth}' \vee \mathsf{AskH}\end{array}\right] \leq \frac{N_U}{N} + \frac{N_S}{N} + q_h \mathsf{Succ}^{\mathsf{cdh}}_{\mathbb{G}}(t')$$

so

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{c-auth}}_{\mathsf{oeke}}(C) \leq{}& \frac{N_U + N_S}{N} + q_h \mathsf{Succ}^{\mathsf{cdh}}_{\mathbb{G}}(t') + \\
&\frac{(2q_e + 2N_U + 3N_S + 3N_P)^2}{2(q-1)} + \frac{q_h^2 + 2N_S}{2^{l_1+1}}
\end{aligned}$$

APPENDIX E. NEW GAME TRANSFORMATIONS

The CryptoVerif proof of OEKE requires new game transformations that we have implemented. We first describe these games transformations, then summarize the proof itself. For a better understanding, we recommend reading Appendix A before reading this appendix.

A. The transformation `move array`

The transformation `move array` X delays the generation of a random value X until the point at which it is first used. This transformation is implemented as a particular case of a cryptographic transformation by the following equivalence:

foreach $i \leq n$ **do** $X \stackrel{R}{\leftarrow} T$;
  (**foreach** $iX \leq nX$ **do** $\mathsf{OX}() :=$ **return**(X) |
   foreach $ieq \leq neq$ **do** $\mathsf{Oeq}(X' : T) :=$
     return$(X' = X)$)
$\approx_{\#\mathsf{Oeq}/|T|}$ $[manual]$
foreach $i \leq n$ **do**
  (**foreach** $iX \leq nX$ **do** $\mathsf{OX}() :=$
    find[**unique**] $j \leq nX$ **suchthat defined**$(Y[j])$
    then return$(Y[j])$ **else** $Y \stackrel{R}{\leftarrow} T$; **return**$(Y)$ |
   foreach $ieq \leq neq$ **do** $\mathsf{Oeq}(X' : T) :=$
    find[**unique**] $j \leq nX$ **suchthat defined**$(Y[j])$
    then return$(X' = Y[j])$ **else return**(false))

where T is the type of X. Two oracles are defined, OX and Oeq. In the left-hand side, OX returns the random X itself. In the right-hand side, OX uses a lookup to test if the random value was already generated; if yes, it returns the previously generated random value $Y[j]$; if no, it generates a

fresh random value Y. Transforming the left-hand side into the right-hand side therefore moves the generation of the random number X to the first call to OX, that is, the first usage of X. The oracle Oeq provides an optimized treatment of equality tests $X' = X$: when the random value X was not already generated, we return false instead of generating a fresh X, so we exclude the case that X' is equal to a fresh X. This case has probability $1/|T|$ for each call to Oeq, so the probability of distinguishing the two games is $\#\mathsf{Oeq}/|T|$. (Notice that, similarly to the reasoning done in Section III-B for the Random Oracle Model, there never exist several choices of j that satisfy the conditions of the **find**s in the right-hand side of this equivalence, so these **find**s can be marked [**unique**] without modifying their behavior.)

B. Extensions of simplification

We have also extended simplification with the following transformations:

1) If some **then** branches of a **find**[**unique**] execute the same code as the **else** branch (up to renaming of variables defined in these branches and that do not have array accesses), and the variables bound in the condition of these **then** branches have no array accesses, then we remove these **then** branches.
   Indeed, these **then** branches have the same effect as the **else** branch. The hypotheses are needed for the following reasons:
   - The renamed variables must not have array accesses because renaming variables that have array accesses requires transforming these array accesses. The transformation `merge_arrays` presented in Section E-C below can rename variables with array accesses.
   - The variables bound in conditions of the removed branches must not have array accesses, because removing the definitions of these variables would modify the behavior of the array accesses.
2) If all branches of **if** or **find** execute the same code (up to renaming of variables defined in these branches and that do not have array accesses), and the variables bound in the conditions of the **then** branches have no array accesses, then we replace that **if** or **find** with its **else** branch.
   In this transformation, we ignore the array accesses that occur in the conditions of the **find** under consideration, since these conditions will disappear after the transformation.
3) If one of the **then** branches of a **find**[**unique**] always succeeds, we keep only that branch.
   Indeed, the other branches are never taken: the **find** aborts when there are several choices.
4) We reorganize a **find**[**unique**] that occurs in a **then** branch of a **find**[**unique**]: we transform

$$\textbf{find}[\textbf{unique}]\ (\bigoplus_{j=1}^{k} \tilde{u}_j \leq \tilde{n}_j\ \textbf{suchthat}\ c_j\ \textbf{then}\ P_j)\ \textbf{else}\ P$$

   where $P_{j_0} = \textbf{find}[\textbf{unique}]\ (\bigoplus_{j'=1}^{k'} \tilde{u}'_{j'} \leq \tilde{n}'_{j'}\ \textbf{suchthat}\ c'_{j'}\ \textbf{then}\ P'_{j'})\ \textbf{else}\ P'$ into

$$\textbf{find}[\textbf{unique}]\ (\bigoplus_{j=1..k, j\neq j_0} \tilde{u}_j \leq \tilde{n}_j\ \textbf{suchthat}\ c_j\ \textbf{then}\ P_j)$$
$$\oplus\ (\bigoplus_{j'=1}^{k'} \tilde{u}_{j_0} \leq \tilde{n}_{j_0}, \tilde{u}'_{j'} \leq \tilde{n}'_{j'}\ \textbf{suchthat}\ c_{j_0} \wedge c'_{j'}\ \textbf{then}\ P'_{j'})$$
$$\textbf{else find}[\textbf{unique}]\ \tilde{u}_{j_0} \leq \tilde{n}_{j_0}\ \textbf{suchthat}\ c_{j_0}\ \textbf{then}\ P'\ \textbf{else}\ P$$

   We advise renaming the variables $\tilde{u}_{j_0}$ to distinct names, since they now have multiple definitions. This transformation cannot be performed when the **find**s are not unique because it might change the probability of taking each branch.
5) We reorganize a **find**[**unique**] that occurs in a condition of a **find**: we transform

$$\textbf{find}[\mathit{unique?}]\ (\bigoplus_{j=1}^{k} \tilde{u}_j \leq \tilde{n}_j\ \textbf{suchthat}\ c_j\ \textbf{then}\ P_j)\ \textbf{else}\ P$$

   where $c_{j_0} = \textbf{defined}(\tilde{M}') \wedge \textbf{find}[\textbf{unique}]\ (\bigoplus_{j'=1}^{k'} \tilde{u}'_{j'} \leq \tilde{n}'_{j'}\ \textbf{suchthat}\ c'_{j'}\ \textbf{then}\ M'_{j'})\ \textbf{else}\ \mathsf{false}$ into

$$\textbf{find}[\mathit{unique?}]\ (\bigoplus_{j=1..k, j\neq j_0} \tilde{u}_j \leq \tilde{n}_j\ \textbf{suchthat}\ c_j\ \textbf{then}\ P_j)$$
$$\oplus\ (\bigoplus_{j'=1}^{k'} \tilde{u}_{j_0} \leq \tilde{n}_{j_0}, \tilde{u}'_{j'} \leq \tilde{n}'_{j'}\ \textbf{suchthat}\ \textbf{defined}(\tilde{M}') \wedge c'_{j'} \wedge M'_{j'}\ \textbf{then}\ P_{j_0})$$
$$\textbf{else}\ P$$

   The indication $[\mathit{unique?}]$ corresponds to either [**unique**] or empty. The **find** is marked [**unique**] after transformation if the outer **find** was [**unique**] before transformation.

For all these transformations, the correctness proof shows that, when the initial game does not execute event NonUnique, the transformed game behaves in the same way as the initial game. We can then apply point 2 of Lemmas 1 and 2 to bound the probability of attack.

C. The transformation `merge_arrays`

The transformation `merge_arrays` $x_{11}\ \ldots\ x_{1n}$, ..., $x_{m1}\ \ldots\ x_{mn}$ merges the variables $x_{j1}, \ldots, x_{jn}$ into a single variable x_{j1} for each $j \leq m$. Each variable x_{jk} must have a single definition. For each $j \leq n$, the variables $x_{j1}, \ldots, x_{jn}$

must have the same type and indices of the same type. They must not be defined for the same value of their indices (that is, x_{jk} and $x_{jk'}$ must be defined in different branches of **if** or **find** when $k \neq k'$). The arrays $x_{j1}, \ldots, x_{jn}$ are merged into a single array x_{j1} for each $j \leq m$. The transformation proceeds as follows:

- If, for each $k \leq n$, x_{1k} is defined above x_{jk} for all $1 < j < m$, we introduce a fresh variable b_k defined by $b_k \leftarrow \mathsf{mark}$ just after the definition of x_{1k}. We call b_k a *branch variable*; it is used to detect that x_{jk} has been defined: $x_{jk}[\tilde{M}]$ is defined before the transformation if and only if $x_{j1}[\tilde{M}]$ and $b_k[\tilde{M}]$ are defined after the transformation, and $x_{j1}[\tilde{M}]$ after the transformation is equal to $x_{jk}[\tilde{M}]$ before the transformation.
- For each **find** that requires that some variables x_{jk} are defined, we leave the branches that do not require the definition of x_{jk} unchanged and we try to transform the other branches $FB_l = \tilde{u}_l \leq \tilde{n}_l$ **suchthat defined**$(\tilde{M}_l) \wedge M_l$ **then** P_l as follows.
  1) We require that, for each l, there exists a distinct k such that the **defined** condition of FB_l refers to x_{jk} for some j but not to $x_{jk'}$ for any other k'. (Otherwise, the transformation fails.) We denote by $l(k)$ the value of l that corresponds to k.
  2) We choose a "target" branch $FB_{\mathrm{T}} = \tilde{u} \leq \tilde{n}$ **suchthat defined**$(\tilde{M}) \wedge M$ **then** P: if the **defined** condition of some branch FB_l refers to x_{j1} for some j, we choose that branch FB_l. Otherwise, we choose any branch FB_l and rename its variables x_{jk} to x_{j1}. We require that the references $x_{j1}[\tilde{M}]$ to the variables x_{j1} in the **defined** condition of the target branch all have the same indices $\tilde{M}$. If the transformation succeeds, we will replace all branches FB_l with the target branch.
  3) The branch FB_{T} after transformation is equivalent to branches $\bigoplus_{k=1}^{n} FB_{\mathrm{T}}\{x_{jk}/x_{j1}, j = 1...m\}$ before transformation. We show that these branches are equivalent to the branches FB_l.
     For each $k \leq n$,
     - if $l(k)$ exists, then we show that $FB_{\mathrm{T}}\{x_{jk}/x_{j1}, j = 1...m\}$ is equivalent to $FB_{l(k)}$. Let $l = l(k)$. We first rename the variables $\tilde{u}_l$ of FB_l to the variables $\tilde{u}$ of the target branch. For simplicity, we still denote by $FB_l = \tilde{u}_l \leq \tilde{n}_l$ **suchthat defined**$(\tilde{M}_l) \wedge M_l$ **then** P_l the obtained branch. Then we show that, if the variables of $\tilde{M}_l$ are defined, then the variables of $\tilde{M}\{x_{jk}/x_{j1}, j = 1...m\}$ are defined, and conversely; $M_l = M\{x_{jk}/x_{j1}, j = 1...m\}$ (knowing the equalities that hold at that program point), and P_l and $P\{x_{jk}/x_{j1}, j = 1...m\}$ execute the same code up to renaming of variables defined in P_l or $P\{x_{jk}/x_{j1}, j = 1...m\}$ and that do not have array accesses.
     - if $l(k)$ does not exist, then we show that $FB_{\mathrm{T}}\{x_{jk}/x_{j1}, j = 1...m\}$ can in fact not be executed, because its condition cannot hold: the variables of $\tilde{M}\{x_{jk}/x_{j1}, j = 1...m\}$ cannot be simultaneously defined or $M\{x_{jk}/x_{j1}, j = 1...m\}$ cannot hold.

     If the transformation above fails and we have introduced branch variables, we replace each condition **defined**$(x_{jk}[\tilde{M}])$ with **defined**$(x_{j1}[\tilde{M}], b_k[\tilde{M}])$.
     If the transformation above fails and we have not introduced branch variables, the whole `merge_arrays` transformation fails.
- The definition of x_{jk} is renamed to x_{j1} and each reference to $x_{jk}[\tilde{M}]$ is renamed to $x_{j1}[\tilde{M}]$.

D. The transformation `merge_branches`

The transformation `merge_branches` extends again the first two extensions of simplification mentioned in Appendix E-B. Instead of applying these transformations to a single **find** at a time, `merge_branches` applies them globally to all **find**s of the game for which the simplification is possible. As a consequence, one can ignore array accesses to all variables in conditions of **find** that will be removed, so more transformations are enabled.

APPENDIX F.
THE PROOF IN CRYPTOVERIF

A. Initial configuration

First, we configure CryptoVerif to iterate the simplification of games at most 3 times, by `set maxIterSimplif = 3`, instead of at most twice by default. The complexity of the intermediate games of OEKE requires more simplifications than in many other examples. One can also iterate simplification until a fixpoint is reached, which is slightly slower but works for all examples.

B. Events Auth and Encrypt

As explained in Section IV-A, we manually introduce events Auth and Encrypt and distinguish cases in hash functions.

C. Automatic Steps

Then, we can use the automatic proof strategy of CryptoVerif, by command `auto`. Basically, this strategy consists in applying all possible cryptographic transformations (coming from equivalences $L \approx_p R$) and simplifying the game after each such transformation. When the transformations fail, they advise syntactic transformations that could make them succeed; these transformations are executed and the

cryptographic transformation is then retried [49, Section 5]. This automatic strategy performs the following transformations:

1) It renames each occurrence of K_u to a distinct name. Indeed, as part of `crypto` icm(enc), the definition of K_u has been copied once for each possible origin of Y_u (a previous encryption/decryption query, or a fresh Y_u if no previous query matches). After this renaming, simplification can simplify many tests of the form $K\_u = K\_s$ that appeared as a result of the transformation of the hash functions; it uses the values of K_u and K_s as well as properties (4), (6), and (7).
2) Using equivalence (10) twice, it replaces the generation of a fresh group element X with the generation of an exponent x and the computation $X \leftarrow \mathsf{exp}(\mathsf{g}, x)$ as a result of decryption when no previous encryption/decryption query matches, in the decryption oracle and in the client.
3) It removes assignments on the copies of K_u created in step 1 above, on K_s (the key of the server), and Kp (the key used in passive eavesdroppings), thus replacing these variables with their values everywhere in the game.
4) It can then apply the CDH assumption (Section III-D). The oracles Oa and Ob are in fact not used in this example (the code that would use them has been removed by introducing events), so the situation is particularly simple: expressions of the form $m = \mathsf{exp}(\mathsf{g}, \mathsf{mult}(a[i], b[j]))$ are replaced with false.
5) At this point, CryptoVerif can bound the probability of breaking all desired properties (secrecy of sk_u, correspondences (1) and (2)). However, the obtained bound depends on the probability of executing the events Auth and Encrypt which are not eliminated yet, so the proof continues in order to eliminate these events.
6) Using the version of equivalence (8) for exp′, we replace the computation of $\mathsf{exp}'(\mathsf{g}, x)$ for a fresh random x with the generation of a random group element X, for the result of the decryption oracle and for the computations of Yp and Xp (the values of Y and X in passive eavesdroppings), Y (in the server), Y_u and X (in the client).

After these transformations, no automatic step can be performed, so the automatic proof stops.

D. Reorganizing random number generations

Using manually guided transformations, we can eliminate the random number generations for Y. We consider the three generations of Y in turn.

- First, Y_u (the value of Y in the client), when it is a fresh random group element. This variable is now named $@6\_X\_416$, so we use the command `move array` "$@6\_X\_416$" to delay its generation. The game is automatically simplified after this command, which reorganizes **find** constructs. Before `move array`, $@6\_X\_416$ is used in the following ways:
  1) to compare it with the argument of encryption queries in the encryption oracle. This usage is transformed using oracle Oeq of the transformation `move array`; no random value is generated.
  2) as a result of decryption queries in the decryption oracle. This usage is transformed using oracle OX; the generation of $@6\_X\_416$ in the client is replaced with a generation of $@2\_Y\_418$ in the decryption oracle.
  3) as a result of the decryption query $\mathsf{dec}(Y^\star, pw)$ in the client. That result is unused, so $@6\_X\_416$ appears only in a **defined** condition, which CryptoVerif leaves unchanged.

  Therefore, in the decryption oracle, when the result was not generated before, we have two cases: either the result was in fact an Y_u whose generation has been delayed, and it is now generated as $@2\_Y\_418$, or the query is really a fresh decryption query, and the result is named $@6\_X\_412$. We would like to merge these two cases. However, `merge_branches` does not succeed directly because these variables have array accesses, so we first apply `merge_arrays` "$@6\_X\_412$" "$@2\_Y\_418$" to merge $@2\_Y\_418$ into $@6\_X\_412$. Furthermore, in some branches that we would like to merge, some random number generations are not ordered in the same way: in the client, the authenticator named $@11\_r\_130$ is generated sometimes before and sometimes after the shared key sk_u. We need to make sure that they are in the same order in all branches for `merge_branches` to succeed. This is done by the transformation `move binder` $@11\_r\_130$, which moves definitions of $@11\_r\_130$ as much as possible downwards in the game. Then, we apply `merge_branches` successfully.
- Second, the value of Y in passive eavesdroppings, Yp. This variable is now named $@6\_X\_413$. We proceed similarly, using `move array` "$@6\_X\_413$", then merge the delayed $@6\_X\_413$ named $@2\_Y\_425$ into the result of decryption $@6\_X\_412$ by `merge_arrays` "$@6\_X\_412$" "$@2\_Y\_425$", and finally apply `merge_branches`.
- Third, the value of Y in the server, now named $@6\_X\_415$. We again proceed similarly, using `move array` "$@6\_X\_415$", then merge the delayed $@6\_X\_415$ named $@2\_Y\_432$ into the result of decryption "$@6\_X\_412$" by

`merge_arrays` "$@6\_X\_412$" "$@2\_Y\_432$". We cannot apply `merge_branches` directly because the condition that triggers the event Auth,

$$\begin{aligned}&\textbf{find } @i\_435 \leq qD, jh \leq qH1 \textbf{ suchthat}\\&\textbf{defined}(@i\_437[@i\_435], x1[jh], x2[jh], x3[jh],\\&\quad x4[jh], @6\_X\_412[@i\_435], @11\_r\_134[jh]) \wedge\\&@i\_437[@i\_435] = iS \wedge \mathsf{U} = x1[jh] \wedge\\&\mathsf{S} = x2[jh] \wedge X\_s = x3[jh] \wedge\\&x4[jh] = @6\_X\_412[@i\_435] \wedge\\&auth\_s = @11\_r\_134[jh] \textbf{ then}\end{aligned} \tag{11}$$

refers to the variable $@i\_437$ which is defined in the condition of a **find** whose branches we would like to merge; this merging is not possible because it would make the definition of $@i\_437$ disappear. So we manually rewrite the condition (11) to remove the reference to $@i\_437$. By

$$\begin{aligned}&\texttt{insert 121 "}\textbf{find } jh' \leq qH1, jd \leq qD\\&\textbf{suchthat defined}(x1[jh'], x2[jh'], x3[jh'],\\&x4[jh'], @11\_r\_134[jh'], m[jd], kd[jd],\\&@6\_X\_412[jd]) \wedge m[jd] = @8\_re\_161 \wedge\\&\mathsf{U} = x1[jh']) \wedge \mathsf{S} = x2[jh']) \wedge X\_s = x3[jh'] \wedge\\&x4[jh'] = @6\_X\_412[jd] \wedge\\&auth\_s = @11\_r\_134[jh'] \wedge kd[jd] = pw \textbf{ then}"\end{aligned} \tag{12}$$

we insert a test just above the **find** (11). The variables jh and $@i\_435$ of the **find** (11) are renamed to jh' and jd respectively in (12) (without change in the meaning), and the condition $@i\_437[@i\_435] = iS$ of (11), that is, $@i\_437[jd] = iS$ is replaced with $m[jd] = @8\_re\_161 \wedge kd[jd] = pw$ in (12). In the considered game, the variable $@i\_437$ is defined in a **find** with condition $m = @8\_re\_161[@i\_437] \wedge kd = pw$, so when $@i\_437[jd]$ is defined, we have $m[jd] = @8\_re\_161[@i\_437[jd]] \wedge kd[jd] = pw$; when $@i\_437[jd] = iS$, we obtain exactly the condition of (12), knowing that the index iS can be omitted because it is the index of the replication above the **find** (12). Explained another way, the **find** of (12) looks for a hash query indexed by jh' and a decryption query indexed by jd, such that the adversary has decrypted the value of $Y^\star$ generated by the server, $@8\_re\_161$, under the correct password pw, obtaining $@6\_X\_412[jd]$ (which is then the correct value of Y) using that decryption query, and then has passed a value $(\mathsf{U}, \mathsf{S}, X\_s, @6\_X\_412[jd], \_)$ to the hash query, obtaining the correct authenticator $auth\_s = @11\_r\_134[jh']$. This corresponds exactly to the situation in which the adversary authenticates to the server by guessing the password. After the command (12), by `insert_event` Auth2 184, we insert the event Auth2 in the **then** branch of the **find** of (12). CryptoVerif can show that the condition (11) implies the condition of (12), so in the **else** branch of (12), the condition (11) never holds, hence by simplification (`simplify`), we remove that **else** branch. Simplification also merges the cases of **find**, without an explicit `merge_branches`. At this point, event Auth does not occur in the current game. The events Encrypt and Auth2 are eliminated and their probabilities are bounded as explained in Section IV-D. (The variables $@8\_re\_161$, $@8\_re\_167$, $@11\_r\_134$, and $@6\_X\_412$ have been renamed to Y_star, re, $hash_1$, and rd respectively there.)