Dealing with Dynamic Key Compromise in CryptoVerif

Bruno Blanchet

Dealing with Dynamic Key Compromise in CryptoVerif

Bruno Blanchet
Inria
Paris, France
bruno.blanchet@inria.fr

***Abstract*—CryptoVerif is a mechanized security protocol verifier sound in the computational model. In this paper, we explore and extend its treatment of dynamic key compromise. First, we present a basic treatment of compromise and explain its limitations. Next, we introduce several extensions in order to remove these limitations: improved proof of secrecy; building different proofs for the various properties to prove; removing code that cannot be executed when the adversary breaks the desired security properties; and guessing tested sessions, values, and branches. We illustrate how these extensions improve the treatment of key compromise on protocols ranging from toy examples to filling gaps in previous large case studies including TLS 1.3 and the WireGuard VPN protocol.**

***Index Terms*—security protocols, computational model, mechanized proofs, key compromise**

I. Introduction

A first step in the verification of a security protocol can be to consider all principals as honest. However, this is rather weak. One generally considers that honest principals may talk to compromised principals, that is, dishonest principals, controlled by the adversary, and one aims to prove security for sessions between honest principals. This is called *static compromise* because the status of a principal (honest or compromised) does not evolve over time. For instance, static compromise is needed to discover the well-known man-in-the-middle attack against the Needham-Schroeder public key protocol [40]. Going further, one can consider *dynamic compromise*: a principal that is initially honest may get compromised at some point, for example because its machine has been successfully attacked, which gives the adversary access to the long-term secret keys stored on that machine. In this case, one typically aims to prove security for sessions executed before the compromise. *Forward secrecy* is a typical property proved in this case: the messages exchanged in a session remain secret even if the long-term keys of the principals involved in that session are later compromised.

Considering dynamic compromise obviously complicates security proofs. In this paper, we present and extend the treatment of dynamic compromise in the protocol verification tool CryptoVerif [18]. This tool relies on the computational model of cryptography [50], which is typically used in the manual proofs of cryptographers, and in which messages are bitstrings, the cryptographic primitives are functions from bitstrings to bitstrings, and the adversary is a probabilistic Turing machine. CryptoVerif produces game-based proofs [14], [49], like those of cryptographers: starting from the initial game representing the protocol to prove in interaction with an adversary, it uses game transformations to generate a sequence of games such that, in the final game, the desired security properties can be proved just by looking syntactically at the game, without using any cryptographic assumption. All game transformations of CryptoVerif are such that, if the security properties are satisfied in the transformed game, then they are also satisfied in the game before transformation. (In most cases, this is because the transformed game is computationally indistinguishable from the game before transformation. However, this is not always true, for instance in the guessing transformation that we introduce in Section V-C.) We can then prove the security properties in the initial game. The applied game transformations are determined either by an automatic proof strategy [18] or by user guidance: in interactive mode, the user gives commands to CryptoVerif to tell it which game transformation to apply. Examples of commands include **crypto** $\langle cryptographic\ assumption \rangle$, which tells CryptoVerif to apply the specified cryptographic assumption, and **simplify**, which tells CryptoVerif to simplify the current game as much as it can. CryptoVerif generates the game after the transformation specified by the given command. CryptoVerif proofs can be interpreted in two ways. First, the proved security properties hold asymptotically, that is, the adversary has a negligible probability of breaking the proved security properties, where the length of keys is determined by a security parameter, the adversary is a probabilistic Turing machine that runs in time polynomial in the security parameter, and *negligible* means asymptotically smaller than all inverses of polynomials in the security parameter. Second, CryptoVerif also provides an explicit formula that bounds the probability of breaking the security properties, as a function of the probability of breaking the cryptographic primitives. This is the *exact security* framework. To simplify the exposition, in this paper, we consider only the asymptotic framework. The details of the computation of probabilities in the exact security framework can be found in the companion technical report [22].

Our contributions deal with the treatment of dynamic key compromise in CryptoVerif. They can be summarized as follows. First, we present a basic treatment of compromise (Section II), which was already used in previous case studies but never explained in detail. Interestingly, this approach does not require any modification of the tool CryptoVerif itself: only the assumptions on primitives are modified. We also present

the limitations of this approach, and ideas on how to remove them. Next, after recalling the definitions of security properties proved by CryptoVerif (Section III), we show how CryptoVerif proves these properties, extending the proof of secrecy in a way useful for dealing with compromise (Section IV). In Section V, we present new commands that users can use to guide the proof and that allow us to remove the limitations of the basic treatment of compromise: focusing on the proof of certain properties; removing code that cannot be executed when the adversary breaks the desired security properties, which allows us to remove the compromise of keys in suitable cases; guessing values, and in particular guessing the tested session. The latter transformation is often used in cryptographic proofs, not necessarily related to compromise, but we apply it in situations related to compromise. Finally, in Section VI, we apply these new commands to several case studies. In particular, we fill gaps in two important previous case studies [16], [38] by showing forward secrecy with respect to the compromise of the pre-shared key for the PSK-DHE (pre-shared key and Diffie-Hellman) handshake of TLS 1.3 Draft 18 [46], which is close to the final version of TLS 1.3 [47], and for the VPN protocol WireGuard [32] integrated in the Linux kernel.

The new commands are implemented in CryptoVerif version 2.07 available at https://cryptoverif.inria.fr. CryptoVerif consists of 68000 lines of OCaml. Additional details on the game transformations and proofs for all results can be found in the companion technical report [22]. The CryptoVerif files for all examples and case studies are available at https://cryptoverif.inria.fr/compromise/.

Related Work: Many other protocol verification tools can deal with the dynamic compromise of keys. At the symbolic level, Scyther [31] has a specific version to deal with compromise [9], [11]. Other tools deal with compromise without specific publications on it. For instance, Tamarin [48] can specify forward secrecy for x by executing a $\mathsf{Secret}(x)$ event and requiring that the arguments of Secret events remain secret provided some principals have not been compromised before the Secret event. ProVerif [20] considers scenarios with stages [23, Section 8]: the protocol is executed in a first stage and the key is compromised in a second stage. This modeling does not catch attacks in which the compromise happens in the middle of a session. All these tools rely on the symbolic model of cryptography, in which cryptographic primitives are considered as ideal blackboxes and the adversary is restricted to apply only a fixed set of these primitives. Hence, the obtained security results are weaker than in the computational model, in which the adversary can be any probabilistic Turing machine. In particular, the symbolic model does not catch attacks that rely on weaknesses of the primitives (*e.g.* weaknesses of encryption in SSH [2]) unless these weaknesses are explicitly modeled, which may require knowing the attack in advance.

At the computational level, EasyCrypt [8] focuses more on cryptographic primitives and schemes than on protocols. It has still been used for proving some protocols such as one-round key exchange [7], e-voting [30], AWS key management [3], and distance bounding [25]. In particular, [7] proves forward secrecy for one-round key exchange protocols. The proof strategy differs from the one we use in CryptoVerif: it uses case distinctions depending on the compromise scenarios in [7, Theorem 1] instead of introducing explicit events for authentication before compromise as we do in Section V. We are confident that the frameworks for cryptographic proofs in Coq, FCF [43], and in Isabelle, CryptHOL [10], could deal with dynamic corruption in security protocols, although to our knowledge they have been used to prove cryptographic schemes (HMAC [44], searchable symmetric encryption [15] for FCF; oblivious transfer [28], sigma protocols and commitment schemes [29] for CryptHOL) but no key exchange protocol so far. Indeed, like EasyCrypt, they can perform more subtle reasoning than CryptoVerif, at the cost of more user effort: the user has to give all games and guide the proof that the games are indistinguishable. That becomes tedious for large protocols, which require many large games. In the tool Squirrel [5], the treatment of forward secrecy is work in progress (personal communication). Squirrel relies on a logic that allows it to use symbolic techniques and be computationally sound. Still, it currently proves a security notion weaker than the standard one: the number of sessions of the protocol must be bounded independently of the security parameter (instead of being polynomial in the security parameter). For instance, a protocol that leaks one bit of a key in each session is then considered not to leak the whole key, because for a fixed number of sessions, the adversary learns a fixed number of bits of the key; when the security parameter is large enough, the length of the key, which is typically proportional to the security parameter, becomes longer than that fixed number of bits. The tool OWL [33] supports static corruptions and a limited form of forward secrecy: the adversary has to commit in advance to the point in time where compromise happens [33, Section 3.4].

Pen-and-paper proofs also deal with dynamic compromise; our work mechanizes these techniques in the tool CryptoVerif. In particular, in the universal composability framework, the approach of [36], which allows dynamic corruption of a signing key inside an ideal functionality, is fairly similar to our basic treatment of compromise in CryptoVerif (Section II). Our more advanced treatment of compromise (Section V) implements in CryptoVerif a way of dealing with dynamic compromise also used in game-based proofs, *e.g.*, the one of [27, Appendix D].

II. Basic Treatment of Compromise

The simplest way to deal with the dynamic compromise of keys in CryptoVerif is to include the possibility to compromise the keys in the assumptions on the cryptographic primitives themselves. We illustrate this approach on the example of ciphertext integrity for a symmetric encryption scheme. This assumption is given in beautified CryptoVerif syntax in Figure 1, with minor simplifications (*e.g.*, we omit proof strategy indications), and explained below.

We consider an encryption scheme that defines a probabilistic encryption function. In CryptoVerif, this function is represented by a deterministic function enc_r that takes as argument the cleartext, the key, and random coins r, and

```
1  equiv(int_ctxt_corrupt(enc))
2  new k : key; (
3    !^{i≤n} Oenc(x : cleartext) :=
4      new r : enc_seed; return(enc_r(x, k, r)) |
5    !^{i'≤n'} Odec(y : ciphertext) :=
6      return(dec(y, k)) |
7    Ocorrupt() := return(k))
8  ≈
9  new k : key; (
10   !^{i≤n} Oenc(x : cleartext) :=
11     new r : enc_seed; let z : ciphertext = enc_r(x, k, r)
12     in return(z) |
13   !^{i'≤n'} Odec(y : ciphertext) :=
14     if defined(corrupt) then return(dec(y, k)) else
15     find j ≤ n suchthat defined(x[j], z[j]) ∧ z[j] = y
16     then return(injbot(x[j])) else return(bottom) |
17   Ocorrupt() := let corrupt : bool = true in return(k)).
```

Fig. 1. Ciphertext integrity with dynamic compromise of keys

returns the corresponding ciphertext. The random coins r represent the probabilistic choices made during encryption. The decryption function dec is deterministic; it takes as argument a ciphertext and a key. It returns the special symbol bottom when decryption fails, and the cleartext when decryption succeeds. The result type of dec is then bitstringbot, representing the union of bitstrings and bottom. Then we have:

$$\mathsf{dec}(\mathsf{enc\_r}(x, k, r), k) = \mathsf{injbot}(x) \tag{1}$$

where injbot is the canonical injection from the set of cleartexts cleartext to bitstringbot, which maps a cleartext to itself. This function is needed for the equality (1) to typecheck. This equality expresses that, when one decrypts a ciphertext with the correct key k, decryption succeeds and returns the cleartext x.

Ciphertext integrity means that, assuming a random key k, an adversary that has access to an encryption oracle under k and an oracle that tests whether decryption with k succeeds has a negligible probability of forging a valid ciphertext, that is, computing a ciphertext that has not been returned by the encryption oracle and that correctly decrypts. A formal definition can be found in [13].

CryptoVerif requires all assumptions on primitives to be specified as indistinguishability properties between two games: $L \approx R$, meaning that the adversary has a negligible probability of distinguishing L from R. CryptoVerif uses such assumptions by replacing L with R inside a bigger game. Therefore, the definition of Figure 1 is of this form. Line 1 just gives the name of the property: int_ctxt means "ciphertext integrity"; corrupt refers to the version of the property that supports dynamic corruption of the key. Lines 2 to 7 define the game L, lines 9 to 17 define the game R.

In each game, we first choose a random key k in the set key by **new** k : key. Then we provide 3 oracles to the adversary: the encryption oracle Oenc, the decryption oracle Odec, and the corruption oracle Ocorrupt. The encryption oracle can be called at most n times, as specified by the replication $!^{i \leq n}$. The index i is called a *replication index* and ranges over $\{1, \ldots, n\}$, which we also write $[1, n]$ for simplicity. Replication bounds such as n are polynomial in the security parameter. Similarly, the decryption oracle can be called at most n' times. The encryption oracle generates fresh coins r and encrypts the received cleartext x using these coins. In R, it additionally stores the ciphertext in variable z (line 11). In CryptoVerif, all variables defined under $!^{i \leq n}$ are implicitly arrays indexed by the replication index $i \in [1, n]$. In particular, in R, the call to Oenc with index i stores the cleartext in $x[i]$ and the ciphertext in $z[i]$. The corruption oracle returns the key k to the adversary. In R, it additionally defines a variable *corrupt* to record that the key has been corrupted. In L, the decryption oracle Odec simply decrypts using the function dec. In R, it distinguishes several cases. If the key has been corrupted, it just decrypts using dec (line 14). We cannot apply ciphertext integrity in this case. Otherwise, it looks for an index j such that $x[j]$ and $z[j]$ are defined and $z[j] = y$. This condition means the oracle Oenc has been called with index j, and that the ciphertext it returned, $z[j]$, is the same as the one given to the decryption oracle, y (**find** construct, line 15). In this case, by (1), the result of decryption is $\mathsf{injbot}(x[j])$. Finally, when no call to Oenc returned y, by ciphertext integrity, decryption must fail, hence Odec returns bottom.

In other words, the games L and R can be distinguished if and only if Odec returns bottom in its last case in R and decryption succeeds in the corresponding call to Odec in L. That corresponds exactly to breaking ciphertext integrity, hence the probability of distinguishing L from R is negligible when the encryption scheme satisfies ciphertext integrity.

Thanks to the presence of oracle Ocorrupt, this property can be applied by CryptoVerif even if the key k is corrupted at some point. Obviously, the property brings useful information only when k is not corrupted yet when we decrypt. The version that does not deal with corruption would remove oracle Ocorrupt and line 14; with this version, the key k must *never* be corrupted for the property to be applicable.

Although it was not detailed in previous papers, this assumption with corruption was used in the case study of WireGuard [38] and similar assumptions including corruption were used for one-wayness in [24] and for the unforgeability of signatures in the case studies of TLS 1.3 [16], of Signal [35], and of the fixed ARINC823 public key protocol of [21].

However, this approach has important limitations:

- This approach works for *computational* assumptions on primitives, that is, assumptions that express that an adversary cannot compute some value (when the key is not compromised). Such assumptions are exploited before the compromise of a key; that remains valid even if the key is compromised in the future. For instance, we can exploit ciphertext integrity to say that the adversary cannot forge a ciphertext before the encryption key is compromised, even if that key is compromised later. Examples of computational properties include ciphertext integrity, unforgeability for signatures [34] and MACs [13],

the computational and gap Diffie-Hellman assumptions (CDH and GDH) [41]. However, this approach does not work for *decisional* assumptions, that is, assumptions that express that the adversary cannot distinguish two games (when the key is not compromised). If the key is compromised later, the adversary becomes able to distinguish the two games in question, so we cannot replace the first game by the second one. Examples of decisional properties include IND-CPA, IND-CCA2 for encryption [12], the decisional Diffie-Hellman (DDH) assumption [41], and the pseudo-random function oracle Diffie-Hellman (PRF-ODH) assumption [26].

- This approach also does not work when the compromised "key" k is used as argument in a sequence of key derivations using hash functions, for instance modeled as random oracles. In this case, k is not directly used in key position in a primitive. We cannot replace the derived key with a random key, because the derived key becomes distinguishable from a random key if k is compromised in the future. This situation happens when k is the pre-shared key in the TLS 1.3 and WireGuard protocols: previous studies of these protocols [16], [38] were unable to prove forward secrecy with respect to the compromise of the pre-shared key in CryptoVerif.
- It does not allow proving in CryptoVerif properties such as the ciphertext integrity with compromise of keys shown in Figure 1 from assumptions that do not allow key compromise. Such proofs would have to be done manually or using another mechanized prover.

In this paper, we show how to remove all these limitations, thanks to extensions that we have implemented in CryptoVerif. Informally, the main idea will be to proceed in two steps. First, we prove an authentication property, assuming the key is not compromised until the end of the session. When a principal is authenticated, that remains true even if a key is compromised later, so, due to the nature of the authentication property, it is enough to prove it without considering the key compromise at all. As a second step, we use that property to prove other properties, including secrecy, in the presence of key compromise. We introduce new commands needed for this approach in Section V, and apply them to case studies in Section VI.

III. Security Properties

A. Preliminaries

The cryptographic games are represented by processes in a probabilistic process calculus detailed in [22, Section 2]. The semantics of this process calculus is defined by a probabilistic reduction relation on semantic configurations. A *trace* Tr of a process Q is a sequence of reductions in this relation, starting from the initial configuration associated to Q. A trace is *full* when its last configuration cannot be reduced. All processes run in polynomial-time in the security parameter, because replication bounds and the size of messages are polynomial in the security parameter and the runtime of all function symbols is polynomial in the size of their arguments.

We denote by φ a *trace property*, that is, a function from traces to $\{\mathsf{true}, \mathsf{false}\}$. We say that Tr satisfies φ, and we write $Tr \vdash \varphi$, when $\varphi(Tr) = \mathsf{true}$. We write $\Pr[Q : \varphi]$ for the total probability of all full traces of Q that satisfy φ.

The processes may execute events e with bitstring arguments $a_1, \ldots, a_m$, written $e(a_1, \ldots, a_m)$, by the instructions $\mathbf{event}\ e(M_1, \ldots, M_m)$ and $\mathbf{event\_abort}\ e$. The latter instruction executes an event e without argument and aborts the game. As shown below, events are used for defining security properties. A *distinguisher* D is a particular trace property that only depends on the sequence of executed events: $Tr \vdash D$ if and only if $\mathcal{E}v \vdash D$, that is, $D(\mathcal{E}v) = \mathsf{true}$ where $\mathcal{E}v$ is the sequence of events executed in Tr. When e is an event, we write e for the distinguisher that is true when event e is executed. For instance, $\Pr[Q : e]$ is the probability that the process Q executes event e.

The adversary against a process Q is represented by an *evaluation context* C, which is basically the adversarial process Q' in parallel with a hole $[\,]$, written $Q' \mid [\,]$ or $[\,] \mid Q'$. We write $C[Q]$ for the context C in which the hole has been replaced with Q, representing the adversarial process Q' running in parallel with Q. An evaluation context C is said to be *acceptable for Q with public variables V* when it satisfies syntactic conditions that allow the combination $C[Q]$ (defined in [22, Section 2.7, Definition 5]) and C has read access to variables defined in Q that are in the set V (via the **find** construct mentioned in Section II).

B. Secrecy

Let us define the secrecy properties proved by CryptoVerif, introduced in [18, Section 4]. Consider a variable x, an array indexed by the indices of replications above the definition of x. We define two secrecy properties for x: one-session secrecy, $\mathsf{1\text{-}ses.secr.}(x)$, intuitively means that each array cell of x is indistinguishable from a uniformly distributed random value; secrecy, $\mathsf{Secrecy}(x)$, intuitively means that the array x is indistinguishable from independent uniformly distributed random values. In particular, one-session secrecy allows several array cells of x to be equal, while secrecy does not. Secrecy corresponds to the "real-or-random" definition of security [1].

To define these notions formally, we define test processes Q_{sp}, where sp is $\mathsf{1\text{-}ses.secr.}(x)$ or $\mathsf{Secrecy}(x)$, explained below.

$$
\begin{aligned}
&Q_{\mathsf{1\text{-}ses.secr.}(x)} = \\
&\quad c_{s0}(); \mathbf{new}\ b : bool; \overline{c_{s0}}\langle\rangle; \\
&\quad (c_s(u_1 : [1, n_1], \ldots, u_m : [1, n_m]); \\
&\quad\ \mathbf{if}\ \mathbf{defined}(x[u_1, \ldots, u_m])\ \mathbf{then} \\
&\quad\ \mathbf{if}\ b\ \mathbf{then}\ \overline{c_s}\langle x[u_1, \ldots, u_m]\rangle\ \mathbf{else}\ \mathbf{new}\ y : T; \overline{c_s}\langle y\rangle \\
&\quad \mid c'_s(b' : bool); \mathbf{if}\ b = b'\ \mathbf{then}\ \mathbf{event\_abort}\ \mathsf{S}\ \mathbf{else} \\
&\quad\ \mathbf{event\_abort}\ \overline{\mathsf{S}})
\end{aligned}
$$

where the channels c_{s0}, c_s, c'_s, the events S, $\overline{\mathsf{S}}$, and the variables $u_1, \ldots, u_m, y, b, b'$ do not occur in Q nor in the public variables V, and the variable x has indices of types $[1, n_1]$, ..., $[1, n_m]$ and its value is of type T.

When the adversary sends a message on channel c_{s0}, the process $Q_{\mathsf{1\text{-}ses.secr.}(x)}$ receives it by the input $c_{s0}()$, chooses a random boolean b by $\mathbf{new}\ b : bool$, and returns control to the adversary by outputting on c_{s0} by the output $\overline{c_{s0}}\langle\rangle$. Then, it allows the adversary to perform one test query on indices $u_1, \ldots, u_m$, by sending these indices on channel c_s. When $x[u_1, \ldots, u_m]$ is not defined, this query fails; otherwise, when $b = \mathsf{true}$, it returns $x[u_1, \ldots, u_m]$ on channel c_s and when $b = \mathsf{false}$, it returns a fresh random value y on channel c_s. Finally, the process $Q_{\mathsf{1\text{-}ses.secr.}(x)}$ allows the adversary to provide a boolean b' on channel c'_s and executes event S when $b = b'$ and event $\overline{\mathsf{S}}$ otherwise. After executing S or $\overline{\mathsf{S}}$, it aborts the game. Intuitively, the goal of the adversary is to guess b, that is, find whether the test query returned the real value $x[u_1, \ldots, u_m]$ or a random value. The boolean b' is the adversary's guess for the value of b.

The process $Q_{\mathsf{Secrecy}(x)}$ is similar to $Q_{\mathsf{1\text{-}ses.secr.}(x)}$ but allows the adversary to perform several test queries that return $x[u_1, \ldots, u_m]$ when $b = \mathsf{true}$ and consistent random values when $b = \mathsf{false}$, that is, when $b = \mathsf{false}$, if the indices $u_1, \ldots, u_m$ have already been queried, then the test query returns the previous answer, else it returns a fresh random value. The CryptoVerif code for this process is given in [22, Section 2.7.1].

Definition 1 ((One-session) secrecy): Let Q be a process, x a variable, and V a set of variables. Let sp be $\mathsf{1\text{-}ses.secr.}(x)$ or $\mathsf{Secrecy}(x)$.

The process Q *satisfies* sp with public variables V ($x \notin V$) when, for all evaluation contexts C acceptable for $Q \mid Q_{sp}$ with public variables V that do not contain S nor $\overline{\mathsf{S}}$, $\Pr[C[Q \mid Q_{sp}] : \mathsf{S}] - \Pr[C[Q \mid Q_{sp}] : \overline{\mathsf{S}}]$ is negligible. ◁

Intuitively, when Q satisfies sp, the adversary has a negligible probability of guessing b, that is, of distinguishing whether the test process Q_{sp} outputs the value of the secret x ($b = \mathsf{true}$) or a random value ($b = \mathsf{false}$).

C. Correspondences

Correspondences are used to model authentication. They are properties of the form "if some events have been executed, then some other events have been executed". Here, correspondences are logical formulas of the form $\forall \widetilde{x} : \widetilde{T}; \psi \Rightarrow \exists \widetilde{y} : \widetilde{T}'; \phi$, where the variables $\widetilde{x}$ are the variables of ψ, $\widetilde{T}$ are their types, the variables $\widetilde{y}$ are the variables of ϕ that do not occur in ψ, and $\widetilde{T}'$ are their types. We use the following logical formulas ϕ:

$\phi ::=$	formula
M	term
$\mathbf{event}(e(M_1, \ldots, M_m))$	(non-injective) event
$\mathbf{inj\text{-}event}(e(M_1, \ldots, M_m))$	injective event
$\phi_1 \wedge \phi_2$	conjunction
$\phi_1 \vee \phi_2$	disjunction

Terms $M, M_1, \ldots, M_m$ in formulas are built from variables in $\widetilde{x}$ and $\widetilde{y}$ and function applications; the variables in $\widetilde{x}$ and $\widetilde{y}$ are distinct from variables of processes. Formulas denoted by ψ are conjunctions of (injective or non-injective) events. (Correspondences have been introduced in CryptoVerif in [17], with a syntax with implicit quantifiers $\psi \Rightarrow \phi$, meaning $\forall \widetilde{x} : \widetilde{T}; \psi \Rightarrow \exists \widetilde{y} : \widetilde{T}'; \phi$ with $\widetilde{x}$, $\widetilde{y}$, $\widetilde{T}$, and $\widetilde{T}'$ as above. Many input files of CryptoVerif still use that syntax.)

The formal definition of correspondences is given in [22, Section 2.7.3]; we give intuition here. We say that a sequence of events (executed by a trace) satisfies $\mathbf{event}(e(M_1, \ldots, M_m))$ when the event $e(M_1, \ldots, M_m)$ occurs in the sequence. A term M is satisfied when it evaluates to true. As usual, $\phi_1 \wedge \phi_2$ is satisfied when ϕ_1 and ϕ_2 are both satisfied, and $\phi_1 \vee \phi_2$ is satisfied when ϕ_1 or ϕ_2 is satisfied. Let us consider as security property sp the correspondence $\forall \widetilde{x} : \widetilde{T}; \psi \Rightarrow \exists \widetilde{y} : \widetilde{T}'; \phi$. When sp does not contain injective events, a sequence of events satisfies sp when, for all values of $\widetilde{x}$, if it satisfies ψ, then there exist values of $\widetilde{y}$ such that it also satisfies ϕ. When sp contains injective events, one additionally requires that each occurrence of the injective events in ψ in the sequence is mapped to a *distinct* occurrence of the injective events in ϕ in the sequence.

Example 1: Consider a protocol with two participants A and B, and suppose that A executes event $e_A(x)$ when it starts the protocol with nonce x, and B executes event $e_B(x)$ when it finishes the protocol with nonce x. In order to model authentication of A to B, we want to prove that when B finishes the protocol, it is sure that A has started the protocol, and that they agree on the value of the nonce x. This property can be modeled by the correspondence

$$\forall x : nonce; \mathbf{event}(e_B(x)) \Rightarrow \mathbf{event}(e_A(x)) \qquad (2)$$

which means that for all x of type $nonce$, if $e_B(x)$ has been executed, then $e_A(x)$ has been executed.

However, this property models a weak form of authentication: there can be many executions of $e_B(x)$ even if event $e_A(x)$ was executed once, that is, A started the protocol only once. In other words, B is not protected against replays. In order to guarantee that B is protected against replays, we need to prove the following injective correspondence

$$\forall x : nonce; \mathbf{inj\text{-}event}(e_B(x)) \Rightarrow \mathbf{inj\text{-}event}(e_A(x)) \qquad (3)$$

which means that each execution of $e_B(x)$ corresponds to a distinct execution of $e_A(x)$: if there are n executions of $e_B(x)$, then there are at least n executions of $e_A(x)$. In other words, each execution of B corresponds to a distinct execution of A, with the same nonce. ◁

Definition 2: Let sp be a correspondence. The process Q *satisfies* sp with public variables V when, for all evaluation contexts C acceptable for Q with public variables V that do not contain the events used by sp, $\Pr[C[Q] : \neg sp]$ is negligible. ◁

A process satisfies sp when the probability that it generates a sequence of events $\mathcal{E}v$ that does not satisfy sp is negligible, in the presence of an adversary represented by the context C.

IV. Proving Security Properties

In order to prove security properties, CryptoVerif first transforms the initial game (the protocol interacting with an adversary), yielding a sequence of games, then it proves the

property itself on the final game. In this section, we explain how CryptoVerif performs this final proof.

In addition to the proof of secrecy and correspondence properties explained below, CryptoVerif can also prove indistinguishability between two games G_1 and G_2 by transforming both games into the same game G_3, using two sequences of games starting from G_1, resp. G_2.

A. Reasoning on Games

CryptoVerif relies on two algorithms in order to reason on games. First, it collects a set $\mathcal{F}_\mu$ of facts that hold at each program point μ in the current game Q_0. In particular, we use the following facts: the boolean term M means that M evaluates to true; $\mathbf{defined}(M)$ means that M is defined (all array accesses in M are defined). In sets of facts, all terms M must be *simple*, that is, contain only replication indices, variables, and function applications. For instance, if the current game Q_0 contains $\mathbf{if}\ M\ \mathbf{then}\ P\ \mathbf{else}\ P'$ and the program point μ is inside P, then $M \in \mathcal{F}_\mu$, since the $\mathbf{then}$ branch can be executed only when M is true; similarly, if μ is inside P', then $\neg M \in \mathcal{F}_\mu$. We do not detail the full computation of $\mathcal{F}_\mu$ since previous versions of this algorithm were already presented in [18, Appendix C.2] and [17, Appendix B.2]. The current algorithm is an extension that relies on the same principles.

The second algorithm is an equational prover: from a set of facts $\mathcal{F}$, this equational prover tries to derive a contradiction by rewriting terms, using an algorithm inspired by Knuth-Bendix completion. It also eliminates collisions between independent random values: for instance, suppose $\mathcal{F}$ contains the equality $n_a = n_b$ where n_a and n_b are two nonces chosen independently and uniformly in a large set $nonce$, where *large* means that $1/|nonce|$ is negligible; the equality $n_a = n_b$ holds only when the two nonces collide, which happens with negligible probability $1/|nonce|$, so the algorithm rewrites $n_a = n_b$ to false and derives a contradiction, by eliminating the collisions between the two nonces. More generally, when the algorithm derives a contradiction, the probability that $\mathcal{F}$ holds is negligible, and we say that "$\mathcal{F}$ yields a contradiction in game Q_0". (We omit the current game Q_0 when it is clear from the context.) Previous versions of this algorithm were presented in [18, Appendix C.5] and [17, Appendix B.3].

B. Secrecy

Let us now define a criterion that proves the secrecy of a variable x. In the first definition of such a criterion [18, Section 4], when x was defined by an assignment $x[i] = y[M]$, CryptoVerif required the whole array y to be secret. We improve this criterion by requiring only that the cells of y that may be stored in x be secret. The other cells of y may leak to the adversary. This extension is important in order to deal with the compromise of keys, since in the presence of key compromise, the adversary can often obtain some session keys, while other session keys remain secret.

Example 2: We illustrate the proof of secrecy on the following toy example:

$$Q_0 = !^{i \leq n} c[i](); \mathbf{new}\ k : \mathsf{key}; \overline{c[i]}\langle\rangle; d[i](compr : \mathsf{bool});$$
$$\mathbf{if}\ compr\ \mathbf{then}\ \overline{d[i]}\langle {}^{\mu_1}k\rangle\ \mathbf{else\ let}\ x : \mathsf{key} = {}^{\mu_2}k\ \mathbf{in}\ {}^{\mu_3}\overline{d[i]}\langle\rangle$$

This process can be executed at most n times, for i from 1 to n, due to the replication $!^{i \leq n}$. The adversary triggers its execution by sending a message to channel $c[i]$, received by the input $c[i]()$. Then it generates a fresh random key k by $\mathbf{new}\ k : \mathsf{key}$ and returns control to the adversary by the output $\overline{c[i]}\langle\rangle$ on channel $c[i]$. Then the adversary can send a boolean value to channel $d[i]$, received by the input $d[i](compr : \mathsf{bool})$ and stored in variable $compr$. If this value is true, the key k is leaked to the adversary by the output $\overline{d[i]}\langle k\rangle$. Otherwise, k is stored in variable x. Our goal is to show that this process satisfies the secrecy of x with public variables $V = \emptyset$. ◁

To prove the secrecy of x in this example, we need to find from which random variable(s) x is defined, k in the example, and to track all usages of the cells of k stored in x, recursively through assignments to other variables if any, to prove that none of these usages leaks x to the adversary. We do this as follows. We write ${}^{\mu}M$ to say that the term M occurs at program point μ. We let $\mathrm{defRand}_\mu(x)$ be the random variable that defines x just before program point μ:

$$\mathrm{defRand}_\mu(x) = \begin{cases} x[\widetilde{i}] & \text{if } \mathbf{new}\ x[\widetilde{i}] : T; {}^{\mu}\ldots \text{ occurs in } Q_0 \\ y[\widetilde{M}] & \text{if } \mathbf{let}\ x[\widetilde{i}] : T = y[\widetilde{M}]\ \mathbf{in}\ {}^{\mu}\ldots \text{ occurs in } Q_0 \text{ and} \\ & y \text{ is defined only by random choices in } Q_0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

When x itself is chosen randomly just before program point μ, $\mathrm{defRand}_\mu(x)$ is simply $x[\widetilde{i}]$, where $\widetilde{i}$ are the *current replication indices* at μ, that is, the indices of replications syntactically above μ in the game. When x is defined by an assignment of a variable $y[\widetilde{M}]$ that is random, $\mathrm{defRand}_\mu(x)$ is that variable. In all other cases, we give up and do not define $\mathrm{defRand}_\mu(x)$. (These other cases can typically be handled by first removing assignments.)

In order to prove that the game Q_0 satisfies the one-session secrecy of x, we use the algorithm $\mathrm{prove}^{\text{1-ses.secr.}(x)}$ defined formally in the full version [22, Section 4.1] and explained here. For each definition of x in Q_0, just before program point μ, we let $y[\widetilde{M}]$ be the variable that defines $x[\widetilde{i}]$ at that point ($y[\widetilde{M}] = \mathrm{defRand}_\mu(x)$), and we show that the probability that $y[\widetilde{M}]$ leaks to the adversary and $\mathcal{F}_\mu$ holds is negligible. (When the facts $\mathcal{F}_\mu$ do not hold, μ is not reached so that definition of x is not executed, hence it can certainly not leak x.) To show that the probability that $y[\widetilde{M}]$ leaks and $\mathcal{F}$ holds is negligible, we proceed as follows:

- If $\mathcal{F}$ yields a contradiction, then the desired result holds: $\mathcal{F}$ itself holds with negligible probability, so a fortiori the probability that $y[\widetilde{M}]$ leaks and $\mathcal{F}$ holds is negligible.
- Otherwise, if $y \in V$, then the desired result does not hold: the adversary can access y because it is a public variable.

- Otherwise, the leak of $y[\widetilde{M}]$ may come from occurrences of $y[\widetilde{M'}]$ inside terms in the game Q_0 that read $y[\widetilde{M}]$ and that leak its value. Hence, we consider each occurrence of $y[\widetilde{M'}]$ in Q_0 and let μ' be its program point.
  - If that occurrence of $y[\widetilde{M'}]$ is in the term M in an assignment $\mathbf{let}\ z[\widetilde{i'}] = M$ in Q_0 and M is built from replication indices, variables, function applications, and conditionals, then this assignment stores in $z[\widetilde{i'}]$ a value that may depend on $y[\widetilde{M'}]$. It may make $y[\widetilde{M}]$ leak only when $z[\widetilde{i'}]$ itself leaks, $\widetilde{M'} = \widetilde{M}$, and the assignment has been executed, so $\mathcal{F}_{\mu'}$ holds. Therefore, we recursively show that the probability that $z[\theta\widetilde{i'}]$ leaks and $\mathcal{F} \cup \theta\mathcal{F}_{\mu'} \cup \{\theta\widetilde{M'} = \widetilde{M}\}$ holds is negligible, where θ is a renaming of the current replication indices at μ' to fresh replication indices. (We rename the replication indices to avoid any clash between the names of replication indices.) We avoid loops by not doing this recursively several times for the same variable z. If we meet again the same variable z in a recursive call, we apply the third case below instead.
  - If that occurrence of $y[\widetilde{M'}]$ is in the argument of an event, then we consider that it does not make $y[\widetilde{M}]$ leak, because the adversary, represented by a context, does not have access to the arguments of events.
  - Otherwise, an occurrence of $y[\widetilde{M'}]$ at μ' may make $y[\widetilde{M}]$ leak only when it is executed with $\widetilde{M'} = \widetilde{M}$, so only when $\widetilde{M'} = \widetilde{M}$ and $\mathcal{F}_{\mu'}$ hold. We show that $\mathcal{F} \cup \theta\mathcal{F}_{\mu'} \cup \{\theta\widetilde{M'} = \widetilde{M}\}$ yields a contradiction, where θ is a renaming of the current replication indices at μ' to fresh replication indices. When this proof succeeds, the probability that $\mathcal{F} \cup \theta\mathcal{F}_{\mu'} \cup \{\theta\widetilde{M'} = \widetilde{M}\}$ holds is negligible, so the probability that $\mathcal{F}$ holds and $y[\theta\widetilde{M'}]$ is executed with $\theta\widetilde{M'} = \widetilde{M}$ is negligible, so the probability that this occurrence of $y[\widetilde{M'}]$ makes $y[\widetilde{M}]$ leak and $\mathcal{F}$ holds is negligible.

In order to prove secrecy, we also define $\mathrm{prove}^{\mathsf{distinct}(x)}$ as for all μ_1, μ_2 that follow a definition of x in Q_0, $z_1 \neq z_2$ or $\theta_1\mathcal{F}_{\mu_1} \cup \theta_2\mathcal{F}_{\mu_2} \cup \{\theta_1\widetilde{M_1} = \theta_2\widetilde{M_2}, \widetilde{i_1} \neq \widetilde{i_2}\}$ yields a contradiction, where $\mathrm{defRand}_{\mu_1}(x) = z_1[\widetilde{M_1}]$, $\mathrm{defRand}_{\mu_2}(x) = z_2[\widetilde{M_2}]$, $\widetilde{i}$ are the current replication indices at the definition of x, θ_1 and θ_2 are two distinct renamings of $\widetilde{i}$ to fresh replication indices, $\widetilde{i_1} = \theta_1\widetilde{i}$, and $\widetilde{i_2} = \theta_2\widetilde{i}$. Intuitively, $\mathrm{prove}^{\mathsf{distinct}(x)}$ guarantees that, for all μ_1, μ_2 that follow a definition of x, if $x[\widetilde{i_1}]$ is defined at μ_1, so $x[\widetilde{i_1}] = z_1[\theta_1\widetilde{M_1}]$, and $x[\widetilde{i_2}]$ is defined at μ_2, so $x[\widetilde{i_2}] = z_2[\theta_2\widetilde{M_2}]$, with $\widetilde{i_1} \neq \widetilde{i_2}$, then the random variables that define x in these two cases, $z_1[\theta_1\widetilde{M_1}]$ and $z_2[\theta_2\widetilde{M_2}]$, are different, that is, $z_1 \neq z_2$ or $\theta_1\widetilde{M_1} \neq \theta_2\widetilde{M_2}$. Therefore, $z_1[\theta_1\widetilde{M_1}]$ is independent of $z_2[\theta_2\widetilde{M_2}]$, so $x[\widetilde{i_1}]$ is independent of $x[\widetilde{i_2}]$. Combining this information with the proof of one-session secrecy, we can prove secrecy of x:

$$\mathrm{prove}^{\mathsf{Secrecy}(x)} = \mathrm{prove}^{\mathsf{1\text{-}ses.secr.}(x)} \wedge \mathrm{prove}^{\mathsf{distinct}(x)}$$

This algorithm is justified by the following proposition, proved in the full version [22, Section 4.1, Proposition 1].

Proposition 1 ((One-session) secrecy): Let sp be $\mathsf{1\text{-}ses.secr.}(x)$ or $\mathsf{Secrecy}(x)$. If prove^{sp}, then Q_0 satisfies sp with public variables V ($x \notin V$). ◁

Example 3: We apply the algorithm to the game Q_0 of Example 2. We denote by $\mathcal{F}\{i'/i\}$ the set $\mathcal{F}$ with i' substituted for i. Recall that k, $compr$, and x are implicitly arrays indexed by i. The game Q_0 contains a single definition of x, just before program point μ_3; x is defined by an assignment from $k[i]$ (the current replication index i is implicit in the process) so $\mathrm{defRand}_{\mu_3}(x) = k[i]$. To prove one-session secrecy of x, $\mathrm{prove}^{\mathsf{1\text{-}ses.secr.}(x)}$, we show that the probability that $k[i]$ leaks and $\mathcal{F}_{\mu_3}$ holds is negligible. The facts $\mathcal{F}_{\mu_3}$ do not yield a contradiction (μ_3 is reachable). We have $k \notin V$. Then we consider the two occurrences of $k[\widetilde{M'}]$ in Q_0, at program points μ_1 and μ_2.

At program point μ_1, the occurrence of k is neither in an assignment or an event, so we are in the third case. We show that $\mathcal{F}_{\mu_3} \cup \mathcal{F}_{\mu_1}\{i_2/i\} \cup \{i_2 = i\}$ yields a contradiction. This is true because $\neg compr[i] \in \mathcal{F}_{\mu_3}$ and $compr[i_2] \in \mathcal{F}_{\mu_1}\{i_2/i\}$: the same cell of k indexed by $i_2 = i$ cannot be both stored in x (before μ_3) and leaked at μ_1.

At program point μ_2, the occurrence of k is in an assignment to x. We show recursively that the probability that $x[i_3]$ leaks and $\mathcal{F}_{\mu_3} \cup \mathcal{F}_{\mu_2}\{i_3/i\} \cup \{i_3 = i\}$ holds is negligible. This is true because $x \notin V$ and x is never used in Q_0: there is no occurrence of $x[\widetilde{M'}]$ to consider, x never leaks.

Hence, we have $\mathrm{prove}^{\mathsf{1\text{-}ses.secr.}(x)}$.

Moreover, $\mathrm{prove}^{\mathsf{distinct}(x)}$ is $k \neq k$ or $\mathcal{F}_{\mu_3}\{i_1/i\} \cup \mathcal{F}_{\mu_3}\{i_2/i\} \cup \{i_1 = i_2, i_1 \neq i_2\}$ yields a contradiction, which is true: $i_1 = i_2$ and $i_1 \neq i_2$ yield a contradiction. Distinct cells of x, of indices i_1 and i_2, come from distinct cells of k, and are thus independent. Therefore, we have $\mathrm{prove}^{\mathsf{Secrecy}(x)}$.

By Proposition 1, Q_0 satisfies the secrecy of x without public variables. ◁

This improved proof of secrecy is useful to prove forward secrecy in a signed Diffie-Hellman protocol, for instance. In this example, the compromise of signature keys can be taken into account by including it in the assumption on signatures.

C. Correspondences

The algorithm that CryptoVerif uses in order to prove correspondences was first presented in [17]. The full version [22, Section 4.2] presents a version that uses exact security rather than asymptotic security and with a few extensions unrelated to key compromise.

V. New Commands

In this section, we present the new commands that we have added to CryptoVerif in order to deal with dynamic key compromise. These commands can be used to guide the tool so that it can perform the proof. Although we focus on key compromise in this paper, these commands are general enough that they can be applied for other purposes.

Example 4: In order to illustrate the usage of these new commands, we use as a running example a Diffie-Hellman key exchange with pre-shared key, for which we prove forward

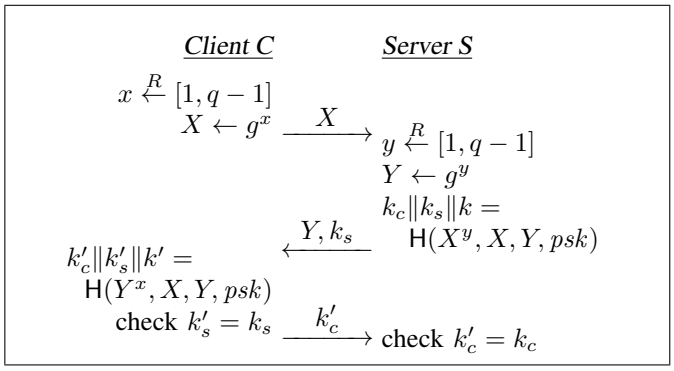

Fig. 2. Running example. The final session key is $k = k'$.

secrecy with respect to the compromise of the pre-shared key. This protocol is a very simplified version of the TLS 1.3 Diffie-Hellman handshake with pre-shared key; it is a kind of example that cannot be handled by the basic treatment of compromise of Section II, because the pre-shared key is not used in key position in a primitive but as argument of a random oracle.

In this protocol, a client C and a server S initially share a pre-shared key psk and exchange messages as shown in Figure 2 where g is a generator of a group[1] $\mathbb{G}$ of prime order q. Furthermore, we assume that the group $\mathbb{G}$ satisfies the gap Diffie-Hellman (GDH) assumption [41], that is, a probabilistic polynomial-time adversary has a negligible probability of computing g^{ab} knowing g^a and g^b for two random exponents $a, b \in [1, q-1]$ and having access to a decisional Diffie-Hellman oracle which, given G, X, Y, Z, returns whether X, Y, Z is a good Diffie-Hellman triple with generator G, that is, whether there exist x and y such that $X = G^x$, $Y = G^y$, and $Z = G^{xy}$.

The client C chooses randomly x in $[1, q-1]$ and sends to the server S the public Diffie-Hellman exponential $X = g^x$. The server chooses randomly y in $[1, q-1]$, computes its public Diffie-Hellman exponential $Y = g^y$, the Diffie-Hellman shared secret g^{xy} as X^y, and three keys k_c, k_s, and k of the same length as $k_c \| k_s \| k = \mathsf{H}(g^{xy}, X, Y, psk)$, where $\|$ is concatenation and H is a random oracle. We recall that a random oracle is a function that returns a fresh random value when it is called on a new argument and returns the previous result when it is called again on the same argument. The key k is the session key for which we want to prove forward secrecy, while k_c and k_s are used in order to provide mutual authentication between the client and the server. The server sends to the client Y and k_s. The client can then compute the Diffie-Hellman shared secret g^{xy} as Y^x and the keys k'_c, k'_s, and k' like the server $k'_c \| k'_s \| k' = \mathsf{H}(g^{xy}, X, Y, psk)$. It verifies that the key k_s it received from the server is equal to the key k'_s it computed. Since the pre-shared key psk is required to compute this key, that authenticates the server to the client. The client also sends the key k'_c to the server, and the server verifies that this key is equal to the key k_c it computed. That authenticates the client to the server.

This protocol is modeled as follows in CryptoVerif:

$cStart(); \mathbf{new}\ hk : \mathsf{hashkey}; \mathbf{new}\ psk : \mathsf{key}; \overline{cStart}\langle\rangle;$
$(\mathsf{Client}(hk, psk) \mid \mathsf{Server}(hk, psk) \mid$
$\mathsf{corruptPSK}(psk) \mid \mathsf{hashoracle}(hk))$

The adversary can start the protocol by sending a message on channel $cStart$, which will be received by the input $cStart()$. Then the protocol chooses randomly a fresh key hk, which models the choice of the random oracle itself. It also chooses the pre-shared key psk, and returns control to the adversary by outputting an empty message on channel $cStart$. Then four processes run in parallel, representing respectively the client, the server, the corruption of the pre-shared key, and the hash oracle. The process $\mathsf{hashoracle}(hk)$, not detailed here, allows the adversary to call the hash oracle H.

The process $\mathsf{corruptPSK}$ allows the adversary to obtain the pre-shared key psk at any time, by sending a message on channel $cCorrupt$. It sends the pre-shared key on channel $cCorrupt$, and records that the pre-shared key has been corrupted by defining the variable $corruptedPSK$:

$\mathbf{let}\ \mathsf{corruptPSK}(psk : \mathsf{key}) = cCorrupt();$
$\quad \mathbf{let}\ corruptedPSK : \mathsf{bool} = \mathsf{true}\ \mathbf{in}\ ^{\mu_1}\overline{cCorrupt}\langle psk\rangle.$

The client is represented by the following process:

$\mathbf{let}\ \mathsf{Client}(hk : \mathsf{hashkey}, psk : \mathsf{key}) =$
$\quad !^{i_c \leq n_c} cC0[i_c](); \mathbf{new}\ x : Z; \mathbf{let}\ X = \mathsf{exp}(g, x)\ \mathbf{in}$
$\quad \overline{cC1[i_c]}\langle X\rangle; cC2[i_c](Y' : \mathsf{G}, k_s : \mathsf{key});$
$\quad \mathbf{let}\ \mathsf{tuple3keys}(k'_c, k'_s, k') =$
$\quad\quad \mathsf{split3}(\mathsf{H}(hk, \mathsf{exp}(Y', x), X, Y', psk))\ \mathbf{in}$
$\quad \mathbf{if}\ k_s = k'_s\ \mathbf{then}$
$\quad \mathbf{if}\ \mathbf{defined}(corruptedPSK)\ \mathbf{then}$
$\quad\quad \overline{cC3[i_c]}\langle k'_c\rangle$
$\quad \mathbf{else}$
$\quad\quad \mathbf{let}\ k\_secret\_c : \mathsf{key} = k'\ \mathbf{in}\ \overline{cC3[i_c]}\langle k'_c\rangle$

The client process takes as argument the keys hk and psk chosen in the initial process. It can be executed at most n_c times, as expressed by the replication $!^{i_c \leq n_c}$. It is triggered when a message is received by the input $cC0[i_c]()$ on channel $cC0[i_c]$ (c stands for channel, C for client). Then it chooses randomly the exponent x. (The type Z represents the type of exponents $[1, q-1]$.) It computes the exponential X, using the exponentiation function exp, and sends it as first real message of the protocol by the output $\overline{cC1[i_c]}\langle X\rangle$. Then the client waits for the second message of the protocol on channel $cC2[i_c]$. When it is received, it is stored in Y', k_s. Then the client computes the keys k'_c, k'_s, k' as specified in the protocol. The function H takes as additional argument the key hk, which models the choice the random oracle. The function split3 splits the output of the random oracle into a tuple of 3 keys. These keys are stored in k'_c, k'_s, k' by matching this tuple with $\mathsf{tuple3keys}(k'_c, k'_s, k')$. The client verifies that $k'_s = k_s$, and when this holds, it will always output the last message of the protocol k'_c by $\overline{cC3[i_c]}\langle k'_c\rangle$. Moreover, we

[1]Formally, $\mathbb{G}$ is a family of prime-order groups indexed by the security parameter, since we work in the asymptotic framework. For simplicity, we name it "a group".

encode forward secrecy of k' as follows: when the pre-shared key is not corrupted yet, which is tested by checking that the variable $corruptedPSK$ is not defined yet, so in the **else** branch of the test **if defined**($corruptedPSK$), the client additionally stores the session key k' in k_secret_c. We are going to show the secrecy of k_secret_c: that shows secrecy of session keys k' for sessions that terminate before the pre-shared key is corrupted, which corresponds to forward secrecy of k'. (The pre-shared key may be corrupted after the end of the session. A more detailed model would give the sessions keys for which we do not prove secrecy to the adversary, to simulate reveal queries on those keys. For simplicity, we omit this point.)

Similarly, the server is represented by the following process:

$$\begin{array}{l}\textbf{let } \mathsf{Server}(hk : \mathsf{hashkey}, psk : \mathsf{key}) = \\ \quad !^{i_s \leq n_s}\, cS1[i_s](X' : \mathsf{G}); \textbf{new } y : Z; \textbf{let } Y = \mathsf{exp}(g, y) \textbf{ in} \\ \quad \textbf{let } \mathsf{tuple3keys}(k_c, k_s, k) = \\ \quad\quad \mathsf{split3}(\mathsf{H}(hk, \mathsf{exp}(X', y), X', Y, psk)) \textbf{ in} \\ \quad \overline{cS2[i_s]}\langle Y, k_s\rangle; cS3[i_s](k'_c : \mathsf{key}); \textbf{if } k_c = k'_c \textbf{ then} \\ \quad \textbf{if defined}(corruptedPSK) \textbf{ then yield else} \\ \quad \textbf{let } k\_secret\_s : \mathsf{key} = k.\end{array}$$

It can be executed at most n_s times, as specified by the replication $!^{i_s \leq n_s}$. It is triggered when it receives the first message of the protocol on channel $cS1[i_s]$ (c stands for channel, S for server). Here, the adversary is assumed to control the network; it is in charge of forwarding messages between the client and the server, possibly modifying them in order to mount attacks. For instance, to run a normal session of the protocol, the adversary should forward the message sent on channel $cC1[i_c]$ by the client to the server on a channel $cS1[i_s]$ for some i_s; then the execution number i_c of the client runs a session of the protocol with the execution number i_s of the server. The server stores the message received on channel $cS1[i_s]$ in X', chooses the exponent y, and computes the exponential Y. It computes the keys k_c, k_s, k much like the client, and outputs the second message of the protocol Y, k_s on channel $cS2[i_s]$ by $\overline{cS2[i_s]}\langle Y, k_s\rangle$. Then it waits for the third message of the protocol on channel $cS3[i_s]$, stores it in k'_c and verifies that $k_c = k'_c$. When this check succeeds, the protocol terminates. Similarly to the client, when the pre-shared key is not corrupted yet, we store the session key k in k_secret_s. We are going to prove the secrecy of k_secret_s to show forward secrecy of the session key k.

The proof of secrecy of k_secret_c and k_secret_s in CryptoVerif relies in particular on the new commands below.◁

A. *focus*

By default, CryptoVerif tries to prove all desired properties simultaneously, using the same sequence of games. The command **focus** $q_1, \ldots, q_m$ tells CryptoVerif to prove only the properties $q_1, \ldots, q_m$ (secrecy and correspondences properties), as a first step. The other properties to prove are (temporarily) ignored. Focusing on the queries $q_1, \ldots, q_m$ has an effect on subsequent game transformations that are allowed: events that do not occur in these queries can be removed; only the active queries are considered in the transformation **success simplify** defined below (Section V-B).

When $q_1, \ldots, q_m$ are proved, CryptoVerif automatically goes back to before the **focus** command to prove the remaining properties. The user can also go back to before the last focus command at any time using the command **undo focus**.

In our approach to key compromise, we use the **focus** command to prove the authentication property mentioned at the end of Section II, as we illustrate in our running example.

Example 5: In order to show the secrecy of the session key k, resp. k', in Example 4, we need to show that, when the client or the server successfully terminates the protocol, the adversary has a negligible probability of performing the same random oracle call to H as the client or the server. Since the pre-shared key psk may be compromised, that amounts to showing that the adversary does not know the Diffie-Hellman value X^y, resp. Y^x, which can be shown using the gap Diffie-Hellman assumption (GDH), when the Diffie-Hellman value is g^{xy} for x and y chosen by the client and the server respectively, that is, when the exponentials match: $X = g^x$, resp. $Y = g^y$. We eliminate the case in which the exponentials do not match by executing events and aborting in this case (as detailed just below); since the games differ when these events are executed, we will need to show that these events happen with negligible probability. For that, we will be able to rely on the secrecy of the pre-shared key, because these events are always executed before the compromise of the pre-shared key.

In more detail, in the client, we introduce a test **find** $j3 \leq n_s$ **suchthat defined**$(X'[j3], Y[j3]) \wedge X = X'[j3] \wedge Y' = Y[j3]$ **then** just before the definition of k_secret_c. This test verifies that the exponentials X, Y' in the client correspond to the exponentials X', Y in some execution of the server of index $j3$. When this test is not true, we execute event client_break_auth, so that the end of the client becomes:

$$\begin{array}{l}\textbf{find } j3 \leq n_s \textbf{ suchthat defined}(X'[j3], Y[j3]) \wedge \\ \quad X = X'[j3] \wedge Y' = Y[j3] \\ \textbf{then let } k\_secret\_c : \mathsf{key} = k' \textbf{ in } \overline{cC3[i_c]}\langle k'_c\rangle \\ \textbf{else } {}^{\mu_2}\textbf{event\_abort } \mathsf{client\_break\_auth}\end{array}$$

This transformation is performed by the CryptoVerif command

insert before "**let** k_secret_c" "**find** $j3 \leq n_s$ **suchthat defined**$(X'[j3], Y[j3]) \wedge X = X'[j3] \wedge Y' = Y[j3]$ **then else event_abort** client_break_auth"

The obtained game differs from the initial game exactly when event client_break_auth is executed, and we will show that this event happens with negligible probability by proving the correspondence **event**(client_break_auth) $\Rightarrow$ false. That will imply that the games are computationally indistinguishable. This is an application of Shoup's lemma [49], which says that, if two games differ only when some event happens, then the probability of distinguishing these games is bounded by the probability of the event. The execution of event client_break_auth corresponds to a breach of authentication of the server to the client: the client successfully terminates

the protocol, although there is no session of the server with matching exponentials.

Similarly, we transform the last line of the server into

$$\textbf{find } j4 \leq n_c \textbf{ suchthat defined}(X[j4]) \wedge X' = X[j4] \textbf{ then let } k\_secret\_s : \mathsf{key} = k \textbf{ else } {}^{\mu_3}\textbf{event\_abort } \mathsf{server\_break\_auth}$$

and that game differs from the previous one only when event server_break_auth is executed.

At this point, we need to prove the secrecy of k_secret_c and k_secret_s and the correspondences $\textbf{event}(\mathsf{client\_break\_auth}) \Rightarrow \mathsf{false}$ and $\textbf{event}(\mathsf{server\_break\_auth}) \Rightarrow \mathsf{false}$. By the command

focus "**query event**(client_break_auth) ⇒ false", "**query event**(server_break_auth) ⇒ false"

we tell CryptoVerif to start by proving the two correspondence properties, without considering the secrecy properties. We continue the proof in Example 6. ◁

The **focus** command is more generally useful when different properties require different proofs.

B. *success simplify*

The transformation **success simplify** extends the existing command **success**, which proves the desired security properties as explained in Section IV. The command **success simplify** first collects information that is known to hold when the adversary breaks at least one of the desired properties. Then, it performs a simplification step: it removes parts of the game that contradict this information and replaces them with **event_abort** adv_loses. Indeed, when these parts of the game are executed, the adversary cannot break any of the security properties to prove, so they can be safely removed. Typically, **success simplify** allows us to remove the key compromise when we only prove authentication before compromise.

Here, we explain a simplified version of this transformation, assuming all active queries are of the form $\textbf{event}(e_i) \Rightarrow \mathsf{false}$, where the events e_i are executed by **event_abort** e_i in the game. This case is sufficient for most examples. The full transformation supports other queries, as explained in the full version [22, Section 5.1.23]. The command **success simplify** computes $\mathcal{L} = \{\theta\mathcal{F}_\mu \mid \textbf{event\_abort } e_i \text{ occurs at } \mu \text{ in the game, } \theta \text{ is a renaming of the current replication indices at } \mu \text{ to fresh replication indices}\}$. When the property $\textbf{event}(e_i) \Rightarrow \mathsf{false}$ is broken, event e_i is executed by **event_abort** e_i at some program point μ, so $\mathcal{F}_\mu$ holds. Therefore, when some property $\textbf{event}(e_i) \Rightarrow \mathsf{false}$ is broken, some set of facts $\mathcal{F} \in \mathcal{L}$ holds. (In the computation of $\mathcal{L}$, we rename the replication indices at μ to fresh indices by θ to avoid colliding indices in the simplification step below.)

In the simplification step, the set $\mathcal{L}$ is used as follows: for each program point μ, if for all $\mathcal{F} \in \mathcal{L}$, $\mathcal{F}_\mu \cup \mathcal{F}$ yields a contradiction, then the code at μ is replaced with **event_abort** adv_loses.

If a security property is broken before the transformation and not after, then a modified program point μ is reached and the adversary breaks the property. Since μ is reached, $\mathcal{F}_\mu$ holds. Since the adversary breaks the property, some $\mathcal{F}$ in $\mathcal{L}$ holds. So $\mathcal{F}_\mu \cup \mathcal{F}$ holds, which has a negligible probability of happening because for all $\mathcal{F} \in \mathcal{L}$, $\mathcal{F}_\mu \cup \mathcal{F}$ yields a contradiction. Therefore, if a security property is broken before the transformation, then it is also broken after the transformation, since all properties are considered up to negligible probability. This is formalized by the following lemma.

Lemma 1: If transformation **success simplify** transforms G into G', the property sp is a secrecy or correspondence property and corresponds to an active query, and G' satisfies sp, then G satisfies sp. ◁

A more general version of this result is proved in the full version [22, Section 5.1.23, Lemma 63].

Example 6: Continuing Example 5, when the events client_break_auth and server_break_auth are executed, the variable $corruptedPSK$ is not defined (these events occur in the **else** branch of the test $\textbf{if defined}(corruptedPSK)$), so the pre-shared key has not been leaked yet. That allows the command **success simplify** to remove the output of the pre-shared key $\overline{cCorrupt}\langle psk\rangle$ and replace it with **event_abort** adv_loses.

In more detail, $\mathcal{L} = \{\mathcal{F}_{\mu_2}\{i'_c/i_c\}, \mathcal{F}_{\mu_3}\{i'_s/i_s\}\}$ since event client_break_auth occurs at program point μ_2 and event server_break_auth occurs at program point μ_3. We have $\neg\textbf{defined}(corruptedPSK) \in \mathcal{F}_{\mu_2}\{i'_c/i_c\}$ and $\neg\textbf{defined}(corruptedPSK) \in \mathcal{F}_{\mu_3}\{i'_s/i_s\}$, because both μ_2 and μ_3 are in the **else** branch of a test $\textbf{if defined}(corruptedPSK)$. Moreover, since the program point μ_1 of $\overline{cCorrupt}\langle psk\rangle$ is under the definition of $corruptedPSK$, we have $\textbf{defined}(corruptedPSK) \in \mathcal{F}_{\mu_1}$. Therefore, $\mathcal{F}_{\mu_1} \cup \mathcal{F}_{\mu_2}\{i'_c/i_c\}$ yields a contradiction and similarly, $\mathcal{F}_{\mu_1} \cup \mathcal{F}_{\mu_3}\{i'_s/i_s\}$ yields a contradiction. The command **success simplify** then replaces the code at μ_1 with **event_abort** adv_loses.

We can then prove $\textbf{event}(\mathsf{client\_break\_auth}) \Rightarrow \mathsf{false}$ and $\textbf{event}(\mathsf{server\_break\_auth}) \Rightarrow \mathsf{false}$ assuming the pre-shared key is never corrupted, since **success simplify** removed that corruption. We apply the random oracle assumption on H by the command **crypto rom(**H**)**. Since the pre-shared key is never corrupted, the adversary has a negligible probability of providing to the random oracle the same arguments as in the client or the server. After simplification of the game, that allows CryptoVerif to show that in the server, the result of the random oracle is always a fresh random value and that in the client, it is either *a)* the same value as in some execution of the server or *b)* a fresh value. By the command **crypto splitter(**split3**) ***, we use that splitting a uniformly chosen bitstring of a fixed length into 3 bitstrings of fixed lengths yields uniformly chosen bitstrings. (The star * applies this property as many times as possible.) After this transformation, the keys k_c, k_s, and k in the server are fresh random values. Furthermore, in case *b)* above, k'_s is a fresh random value, so the test $k'_s = k_s$ in the client has a negligible probability of succeeding. So the only case that allows the test $k'_s = k_s$ in the client to succeed with non-negligible probability is *a)*: the random oracle returned

the same result as in some execution of the server. In this case, the arguments of the random oracle are also the same in the client and in the server, so the test **find** $j3 \leq n_s$ **suchthat defined**$(X'[j3], Y[j3]) \wedge X = X'[j3] \wedge Y' = Y[j3]$, introduced in Example 5, succeeds. The **else** branch of that test can then be removed, which removes the event client_break_auth.

A similar reasoning allows CryptoVerif to remove the event client_break_auth in the server. Then the command **success** proves the correspondences **event**(client_break_auth) $\Rightarrow$ false and **event**(server_break_auth) $\Rightarrow$ false since the two considered events do not occur in the current game.

Since all active queries are proved, CryptoVerif goes back to before the **focus** command and tries to prove secrecy of k_secret_c and k_secret_s in that game. In the game before the **focus** command, the pre-shared key may be corrupted, so we cannot rely on its secrecy to show that the adversary cannot compute the same random oracle calls as the client and the server. We rely on the GDH assumption instead.

As explained at the beginning of Example 5, in order to apply GDH, we must isolate the cases in which the computed Diffie-Hellman value Y^x or X^y is g^{xy} for x, y chosen by the client and the server respectively. We do this by introducing tests on the received exponentials similar to those introduced in Example 5, but earlier in the process, as soon as the exponentials are received, so that the cases are distinguished before computing Y^x (resp. X^y). Precisely, we first introduce a test after the input $cS1[i_s](X' : \mathsf{G})$ in the server, to distinguish whether the received exponential X' comes from the client or not: **find** $j' \leq n_c$ **suchthat defined**$(X[j']) \wedge X' = X[j']$ **then**. After that, the server contains two copies of the code from **new** $y : Z$, one when the test succeeds, one when it fails. We rename the variable Y to two distinct names in these two copies: Y_2 when the test succeeds, Y_3 when it fails. Similarly, we introduce a test after the input $cC2[i_c](Y' : \mathsf{G}, k_s : \mathsf{key})$ in the client, to distinguish whether the received exponential Y' comes from a session of the server that received the exponential X from the client: **find** $j'' \leq n_s$ **suchthat defined**$(Y_2[j''], X'[j'']) \wedge X = X'[j''] \wedge Y' = Y_2[j'']$ **then**. By the tests already introduced in the client and in the server in Example 5, the variables k_secret_c and k_secret_s that we want to prove secret are defined only when the two tests above succeed. (Otherwise, we execute events client_break_auth or server_break_auth.) Then we apply the random oracle assumption on H and the GDH assumption. When the test **find** $j' \leq n_c$ **suchthat defined**$(X[j']) \wedge X' = X[j']$ **then** (introduced after the input on $cS1$) succeeds, we have $X' = X[j'] = g^{x[j']}$, so the value of $\mathsf{exp}(X', y)$ in the server is $g^{x[j']y}$, and the adversary has a negligible probability of computing it by GDH. Therefore, the adversary has a negligible probability of the computing the argument of the random oracle H, so the result of the random oracle is indistinguishable from a fresh random value to the adversary. Therefore, k, and so k_secret_s are indistinguishable from fresh random values to the adversary. This is the main argument that allows CryptoVerif to prove the secrecy of k_secret_s. Similarly, when the test **find** $j'' \leq n_s$ **suchthat defined**$(Y_2[j''], X'[j'']) \wedge X = X'[j''] \wedge Y' = Y_2[j'']$ **then** (introduced after the input on $cC2$) succeeds, the value of $\mathsf{exp}(Y', x)$ in the client is $g^{xy[j'']}$, and the adversary has a negligible probability of computing it by GDH. By the same reasoning, that allows CryptoVerif to prove the secrecy of k_secret_c. $\triangleleft$

Generalizing the example above, we prove forward secrecy with respect to the compromise of a key k_0 as follows. We introduce events (such as client_break_auth and server_break_auth above), executed when authentication does not hold (*i.e.*, the protocol successfully terminates but the messages seen by the client and server do not match) while the key k_0 is not compromised yet. Using **focus**, we focus on proving that these events happen with negligible probability. Then, by **success simplify**, we remove the compromise of k_0 (since the considered events are never executed once k_0 is compromised). Then we show that these events happen with negligible probability, by relying on the secrecy of k_0. After that, we come back to before the **focus** command and prove the remaining queries, exploiting that the cases in which authentication does not hold have been removed by introducing events.

This proof technique is similar to some pen-and-paper proofs of forward secrecy, such as the one of [27, Appendix D]. (The events $\mathsf{EncryptBC}_k$ and AuthBC_k of [27, Appendix D] play the same role as client_break_auth and server_break_auth here.)

Example 7: We consider again the example of Section II, and show the model of ciphertext integrity with compromise (Figure 1) from a definition without compromise. Basically, we transform the left and right-hand sides of the indistinguishability property into the following game G_3:

new k : key; (
 !$^{i \leq n}$Oenc(x : cleartext) :=
  new r : enc_seed; **let** z : ciphertext = enc_r(x, k, r)
  in return(z) |
 !$^{i' \leq n'}$Odec(y : ciphertext) :=
  if defined$(corrupt)$ **then return**$(\mathsf{dec}(y, k))$ **else**
  find $j \leq n$ **suchthat defined**$(x[j], z[j]) \wedge z[j] = y$
  then return$(\mathsf{injbot}(x[j]))$ **else**
  if $dec(y, k) <> bottom$ **then** $^{\mu}$**event_abort** disting
  else return(bottom) |
 Ocorrupt() := **let** $corrupt$: bool = true **in** $^{\mu_1}$**return**(k)).

Using (1), we can see that the left-hand side L of Figure 1 differs from G_3 only when event disting is executed. Similarly, the right-hand side R of Figure 1 differs from G_3 only when event disting is executed. Hence, the probability of distinguishing L from R is bounded by the probability of disting in G_3. We show that this probability is negligible by proving the correspondence query **event**(disting) $\Rightarrow$ false. To do that, we first apply the transformation **success simplify**. Intuitively, when the correspondence **event**(disting) $\Rightarrow$ false is broken, event disting is executed, so $corrupt$ is not defined; therefore, the program point μ_1 has not been executed. That allows us to replace $^{\mu_1}$**return**(k) with **event_abort** adv_loses. Formally, $\mathcal{L} = \{\mathcal{F}_{\mu}\{i''/i'\}\}$, where event disting occurs at μ. We have **defined**$(corrupt) \in \mathcal{F}_{\mu_1}$

and $\neg\mathbf{defined}(corrupt) \in \mathcal{F}_\mu\{i''/i'\}$ (since μ is in the **else** branch of **if** $\mathbf{defined}(corrupt)$), so $\mathcal{F}_{\mu_1} \cup \mathcal{F}_\mu\{i''/i'\}$ yields a contradiction, and we replace the code at μ_1 with **event_abort** adv_loses. In the transformed game G_4, the key k is never corrupted, so we can apply the standard ciphertext integrity assumption without corruption show that the probability of disting is negligible and conclude that the probability of distinguishing L from R is negligible. ◁

C. *guess* i

The transformation **guess** i consists in guessing the tested session of a principal in a protocol, a frequent step in cryptographic proofs: properties (e.g. one-session secrecy) may be defined using a test query, and guessing the tested session means guessing on which session the adversary is going to perform its test query. More generally, by guessing the tested session, we prove security properties for one session (the tested one) and infer that they hold in all sessions by symmetry. In CryptoVerif, a replicated process $!^{i\leq n}Q$ represents n sessions (or instances, or copies) of the principal represented by process Q. A priori, CryptoVerif proves security properties for all sessions of Q. By guessing the tested session of Q, we prove the security properties only for one session of Q, the tested one, for which the replication index i is equal to the guessed value i_{tested}. Formally, we consider a game G and define a transformed game G' by replacing $!^{i\leq n}Q$ with $!^{i\leq n}Q'$ where Q' is obtained from Q by replacing the processes P under the first inputs with **if** $i = i_{\text{tested}}$ **then** P' **else** P'' and i_{tested} is a constant. We distinguish the process executed in the tested session, P', on which we are going to prove security properties, from the process P'' for other sessions which are executed, but for which we do not prove security properties. The process P' is obtained from P by

- duplicating all events: **event** $e(\widetilde{M})$ is replaced with **event** $e(\widetilde{M})$; **event** $e'(\widetilde{M})$ and **event_abort** e is replaced with **event** e; **event_abort** e'. We require that in the game G, the same event e cannot occur both under the modified replication $!^{i\leq n}Q$ and elsewhere in the game. (Otherwise, queries that use e are left unchanged.)
- duplicating definitions of every variable x used in queries for secrecy and one-session secrecy: **let** $x' = x$ **in** is added after each definition of x. We require that in the game G, the same variable x used in queries for secrecy or one-session secrecy cannot be defined both under the modified replication $!^{i\leq n}Q$ and elsewhere in the game. (Otherwise, the considered query is left unchanged.)

The process P'' is obtained from P by duplicating definitions of every variable x used in queries for secrecy (not one-session secrecy): **let** $x'' = x$ **in** is added after each definition of x.

We replace variables x in secrecy and one-session secrecy queries with their duplicated version x'. For secrecy queries, the duplicated version x'' is added to public variables. In both cases, we prove (one-session) secrecy for the variable x' defined in the tested session. In case of one-session secrecy, that is enough: it shows that x' is indistinguishable from a random value, and that proves one-session secrecy of x for all sessions by symmetry. However, for secrecy, we additionally want to show that the values of x in the various sessions are independent of each other; this is achieved by considering the value of x in sessions other than the tested session (that is, x'') as public: if x' is indistinguishable from random even when x'' is public, then x' is independent of x''.

In correspondence queries without injective events, we replace *one* non-injective event e before the arrow $\Rightarrow$ with its duplicated version e'. Hence, we prove the query for the tested session, which uses event e'. The proof is valid for all sessions by symmetry.

Other queries are left unchanged: secrecy and one-session secrecy queries for variables not defined under the modified replication, non-injective correspondence queries with no event before the arrow $\Rightarrow$ under the modified replication (the previous queries prove properties about other principals than the one for which we guess the tested session) as well as correspondence queries with injective events (see details below). It is clear that these queries are not affected by the transformation.

Lemma 2: Suppose the game G is transformed into G' by the transformation **guess** i, where i is a replication index bounded by n. Below, we consider only the modified queries.

Let sp and sp' be respectively a correspondence without injective events and its transformed correspondence. If G' satisfies sp', then G satisfies sp.

If G' satisfies the one-session secrecy of x' with public variables V ($x, x', x'' \notin V$), then G satisfies the one-session secrecy of x with public variables V.

If G' satisfies the secrecy of x' with public variables $V \cup \{x''\}$ ($x, x', x'' \notin V$), then G satisfies the secrecy of x with public variables V. ◁

A version of this result for exact security is proved in the full version [22, Section 5.1.17, Lemma 58]. In the exact security framework, the probability of breaking a property in the initial game G is basically n times the probability of breaking the matching property in the transformed game G'.

Example 8: We consider the following protocol:

$$\begin{array}{ll} B \to A: & \{na\}_{pkA} \\ A \to B: & na \end{array}$$

The key pkA is the public encryption key of participant A. The participant B chooses a fresh random nonce na and sends it to A encrypted under pkA, using an IND-CCA2 public-key encryption scheme. Only A can decrypt this message, using her secret key skA, and A replies by sending the nonce na to B. When B sees this nonce, B knows that the first message has been decrypted; therefore A has decrypted it: that authenticates A to B.

This protocol is related to compromise in that the secrecy of the nonce na is essential to the proof, but the nonce is leaked (compromised) at the end of the protocol. We use **guess** and the approach for dealing with compromise outlined at the end of Section II to prove its security.

The role of B is modeled by the following process:

$!^{i_B\leq n_B} c3[i_B](); \mathbf{new}\ na : \mathsf{nonce}; \overline{c4[i_B]}\langle \mathsf{enc}(\mathsf{pad}(na), pkA)\rangle; c5[i_B](= na); \mathbf{event}\ e_B(na)$

The replication $!^{i_B \leq n_B}$ says that this process can be executed at most n_B times. It is triggered when a message is received by the input $c3[i_B]()$ on channel $c3[i_B]$. Then it generates a fresh random nonce na, pads it to the length of an encryption block by the function pad, and encrypts it under pkA. The ciphertext is sent on channel $c4[i_B]$. When the process receives a message equal to na on channel $c5[i_B]$, it executes event $e_B(na)$, meaning intuitively that B successfully terminates the protocol with nonce na. A similar event $e_A(na)$ is executed by A after she decrypts B's message, meaning that A was involved in the protocol with nonce na. Our goal here is to show the correspondence (2) of Example 1. As in Example 4, the adversary controls the network. It is in charge of relaying messages between A and B, possibly modifying them in order to mount attacks.

The first step of the proof is to guess the tested session for B, by **guess** i_B. That transforms the code for B into

$!^{i_B \leq n_B} c3[i_B]();$
if $i_B = i_{B\text{tested}}$ **then**
  new $na : \mathsf{nonce}; \overline{c4[i_B]}\langle \mathsf{enc}(\mathsf{pad}(na), pkA)\rangle;$
  $c5[i_B](= na); \mathbf{event}\ e_B(na); \mathbf{event}\ e'_B(na)$
else
  new $na : \mathsf{nonce}; \overline{c4[i_B]}\langle \mathsf{enc}(\mathsf{pad}(na), pkA)\rangle;$
  $c5[i_B](= na); \mathbf{event}\ e_B(na)$

without affecting the rest of the process. The query to prove becomes $\forall x : nonce; \mathbf{event}(e'_B(x)) \Rightarrow \mathbf{event}(e_A(x))$, using e'_B instead of e_B, so we prove the correspondence for the tested session, of index $i_B = i_{B\text{tested}}$. Then we distinguish whether the nonce na has been generated in the tested session or not, by renaming na to na_3 in the tested session and to na_2 in the others. Next, we apply the IND-CCA2 assumption on encryption, only in the tested session. That replaces the encryption of $\mathsf{pad}(na_3)$ with the encryption of a 0 block Zb, and adapts the decryption accordingly in A: when the message received by A is that encryption of Zb sent by B in the tested session, we replace the decryption with $\mathsf{pad}(na_3)$. Otherwise, we decrypt normally. Next, we insert a **find** just before e'_B that tests whether $e_A(na_3)$ has been executed; its **then** branch still executes e'_B while its **else** branch executes **event_abort** bad. That game differs from the previous one only when event bad is executed, and we are going to show that its probability is negligible. In that game, the correspondence $\forall x : nonce; \mathbf{event}(e'_B(x)) \Rightarrow \mathbf{event}(e_A(x))$ is proved since the **find** guarantees the execution of e_A with the appropriate argument. To show that the probability of bad is negligible, we use the command **success simplify**, which removes the output of na_3 in A (when na_3 is sent, $e_A(na_3)$ has been executed, so bad will not be executed). For that, the previous application of **guess** is crucial: the nonces na_2 for the other sessions are still output. Finally, a dependency analysis on na_3 shows that the adversary has no information on na_3, so the input $c5[i_B](= na_3)$ has a negligible probability of succeeding: the code that follows it is removed, which removes event bad and concludes the proof. ◁

In the **guess** transformation, we cannot modify correspondence queries with injective events, because, in case we prove the query only for the tested session, two executions of some injective event e with different indices i could be mapped to the same events in the conclusion of the query. As a counter-example, consider the query: $\forall i : [1, n], x : T'; \mathbf{event}(e_1(i, x)) \wedge \mathbf{inj\text{-}event}(e_2(x)) \Rightarrow \mathbf{inj\text{-}event}(e_3())$ with events $e_1(i_1, x_1)$, $e_1(i_2, x_2)$, $e_2(x_1)$, $e_2(x_2)$ and e_3 each executed once. This query is false: we have two executions of e_2 (with matching executions of e_1) for a single execution of e_3. That contradicts injectivity. However, the query is true if we restrict ourselves to one value of i (the index of the tested session), because we consider $e_1(i_1, x_1)$, $e_2(x_1)$ and e_3 for $i = i_1$ and $e_1(i_2, x_2)$, $e_2(x_2)$ and e_3 for $i = i_2$. Requiring the same value of x in e_1 and e_2 restricts the events e_2 that we consider when we guess the session for e_1. Therefore, proving the query for the tested session does not allow us to prove it in the initial game.

Hence, for correspondences with injective events $\forall \widetilde{x} : \widetilde{T}; \psi \Rightarrow \exists \widetilde{y} : \widetilde{T'}; \phi$, we proceed as follows: we define a non-injective correspondence $\mathrm{noninj}(\forall \widetilde{x} : \widetilde{T}; \psi \Rightarrow \exists \widetilde{y} : \widetilde{T'}; \phi)$ simply obtained by replacing injective events with non-injective events, and we try to prove that $\mathrm{noninj}(\forall \widetilde{x} : \widetilde{T}; \psi \Rightarrow \exists \widetilde{y} : \widetilde{T'}; \phi)$ implies $\forall \widetilde{x} : \widetilde{T}; \psi \Rightarrow \exists \widetilde{y} : \widetilde{T'}; \phi$ in the current game. If that proof succeeds, we just have to prove $\mathrm{noninj}(\forall \widetilde{x} : \widetilde{T}; \psi \Rightarrow \exists \widetilde{y} : \widetilde{T'}; \phi)$ and we can apply the **guess** transformation for non-injective correspondences. Otherwise, we simply leave the query $\forall \widetilde{x} : \widetilde{T}; \psi \Rightarrow \exists \widetilde{y} : \widetilde{T'}; \phi$ unchanged.

The proof that $\mathrm{noninj}(\forall \widetilde{x} : \widetilde{T}; \psi \Rightarrow \exists \widetilde{y} : \widetilde{T'}; \phi)$ implies $\forall \widetilde{x} : \widetilde{T}; \psi \Rightarrow \exists \widetilde{y} : \widetilde{T'}; \phi$ is a modified version of the proof of injective correspondences. Intuitively, it shows that, if we have different executions of injective events in ψ, then the arguments of each injective event in ϕ must be different, which implies different executions of this event. For this reason, the arguments of events in ϕ must be chosen so that they are actually different in this case; for instance, in Example 9 below, the presence of na as argument of events is essential for the proof to work. This idea generalizes the one used in [21, Lemma 1] to infer an injective correspondence from a non-injective one in the symbolic model, using an argument that appears both in injective events in ψ and ϕ, and such that this argument has a different value for each execution an injective event in ψ. Details are given in the full version [22, Section 5.1.17, Proposition 4].

Example 9: Continuing Example 8, we now want to prove correspondence (3) of Example 1. In the game before guessing i_B, distinct executions of e_B have distinct values of their argument na by eliminating collisions between nonces, so assuming (2), they correspond to executions of e_A also with distinct arguments, therefore distinct executions of e_A. Therefore, (2) implies (3). We prove (2) as in Example 8, and conclude that (3) also holds. ◁

CryptoVerif also supports other guessing transformations, to guess the value of a variable and the branch taken in a test. These transformations are detailed in the full version [22,

Sections 5.1.18 and 5.1.19].

D. Implementation details

Most game transformations are implemented as a separate module, implementing a function that takes as input the initial game and returns the transformed game, plus a description of details of what has been transformed, and for the exact security framework, the computation of the probability of success of an attack. The transformation **guess** i fits in this case. The transformation **success simplify** is implemented by modifying the command **success**, which proves security properties (Section IV), so that it collects information valid when the security properties are broken. This information is then used in a modified version of the transformation **simplify** to perform the actual transformation. The command **focus** is not a game transformation, it just modifies the active queries.

VI. Applications

In this section, we present applications of the new techniques and game transformations to practical case studies.

A. Forward Secrecy with respect to the Compromise of the Pre-shared Key in TLS 1.3 and WireGuard

In a previous verification of TLS 1.3 Draft 18 using CryptoVerif [16], the authors proved forward secrecy with respect to the compromise of signature keys, by allowing compromise in the model of the signature itself. However, they could not prove forward secrecy with respect to the compromise of the pre-shared key for the PSK-DHE handshake [16, Section VI.C]. Basically, that handshake derives a key by hashing together a pre-shared key (PSK) and a Diffie-Hellman shared secret coming from a Diffie-Hellman key exchange with ephemerals (DHE). We cannot include the compromise of the PSK in the model of a primitive, so our new approach is necessary. We prove this property by relying on the same technique as in our running example (Examples 4 to 6).

Using a similar approach, we prove forward secrecy with respect to the compromise of the pre-shared key in WireGuard, which could not be proved in [38, Section VI].

B. PRF-ODH with Compromise

The PRF-ODH assumption (pseudo-random function oracle Diffie-Hellman) [26] is an assumption on a combination of a Diffie-Hellman exponentiation and a hash function h. We write g for a generator of the Diffie-Hellman group. CryptoVerif supports two variants of PRF-ODH. The first one says that an adversary that has g^a and g^b for random a, b has a negligible probability of distinguishing $x \mapsto h(g^{ab}, x)$ from a random function. The second one is stronger: it additionally gives the adversary access to oracles $\mathsf{PRFDHa}(x, Y)$, which returns $h(Y^a, x)$ when $Y^a \neq g^{ab}$ and $\perp$ otherwise, and $\mathsf{PRFDHb}(x, X)$, which returns $h(X^b, x)$ when $X^b \neq g^{ab}$ and $\perp$ otherwise. (These oracles return $\perp$ when they would compute $h(g^{ab}, x)$, which the adversary tries to distinguish from random. When the Diffie-Hellman structure is a prime-order group, the condition $Y^a \neq g^{ab}$ is equivalent to $Y = g^b$. However, this is not true in structures like Curve25519 and Curve448 [37], see [39, Section 3.3]. The presence of these oracles allows one to prove the protocol secure even if the adversary can test whether its exponential Y or X is the expected exponential, for instance by seeing whether the session continues.) The first variant can be proved from CDH and the random oracle model, or from DDH and PRF (pseudo-random function); the second one from GDH and the random oracle model. The first variant is basically PRF-ODHnn in [26], the second one PRF-ODHmm.

The PRF-ODH assumption is a decisional assumption. Hence, in case an exponent a or b is compromised at some point, we cannot apply it for this exponent. Therefore, we cannot use the basic treatment of compromise to prove forward secrecy with respect to the compromise of some exponent; we need the new commands of Section V. We illustrate this point on the Noise NK protocol [42], which relies on a long-term static Diffie-Hellman key for authentication of the responder. We prove forward secrecy of the encrypted payloads from the third message with respect to the compromise of this key, assuming the second variant of PRF-ODH and using **focus** and **success simplify** as in our running example (Examples 4 to 6). In contrast, assuming GDH and the random oracle model, we prove the same property using the basic treatment of compromise: we can apply GDH when the static key is not compromised yet, because it is a computational property.

As other usages of PRF-ODH, we also considered TLS 1.3 and a variant of the Diffie-Hellman authenticated KEM (Key Encapsulation Mechanism) of HPKE (Hybrid Public-Key Encryption) [4], [6]; in these examples, the new commands **focus** and **success simplify** are needed only when we consider the dynamic compromise of the pre-shared key in TLS 1.3, as in Section VI-A, because the exponents are not compromised. (We leave compromised ephemeral exponents for future work.)

C. Forward Secrecy for OEKE

In the password-based key exchange OEKE [27], we prove forward secrecy with respect to the compromise of the password. This property was proved on paper in [27], but was not proved in the previous CryptoVerif study of this protocol [19]. This proof relies on the computational Diffie-Hellman (CDH) assumption. We recall that, given a group $\mathbb{G}$ of prime order q, with a generator g, an adversary succeeds against CDH when it computes g^{ab} knowing g^a and g^b for two random exponents $a, b \in [1, q-1]$. Our proof of forward secrecy relies on **focus** and **success simplify** as in our running example (Examples 4 to 6); it also uses the guessing transformations, to guess the tested session and its partner in order to apply the CDH assumption to a single pair of exponents a and b, and guess the result of a test $Z = X^b$ for that exponent b, so that this test is removed, which is needed to apply the CDH assumption to a and b since this assumption allows the adversary to know g^a and g^b but not the result of a test $Z = X^b$.

Model	#f	#c	runtime
VI-A TLS 1.3	1	34	1 min 39 s
WireGuard	12	45 to 47	4 h 40 min
VI-B NoiseNK with PRF-ODH	1	27	31 s
NoiseNK with GDH+ROM	1	16	19 s
TLS 1.3 with PRF-ODH			
- Key schedule lemmas	4	0	1 s
- Initial handshake	1	17	3 min 26 s
- PSK without corruption	1	0	13 min 32 s
- PSK with corruption	1	30	6 min 53 s
HPKE with PRF-ODH	3	34 to 53	30 s
VI-C OEKE	1	67	31 s
VI-D WireGuard	1	108	21 min 25 s

Fig. 3. Guidance and runtime for the applications

D. Grouping Compromise Scenarios

The initial verification of WireGuard using CryptoVerif [38] considers 4 corruption scenarios for the Diffie-Hellman exponents s_i, s_r (static keys for the initiator and responder) and e_i, e_r (ephemeral keys for the initiator and responder) in separate CryptoVerif files, allowing the compromise of all these keys without compromising both the ephemeral and static keys of the same principal. However, in practice, one does not know a priori which corruption scenario will happen, so we need to manually combine those results to get a security result valid in all scenarios. In this work, we mechanize this part of the proof: we group all these scenarios in a single file by corrupting either s_i or e_i (resp. s_r or e_r) depending on a boolean test and guessing which branch is taken. Then CryptoVerif gives us directly a single security result valid in all scenarios.

E. Guidance and Runtime

Figure 3 gives, for each application of this section, the number of CryptoVerif input files in column #f (separate files may be used for different properties, compromise scenarios, or protocol variants); the number of commands given to guide the proof in each input file in column #c, not counting the commands that just display the current game; and the total runtime of CryptoVerif in all input files. The examples that require the advanced treatment of compromise of Section V (examples of Sections VI-A and VI-C; NoiseNK with PRF-ODH, TLS 1.3 with pre-shared key (PSK) and corruption in Section VI-B) require fairly detailed guidance, with more commands than similar examples without compromise, basically because we combine two proofs, the proof of authentication before compromise, and the proof of the other properties. The guidance is still much lighter than in tools like EasyCrypt, FCF, and CryptHOL because CryptoVerif generates the cryptographic games. When the number of commands is 0, the proof is fully automatic. The runtime for WireGuard in Section VI-A is specially high because it considers the compromise of the pre-shared key in a variety of other compromise scenarios and two variants of the protocol. In Section VI-D, we do not consider the compromise of the pre-shared key and consider a single variant of the protocol. The higher number of commands in this example comes from grouping the 4 compromise scenarios: the commands for each scenario have to be included.

VII. Conclusion

CryptoVerif now provides two proof strategies in order to deal with dynamic key compromise. The basic one (Section II) consists in including the compromise of the key in the specification of cryptographic primitive itself. It allows us to deal with dynamic key compromise without modifying the tool CryptoVerif itself, but it has limitations, detailed in Section II, that the second strategy removes. The advanced strategy (Section V) relies on new commands and game transformations: it consists in first focusing on the proof of authentication properties, for which the compromise of the key can be removed, and then using these properties in order to prove forward secrecy in the presence of dynamic compromise. This second strategy is illustrated in detail in Examples 4 to 6. This strategy allowed us to fill gaps in previous case studies, proving in particular the forward secrecy with respect to the compromise of the pre-shared key in the PSK-DHE handshake of TLS 1.3 Draft 18 and in the VPN protocol WireGuard. The new game transformations are general enough that they can also have other applications than dynamic compromise. In particular, guessing the tested session is a common step in cryptographic proofs and will be useful in other contexts.

CryptoVerif still has limitations. In particular, the size of games tends to grow too fast, which limits its ability to deal with large examples, especially because, when there is a test, the code is duplicated from the test until the end of protocol. Planed improvements include allowing game transformations to work without duplicating code in this way; allowing internal oracle calls in games, in order to share code between different parts of the game; using composition results in order to make proofs more modular. Moreover, some game transformations could be generalized. For instance, a transformation merges branches of a test when they execute the same code; the detection that several branches execute equivalent code could be made more flexible, by allowing reorderings of instructions for instance. CryptoVerif only considers blackbox adversaries: it does not support proofs that manipulate the code of the adversary, such as the forking lemma [45].

Acknowledgments: We thank Charlie Jacomme for helpful comments on a draft of this paper and David Pointcheval for help regarding the proof of forward secrecy of OEKE. This work benefited from funding managed by the French National Research Agency under the France 2030 programme with the reference ANR-22-PECY-0006 (PEPR Cybersecurity SVP).

References

[1] 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, Mar. 2006.

[2] M. R. Albrecht, K. G. Paterson, and G. J. Watson, "Plaintext recovery attacks against SSH," in *IEEE S&P'09*. IEEE, May 2009, pp. 16–26.

[3] J. B. Almeida, M. Barbosa, G. Barthe, M. Campagna, E. Cohen, B. Grégoire, V. Pereira, B. Portela, P.-Y. Strub, and S. Tasiran, "A machine-checked proof of security for AWS key management service," in *CCS'19*. ACM, Nov. 2019, pp. 63–78.

[4] J. Alwen, B. Blanchet, E. Hauck, E. Kiltz, B. Lipp, and D. Riepel, "Analysing the HPKE standard," in *Eurocrypt 2021*, ser. LNCS, A. Canteaut and F.-X. Standaert, Eds., vol. 12696. Springer, Oct. 2021, pp. 87–116.

[5] D. Baelde, S. Delaune, A. Koutsos, C. Jacomme, and S. Moreau, "An interactive prover for protocol verification in the computational model," in *IEEE S&P'21*. IEEE, May 2021, pp. 537–554.

[6] R. L. Barnes, K. Bhargavan, B. Lipp, and C. A. Wood, "Hybrid Public Key Encryption," Internet Research Task Force (IRTF), RFC 9180, Feb. 2022. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9180.html

[7] G. Barthe, J. M. Crespo, Y. Lakhnech, and B. Schmidt, "Mind the gap: Modular machine-checked proofs of one-round key exchange protocols," in *Eurocrypt 2015*, ser. LNCS, E. Oswald and M. Fischlin, Eds., vol. 9057. Springer, Apr. 2015, pp. 689–718.

[8] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, "Computer-aided security proofs for the working cryptographer," in *CRYPTO 2011*, ser. LNCS, P. Rogaway, Ed., vol. 6841. Springer, Aug. 2011, pp. 71–90.

[9] D. Basin and C. Cremers, "Modeling and analyzing security in the presence of compromising adversaries," in *ESORICS'10*, ser. LNCS, vol. 6345. Springer, 2010, pp. 340–356.

[10] D. Basin, A. Lochbihler, and S. R. Sefidgar, "CryptHOL: Game-based proofs in higher-order logic," *Journal of Cryptology*, vol. 33, pp. 494–566, 2020.

[11] D. A. Basin and C. J. Cremers, "Degrees of security: Protocol guarantees in the face of compromising adversaries," in *CSL'10*, ser. LNCS, vol. 6247. Springer, 2010, pp. 1–18.

[12] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway, "Relations among notions of security for public-key encryption schemes," in *CRYPTO 1998*, ser. LNCS, H. Krawczyk, Ed., vol. 1462. Springer, Aug. 1998, pp. 26–45.

[13] M. Bellare and C. Namprempre, "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm," in *ASIACRYPT'00*, ser. LNCS, T. Okamoto, Ed., vol. 1976. Springer, Dec. 2000, pp. 531–545.

[14] M. Bellare and P. Rogaway, "The security of triple encryption and a framework for code-based game-playing proofs," in *Eurocrypt 2006*, ser. LNCS, S. Vaudenay, Ed., vol. 4004. Springer, May 2006, pp. 409–426, extended version available at http://eprint.iacr.org/2004/331.

[15] L. Beringer, A. Petcher, K. Q. Ye, and A. W. Appel, "Verified correctness and security of OpenSSL HMAC," in *24th Usenix Security Symposium*. USENIX Association, Aug. 2015, pp. 207–221.

[16] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the TLS 1.3 standard candidate," in *IEEE S&P'17*. IEEE, May 2017, pp. 483–503.

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

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

[19] ——, "Automatically verified mechanized proof of one-encryption key exchange," in *CSF'12*. IEEE, Jun. 2012, pp. 325–339.

[20] ——, "Modeling and verifying security protocols with the applied pi calculus and ProVerif," *Foundations and Trends in Privacy and Security*, vol. 1, no. 1–2, pp. 1–135, Oct. 2016.

[21] ——, "Symbolic and computational mechanized verification of the ARINC823 avionic protocols," in *CSF'17*. IEEE, Aug. 2017, pp. 68–82.

[22] ——, "Cryptoverif: a computationally-sound security protocol verifier (initial version with communication on channels)," Inria, Research report RR-9525, Oct. 2023, available at https://inria.hal.science/hal-04246199.

[23] B. Blanchet, M. Abadi, and C. Fournet, "Automated verification of selected equivalences for security protocols," *Journal of Logic and Algebraic Programming*, vol. 75, no. 1, pp. 3–51, Feb.–Mar. 2008.

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

[25] I. Boureanu, C. C. Drăgan, F. Dupressoir, D. Gérault, and P. Lafourcade, "Mechanised models and proofs for distance-bounding," in *CSF'21*. IEEE, 2021.

[26] J. Brendel, M. Fischlin, F. Günther, and C. Janson, "PRF-ODH: Relations, instantiations, and impossibility results," in *CRYPTO 2017*, ser. LNCS, vol. 10403. Springer, Aug. 2017, pp. 651–681.

[27] E. Bresson, O. Chevassut, and D. Pointcheval, "Security proofs for an efficient password-based key exchange," in *CCS '03*, V. Atluri, Ed. ACM, 2003, pp. 241–250, we refer to the long version, available at https://eprint.iacr.org/2002/192.pdf.

[28] D. Butler, D. Aspinall, and A. Gascón, "Formalising oblivious transfer in the semi-honest and malicious model in CryptHOL," in *CPP'20*. ACM, Jan. 2020, pp. 229–243.

[29] D. Butler, A. Lochbihler, D. Aspinall, and A. Gascón, "Formalising σ-protocols and commitment schemes using CryptHOL," *Journal of Automated Reasoning*, vol. 65, no. 4, pp. 521–567, Apr. 2021.

[30] V. Cortier, C. C. Drăgan, F. Dupressoir, B. Schmidt, P.-Y. Strub, and B. Warinschi, "Machine-checked proofs of privacy for electronic voting protocols," in *IEEE S&P'17*. IEEE, 2017, pp. 993–1008.

[31] C. J. Cremers, "The Scyther Tool: Verification, falsification, and analysis of security protocols," in *CAV'08*, ser. LNCS, vol. 5123/2008. Springer, 2008, pp. 414–418.

[32] J. A. Donenfeld, "WireGuard: Next generation kernel network tunnel," in *NDSS'17*, 2017, we use the whitepaper version for our analysis, which differs in how the MACs are defined: https://www.wireguard.com/papers/wireguard.pdf, Nov. 2nd, 2017, draft revision ceb3a49.

[33] J. Gancher, S. Gibson, P. Singh, S. Dharanikota, and B. Parno, "OWL: Compositional verification of security protocols via an information-flow type system," in *IEEE S&P'23*. IEEE, May 2023, pp. 1114–1131.

[34] S. Goldwasser, S. Micali, and R. Rivest, "A digital signature scheme secure against adaptative chosen-message attacks," *SIAM Journal of Computing*, vol. 17, no. 2, pp. 281–308, Apr. 1988.

[35] N. Kobeissi, K. Bhargavan, and B. Blanchet, "Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach," in *EuroS&P'17*. IEEE, Apr. 2017, pp. 435–450.

[36] R. Küsters and D. Rausch, "A framework for universally composable Diffie-Hellman key exchange," in *IEEE S&P'17*. IEEE, May 2017, pp. 881–900.

[37] A. Langley, M. Hamburg, and S. Turner, "Elliptic curves for security," Internet Requests for Comments, RFC Editor, RFC 7748, Jan. 2016. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7748.html

[38] B. Lipp, B. Blanchet, and K. Bhargavan, "A mechanised cryptographic proof of the WireGuard virtual private network protocol," in *EuroS&P'19*. IEEE Computer Society, Jun. 2019, pp. 231–246.

[39] ——, "A mechanised cryptographic proof of the WireGuard virtual private network protocol," Inria, Research report RR-9269, Apr. 2019, available at https://hal.inria.fr/hal-02100345.

[40] G. Lowe, "Breaking and fixing the Needham-Schroeder public-key protocol using FDR," in *TACAS'96*, ser. LNCS, vol. 1055. Springer, 1996, pp. 147–166.

[41] T. Okamoto and D. Pointcheval, "The gap-problems: a new class of problems for the security of cryptographic schemes," in *PKC'2001*, ser. LNCS, K. Kim, Ed., vol. 1992. Springer, Feb. 2001, pp. 104–118.

[42] T. Perrin, "The Noise protocol framework," Jul. 2018, https://noiseprotocol.org/noise.html.

[43] A. Petcher and G. Morrisett, "The foundational cryptography framework," in *POST'15*, ser. LNCS, R. Focardi and A. C. Myers, Eds., vol. 9036. Springer, Apr. 2015, pp. 53–72.

[44] ——, "A mechanized proof of security for searchable symmetric encryption," in *CSF'15*. IEEE, Jul. 2015, pp. 481–494.

[45] D. Pointcheval and J. Stern, "Security proofs for signature schemes," in *Eurocrypt 1996*, ser. LNCS, U. Maurer, Ed., vol. 1070. Springer, May 1996, pp. 387–398.

[46] E. Rescorla, "The Transport Layer Security (TLS) protocol version 1.3, draft-ietf-tls-tls13-18," https://tools.ietf.org/html/draft-ietf-tls-tls13-18, Oct. 2016.

[47] ——, "The Transport Layer Security (TLS) protocol version 1.3," RFC 8446, https://www.rfc-editor.org/rfc/rfc8446, Aug. 2018.

[48] B. Schmidt, S. Meier, C. Cremers, and D. Basin, "Automated analysis of Diffie-Hellman protocols and advanced security properties," in *CSF'12*. IEEE, Jun. 2012, pp. 78–94.

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

[50] A. C. Yao, "Theory and applications of trapdoor functions," in *FOCS'82*. IEEE, Nov. 1982, pp. 80–91.