FORSAKES: A Forward-Secure Authenticated Key Exchange Protocol Based on Symmetric Key-Evolving Schemes

Mohammad Sadeq Dousti and Rasool Jalili

Data & Network Security Lab, Department of Computer Engineering, Sharif University of Technology, Tehran, Iran

March 1, 2014

Abstract

This paper suggests a model and a definition for forward-secure authenticated key exchange (AKE) protocols, which can be satisfied without depending on the Diffie-Hellman assumption. The basic idea is to use key-evolving schemes (KES), where the long-term keys of the system get updated regularly and irreversibly. Protocols conforming to our model can be highly efficient, since they do not require the resource-intensive modular exponentiations of the Diffie-Hellman protocol. We also introduce a protocol, called FORSAKES, and prove rigorously that it is a forward-secure AKE protocol in our model. FORSAKES is a very efficient protocol, and can be implemented by merely using hash functions.

Keywords. Authenticated Key Exchange Protocol, Forward Security, Key Evolving Schemes, Provable Security, Security Model.

1 Introduction

Establishing *secure channels* is a prominent problem in secure networks. Transmitting confidential data, or ensuring its integrity over a public channel is impossible without secure channels. While communicating parties can utilize their long-term keys (LTKs) for confidentiality or integrity, it is generally considered a bad practice. The standard approach is to first exchange *ephemeral keys*, and then use these keys to establish secure channels.

On the surface, designing efficient and secure authenticated key-exchange (AKE) protocols does not seem to be a complicated task. However, the history shows otherwise. One of the earliest key exchange protocols, the Needham–Schroeder protocol [1], was soon found to have a subtle flaw [2]. Another example is the Otway–Rees [3] protocol, which is shown to be susceptible to *typing attacks* [4, 5]. In another work, Bird *et al.* [6] formulated an attack called *parallel session attack*, and showed that several protocols were vulnerable to it. In particular, the attack was successfully mounted on several proposed ISO-9798 protocols. It is interesting to know that, after over 20 years, the ISO-9798 family of protocols is still open to some attacks [7]. A survey of attacks against entity authentication and AKE protocols can be found in [8, 9].

One of the main reasons for the existence of such attacks is the concurrent nature of AKE protocols. The designers of vulnerable AKE protocols often fail to account for all possible scenarios which can happen in a concurrent setting. A more important reason, identified by cryptographers after a while, was the lack of a proper model and definition for AKE protocols.

A *model* specifies the resources (time, space, etc.) available to each party and the adversary, the way they communicate, and the special abilities at the adversary's disposal (capturing all possible

attacks in a general way). A *definition* specifies what it means for the protocol under consideration to be *secure* in the model. As soon as the model and definition are determined, the security of many protocols can be formally proven or refuted.

While providing a certain level of assurance, a security proof is not the panacea. In particular:

- Proving security theorems and verifying the proofs are daunting tasks, and often prone to errors themselves;
- It is possible that the model fails to capture certain attacks, or the definition is inadequate for some security requirements;
- The proofs are sometimes misinterpreted. For instance, an *asymptotic* proof of security might be of limited significance in practice, where *concrete* proofs of security are needed [10].

A prime example of the first two items is apparent in the work of Krawczyk [11]: He puts forward a general model for AKE protocols, as well as a security definition. Based on this model/definition, an efficient AKE protocol called MQV [12,13] is analyzed, and several security flaws are detected. MQV is then updated into an improved version, called HMQV. Finally, the security of HMQV is proven formally. However, Menezes [14] points out to a few mistakes in the proof, as well as the failure of the model to capture several practical attacks. Subsequently, Krawczyk added an appendix to the full version of his paper [11], discussing how to evade such flaws. His model has since been a de facto standard in designing provably secure AKE protocols (see Section 1.2 for a survey).

The bottom line is that one must be careful while proposing a new security model or definition, and should take every possible measure to write sound security proofs. That said, a provably secure AKE protocol is certainly preferable than one which is designed and analyzed in an ad hoc manner, due to the delicate nature of these protocols, as discussed above. For this reason, researchers started to propose models and definitions for AKE protocols in different settings, for which ad hoc protocols already existed. Some of these settings are as follows: Two-party setting, three-party setting, public-key setting, group key exchange, and password-based AKE. We survey these settings in Section 1.2.

In this paper, we put forward a model and definition for a class of AKE protocols, which were previously designed ad hoc, and then prove the security of our proposed protocol. Informally, this class contains *lightweight* AKE protocols which provide *forward security*. By lightweight, we mean the protocols which do not use heavy operations such as modular exponentiation. Forward security, also called *perfect forward secrecy* (PFS), is an important property of AKE protocols. It was first defined by Günther [15], used famously in the Station-to-Station protocol (STS) [16], and formalized by [11,17–19]. This concept is defined informally below:

Informal Definition. An AKE protocol is said to be *forward secure* if, even if the long-term keys (LTKs) are revealed to the adversary, the ephemeral keys generated prior to the exposure of the LTKs remain protected from the adversary.

Most (if not all) provably-secure AKE protocols satisfying the forward security property use a variant of the Diffie-Hellman (DH) protocol [20], which requires the *heavy* modular exponentiations. On the other hand, forward secure and *lightweight* AKE protocols are often designed *ad hoc*. The basic idea of such protocols is not to use DH, but to modify the LTKs regularly.

Unfortunately, ad hoc protocols are often prone to attacks. A famous "ultra lightweight" AKE protocol, called SASI [21], provides a good example. SASI attempts to provide forward security without depending on DH, but is found to be flawed by several researchers [22–24]. While proven insecure, SASI (and similar protocols) follow a remarkable approach to forward security: They update the LTKs regularly and irreversibly. If the adversary gets hold of an updated LTK K_{new}, she will be unable to infer the previous LTK K_{old}, as doing so requires inverting the one-way function. Since the ephemeral keys depend essentially on K_{old}, it must be impossible for the adversary to obtain the ephemeral keys from K_{new}. We will use the term *Key-Evolving Schemes* (KES) for protocols which

update their LTKs at specific occasions. In this paper, we are interested in symmetric KES protocols, where each pair of parties may have a pre-shared symmetric key.

Types of KES. The literature distinguishes two types of KES:

1. LTKs are updated after a *specific event* in the system. For instance, AKE protocols can update the LTK upon the exchange of each ephemeral key. This is the approach that SASI and many other AKE protocols follow.
2. LTKs are updated at *specific time intervals*. A good example of this approach is RSA SecurID® (for description and analysis, see [25–28]). This technique is also used in forward-secure public-key encryption [29], though the latter uses a DH-based construction.

Updating LTKs after each key exchange provides forward security in its entirety. However, it is only suitable for the case of smart cards or security tokens, where the device performs no more than one instance of an AKE protocol at each moment. Consider what happens if there are multiple instances of the AKE protocol between two parties, and one instance is completed: The LTKs will get updated, which in turn renders all other instances of the AKE protocol invalid. Therefore, the first solution is not viable for networks such as the Internet, where each party might run several concurrent instance of the AKE protocol with another party (for instance, consider the communications between two routers, or two security gateways).

The second approach, which we adopt in this paper, is more promising for concurrent settings. However, it only satisfies the forward security property partially.[1] As explained in [29], the lifetime of the system is divided into many time stages. At the beginning of each time stage, the LTKs get updated. If the adversary gets hold of an LTK K in time stage T, she must be unable to find the ephemeral keys generated using K in time stage $T-1$ or earlier. On the other hand, the security of ephemeral keys generated using K in time stage T is not guaranteed. The faster the LTKs get updated, the more forward security is satisfied. However, if the update frequency is too high, there will be no time for the actual key exchange to take place. Therefore, the update frequency should be set reasonably to prevent loss of functionality while preserving the forward security. Depending on the unique requirements of the system, update frequencies of once per minute, once per hour, or once per day might be appropriate. However, an update frequency of once per *second* seems inadequate.

Another issue which might arise is the use of time in security protocols. It is important to note that the precise value of the time is of no significance to our model, as we will not deal with time stamps. The important issue is to keep the the time synchrony between the parties, such that they perceive the same *time stage* all the time. The current technology allows us to produce devices, such as the RSA SecurID®, which can keep the time synchrony for a long time. Furthermore, when the update frequency decreases (say, once a day), an asynchrony of a few seconds might be acceptable.

1.1 Contributions

The contributions of this paper are as follows:

- We put forward a new model for AKE protocols, where the LTKs evolve over time. We also define what it means for an AKE protocol to be secure in our model, satisfying the forward security property.
- Our model/definition has an algorithmic flavor. That is, we first explain the concept in plain English, and then try to algorithmically describe it. The significance of this approach is that our definitions can be incorporated into tools for automatic verification of security protocols.

[1] Note that SecurID® uses a hybrid approach: It updates LTKs on a regular basis, but allows only a single use of the LTK in each time interval. If a second authentication is required, the user should wait for the next time interval. Therefore, SecurID® provides full forward security, at the cost of being unable to handle concurrent authentications.

- We design an AKE protocol, and rigorously prove its security within the model and according to the above definition. The protocol is aptly named FORSAKES, because it is a Forward Secure AKE based on KES. (Recall that AKE stands for *Authenticated Key Exchange*, and KES stands for *Key-Evolving Scheme*.)
- FORSAKES is designed in the random oracle model (Section 2) without any assumptions. Most AKE protocols depend on some cryptographic assumption, even if they use the random oracle model. For instance, [11, 18, 30] are proven secure in the random oracle model, but use some form of the Diffie-Hellman assumption as well (e.g., Decisional Diffie-Hellman or Gap Diffie-Hellman).
- Since we prove the security of FORSAKES without any assumptions, it is *unconditionally secure* in the random oracle model. In other words, there is no restriction on the running time of the adversary, and the security proof holds even for infinitely powerful adversaries. (While the adversary can have infinite running time, she is not free to make as many query as she likes to the system or the random oracle. See the proofs for more information.)
- We use the random oracle *model* only to simplify the proofs. However, since we do not use the facilities provided by the random oracle model (such as viewing or programming the adversary's queries), it is possible to replace the random oracle with a *pseudorandom function* [31], to achieve a secure AKE protocol in the *standard model*. See Section 6 for more information.
- While there are efficient and provably secure pseudorandom function [32], using the random oracle heuristic and replacing all instances of the random oracle with a hash function such as SHA-1 yields a more practical protocol. In the case of FORSAKES, we can obtain a protocol which uses nothing but hash functions, which is particularly suitable for constrained devices such as smart cards or tokens. See Section 6 for more information.

1.2 Related Work

Early papers on secure authentication were either ad hoc, or adopted the zero-knowledge model of Feige, Fiat, and Shamir [33], which was suitable only for smart-card identification. In this paper, we are not concerned with such approach. A comprehensive account of papers on zero-knowledge identification can be found in [34, Section 2.2].

Formalization of a model and definition for both entity authentication and AKE protocols started with the seminal work of Bellare and Rogaway [35]. They recognized that each party can take part in multiple instances of the protocol (each of which was called a *session*), and modeled each session as an oracle, to which the adversary could make three types of queries: **send**, **reveal**, and **test**. A **send** query allowed the adversary to deliver a specific message to some session of her choice, and observe the result. A **reveal** query gave the adversary the session key of a specific session. A **test** query flipped a coin, and based on the result, gave the adversary either a random value or the session key of the target session. The goal of the adversary was to *distinguish* whether she is given the random value, or the actual session key.

The ideas of [35] were notable is several ways: (1) They defined the security of AKE protocols based on a distinguishability game. (2) They defined a notion of *session freshness*. If the adversary makes a **reveal** query to the target session before or after the **test** query, she can obtain the session key, and distinguish whether she is given a random value or the actual session key. Therefore, a revealed session is no longer *fresh*. This is also the case if the "partner session" of the target session is revealed. Therefore, there was a need to define *partnership*. (3) They defined partnership via the concept of *matching conversations*. It simply means that two sessions are partners if every message that one sends is received by the other, and vice versa (except possibly the last message, which can always be deleted by the adversary, without being detected by the sender).

Notice that [35] modeled the two-party authentication/AKE based on symmetric keys. The present paper uses a similar model. Furthermore, [35] uses random oracles in their proofs, similar to the proofs

of this paper. However, the AKE definition of [35] does not support the concept of *forward security*. Actually, their model does not allow the adversary to obtain the long-term keys at all.

It is notable that Blake-Wilson *et al.* [17, 36] adapted the Bellare–Rogaway model [35] for the case of *asymmetric* keys. Accordingly, their definition was updated to recognize the forward security property, as well as protection against new types attacks which are meaningful only in an asymmetric setting, such as *Key-Compromise Impersonation* (KCI) or *Unknown Key-Share* (UKS) attacks.

Another change of model was due to Bellare and Rogaway [37] themselves: They proposed a model for the case of three-party AKE, which is the setting used in the Needham–Schroeder protocol [1]. Their model supported a new type of query, **corrupt**, which allowed the adversary to obtain, and even set the long-term keys. An important achievement of their work was to differentiate between the notions of mutual authentication and authenticated key exchange, and to provide a protocol called 3PKD, which only satisfied the latter.

The notion of partnership in [37] was modeled via an "existentially guaranteed partnering function," a notion that the authors called *unintuitive* later [18]. Furthermore, the specific partnering function used in the proof of security of 3PKD was later found to be flawed [38].

Shoup and Rubin [39] changed the three-party AKE model of [37] by adding yet another type of query, **access**. This query allowed modeling a virus on the host machine, accessing the smart card which contained the long-term key.

Several later papers, like [40, 41], tried to model AKE protocols using the notion of *simulatability*, rather than *indistinguishability*. The former approach is common in modeling secure multiparty computations. This approach was criticized by Canetti and Krawczyk as being "over-restrictive", since "it ruled out protocols that seem to provide sufficient security" [19].

Bellare, Pointcheval, and Rogaway [18] proposed yet another model for AKE protocols, this time formalizing password-based AKE (PAKE) protocols. Passwords are considered low-entropy secrets, and the model and security definition should prevent offline dictionary attacks against PAKE protocols. The also utilized the notion of *session identifiers* as a means of defining partnership, and took an *algorithmic approach* towards formalizing the model and definition. In this paper, we will follow a similar algorithmic approach.

Canetti and Krawczyk [19] used a technique introduced in [40] to define the AKE security in an indistinguishability framework: They considered two models, one which provided authenticated channels, while the other did not provide such luxury. The technique was to propose a general *compiler*, to convert any protocol secure in the former model to one that was secure in the latter model. The model proposed in [19] is generally called the CK model (CK stands for the initials of the authors). The CK model is suitable for two-party AKE in an asymmetric (public-key) setting. It has several important features:

- The CK model allows for modular design of AKE protocols. The protocol designer can design a protocol in the *authenticated channel* model, prove its security, and then easily compile the protocol into one which is secure in the *unauthenticated channel* model.
- The authors formalized the notion of *secure channels*, and showed that an AKE protocol proven secure in their AKE model could be *composed* with any protocol proven secure in their secure channel model. The composability theorem shows that any CK-secure AKE protocol can be used to establish a CK-secure channel. Similarly, the composability of the Bellare–Rogaway model [35] with secure channel protocols was later established [42]. See also [43] for another composable model.
- The CK model was to become later known as the de facto standard in AKE security models. Many other models can be considered as a variant of the CK model.

Canetti and Krawczyk proposed yet another security model for AKE protocols [44], this time in the broader context of the *Universal Composability* (UC) framework [45]. Any protocol proven secure in the UC framewok can be composed with other protocols, even insecure ones. This is important

for the Internet protocols, where the protocol designer cannot guarantee that the protocols executed concurrently with his protocol are all secure. It was later shown that the UC-definition of AKE protocols is flawed [46], and the flaw was corrected accordingly. A model for password-based AKE protocols in the UC framework was later proposed [47, 48] as well.

In his famous paper, Krawczyk [11] proposed an improvement to the CK model, which became known as the CK$^+$ model. In this model, the adversary was allowed to reveal party or session information via three different types of queries: **state-reveal** queries, **session-key** queries, and party **corruption** queries. These types of queries provide the adversary with a great flexibility.

LaMacchia, Lauter, and Mityagin [30] spotted several weaknesses in the CK and CK$^+$ models. Specifically, the CK model does not specify the precise result of the **state-reveal** queries: "An important point here is what information is included in the local state of a session; this is to be specified by each KE protocol" [19]. Furthermore, the security definition in both models does not allow an adversary to use his full potential. Therefore, LaMacchia *et al.* put forward a new security model/definition, called the *extended CK* (eCK). The eCK model replaced the **state-reveal** query of CK/CK$^+$ with an **Ephemeral Key Reveal** query. The latter query is specific to Diffie-Hellman protocols, and allows the adversary to reveal the private exponent α in a $g^\alpha \bmod p$ flow of the Diffie-Hellman protocol. The security definition of LaMacchia *et al.* considered a session fresh, if either the long-term key or the private exponent of that session is revealed, but not both (and the same should hold for the partner session). This definition greatly increased the power of the adversary, and for two years it was assumed to be the strongest AKE model/definition. However, Cremers [49] showed that the **state-reveal** query of CK/CK$^+$ is stronger than the **Ephemeral Key Reveal** query of eCK, and therefore the models are incomparable. Cremers used the vagueness in the definition of the **state-reveal** query to reveal the *intermediate results* of Diffie-Hellman computations in the NAXOS protocol of LaMacchia *et al.*, thus showing that NAXOS is insecure in the CK/CK$^+$ model.

In 2010, Sarr *et al.* [50] tried to amplify the eCK model, by proposing another model in which the adversary could reveal the intermediate results. Their model considered an implementation approach where the private exponents of the Diffie-Hellman protocol were computed during the idle time of a machine (i.e., before the protocol), and were stored in the RAM, which is considered an insecure storage. They called their model *strengthened eCK* (seCK).

One year later, Yoneyama and Zhao [51] showed a flaw in the security proofs of Sarr *et al.* [50], and concluded that achieving secure protocols in the seCK model is very hard, if not impossible.

The above survey shows the delicate nature of proposing security models and definitions for AKE protocols. The reader interested in further comparison of these models and definitions can consult [9, 49, 52–54].

1.3 Organization

The rest of this paper is organized as follows: Section 2 defines the concepts and notation used throughout this paper. Section 3 presents the FORSAKES protocol. In Section 4, we put forward our new security model and definition for AKE protocols with a symmetric key-evolving scheme. Section 5 provides a rigorous proof of the security of FORSAKES according to the model/definition presented in Section 4. Section 6 discusses the issues regarding the implementation of FORSAKES in practice. Finally, Section 7 concludes the paper, and explains the future work.

This paper has an appendix, A, where we prove the security of a message authentication code (MAC) based on random oracles. This proof is incorporated in the security proof of FORSAKES, in Section 5.

2 Preliminaries

For $c \in \mathbb{N}$, let $[c]$ denote the set $\{1, 2, \ldots, c\}$. For a finite set S, the notion $e \leftarrow_R S$ means that the element e is picked from S randomly.

Let $\binom{n}{k}$ denote the number of k-subsets of an n-set. By convention, $\binom{n}{k} = 0$ if $k > n$. Otherwise, $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

We use $n \in \mathbb{N}$ as the security parameter, meaning that the resources (time, space, etc.) available to all parties and algorithms are measured in n. As is customary in cryptography, n will be provided to the algorithms in *unary notation* 1^n (i.e., 1 is repeated n times). This is because the resources are actually accounted based on the *length* of the input.

Throughout the paper, we use four functions $\ell, r, k, K \colon \mathbb{N} \to \mathbb{N}$, where $\ell(n)$ is the length of entity identifiers in the system, $r(n)$ is an upper bound on the number of random bits used by each entity, $k(n)$ is the length of ephemeral keys, and $K(n)$ is the length of long-term keys. We assume that the exists a polynomial $p \colon \mathbb{N} \to \mathbb{N}$, such that $n \leq \ell(n), r(n), k(n), K(n) \leq p(n)$ for all $n \in \mathbb{N}$. When the context is clear, we may drop the parameter n. For instance, we may simply write r instead of $r(n)$.

Let $\{0,1\}^*$ denote the set of all finite binary strings, and $\{0,1\}^n$ denote the set of all binary strings of length n. The length of a string $x \in \{0,1\}^*$ is denoted by $|x|$. The special symbol $\lambda \in \{0,1\}^*$ denotes the empty string, i.e., $|\lambda| = 0$. For two strings $x, y \in \{0,1\}^*$, let $x \,||\, y$ denote the concatenation of x and y. The result of the concatenation of λ with any string is the string itself. Another special symbol is $*$. It is called the *wildcard* symbol, and matches a single bit. For instance, the result of the comparison $1{*}{*}0 = 1010$ is true.

We may occasionally assign 0 to a Boolean value to denote that it is logically false. Similarly, 1 is to be interpreted as logical true.

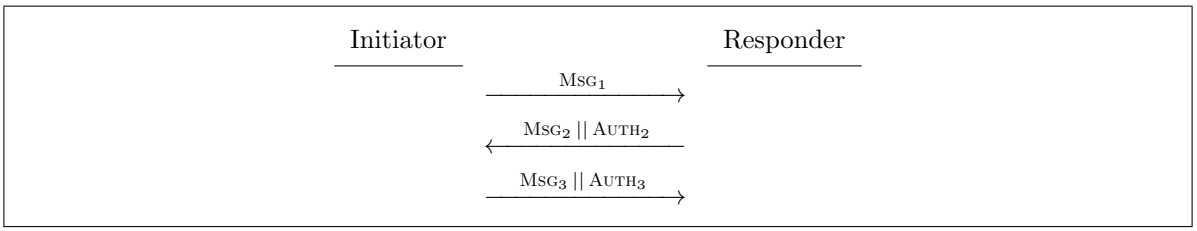
A function $f \colon \mathbb{N} \to \mathbb{R}$ is called *negligible* if it vanishes faster than the inverse of any positive polynomial. That is, $f(n) < n^{-c}$ for all c and all sufficiently large $n \in \mathbb{N}$.

We use the *random-oracle model* (ROM) [55], where all parties, including the adversary, have access to a *random* function $\mathcal{O} \colon \{0,1\}^* \to \{0,1\}^{k(n)}$. Once $\mathcal{O}$ is queried on some value $x \in \{0,1\}^*$ for the first time, a random $k(n)$-bit value y is chosen and returned. Notice that y is independent of x and the identity of the entity making the query. From this point on, $\mathcal{O}$ will return y if it is queried again on x. It is instrumental to think of $\mathcal{O}$ as an *ideal* hash function, having properties such as one-wayness and collision resistance. The ROM will greatly simplify the security proofs, but as discussed in Section 6, we can replace $\mathcal{O}$ with pseudorandom functions.

Convention. In writing the pseudocodes, we will assume *minimal evaluation*: In conditional statement, the conjuncts or disjuncts are evaluated from left to right. As soon as the validity of the statement is either proven or refuted, the remaining conjuncts or disjuncts are ignored. For instance, consider the statement `if(a and b)`: Assuming `a` is false, the value of `b` is ignored. The *minimal evaluation* allows us to write shorter pseudocodes. For instance, in the above example, the variable of `b` may be undefined unless `a` is true. If we had not used the minimal evaluation convention, we should have written two separate **if** statements: The first one evaluated `a`, and only if it were true, the value of `b` was evaluated.

3 The FORSAKES Protocol

To better understand the theoretical foundations of the AKE security model and definition presented in the next section, let us first describe the FORSAKES protocol, which is more practically oriented. Protocol 1 illustrates FORSAKES on a high level.



Protocol 1. The proposed AKE protocol. Details are described in Section 3.

System Time Stage. Let T be a variable denoting the current *time stage* of the system. T is assumed to be 1 in the system's *epoch*[2], and is incremented every τ seconds. In this paper, we picked T to be a *64-bit unsigned integer*. Therefore, our system supports $2^{64} \approx 10^{20}$ different time stages. However, this choice is quite conservative, and integers with much shorter bit lengths are usually appropriate as well.

Long-Term Keys. Suppose that FORSAKES initiator has identifier id_x, and the responder has identifier id_y. Moreover, assume that they had shared a long-term key K_{xy}^{θ} in time stage $T = \theta$. This key is updated whenever the time stage is incremented. Let $K_{xy}^{\theta+1}$ denote the new key. In FORSAKES, we have $K_{xy}^{\theta+1} \leftarrow \mathcal{O}(K_{xy}^{\theta})$. Notice that the random oracle $\mathcal{O}$ can be considered as an ideal hash function, which in particular, is ideally one-way. Therefore, $K_{xy}^{\theta+1}$ reveals no information about K_{xy}^{θ}.

Session State. As pointed out earlier, FORSAKES supports many concurrent key exchanges to run at a time. Each instance is called a *session*. The session information, also called the *session state*, is kept in memory. The sessions use independent randomness, and their states should be stored separately.

Each party keeps an internal session counter c, which is initially 0. Before a new session state is stored, the session counter is incremented.

The variable st_z^i stores the session state on party $z \in \{x, y\}$, whose session counter is i. It is composed of the following variables:

- $\mathsf{sid}_z^i \in \{*, 0, 1\}^{2r}$: The session identifier. It is the (ordered) concatenation of two nonces: The nonce sent by the initiator, and the nonce sent by the receiver. Let us denote by $\mathsf{rnd}_z^i \in \{0,1\}^r$ the randomness assigned to session i on party x. The nonce of each session is simply the session randomness. Before the initiator receives the nonce of the responder, he sets the nonce of the responder to $*^r$ (i.e., a wildcard string of length r).
- $\mathsf{role}_z^i \in \{\texttt{'I'}, \texttt{'R'}\}$: The role of the party in the current session, which is either the initiator (`'I'`) or the responder (`'R'`).
- $\mathsf{pid}_z^i \in \{0,1\}^{\ell}$: The identifier of the session partner.
- $T_z^i \in \{0,1\}^{64}$: The time stage in which the session is initiated. We will see that FORSAKES is designed so that all three messages should be exchanged within one time stage. Otherwise, the session state will be deleted.
- $sk_z^i \in \{0,1\}^k$: The first part of the ephemeral key, known as the *session key*. As is the case with all secure AKE protocols, the session key will not be used during the FORSAKES. Instead, it will be used used for confidentiality in the secure channel established by the AKE protocol. Therefore,

[2]The system's epoch can be any instant in time. For instance, the Unix Epoch is the time 00:00:00 UTC on January 1, 1970.

Algorithm 1. Initiator's handling of session initiation.

$\mathsf{pid}_x^c \leftarrow \mathrm{id}_y$; // Partner ID, set by the caller.

Incoming Message: None (0)

1: $c \leftarrow c + 1$;
2: $\mathsf{sid}_x^c \leftarrow \mathsf{rnd}_x^c \,||\, *^r$; // '*' is a wildcard
3: $\mathsf{role}_x^c \leftarrow$ 'I';
4: $T_x^c \leftarrow T$;
5: $sk_x^c \leftarrow ik_x^c \leftarrow \lambda$;
6: $\mathsf{acc}_x^c \leftarrow \lambda$; // Session fate is undecided yet.
7: $st_x^c \leftarrow \mathsf{sid}_x^c \,||\, \mathsf{role}_x^c \,||\, \mathsf{pid}_x^c \,||\, T_x^c \,||\, sk_x^c \,||\, ik_x^c \,||\, \mathsf{acc}_x^c$;
8: $\mathrm{Msg}_1 \leftarrow 1 \,||\, \mathrm{id}_x \,||\, \mathsf{pid}_x^c \,||\, T_x^c \,||\, \mathsf{rnd}_x^c$;
9: **return** Msg_1

it is important to protect the session key properly. The next section shows that the security of an AKE protocol is defined based on the proper protection of the session key.

- $ik_z^i \in \{0,1\}^k$: The second part of the ephemeral key, known as the *integrity key*. Once computed, this key protects the integrity of FORSAKES messages.
- $\mathsf{acc}_z^i \in \{\lambda, 0, 1\}$: The acceptance decision of the session. In the beginning, the acceptance is undecided (λ). When the session decides it acceptance, acc_z^i will be either false (0) or true (1).

Next, we will explain how each message of FORSAKES is handled by the corresponding party.

Session Initiation. The initiator does not receive any message from *another party*. However, the initiator session is supposedly called by some "higher-level application," which can be thought as the *zeroth message* of the protocol. The caller determines the identifier of the desired partner. Let us assume that the ID of the initiator is id_x, and the ID of the partner (i.e., the responder) is id_y. The algorithm used by the initiator to handle session initiation, and generation of the first protocol message, is described by Algorithm 1.

The initiator first sets the session partner ID to id_y, and then increases the session counter c. Next, the session ID is set to $\mathsf{rnd}_x^c \,||\, *^r$. Recall that rnd_x^c is the current session's randomness, while $*^r$ is a wildcard string of length r. The reason of using a wildcard string becomes clear when we discuss how the initiator handles the second message.

In the next step, the initiator sets the session time stage to T, the ephemeral keys are set to empty, and the acceptance state is set to undecided (λ). Finally, the session state and Msg_1 are set.

Notice that the session state includes every other state variable:

$$st_x^c \leftarrow \mathsf{sid}_x^c \,||\, \mathsf{role}_x^c \,||\, \mathsf{pid}_x^c \,||\, T_x^c \,||\, sk_x^c \,||\, ik_x^c \,||\, \mathsf{acc}_x^c \ .$$

It will be the only variable that each session stores. That is, there will be no need to store other variables, such as sid_x^c, separately. However, we will keep using other variables later, when the session receives the second message. Here, it is implicitly assumed that session state is interpreted by the party to its constituent parts.

Let us explain the construction of the first protocol message as well:

$$\mathrm{Msg}_1 \leftarrow 1 \,||\, \mathrm{id}_x \,||\, \mathsf{pid}_x^c \,||\, T_x^c \,||\, \mathsf{rnd}_x^c \ . \tag{1}$$

Notice that the message begins with 1, indicating that this is the first message of the protocol. It helps the receiver—who may have participated in multiple concurrent sessions—to distinguish the incoming message. It can prevent *typing attacks* [4, 5], where the adversary uses the syntactical similarity of protocol messages to reorder them and mount an attack.

Algorithm 2. Responder's handling of the first message.

Incoming Message: $m = 1 \,||\, \mathrm{id}_s \,||\, \mathrm{id}_r \,||\, T_s \,||\, \mathsf{rnd}_s$

if $\left(\mathrm{id}_r \neq \mathrm{id}_y \;\text{ or }\; T_s \neq T\right)$
  return;
$d \leftarrow d + 1$;
$\mathsf{pid}_y^d \leftarrow \mathrm{id}_s$;
$\mathsf{sid}_y^d \leftarrow \mathsf{rnd}_s \,||\, \mathsf{rnd}_y^d$;
$\mathsf{role}_y^d \leftarrow$ `'R'`;
$T_y^d \leftarrow T$;
let K_{yx}^T be the LTK shared between id_y and pid_y^d;
$sk_y^d \leftarrow \mathcal{O}(K_{yx}^T \,||\, 0 \,||\, \mathsf{sid}_y^d)$;
$ik_y^d \leftarrow \mathcal{O}(K_{yx}^T \,||\, 1 \,||\, \mathsf{sid}_y^d)$;
$\mathsf{acc}_y^d \leftarrow \lambda$; // `Session fate is undecided yet.`
$st_y^d \leftarrow \mathsf{sid}_y^d \,||\, \mathsf{role}_y^d \,||\, \mathsf{pid}_y^d \,||\, T_y^d \,||\, sk_y^d \,||\, ik_y^d \,||\, \mathsf{acc}_y^d$;
$\textsc{Msg}_2 \leftarrow 2 \,||\, \mathrm{id}_y \,||\, \mathrm{id}_x \,||\, T_y^d \,||\, \mathsf{sid}_y^d$;
$\textsc{Auth}_2 \leftarrow \mathcal{O}(ik_y^d \,||\, \textsc{Msg}_2)$;
return $\textsc{Msg}_2 \,||\, \textsc{Auth}_2$;

The message includes the identifier of the sender (id_x), as well as the identifier of the partner (pid_x^c). We include these two identifiers, in the order specified, in all messages of FORSAKES. This prevents *parallel session attacks* [6], where the attacker opens two sessions with a single party in parallel. She then sends every message he receives from the first session in the second session, and vice versa.

$\textsc{Msg}_1$ includes the current time stage of the initiator. This inclusion is because the receiver should reject the message if it is delivered in a different time stage.

Finally, $\textsc{Msg}_1$ includes the nonce of the initiator (rnd_x^c). The nonce is used to prevent *replay attacks*. Furthermore, we will see that the nonce of both parties affects the ephemeral keys in an essential way.

Handling the First Message. The first message of the protocol is specified in Equation 1. Upon receiving this message, the receiver verifies the syntax, and if approved, follows Algorithm 2. Notice that the incoming message is described as $m = 1 \,||\, \mathrm{id}_s \,||\, \mathrm{id}_r \,||\, T_s \,||\, \mathsf{rnd}_s$, where the subscripts s and r denote the sender and receiver, respectively. For instance, id_s means the sender ID, while id_r means the sender ID.

The receiver first checks whether the message is intended for him ($\mathrm{id}_r = \mathrm{id}_y$), and whether the time stage in which the first message was generated is the same as the local time stage ($T_s = T$). If either condition fails, the responder executes the **return** command, meaning that he is no longer interested in this message.

Remark 1. An alternative approach is to return an error message to the initiator, but it might be wasteful of the computation time and bandwidth. The protocol designer should customize this part, if desired. ◁

Next, the responder increments his session counter d. While not necessary, we called the receiver's session counter d, to distinguish it from the initiator's session counter c.

The receiver sets the partner ID to id_s, the session ID to $\mathsf{rnd}_s \,||\, \mathsf{rnd}_y^d$, the session role to `'R'`, and the session time stage to T. We also assume that K_{yx}^T is the current value of the LTK shared between the initiator and the responder. (If such a key does not exist, the incoming message will be rejected.)

Algorithm 3. Initiator's handling of the second message.

Incoming Message: $m = \overbrace{2 \,||\, \mathrm{id}_s \,||\, \mathrm{id}_r \,||\, T_s \,||\, \mathsf{sid}}^{\textsc{Msg}_2} \,||\, \textsc{Auth}_2$

if $\left(\mathsf{sid}_x^i \neq \mathsf{sid} \text{ for all } i \in [c]\right)$
  return; // No session matches with m.
if $\left(\mathsf{sid}_x^i = \mathsf{sid} \text{ for more than one } i \in [c]\right)$
  $st_x^i \leftarrow \lambda$ for all such i's;
  return; // Unique match is required.
let $i \in [c]$ be the unique value such that $\mathsf{sid}_x^i = \mathsf{sid}$;
if $\left(\mathrm{id}_s \neq \mathsf{pid}_y^i \text{ or } \mathrm{id}_r \neq \mathrm{id}_x \text{ or } T_s \neq T \text{ or } T_s \neq T_x^i \text{ or } \mathsf{role}_x^i \neq \text{`I'} \text{ or } \mathsf{acc}_x^i \neq \lambda\right)$
  $st_x^i \leftarrow \lambda$;
  return;
let K_{xy}^T be the LTK shared between id_x and pid_x^i;
$\mathsf{sid}_x^i \leftarrow \mathsf{sid}$; // No more wildcards.
$sk_x^i \leftarrow \mathcal{O}(K_{xy}^T \,||\, 0 \,||\, \mathsf{sid}_x^i)$;
$ik_x^i \leftarrow \mathcal{O}(K_{xy}^T \,||\, 1 \,||\, \mathsf{sid}_x^i)$;
if $\left(\textsc{Auth}_2 \neq \mathcal{O}(ik_x^i \,||\, \textsc{Msg}_2)\right)$
  $st_x^i \leftarrow sk_x^i \leftarrow ik_x^i \leftarrow \lambda$;
  return;
$\mathsf{acc}_x^i \leftarrow 1$;
$st_x^i \leftarrow \mathsf{sid}_x^i \,||\, \mathsf{role}_x^i \,||\, \mathsf{pid}_x^i \,||\, T_x^i \,||\, sk_x^i \,||\, ik_x^i \,||\, \mathsf{acc}_x^i$;
$\textsc{Msg}_3 \leftarrow 3 \,||\, \mathrm{id}_x \,||\, \mathsf{pid}_x^i \,||\, T_x^i \,||\, \mathsf{sid}_x^i$;
$\textsc{Auth}_3 \leftarrow \mathcal{O}(ik_x^i \,||\, \textsc{Msg}_3)$;
return $\textsc{Msg}_3 \,||\, \textsc{Auth}_3$

It is now possible to compute the ephemeral keys based on the current LTK, as well as the nonces of both sessions (note that the session ID is the concatenation of nonces):

$$sk_y^d \leftarrow \mathcal{O}(K_{yx}^T \,||\, 0 \,||\, \mathsf{sid}_y^d)$$
$$ik_y^d \leftarrow \mathcal{O}(K_{yx}^T \,||\, 1 \,||\, \mathsf{sid}_y^d)\,.$$

After setting the ephemeral keys, the acceptance state of the session is set to undecided (λ), and the session state is constructed. Then, the second message is generated with a similar syntax to the first message. There are two differences, though: (1) The second message includes the session ID rather than a single nonce; and (2) The second message is authenticated using the *integrity key*. The construction of the message and the authenticator is as follows:

$$\textsc{Msg}_2 \leftarrow 2 \,||\, \mathrm{id}_y \,||\, \mathrm{id}_x \,||\, T_y^d \,||\, \mathsf{sid}_y^d$$
$$\textsc{Auth}_2 \leftarrow \mathcal{O}(ik_y^d \,||\, \textsc{Msg}_2)\,.$$

Finally, the responder sends $\textsc{Msg}_2 \,||\, \textsc{Auth}_2$ to the initiator.

Handling the Second Message. After verifying the message syntax, the initiator executes Algorithm 3 to handle the second message. It is first verified whether there exists a session whose identifier matches the session ID included in the incoming message. It is here that *wildcard matching* is important: Notice that the initiator only knows the first part of the session ID, and the second part is chosen by the responder. If no match is found, the incoming message is rejected (via **return**).

Next, it is verified that exactly one match exists. If there are two or more sessions whose ID's matches the sid on the incoming message, all such sessions will be deleted by setting them to empty

Algorithm 4. Receiver's handling of the third message.

Incoming Message: $m = \overbrace{3 \,||\, \mathrm{id}_s \,||\, \mathrm{id}_r \,||\, T_s \,||\, \mathsf{sid}}^{\textsc{Msg}_3} \,||\, \textsc{Auth}_3$

1: **if** $\left(\mathsf{sid}_y^i \neq \mathsf{sid} \text{ for all } i \in [d]\right)$
2: **return**; // No session matches with m.
3: **if** $\left(\mathsf{sid}_y^i = \mathsf{sid} \text{ for more than one } i \in [d]\right)$
4: $st_y^i \leftarrow \lambda$ for all such i's;
5: **return**; // Unique match is required.
6: **let** $i \in [d]$ be the unique value such that $\mathsf{sid}_y^i = \mathsf{sid}$;
7: **if** $\left(\mathrm{id}_s \neq \mathsf{pid}_y^i \text{ or } \mathrm{id}_r \neq \mathrm{id}_y \text{ or } T_s \neq T \text{ or } T_s \neq T_y^i \text{ or } \mathsf{role}_y^i \neq \text{'R'} \text{ or } \mathsf{acc}_y^i \neq \lambda\right)$
8: $st_y^i \leftarrow sk_y^i \leftarrow ik_y^i \leftarrow \lambda$;
9: **return**;
10: **if** $\left(\textsc{Auth}_3 \neq \mathcal{O}(ik_y^i \,||\, \textsc{Msg}_3)\right)$
11: $st_y^i \leftarrow sk_y^i \leftarrow ik_y^i \leftarrow \lambda$;
12: **return**;
13: $\mathsf{acc}_y^i \leftarrow 1$;
14: $st_y^i \leftarrow \mathsf{sid}_y^i \,||\, \mathsf{role}_y^i \,||\, \mathsf{pid}_y^i \,||\, T_y^i \,||\, sk_y^i \,||\, ik_y^i \,||\, \mathsf{acc}_y^i$;

string, and rejecting the incoming message (via **return**). This event should occur only with exponentially small probability (see Corollary 1 on page 26).

Remark 2. Notice that deleting a sensitive object by freeing the memory allocated is a bad security practice, because the traces of that object can reside in the memory for a long time. Therefore, we urge the implementors to securely erase (wipe) the objects from the memory. This paper adopts the convention that assigning the empty string λ to a previously initialized variable denotes the secure wipe of the memory allocated to it. ◁

Now that the *unique* session state matching the incoming message is found, six more verifications are performed: (1) The ID of the sender matches the ID of the session partner, (2) the ID of the receiver matches the ID of the current party, (3) the time stage of the sender matches the current time stage, (4) the current time stage matches the session time stage, (5) the role of the current session is `'I'`, meaning that it anticipated the second protocol message, and (6) the session did not accept or reject previously. If any of these conditions does not hold, the session state is securely wiped, and the incoming message is rejected (via **return**).

Let us assume that K_{xy}^T is the current value of the LTK shared between the initiator and the responder. (If such a key does not exist, the state is securely wiped, and incoming message will be rejected.) By symmetry, we have $K_{xy}^T = K_{yx}^T$.

Next, the initiator updates the session ID to sid. This effectively replaces the wildcard string with a proper value. He will then set the ephemeral keys, and verifies the integrity of the second message by comparing $\textsc{Auth}_2$ to the value it must be. If the verification fails, the ephemeral keys and the session state are securely wiped, and the incoming message is rejected (via **return**).

Otherwise, the acceptance state is set to true, and the session state as well as the third message are computed.

Handling the Third Message. The responder verifies the third message syntactically, and if it is OK, executes Algorithm 4. The algorithm first checks whether the sid on the incoming message matches a *unique* session state. If not, secure wipes will take place, and the incoming message is rejected (via **return**).

Next, six more verifications are performed: (1) The ID of the sender matches the ID of the session partner, (2) the ID of the receiver matches the ID of the current party, (3) the time stage of the sender matches the current time stage, (4) the current time stage matches the session time stage, (5) the role of the current session is `'R'`, meaning that it anticipated the third protocol message, and (6) the session did not accept or reject previously. If any of these conditions does not hold, the session state is securely wiped, and the incoming message is rejected (via **return**).

As a final verification, the integrity key is used to verify $\textsc{Auth}_3$ on $\textsc{Msg}_3$. If the verification fails, the session state is securely wiped, and the incoming message is rejected (via **return**).

Ultimately, the receiver accepts, and updates the session state. This step concludes the description of FORSAKES, but many practical issues are left to be discussed in Section 6.

4 Security Model & Definition

Now that we saw a practical AKE protocol (FORSAKES), it will be easier to understand the security model and definition for general AKE protocols. In what follows, we first model a powerful adversary in Section 4.1. The adversary can arbitrarily create new parties, share a long-term key between any pair of parties, deliver arbitrary messages to them at any time, receive the response as well as information about their internal state, get access to the long-term and session keys, and so on. In the next step, we define what it means for an AKE protocol to be secure in the model. To this end, we define a game between the adversary, and a hypothetical entity called the *challenger*. The game provides the adversary with all the abilities specified in the model. The goal of the adversary is to distinguish any session key of her choice, from a random value provided by the challenger. The security definition is as permissive on the adversary as possible. In other words, we deem the adversary successful if she succeeds in any *non-obvious* way. Examples of obvious winning strategies are when the adversary reveals the session key of the target session, or a session partnered to the target session. Section 4.2 formalizes these obvious strategies, by defining what it means for a session to be partnered to another session, and how a fresh session is defined. The actual definition of secure AKE protocols is described in Section 4.3.

4.1 A Model for Key Exchange Protocols

As in the previous work, we put the adversary in "the center of the universe." No message is delivered without the permission of the adversary. The adversary can eavesdrop on, forge, redirect, replicate, change, delete, and delay messages. Moreover, the adversary is free to obtain the internal state of each session, or even acquire the long-term key of any pair of parties.

An innovation in our model is the ability of the adversary to dynamically create a network of interconnected parties. In other words, the adversary can freely register new parties in the system, and ask the system to share a long-term key between any pair of registered parties in the system.

Clock Model. In two-party protocols, every pair of parties wishing to share a session key may need to have synchronized clocks. It is rarely a problem if the shared clock between A and B differs from that of B and C. However, incorporating different shared clocks between various pairs of parties may result in unnecessary clutter in the model. We therefore assume all parties in the system share a universal clock. The system starts in time stage 1, which is incremented every τ seconds. The value of the time stage is stored by the variable T.

4.1.1 Adversarial & Communication Model

The interactions between the adversary and the protocol entities can be modeled as a thought experiment. The thought experiment (or the game) is played between a hypothetical *challenger* $\mathcal{C}$ and

Table 1. Global variables stored by $\mathcal{C}$.

Variable	Description
T	The time stage of the system, which is initially 1.
N	The number of the parties in the system, which is initially 0.
$\mathcal{ID}$	A function mapping the real identifiers of parties to their ordinal identifiers. It is initially empty.

Table 2. Party-specific variables stored by $\mathcal{C}$.

Variable	Description
id_x	The $\ell(n)$-bit string identifier of party x.
$\mathscr{P}_x$	The set of parties who share an LTK with party x.
sess_x	The set of sessions created on party x.
K_{xy}^{θ}	The value of the LTK between x and y in time stage θ.
$\mathsf{revealTime}_{xy}$	An integer denoting the time stage in which the LTK between x and y is revealed to the adversary.

the *adversary* $\mathcal{A}$. The challenger allows the adversary to make a set of predefined queries, to which $\mathcal{C}$ answers accordingly. The role of the challenger is to keep the state information, to verify the queries, to run the protocol, and to respond to the adversary.

Input of $\mathcal{C}$. The input of $\mathcal{C}$ is the security parameter n, in the unary form 1^n. Furthermore, $\mathcal{C}$ has black-box access to the AKE protocol Π. We denote this fact by $\mathcal{C}^{\Pi}(1^n)$.

State Information of $\mathcal{C}$. Let us discuss the state information which $\mathcal{C}$ keeps. As described below, the state changes when either $\mathcal{A}$ or the universal clock makes a query to $\mathcal{C}$.

The global state variables are detailed in Table 1. The variable $\mathcal{ID}$ requires an explanation: Each party has two types of identifiers: An *ordinal* number, and an $\ell(n)$-bit *binary string*. The ordinal number determines the order at which parties are created in the system. For instance, the fifth party registered in the system receives the ordinal 5. This identifier is for internal use by $\mathcal{A}$ and $\mathcal{C}$, and is not used by the protocol itself. In contrast, the protocol uses the $\ell(n)$-bit identifiers. The set $\mathcal{ID}$ contains both type of identifiers, paired together.

The challenger $\mathcal{C}$ also keeps state variables which are specific to one or more parties. These variables are explained in Table 2. Finally, $\mathcal{C}$ keeps session-specific state variables, which are described in Table 3. Notice that (x, s) denotes a session identified by s on party x.

Queries Allowed by $\mathcal{C}$. Below, we list and describe the queries allowed by $\mathcal{C}$. Except the first query (which is made by the universal clock), all other queries are made by the adversary. For the sake of simplicity (and *not* security), we assume that the adversary loses the game if she makes queries which are "obviously wrong." For instance, the adversary loses the game if she asks $\mathcal{C}$ to share an LTK between two non-existent parties in the system. In such cases, the challenger immediately outputs 0 and aborts the game (0 means the adversary has lost the game).

▶ The TimeEvent( ) function. The universal clock calls this function at regular intervals (every τ seconds). Upon receiving this query, $\mathcal{C}$ updates all long-term keys in the system by calling UpdateLTK(T, K_{xy}^{T}). In FORSAKES, the function UpdateLTK simply ignores its first argument, and performs an

Table 3. Session-specific variables stored by $\mathcal{C}$.

Variable	Description
pid_x^s	The *ordinal* identifier of the party to whom the session (x, s) is partnered.
rnd_x^s	The randomness used in the session (x, s).
role_x^s	The role of the session (x, s), which is either `'I'` (initiator) or `'R'` (responder).
sk_x^s	The session key of the session (x, s).
skTime_x^s	The time stage in which the session key of the session (x, s) is set.
acc_x^s	The acceptance state of the session (x, s). It is λ if the decision is yet to be made, 0 if it has rejected, and 1 if it has accepted.
state_x^s	The state of the session (x, s). We assume that this variable encodes, among other information, the value of sid_x^s, pid_x^s, sk_x^s, and acc_x^s.
$\mathsf{exposed}_x^s$	A Boolean variable, indicating whether the adversary has exposed the session (x, s).

Algorithm 6. The Register( ) function.

```
1: Register( )
2:   N ← N + 1;
3:   id_N ←_R {0,1}^ℓ(n);
4:   ID ← ID ∪ {(id_N, N)};
5:   sess_N ← P_N ← ∅;
6:   return id_N;
```

time-independent update: $\mathsf{UpdateLTK}(T, K_{xy}^T) \stackrel{\text{def}}{=} \mathcal{O}(K_{xy}^T)$. Immediately after the new key K_{xy}^{T+1} is computed, the old key K_{xy}^T is wiped securely (See Remark 2).

After all LTKs are updated, $\mathcal{C}$ starts a new time stage by increasing T, and notifies the adversary by calling $\mathsf{Notify}(\mathcal{A})$.

The pseudocode of the TimeEvent( ) function is described in Algorithm 5.

Algorithm 5. The TimeEvent( ) function.

```
1: TimeEvent( )
2:   for (x ← 1 to N)
3:     for (y ∈ P_x)
4:       K_xy^{T+1} ← UpdateLTK(T, K_xy^T);
5:       K_xy^T ← λ;                          // secure wipe of the LTK
6:   T ← T + 1;
7:   Notify(A);
```

▶ The Register( ) function. The adversary calls this function to introduce a new party into the system. Upon receiving this query, $\mathcal{C}$ increments N, and generates a random $\ell(n)$-bit identifier id_N. The pair (id_N, N) is added to the set $\mathcal{ID}$, and the set of sessions on N (sess_N) and the set of parties who share an LTK with N ($\mathcal{P}_N$) are set to empty. Finally, id_N is returned to the adversary. Notice that the adversary can keep track of N and $\mathcal{ID}$ by herself, and therefore $\mathcal{C}$ does not bother to return these values.

The pseudocode of the Register( ) function is described in Algorithm 6.

▶ The $\mathsf{ShareLTK}(x, y)$ function. The adversary calls this function to share an LTK between the parties whose *ordinal* identifiers are x and y. The challenger first checks if either x or y is nonexistent, whether they are identical, and whether they have already shared a key. If either of these conditions hold, the adversary loses the game. Otherwise, a random $K(n)$-bit key is shared between x and y, and they will be added to the partner set of each other. Notice that by symmetry, $K^T_{xy} = K^T_{yx}$. Finally, the reveal time of both keys is set to infinity.

The pseudocode of the $\mathsf{ShareLTK}(x, y)$ function is described in Algorithm 7.

Algorithm 7. The $\mathsf{ShareLTK}(x, y)$ function.

$\mathsf{ShareLTK}(x \in \mathbb{N}, y \in \mathbb{N})$
  if $\big(x > N$ or $y > N$ or $x = y$ or $y \in \mathscr{P}_x\big)$
    output 0 and **abort**;
  $K^T_{yx} \leftarrow K^T_{xy} \leftarrow_R \{0,1\}^{K(n)}$;
  $\mathscr{P}_x \leftarrow \mathscr{P}_x \cup \{y\}$ and $\mathscr{P}_y \leftarrow \mathscr{P}_y \cup \{x\}$;
  $\mathsf{revealTime}_{xy} \leftarrow \mathsf{revealTime}_{yx} \leftarrow +\infty$;

▶ The $\mathsf{Send}(x, s, y, m)$ function. The adversary calls this function to send a message m to (x, s), claiming this message comes from party y. To start the protocol on the initiator side, the adversary sends a zero message ($m = 0$). It is implicitly assume that $m = 0$ is not a valid protocol message (which is the case for virtually all protocols), and therefore it is not misinterpreted by the receiving party.

If party x or y do not exist in the system, or if y does not have an LTK with x, the adversary loses the game.

Next, the challenger checks whether the session (x, s) exists, and if not, initializes (or changes) the following variables:

- sess_x: The session s is added to this set.
- skTime^s_x: It is set to 0, which means that the session key is not generated yet.
- rnd^s_x: The randomness of the current session is picked randomly from $\{0,1\}^{r(n)}$.
- role^s_x: If the incoming message is 0, the session role is set to `'I'`. Otherwise, it is set to `'R'`.
- $\mathsf{exposed}^s_x$: It is set to false (0), as the session is not exposed yet.

Let Π denote the AKE protocol executed by a single party. To run (x, s) on the incoming message m (allegedly from y), the challenger feeds Π with the following pieces of information:

- the long-term key (K^T_{xy});
- the current time stage (T);
- the current session state (state^s_x);
- the session randomness (rnd^s_x);
- the identifiers of the current party and its session partner (id_x and id_y); and
- the incoming message (m).

The output of Π is the outgoing message (m'), and the updated session state (state^s_x), and the updated randomness (rnd^s_x):

$$(m', \mathsf{state}^s_x, \mathsf{rnd}^s_x) \leftarrow \Pi(K^T_{xy}, T, \mathsf{state}^s_x, \mathsf{rnd}^s_x, \mathrm{id}_x, \mathrm{id}_y, m)\,.$$

The reason why the randomness gets updates is explained in the description of the $\mathsf{ExposeSS}$ function (see below).

Algorithm 8. The $\mathsf{Send}(x, s, y, m)$ function.

1: $\mathsf{Send}(x \in \mathbb{N}, s \in \mathbb{N}, y \in \mathbb{N}, m \in \{0,1\}^*)$
2: **if** $\big(x > N$ or $y > N$ or $y \notin \mathscr{P}_x\big)$
3: **output** 0 and **abort**;
4: **if** $(s \notin \mathsf{sess}_x)$
5: $\mathsf{sess}_x \leftarrow \mathsf{sess}_x \cup \{s\}$;
6: $\mathsf{skTime}_x^s \leftarrow 0$;
7: $\mathsf{rnd}_x^s \leftarrow_R \{0,1\}^{r(n)}$;
8: **if** $(m = 0)$
9: $\mathsf{role}_x^s \leftarrow I$;
10: **else**
11: $\mathsf{role}_x^s \leftarrow R$;
12: $\mathsf{exposed}_x^s \leftarrow 0$;
13: $(m', \mathsf{state}_x^s, \mathsf{rnd}_x^s) \leftarrow$
$\Pi(K_{xy}^T, T, \mathsf{state}_x^s, \mathsf{rnd}_x^s, \mathrm{id}_x, \mathrm{id}_y, m)$;
14: $(\mathsf{sid}_x^s, \mathsf{pid}_x^s, sk_x^s, \mathsf{acc}_x^s) \leftarrow f_{\mathcal{ID}}(\mathsf{state}_x^s)$;
// We implicitly assume that $\mathsf{pid}_x^s = y$.
15: **if** $\big(\mathsf{skTime}_x^s = 0$ and $sk_x^s \neq \lambda\big)$
16: $\mathsf{skTime}_x^s \leftarrow T$;
17: **return** $(m', \mathsf{sid}_x^s, \mathsf{pid}_x^s, \mathsf{skTime}_x^s, \mathsf{acc}_x^s)$;

As stated in Table 3, the session state encodes other values, including the session ID, the partner ID, the session key, and the acceptance state of the session. These values are obtained by applying some efficient function $f_{\mathcal{ID}}$ to state_x^s:

$$(\mathsf{sid}_x^s, \mathsf{pid}_x^s, sk_x^s, \mathsf{acc}_x^s) \leftarrow f_{\mathcal{ID}}(\mathsf{state}_x^s)$$

Notice that the subscript $\mathcal{ID}$ in $f_{\mathcal{ID}}$ means that f uses the set $\mathcal{ID}$ to translate the string partner ID to the corresponding ordinal value. We implicitly assume that $\mathsf{pid}_x^s = y$, and do not check for this condition explicitly.

Next, the value of skTime_x^s is set: If it is still 0, but the session key is set, then skTime_x^s is set to the current value of the time stage.

Upon the completion of the Send function, the challenger $\mathcal{C}$ returns the tuple $(m', \mathsf{sid}_x^s, \mathsf{pid}_x^s, \mathsf{skTime}_x^s, \mathsf{acc}_x^s)$ to the adversary. Notice that the returned value does *not* include the session state (state_x^s) or the session randomness (rnd_x^s).

The pseudocode of the $\mathsf{Send}\,(x, y, s, m)$ function is described in Algorithm 8.

▶ The $\mathsf{ExposeSS}(x, s)$ function. This function is used to model the leakage of session-specific information to the adversary.

If either party x or session (x, s) is non-existent, the adversary loses the game. Otherwise, $\mathsf{exposed}_x^s$ is set to 1, and the pair $(\mathsf{state}_x^s, \mathsf{rnd}_x^s)$ is returned to the adversary. That is, the adversary will get hold of both the session state and the randomness used for this session.

In many key-exchange protocols, the exposure of the session randomness can have a devastating result on the security of the protocol. For instance, in the Diffie–Hellman key exchange, the private exponent of each session equals (or can be obtained directly from) the session randomness. Therefore, the protocol Π is given a chance to affect rnd_x^s in line 13 of Algorithm 8. As a general guideline, the protocol designers should securely wipe the session randomness as soon as it is no longer needed. However, some protocols use all the randomness in public communications, and do not need to wipe it. For instance, FORSAKES uses the session randomness as nonce, which is sent in cleartext.

The pseudocode of the $\mathsf{ExposeSS}(x, s)$ function is described in Algorithm 9.

Algorithm 9. The $\mathsf{ExposeSS}(x, s)$ function.

```
1: ExposeSS(x ∈ ℕ, s ∈ ℕ)
2:   if (x > N or s ∉ sess_x)
3:     output 0 and abort;
4:   exposed_x^s ← 1;
5:   return (state_x^s, rnd_x^s);
```

▶ The $\mathsf{RevealLTK}(x, y)$ function. The adversary calls this function to receive the long-term key between parties x and y. If either party is non-existent, or they do not have an LTK, the adversary loses the game. Otherwise, the variables $\mathsf{revealTime}_{xy}$ and $\mathsf{revealTime}_{yx}$ are set to T, and K_{xy}^T is returned.

The pseudocode of the $\mathsf{RevealLTK}(x, y)$ function is described in Algorithm 10.

Algorithm 10. The $\mathsf{RevealLTK}(x, y)$ function.

```
1: RevealLTK(x ∈ ℕ, y ∈ ℕ)
2:   if (x > N or y > N or y ∉ 𝒫_x)
3:     output 0 and abort;
4:   revealTime_xy ← revealTime_yx ← T;
5:   return K_xy^T;
```

In most papers, the adversary is given access to a $\mathsf{Corrupt}(\,)$ query, which returns *all* long-term keys stored on a single party. $\mathsf{RevealLTK}(\,)$ is a more flexible query, since the adversary can pick the exact LTK she wants to obtain. Moreover, since the adversary knows the set $\mathscr{P}_x$ of parties to whom x shares an LTK,[3] she can effectively corrupt x by running $\mathsf{RevealLTK}(\,)$ on x and every party in $\mathscr{P}_x$.

General Assumptions. We assume that Π and $f_{\mathcal{ID}}$ satisfy the following rules for all $s, x \in \mathbb{N}$:

- If Π detects an error, it returns an empty string as the state and the outgoing message.
- On input an empty string, $f_{\mathcal{ID}}$ outputs the follow tuple: $(\mathsf{sid}_x^s, \mathsf{pid}_x^s, sk_x^s, \mathsf{acc}_x^s) = (\lambda, -1, \lambda, 0)$. Note especially that the partner ID is set to -1 (invalid party), and the acceptance decision is set to 0 (false).
- If m is the last message of the protocol, Π outputs the empty message $m' = \lambda$ as the outgoing message.

If Π does not detect an error, the following assumptions are also true:

- pid_x^s is set as soon as (x, s) receives the first message, and will *not* change to any other value during the lifetime of (x, s).
- Neither sk_x^s nor sid_x^s will change after skTime_x^s is set to T in line 16.

4.2 Session Partnership and Freshness

Before defining what it means for an AKE protocol to be secure, we need to define two central notions: Session partnership, and session freshness. Our notion of session partnership, presented in Definition 1, is adopted from [18].

Definition 1 (Session Partnership). Two sessions (x, s) and (y, t) are called *partners* if the following conditions hold:

[3] This is because the parties share an LTK if and only if the adversary uses the $\mathsf{ShareLTK}(\,)$ query.

1. $sk_x^s = sk_y^t \neq \lambda$;
2. $\mathsf{sid}_x^s = \mathsf{sid}_y^t \neq \lambda$;
3. $\mathsf{role}_x^s \neq \mathsf{role}_y^t$;
4. $\mathsf{pid}_x^s = y$ and $\mathsf{pid}_y^t = x$;
5. $\mathsf{sid}_z^u \neq \mathsf{sid}_x^s$ for all $(z, u) \in \mathbb{N}^2 - \{(x, s), (y, t)\}$. This condition does *not* include wild-card matches.

□

The first and second conditions state that both sessions should output the same non-empty *session keys* and *session identifiers*. The third condition requires that the sessions have different roles; i.e., one is the initiator and the other is the responder. The forth condition states that the sessions recognize the other party as the partner. Finally, the fifth condition requires that no other session besides (x, s) and (y, t) outputs the same session identifier.

The rules stated for partnership are pretty strict. For instance, if the adversary succeeds in making two sessions agree on different session keys or session identifiers, they are no longer considered partners. As an another example, consider the case where the adversary observes the session ID of two sessions, and succeeds in making a third session output the same session identifier. In this case, none of the three sessions will be partnered to each other.

$\mathcal{C}$ uses the function $\mathsf{FindPartnerSession}(x, s)$ in Algorithm 11 to find the partner of the session (x, s), if one exists according to Definition 1. It is assumed that $\mathcal{C}$ has already made the usual sanity checks, and is assured that the session (x, s) exists in the system. The output of this function is a numeric value, denoting the *ordinal* t of the partner session on the party $y = \mathsf{pid}_x^s$. In other words, (y, t) is the session partnered to (x, s). The value of t will be -1 if no partner session exists.

Algorithm 11. The $\mathsf{FindPartnerSession}(x, s)$ function.

```
1:  FindPartnerSession(x ∈ ℕ, s ∈ ℕ)
2:    if (sid_x^s = λ or sk_x^s = λ)
3:      return −1;
4:    c ← 0                                          // number of partners
5:    for z ← 1 to N
6:      for all (u ∈ sess_z)
7:        if ((z, u) ≠ (x, s) and sid_z^u = sid_x^s)
8:          c ← c + 1;
9:          t ← u;
10:   if (c ≠ 1)                                     // no unique partner?
11:     return −1;
12:   y ← pid_x^s;
13:   if (sk_y^t ≠ sk_x^s or sid_y^t ≠ sid_x^s or
            role_y^t = role_x^s or pid_y^t ≠ x)
14:     return −1;
15:   return t;
```

The idea used in Algorithm 11 is as follows: We first count the number of sessions in the system, except (x, s), whose session identifier is sid_x^s. According to Definition 1, there must be a unique session, beyond (x, s), with session identifier sid_x^s. Therefore, if we count zero such sessions, or more than one such session, then (x, s) has no partner session. Otherwise, the session with identifier sid_x^s is examined for other conditions in Definition 1.

Now that the concept of partner sessions is established, one can define the concept of a *fresh session*. Intuitively, if a session is *unfresh*, then the adversary knows enough information about that

session to trivially deduce the session key. For instance, if the adversary exposes a session after the session key is established, then she already knows the session key. The same holds if the adversary exposes the partner session of a session. Therefore, it is important to properly define the partner session, as we just did.

In Section 4.3, we will define the security of AKE protocols as follows: We will allow the adversary to target any fresh session, and receive either the session key or a random key. Her goal will then be to distinguish the two cases. Consequently, it is important to define the concept of fresh sessions as loosely as possible, so that when a protocol is proved secure in our model, it resist a variety of attacks. Our definition of freshness is presented in Definition 2.

Definition 2 (Freshness). A session (x, s) is called *fresh* if the following conditions hold (here, $y = \mathsf{pid}_x^s$):

1. $\mathsf{skTime}_x^s < \mathsf{revealTime}_{xy}$;
2. $\mathsf{exposed}_x^s = 0$;
3. Let $t \leftarrow \mathsf{FindPartnerSession}(x, s)$. If $t \neq -1$, then $\mathsf{skTime}_y^t \geq \mathsf{revealTime}_{yx}$ and $\mathsf{exposed}_y^t = 0$. □

The first condition is a *forward security* requirement: If the LTK is revealed in a time stage later than the one in which the session key is established, then the session key should remain protected. Notice that the initial values for skTime_x^s and $\mathsf{revealTime}_{xy}$ are 0 and $+\infty$, respectively. Therefore, condition 1 holds even if the session key is yet to be established, or the LTK is not revealed. The second condition verifies that the session is not exposed. Finally, the third condition examines the case when the session has a partner: In this case, the partner session should satisfy the first two conditions above.

$\mathcal{C}$ uses the function $\mathsf{fresh}(x, s)$ in Algorithm 12 to find whether the session (x, s) is fresh, according to Definition 2.

Algorithm 12. The $\mathsf{fresh}(x, s)$ function.

```
1: fresh(x ∈ ℕ, s ∈ ℕ)
2:   if (skTime_x^s ≥ revealTime_xy or exposed_x^s = 1)
3:     return 0;
4:   y ← pid_x^s;
5:   t ← FindPartnerSession(x, s);
6:   if (t ≠ −1)
7:     if (exposed_y^t = 1 or skTime_y^t ≥ revealTime_yx)
8:       return 0;
9:   return 1;
```

4.3 AKE Security Definition

Up until now, we defined a general model for interaction between the parties and the adversary. Given this model, it is straightforward to give the definition of what it means for an authenticated key exchange (AKE) protocol to be secure. We first need to augment the challenger $\mathcal{C}$ with two more queries. We denote the augmented challenger by $\mathcal{D}$. The new queries are the Test query, and the Guess query. Contrary to previous queries, the adversary can make the Test and Guess queries only once. Moreover, the Guess query is the last query in the system, after which $\mathcal{D}$ announces whether the adversary wins, and aborts the game.

In a Test query, the adversary specifies a *target session*, which must be a fresh one containing a session key. Next, $\mathcal{D}$ flips a coin, and depending on the result, answers with either the session key, or a random value. The adversary continues the game by making arbitrary queries (except Test and Guess,

which can only be made once). Her goal is to gather as much information as possible, to find out whether the value returned by the Test function is the actual session key, or merely a random value. Notice that in this process, she cannot make any query which makes the target session non-fresh; otherwise, she loses the game as soon as she makes a Guess query.

At some point in time, the adversary makes a Guess query, in which she announces her guess of the coin flipped during the Test query. If the target session is still fresh, and the guess is correct, the adversary wins the game. Otherwise, she loses.

Let us describe the queries in more detail. We will assume that $\mathcal{D}$ keeps three new global variables beyond Table 1: X, S, and b. The pair (X, S) denotes the target session, and b specifies the random coin. The initial value for b is -1.

▶ The $\mathsf{Test}(x, s)$ function. By making this query, the adversary specifies that (x, s) is her target session of choice. If the party x or session (x, s) is non-existent, or the adversary has already made a Test query, or the target session is unfresh, or the session key is not established yet, she loses the game. (The second condition is checked by verifying $b \neq -1$, as the value of b will be set to either 0 or 1 by this function).

Next, the global variables X and S are set to x and s, respectively. The value of b is then set according to a random coin flip. If b is 0, the session key of (x, s) is returned to the adversary. Otherwise, a random $k(n)$-bit binary string is returned to the adversary. (Recall that $k(n)$ is the length of the session key.)

The pseudocode of the $\mathsf{Test}(x, s)$ function is described in Algorithm 13.

Algorithm 13. The $\mathsf{Test}(x, s)$ function.

```
1:  Test(x ∈ ℕ, s ∈ ℕ)
2:    if (x > N or s ∉ sess_x or b ≠ −1 or ¬fresh(x, s) or sk_x^s = λ)
3:      output 0 and abort;
4:    (X, S) ← (x, s);
5:    b ←_R {0, 1};
6:    if (b = 0)
7:      key ← sk_x^s;
8:    else
9:      key ←_R {0, 1}^{k(n)};
10:   return key;
```

▶ The $\mathsf{Guess}(b')$ function. By calling this function, the adversary announces her guess of the random coin b. As a sanity check, the function checks whether a Test query is made before (if so, $b \neq -1$). This is done because otherwise no target session is specified. Next, it is checked that the target session is fresh. If either of of the conditions is false, the adversary loses.

The rest is pretty simple: If the adversary's guess is correct ($b' = b$), the adversary wins and Guess outputs 1. Otherwise, the adversary loses and Guess outputs 0. In both cases, $\mathcal{D}$ aborts the game, as Guess is the last possible query in the game.

The pseudocode of the $\mathsf{Guess}(b')$ function is described in Algorithm 14.

Algorithm 14. The $\mathsf{Guess}(b')$ function.

```
1:  Guess(b′ ∈ {0, 1}*)
2:    if (b = −1 or b′ ≠ b or ¬fresh(X, S))
3:      output 0 and abort;
4:    output 1 and abort;
```

Denote by $\langle \mathcal{D}^{\Pi}, \mathcal{A} \rangle(n)$ the single-bit output of the challenger $\mathcal{D}$, when the protocol is Π, the adversary is $\mathcal{A}$, and the security parameter is n. We assume that $\mathcal{D}$ outputs 0 if $\mathcal{A}$ halts before $\mathcal{D}$ returns any value. Define the *AKE-advantage* of $\mathcal{A}$ by the following equation:

$$\mathsf{Adv}^{\mathrm{ake}}_{\mathcal{A},\mathcal{D},\Pi}(n) \stackrel{\mathrm{def}}{=} \Pr\left[\langle \mathcal{D}^{\Pi}, \mathcal{A} \rangle(n) = 1\right] - \frac{1}{2}\,, \tag{2}$$

where the probability is taken over the random coins of $\mathcal{D}$ and $\mathcal{A}$. We can now define the security of AKE protocols.

Definition 3 (Secure AKE). An AKE protocol Π is called $(n, T(n), \epsilon(n))$-*secure* if:

$$\max_{\mathcal{A}} \left\{ \mathsf{Adv}^{\mathrm{ake}}_{\mathcal{A},\mathcal{D},\Pi}(n) \right\} < \epsilon(n)\,, \tag{3}$$

where the maximum is taken over all probabilistic algorithms whose sum of running time and code size is at most $T(n)$.

We call Π a *secure AKE* if it is $(n, T(n), \epsilon(n))$-secure for all positive polynomials $T(\cdot)$ and $1/\epsilon(\cdot)$, and all sufficiently large n. □

The last sentence requires clarification. First, if $1/\epsilon(\cdot)$ is a polynomial, then $\epsilon(\cdot)$ is an inverse polynomial. Therefore, the requirement basically states that any adversary whose running time is a polynomial must have an advantage in winning the game which is smaller than the inverse of any polynomial (i.e., the advantage must be *negligible*).

5 Proving the AKE Security of FORSAKES

In this section, we provide proofs of AKE security for the FORSAKES protocol. The proof is quite involved; therefore, it is divided into several parts. The first part, presented in Section 5.1, is in fact a general proof, and is not specific to FORSAKES. It shows that if the adversary creates (polynomially) more than two parties in the system, her AKE-advantage does not change by more than a polynomial. Section 5.2 presents several lemmas about FORSAKES in a multiparty setting. Finally, Section 5.3 uses the aforementioned proofs to establish the security of FORSAKES in a two-party setting.

5.1 Reducing a Multi-Party Setting to a Two-Party Setting

Let $q_{\mathrm{reg}} \stackrel{\mathrm{def}}{=} q_{\mathrm{reg}}(n)$ and $q_{\mathrm{ltk}} \stackrel{\mathrm{def}}{=} q_{\mathrm{ltk}}(n)$ denote upper bounds on the number of Register and ShareLTK queries that the adversary makes to $\mathcal{D}$. Furthermore, let $q \stackrel{\mathrm{def}}{=} q(n)$ be an upper bound on the total number of queries that the adversary makes to $\mathcal{D}$.

Theorem 1 states that the adversary can achieve essentially the same AKE-advantage (up to a polynomial), if she registers exactly two parties in the system, rather than polynomially many.

Theorem 1. *Let Π be any AKE protocol (not necessarily secure). For all $n \in \mathbb{N}$ and any $(q_{\mathrm{reg}}, q_{\mathrm{ltk}})$–adversary $\mathcal{A}$ against Π which runs in time at most $T(n)$, there exists a $(2,1)$–adversary $\mathcal{S}$ which runs in time at most $T_{\mathcal{S}}(n) = T_{\mathcal{A}}(n) + q(n) \cdot \mathrm{poly}(n)$, such that:*

$$\mathsf{Adv}^{\mathrm{ake}}_{\mathcal{S},\mathcal{D},\Pi}(n) \geq \frac{\mathsf{Adv}^{\mathrm{ake}}_{\mathcal{A},\mathcal{D},\Pi}(n)}{(q_{\mathrm{reg}}(n))^2}\,.$$

The proof essentially results from three facts in the modeling:

1. The long-term keys (K^T_{xy}) are generated independently from each other;
2. The randomness of sessions (rnd^s_x) are generated independently from each other;
3. The session states (state^s_x) are stored and updated independently from each other.

Algorithm 15. The simulator $\mathcal{S}$.

Initialization

$\alpha \leftarrow_R [q_{\text{reg}}]$ and $\beta \leftarrow_R [q_{\text{reg}}] - \{\alpha\}$;
$N, p, p_\alpha, p_\beta \leftarrow 0$;

Query-Response. Respond to queries as follows:

TimeEvent ( )
  execute the query as specified in Algorithm 5;

Register ( )
  if $(N+1 \in \{\alpha, \beta\})$
    $N \leftarrow N+1$;
    $p \leftarrow p+1$ and $p_N \leftarrow p$;
    forward the query to $\mathcal{D}$ and **receive** id_N;
  else
    execute Algorithm 6 and **receive** id_N;
  return id_N to $\mathcal{A}$;

ShareLTK (x, y)
  if $(\{\alpha, \beta\} = \{x, y\})$
    forward ShareLTK(p_x, p_y) to $\mathcal{D}$;
  else
    execute the query as specified in Algorithm 7;

Send (x, s, y, m)
  if $(\{\alpha, \beta\} = \{x, y\})$
    forward Send(p_x, s, p_y, m) to $\mathcal{D}$;
    receive $(m', \mathsf{sid}_x^s, \mathsf{pid}_x^s, \mathsf{skTime}_x^s, \mathsf{acc}_x^s)$;
    $\mathsf{pid}_x^s \leftarrow y$;
  else
    execute the query as specified in Algorithm 8;
    receive $(m', \mathsf{sid}_x^s, \mathsf{pid}_x^s, \mathsf{skTime}_x^s, \mathsf{acc}_x^s)$;
  return $(m', \mathsf{sid}_x^s, \mathsf{pid}_x^s, \mathsf{skTime}_x^s, \mathsf{acc}_x^s)$ to $\mathcal{A}$;

ExposeSS (x, s)
  $y \leftarrow \mathsf{pid}_x^s$;
  if $(\{\alpha, \beta\} = \{x, y\})$
    forward ExposeSS(p_x, s) to $\mathcal{D}$;
    receive $(\mathsf{state}_x^s, \mathsf{rnd}_x^s)$;
  else
    execute the query as specified in Algorithm 9;
    receive $(\mathsf{state}_x^s, \mathsf{rnd}_x^s)$;
  return $(\mathsf{state}_x^s, \mathsf{rnd}_x^s)$ to $\mathcal{A}$;

RevealLTK (x, y)
  if $(\{\alpha, \beta\} = \{x, y\})$
    forward RevealLTK(p_x, p_y) to $\mathcal{D}$;
    receive K_{xy}^T;
  else
    execute Algorithm 10 and **receive** K_{xy}^T;
  return K_{xy}^T to $\mathcal{A}$;

Test (x, s)
  $y \leftarrow \mathsf{pid}_x^s$;
  if $(\{\alpha, \beta\} = \{x, y\})$
    forward Test(p_x, s) to $\mathcal{D}$ and **receive** *key*;
    return *key* to $\mathcal{A}$;
  else
    $\mathcal{S}$ halts with failure; // `simulation fails`

Guess (b')
  forward Guess(b') to $\mathcal{D}$;

Since the adversary $\mathcal{A}$ creates at most q_{reg} parties, the number of long-term keys in the system will be at most $\binom{q_{\text{reg}}}{2} \leq q_{\text{reg}}^2$.

The high-level strategy of the proof is as follows: First, the adversary $\mathcal{S}$ randomly picks two *distinct* integers α and β from the set $\{1, \ldots, q_{\text{reg}}\}$. The values α and β will denote the ordinal identities of two parties, which we call *special parties*. Next, $\mathcal{S}$ starts to *simulate* the game between $\mathcal{A}$ and $\mathcal{D}$. It answers all queries of $\mathcal{A}$ by itself, except those queries related to the special parties. The latter type of queries are forwarded to $\mathcal{D}$, and the answers are relayed back to $\mathcal{A}$. In this way, $\mathcal{S}$ will make at most two Register( ) queries, and at most one ShareLTK( ) query.

In the final stage, if $\mathcal{A}$ picks a session between the special parties as the test session, $\mathcal{S}$ will win

with the same probability that $\mathcal{A}$ wins. Otherwise, $\mathcal{S}$ loses. It will be shown that the probability of the former event is at least $1/q_{\text{reg}}^2$, and therefore the theorem follows.

We now prove the theorem more rigorously.

Proof (of Theorem 1). Let $\mathcal{S}$ be the adversary described by Algorithm 15. Since the main purpose of $\mathcal{S}$ is to simulate a multi-party AKE setting for $\mathcal{A}$, we call it a *simulator*. Before going into the details of the simulation, let us note that the following conventions were used to express this algorithm more briefly:

1. Only the state variables used *explicitly* in the algorithm were defined and initialized. For instance, the state variable T is used *implicitly* by the TimeEvent( ) function (lines 3–4 of Algorithm 15), but since this use is not explicit, the variable T is neither defined nor initialized. The corresponding definitions can be found in Section 4.
2. No sanity check is performed by $\mathcal{S}$. As an example, when $\mathcal{A}$ makes a ShareLTK(x, y), it must be verified that identities x and y exist in the system by checking whether $x \leq N$ and $y \leq N$. Moreover, x and y should not have already shared a long-term key. Such sanity checks are assumed to be performed *implicity* by $\mathcal{S}$, as specified by the rules in Section 4.

Below, each part of the simulator is explained in details.

▶ Initialization, lines 1–2. $\mathcal{S}$ first picks the ordinal identity of the two special parties. This is done by picking two distinct random integers α and β from the set $[q_{\text{reg}}] = \{1, \ldots, q_{\text{reg}}\}$. To make sure that the numbers are distinct, α is first selected randomly from $[q_{\text{reg}}]$, and then β is picked from $[q_{\text{reg}}] - \{\alpha\}$.

Next, the state variables required for performing the experiment are defined and initialized. N keeps the number of parties which will be registered by $\mathcal{A}$. Other state variables (p, p_α, and p_β) keep a *mapping* between the ordinal identities of the special parties in the game between $\mathcal{A}$ and $\mathcal{S}$, and the game between $\mathcal{S}$ and $\mathcal{D}$. The mapping is detailed in the description of the Register( ) query below.

▶ TimeEvent( ), lines 3–4. A TimeEvent( ) query is made by $\mathcal{D}$. The simulator $\mathcal{S}$ answers this query by running Algorithm 5, which involves incrementing T, notifying $\mathcal{A}$, and updating all long-term keys.

▶ Register( ), lines 5–12. When $\mathcal{A}$ makes a Register( ) query, $\mathcal{S}$ first checks whether this is either the α^{th} or β^{th} query of type Register (the *if statement* at line 6). If this is the case, N and p are incremented, and p_N is set to p. Example 1 illustrates how the mapping works.

Example 1. Assume that $\mathcal{A}$ is an adversary which makes *at most* twenty Register( ) queries. Therefore, $\mathcal{S}$ picks α and β randomly and distinctly from the set $\{1, \ldots, 20\}$. Assume that $\alpha = 18$ and $\beta = 13$. When $\mathcal{A}$ makes the thirteenth Register( ) query, $\mathcal{S}$ assigns $p \leftarrow p + 1 = 1$, and $p_{13} \leftarrow p = 1$, and registers the first party with $\mathcal{D}$. When $\mathcal{A}$ makes the eighteenth Register( ) query, $\mathcal{S}$ assigns $p \leftarrow p + 1 = 2$, and $p_{18} \leftarrow p = 2$, and registers the second party with $\mathcal{D}$. ◁

Next, the query is forwarded to $\mathcal{D}$, and the identifier id_N is received.

On the other hand, if this is neither the α^{th} nor β^{th} query of type Register, it is treated ordinarily via a call to Algorithm 6, where id_N is computed. Finally, id_N is returned to the adversary $\mathcal{A}$.

Notice that from the viewpoint of $\mathcal{A}$, the value of id_N is distributed identically, regardless of whether it $N + 1 \in \{\alpha, \beta\}$ or not. Therefore, $\mathcal{S}$ simulates the Register( ) query perfectly.

▶ ShareLTK(x, y), lines 13–17. The simulator $\mathcal{S}$ first checks whether $\{x, y\} = \{\alpha, \beta\}$, which means either "$x = \alpha$ and $y = \beta$" or "$x = \beta$ and $y = \alpha$". If this is the case, the adversary $\mathcal{A}$ wants a long-term key to be shared between the special parties. Therefore, the query is forwarded to $\mathcal{D}$, with one subtlety: $\mathcal{S}$ makes the proper mapping, and sends ShareLTK(p_x, p_y) to $\mathcal{D}$. Otherwise, Algorithm 7 is executed.

Nothing is sent back to the adversary as the returned value. Internally, either calling $\mathcal{D}$ or calling Algorithm 7 creates a random and independent key between the corresponding parties. Therefore, $\mathcal{S}$ simulates the ShareLTK( ) query perfectly.

▶ $\mathsf{Send}(x, s, y, m)$, lines 18–26. The simulator $\mathcal{S}$ first checks whether $\{x, y\} = \{\alpha, \beta\}$.

- If this is the case, $\mathcal{A}$ wants to send a message between two special parties. This is handled by making the proper mapping (x to p_x and y to p_y), and forwarding the query to $\mathcal{D}$. After receiving the answer $(m', \mathsf{sid}_x^s, \mathsf{pid}_x^s, \mathsf{skTime}_x^s, \mathsf{acc}_x^s)$ from $\mathcal{D}$, the simulator modifies pid_x^s. It is because in the returned value, $\mathsf{pid}_x^s = p_y$; whereas p_y should be mapped back to y. This is consistent with the comment on line 14 of Algorithm 8.
- If $\{x, y\} \neq \{\alpha, \beta\}$, then $\mathcal{A}$ wants to send a message between two non-special parties, or between a special and a non-special party. In this case, Algorithm 8 is executed, and the result $(m', \mathsf{sid}_x^s, \mathsf{pid}_x^s, \mathsf{skTime}_x^s, \mathsf{acc}_x^s)$ is received.

In either case, the tuple $(m', \mathsf{sid}_x^s, \mathsf{pid}_x^s, \mathsf{skTime}_x^s, \mathsf{acc}_x^s)$ is returned to $\mathcal{A}$. This tuple is generated by running the protocol Π, and then extracting the information from state_x^s via the function $f_{\mathcal{ID}}$ (see lines 13–16 of Algorithm 8). Notice that the inputs to Π are distributed identically regardless of whether $\{x, y\} = \{\alpha, \beta\}$ or not: The long-term key and the session randomness are always uniformly random, and the session state state_x^s is initially empty. The time stage T is incremented independently, and the rest of the inputs to Π (i.e., id_x, id_y, and m) are determined by the adversary.

Therefore, from the viewpoint of $\mathcal{A}$, the output of Send is identically distributed regardless of whether $\{x, y\} = \{\alpha, \beta\}$ or not, and $\mathcal{S}$ simulates this query perfectly.

▶ $\mathsf{ExposeSS}(x, s)$, lines 27–35. The simulator first finds the partner of the session (x, s) by letting $y \leftarrow \mathsf{pid}_x^s$. If both partners are the special parties, the $\mathsf{ExposeSS}$ query is forwarded to $\mathcal{D}$ (making proper mappings), and the results are returned to $\mathcal{A}$. Otherwise, Algorithm 9 is executed, and the results are returned to $\mathcal{A}$.

In both cases, the returned value is of the form $(\mathsf{state}_x^s, \mathsf{rnd}_x^s)$. As explained above (while describing the way a Send query is treated), these values are identically distributed regardless of whether $\mathcal{A}$ is dealing with special parties or not. Therefore, $\mathcal{S}$ simulates this query perfectly.

▶ $\mathsf{RevealLTK}(x, y)$, lines 36–42. If this query is made for the key shared between the special parties, $\mathcal{D}$ will respond the query ($\mathcal{S}$ makes proper mappings beforehand). Otherwise, Algorithm 10 is executed.

In both cases, a long-term key is returned, which is distributed randomly, and is consistent with the rest of $\mathcal{A}$'s view. Therefore, $\mathcal{S}$ simulates this query perfectly.

▶ $\mathsf{Test}(x, s)$, lines 43–49. This is the only query where $\mathcal{S}$ may fail to simulate the view of $\mathcal{A}$. The simulator first finds the partner of the *test session* (x, s) by letting $y \leftarrow \mathsf{pid}_x^s$. If the partners of the test session are the special parties, the query is forwarded to $\mathcal{D}$ (making proper mappings), and the result is returned to $\mathcal{A}$.

Otherwise, the simulator fails. This failure is not because $\mathcal{S}$ cannot continue the simulation; rather, continuing the simulation is *pointless*. This is because $\mathcal{S}$ should make at most two register queries to $\mathcal{D}$, and then attempt to distinguish a random value from the session key of one of the sessions between these two parties, using $\mathcal{A}$ as a guide. If $\mathcal{A}$ picks a test session whose partners are not the special parties, then $\mathcal{S}$ cannot attain its goal, and fails as a result.

Since $\mathcal{S}$ perfectly simulates the whole experiment up to a Test query, $\mathcal{A}$ has no way of distinguishing special and non-special parties. Therefore, as $\mathcal{A}$ has registered at most q_{reg} parties before a Test query, there are at most $\binom{q_{\mathrm{reg}}}{2} \leq q_{\mathrm{reg}}^2$ pairs of parties in the system. Consequently, the probability that both partners of the test session are special parties is at least $1/q_{\mathrm{reg}}^2$.

As a result, the probability that the simulation does not fail is at least $1/q_{\mathrm{reg}}^2$. Notice that if this is the case, the view of the adversary $\mathcal{A}$ is simulated perfectly.

▶ $\mathsf{Guess}(b')$, lines 50–51. The simulator simply forwards the guess b' to $\mathcal{D}$. This will finish the simulation, as $\mathcal{D}$ finishes the game as soon as it receives the Guess query.

Conditioned on the fact that the simulation does not fail, $\mathcal{S}$ forwards a Guess query, and wins with the same advantage of $\mathcal{A}$. We just proved that the simulation does not fail with probability at least $1/q_{\text{reg}}^2$. Therefore,

$$\mathsf{Adv}^{\text{ake}}_{\mathcal{S},\mathcal{D},\Pi}(n) \geq \frac{\mathsf{Adv}^{\text{ake}}_{\mathcal{A},\mathcal{D},\Pi}(n)}{q_{\text{reg}}^2} .$$

Notice that $\mathcal{S}$ answers each query of $\mathcal{A}$ with at most a polynomial overhead. Therefore, the running time $T_{\mathcal{S}}(n)$ of $\mathcal{S}$ is $T_{\mathcal{A}}(n) + q(n) \cdot \text{poly}(n)$. ■

Theorem 1 has an important implication: Since for polynomial-time adversaries, the number of q_{reg} queries are at most a polynomial in n, the advantage of the adversary in a multi-party setting is at most polynomially more than her advantage in a two-party setting.

Furthermore, consider practical settings, where the number of parties in the system is at most on the scale of ten thousand. If the advantage of the adversary in the two-party setting is 2^{-256}, then her advantage with ten-thousand parties will be at most $2^{-256} \times (10,000)^2 \approx 2^{-229}$, which is still a very low advantage.

5.2 FORSAKES in a Multi-Party Setting

In this section, we investigate FORSAKES in a multi-party setting. We start by considering the probability that two parties receive the same identifier.

Fact 1. *Let the adversary register at most $q_{\text{reg}} \stackrel{\text{def}}{=} q_{\text{reg}}(n)$ parties, and assume the system uses random $\ell \stackrel{\text{def}}{=} \ell(n)$-bit identifiers. Then, the probability that at least two identifiers are equal is at most $\binom{q_{\text{reg}}}{2} 2^{-\ell} \leq q_{\text{reg}}^2 2^{-\ell}$.*

In the following, we assume that the identifiers are unique. Later, in Theorem 2, we account for the probability stated in Fact 1.

Fact 2. *The probability that two particular sessions in FORSAKES output the same nonce is $2^{-r(n)}$.*

Notice that Fact 2 fact holds even if the adversary obtains the long-term keys and the session keys. This is because the adversary has no control over the nonce generated by any session: The session simply reads a string of length $r \stackrel{\text{def}}{=} r(n)$ from his random tape, and outputs it.

Let $\sigma \stackrel{\text{def}}{=} \sigma(n)$ be an upper bound on the number of sessions the adversary creates on the parties. If the adversary issues at most $q_{\text{snd}}(n)$ queries of type Send, then $\sigma(n) \leq q_{\text{snd}}(n)$. This is because new sessions can only be created via a Send query.

The following corollary is immediate by noting that in a system with σ sessions, there are $\binom{\sigma}{2}$ pairs of sessions.

Corollary 1. *In an execution of FORSAKES with $\sigma \stackrel{\text{def}}{=} \sigma(n)$ sessions, the probability that (at least) two sessions output the same nonce is at most $\binom{\sigma}{2} 2^{-r} \leq \sigma^2 2^{-r}$.*

Let us call an execution of FORSAKES *colliding* if at least two sessions output the same nonce. Otherwise, the execution is called *non-colliding*. A FORSAKES session ID is composed of the concatenation of the initiator and responder nonces. The adversary can affect either of the nonces (via the Send query), but not both. Therefore, we obtain the following corollary.

Corollary 2. *In a non-colliding execution of FORSAKES, no two sessions output the same session ID.*

In the rest of this section, we assume that the execution is non-colliding. The influence of the collision, stated in Corollary 1, will be accounted later in Theorem 2. Consequently, we will assume that the fifth condition of Definition 1 never holds.

Properties of FORSAKES messages. Let $\mathcal{M}$ be the set of nonempty messages which the adversary receives from parties. In FORSAKES, the first message is not equipped with an integrity mechanism (such as a MAC), and it can be forged easily. However, the second and third messages are authenticated. Assume the adversary issues a $\mathsf{Send}(x, s, y, m)$ query, where m is the second or third protocol message (i.e., it is prefixed with either '2' or '3'). Lemma 1 and Lemma 2 (in the next section) consider two separate cases, depending on whether m belongs to $\mathcal{M}$ or not.

Lemma 1. *Let $m \in \mathcal{M}$ be the second or third message of FORSAKES, delivered via $\mathsf{Send}(x, s, y, m)$. Assume that after the delivery, $\mathsf{acc}_x^s = 1$. Then, the probability of the following events is 0 (assuming a non-colliding execution):*

1. *m was generated by any party other than y;*
2. *m was destined at any session other than (x, s).*

The lemma holds even if all long-term and session keys in the system are known to the adversary.

Proof. First, notice that since the message m is generated by some party in the system, it is not important which keys are known to the adversary; she merely takes the delivery.

Any second-or-third FORSAKES message $m \in \mathcal{M}$ is authenticated. Since m includes the identifiers of the sender and receiver respectively, (x, s) rejects m if the sender is any party other than y. Consequently, $\mathsf{acc}_x^s = 1$ shows that the sender must have been y, and case (1) is ruled out.

The second and the third messages of FORSAKES carry the session identifier. Since the execution is assumed to be non-colliding, it is impossible that the message have been destined at any session other than (x, s), and $\mathsf{acc}_x^s = 1$. Therefore, case (2) is ruled out as well. ■

Remark 3. Lemma 1 shows, in particular, that *parallel session attacks* [6] are impossible against FORSAKES. ◁

The next section considers the case $m \notin \mathcal{M}$, and concludes the proof of the AKE security of FORSAKES.

5.3 Proof of AKE-Security of FORSAKES in a Two-Party Setting

In Section 5.1, we proved that any efficient adversary $\mathcal{A}$ against an AKE protocol in a *multi-party* system can be reduced to a an efficient adversary $\mathcal{S}$ against an AKE protocol in a *two-party* system, such that the running times and advantages of $\mathcal{A}$ and $\mathcal{S}$ are identical up to a polynomial. Therefore, this section restricts the adversaries to those registering at most two parties.

We also assume that the adversary against a two-party system does not make "foolish" actions which result in immediate loss of the game. Several of such assumptions are detailed below:

- Any adversary registering less than two parties has an AKE advantage of 0, since there will be no test session to attack. Therefore, we assume that the adversary registers exactly two parties. Let x and y denote the ordinal identifiers of the two registered parties, in arbitrary order. That is, $\{x, y\} = \{1, 2\}$.
- If the adversary does not share any long-term keys between x and y, or she tries to share more than one long-term keys between them, she will have an AKE advantage of 0. Therefore, we assume that the adversary registers exactly one long-term key $K_{xy}^T = K_{yx}^T$ between x and y.
- Let (X, S) denote the test session, and (Y, S^*) be the session partnered to it (the latter does not necessarily exist). If the adversary exposes (X, S) or (Y, S^*), then she will have an AKE advantage of 0. Therefore, we assume that the adversary does not make the queries $\mathsf{ExposeSS}(X, S)$ or $\mathsf{ExposeSS}(Y, S^*)$.

Algorithm 16. The algorithm which $\mathcal{F}^{\mathcal{O},\mathcal{R}_K^{\mathcal{O}},\mathcal{T}_K^{\mathcal{O}},\mathcal{V}_K^{\mathcal{O}}}(1^n)$ uses to handle Π (i.e., protocol FORSAKES). We assume that the input is well formed, and do not check for syntactical issues. Note that in this experiment, $\mathcal{F}$ only uses the $\mathcal{T}_K^{\mathcal{O}}$ and $\mathcal{V}_K^{\mathcal{O}}$ oracles.

Function $\Pi(-, T, \mathsf{state}_x^s, \mathsf{rnd}_x^s, \mathrm{id}_x, \mathrm{id}_y, m)$

$m' \leftarrow st \leftarrow \lambda$;

Case $m = 0$:

$\mathsf{sid}_x^s \leftarrow \mathsf{rnd}_x^s \,||\, *^r$; // '*' is a wildcard

$m' \leftarrow 1 \,||\, \mathrm{id}_x \,||\, \mathrm{id}_y \,||\, T \,||\, \mathsf{rnd}_x^s$;

$st \leftarrow \mathsf{sid}_x^s \,||\, \texttt{'I'} \,||\, \mathrm{id}_y \,||\, T \,||\, 0^k \,||\, 0^k \,||\, \lambda$;

Case $m = 1 \,||\, \mathrm{id}_s \,||\, \mathrm{id}_r \,||\, T \,||\, n_s$:

if $\left(\mathrm{id}_s \neq \mathrm{id}_y \text{ or } \mathrm{id}_r \neq \mathrm{id}_x\right)$

  return (m', st);

$\mathsf{sid}_x^s \leftarrow n_s \,||\, \mathsf{rnd}_y$;

$\mathrm{MSG}_2 \leftarrow 2 \,||\, \mathrm{id}_x \,||\, \mathrm{id}_y \,||\, T \,||\, \mathsf{sid}_x^s$;

$\mathrm{AUTH}_2 \leftarrow \mathcal{T}_K^{\mathcal{O}}(\mathrm{MSG}_2, 1 \,||\, \mathsf{sid}_x^s)$;

$m' \leftarrow \mathrm{MSG}_2 \,||\, \mathrm{AUTH}_2$;

$st \leftarrow \mathsf{sid}_x^s \,||\, \texttt{'R'} \,||\, \mathrm{id}_y \,||\, T \,||\, 0^k \,||\, 0^k \,||\, \lambda$;

Case $m = \overbrace{2 \,||\, \mathrm{id}_s \,||\, \mathrm{id}_r \,||\, T \,||\, \mathsf{sid}}^{\mathrm{MSG}_2} \,||\, \mathrm{AUTH}_2$:

if $\left(\mathrm{id}_s \neq \mathrm{id}_y \text{ or } \mathrm{id}_r \neq \mathrm{id}_x \text{ or } \mathsf{sid} \neq \mathsf{sid}_x^s\right)$

  return (m', st);

$\mathsf{sid}_x^s \leftarrow \mathsf{sid}$; // No more wildcards.

if $\left(\neg \mathcal{V}_K^{\mathcal{O}}(\mathrm{MSG}_2, 1 \,||\, \mathsf{sid}_x^s, \mathrm{AUTH}_2)\right)$

  return (m', st);

if $\left(m \notin \mathcal{M} \text{ and } \mathsf{fresh}(x, s)\right)$

  output $(\mathrm{MSG}_2, 1 \,||\, \mathsf{sid}_x^s, \mathrm{AUTH}_2)$ and **abort**;

$\mathrm{MSG}_3 \leftarrow 3 \,||\, \mathrm{id}_x \,||\, \mathrm{id}_y \,||\, T \,||\, \mathsf{sid}_x^s$;

$\mathrm{AUTH}_3 \leftarrow \mathcal{T}_K^{\mathcal{O}}(\mathrm{MSG}_3, 1 \,||\, \mathsf{sid}_x^s)$;

$m' \leftarrow \mathrm{MSG}_3 \,||\, \mathrm{AUTH}_3$;

$st \leftarrow \mathsf{sid}_x^s \,||\, \texttt{'I'} \,||\, \mathrm{id}_y \,||\, T \,||\, 0^k \,||\, 0^k \,||\, 1$;

Case $m = \overbrace{3 \,||\, \mathrm{id}_s \,||\, \mathrm{id}_r \,||\, T \,||\, \mathsf{sid}}^{\mathrm{MSG}_3} \,||\, \mathrm{AUTH}_3$:

if $\left(\mathrm{id}_s \neq \mathrm{id}_y \text{ or } \mathrm{id}_r \neq \mathrm{id}_x \text{ or } \mathsf{sid} \neq \mathsf{sid}_x^s\right)$

  return (m', st);

if $\left(\neg \mathcal{V}_K^{\mathcal{O}}(\mathrm{MSG}_3, 1 \,||\, \mathsf{sid}_x^s, \mathrm{AUTH}_3)\right)$

  return (m', st);

if $\left(m \notin \mathcal{M} \text{ and } \mathsf{fresh}(x, s)\right)$

  output $(\mathrm{MSG}_3, 1 \,||\, \mathsf{sid}_x^s, \mathrm{AUTH}_3)$ and **abort**;

$m' \leftarrow \lambda$;

$st \leftarrow \mathsf{sid}_x^s \,||\, \texttt{'R'} \,||\, \mathrm{id}_y \,||\, T \,||\, 0^k \,||\, 0^k \,||\, 1$;

return (m', st);

- If the adversary reveals either the long-term keys K_{xy}^T or K_{yx}^T before or at the same time stage that the test session generates the session key, she will have an AKE advantage of 0. Therefore, we assume that the adversary either does not make $\mathsf{RevealLTK}(x, y)$ or $\mathsf{RevealLTK}(y, x)$ query, or makes either of these queries in a time stage after the session key of the test session is generated.

Lemma 2 pertain to the case $m \notin \mathcal{M}$, where the adversary $\mathcal{A}$ successfully forges a second-or-third message of FORSAKES. It is the "dual" of Lemma 1 in the previous section, which considered the case $m \in \mathcal{M}$. On a high level, Lemma 2 converts an adversary who successfully delivers a second-or-third message $m \notin \mathcal{M}$ (without being detected), to an adversary who forges a message authentication code. For discussions related to the construction and security of a random-oracle based MAC, see A. Let $q_{\mathrm{ro}}(n)$ denote an upper bound on the number of the queries which the adversary makes to the random oracle.

Lemma 2. *Let $\mathcal{A}$ be an adversary against FORSAKES, who succeeds with probability $\epsilon_{\mathcal{A}}(n)$ in delivering a second-or-third message $m \notin \mathcal{M}$ in a non-colliding execution, to a fresh session (x, s), after*

which $\mathsf{acc}_x^s = 1$.

Then, there exists a $(q_{\text{ro}}, 2\sigma + 2, q_{\text{snd}}, q_{\text{snd}})$*–forger* $\mathcal{F}$ *against RO-Multi-MAC (Construction 2) which wins the* Multi-MAC-Forge *(Algorithm 17) with probability* $\epsilon_{\mathcal{F}}(n) = \epsilon_{\mathcal{A}}(n)$.

Furthermore, the running times of $\mathcal{A}$ *and* $\mathcal{F}$ *are related by* $T_{\mathcal{F}}(n) = T_{\mathcal{A}}(n) + q(n) \cdot \text{poly}(n)$.

Proof. By construction, FORSAKES sessions created at time stage T will not accept any message at time stage $T + 1$ or later. Therefore, with no loss of generality, we limit the scope to one time stage; that is, the long-term key does not get updated.

Let $\mathcal{F}$ be the adversary against the RO-Multi-MAC, who takes part in the Multi-MAC-Forge experiment. By definition of this experiment, $\mathcal{F}$ has access to four oracles: $\mathcal{O}$, $\mathcal{R}_K^{\mathcal{O}}$, $\mathcal{T}_K^{\mathcal{O}}$, and $\mathcal{V}_K^{\mathcal{O}}$. During the experiment, $\mathcal{F}$ uses $\mathcal{A}$ as a black-box, and simulates for $\mathcal{A}$ a two-party execution of FORSAKES, and answers to $\mathcal{A}$'s queries as follows (as before, we assume that $\mathcal{A}$ does not make an invalid query, which makes her lose the authentication game):

- *Random oracle queries:* The queries are forwarded to $\mathcal{F}$'s random oracle, and the answers are returned to $\mathcal{A}$.
- $\mathsf{Register}(\,)$: A random id $\in \{0,1\}^\ell$ is selected by $\mathcal{F}$. This id is then returned to $\mathcal{A}$ by $\mathcal{F}$.
- $\mathsf{ShareLTK}(x, y)$: Nothing is actually done, since the Multi-MAC-Forge experiment has already picked a random key in the initialization phase.
- $\mathsf{Send}(x, s, y, m)$: This query is handled by $\mathcal{F}$ as in Algorithm 8; however, when the protocol Π (i.e., FORSAKES) is called, $\mathcal{F}$ runs Algorithm 16. The idea behind this algorithm is simple: It does not have access to the LTK, but uses the tag-generation oracle $\mathcal{T}_K^{\mathcal{O}}$ to generate Auth_2 and Auth_3 (see Protocol 1), and uses the tag-verification oracle $\mathcal{V}_K^{\mathcal{O}}$ to verify them. It also checks whether the incoming message belongs to the set $\mathcal{M}$. If a second-or-third message has a valid tag, but is not in $\mathcal{M}$, it means that $\mathcal{A}$ has successfully forged a valid message (assuming the session is fresh). In this case, the forger $\mathcal{F}$ outputs the forgery just found, and finishes the game successfully.
- $\mathsf{ExposeSS}(x, s)$: First, $\mathcal{F}$ uses the $\mathcal{R}_K^{\mathcal{O}}$ oracle to get the session and integrity keys. This is done by setting $sk_x^s \leftarrow \mathcal{R}_K^{\mathcal{O}}(0 \,||\, \mathsf{sid}_x^s)$ and $ik_x^s \leftarrow \mathcal{R}_K^{\mathcal{O}}(1 \,||\, \mathsf{sid}_x^s)$. Next, state_x^s is updated, by setting sk_x^s and ik_x^s into the session and integrity key "placeholders" of state_x^s. Finally, the pair $(\mathsf{state}_x^s, \mathsf{rnd}_x^s)$ is returned to $\mathcal{A}$.
- $\mathsf{RevealLTK}(x, y)$: This query is not allowed at the current time stage, since it makes all sessions unfresh. However, $\mathcal{A}$ can be given the value of the long-term key at the next time stage. To this end, $\mathcal{F}$ simply queries the $\mathcal{R}_K^{\mathcal{O}}$ oracle at λ, to receive the key $K' = \mathcal{R}_K^{\mathcal{O}}(\lambda) = \mathcal{O}(K \,||\, \lambda) = \mathcal{O}(K)$. This is the value of K in the next time stage, and is returned to $\mathcal{A}$.
- $\mathsf{Test}(x, y)$: First, $\mathcal{F}$ flips a random coin b. If $b = 1$, a random $k(n)$-bit binary string is returned to $\mathcal{A}$. Otherwise, $\mathcal{F}$ sets $sk_x^s \leftarrow \mathcal{R}_K^{\mathcal{O}}(0 \,||\, \mathsf{sid}_x^s)$, and returns sk_x^s to $\mathcal{A}$.
- $\mathsf{Guess}(b')$: This query finishes the game with failure, since it is the ultimate query of $\mathcal{A}$, and she has not forge any messages yet.

Since the execution is non-colliding, Lemma 1 shows that no messages output by sessions can be delivered with a fake source, or at a fake destination. The forger $\mathcal{F}$ simulates the view of $\mathcal{A}$ successfully. $\mathcal{F}$ succeeds if and only if $\mathcal{A}$ succeeds. Therefore, $\epsilon_{\mathcal{F}}(n) = \epsilon_{\mathcal{A}}(n)$.

The maximum number of queries $\mathcal{F}$ makes to its oracles are as follows:

- q_{ro} queries at $\mathcal{O}$.
- $2\sigma + 2$ queries at $\mathcal{R}_K^{\mathcal{O}}$. There are at most σ sessions on the system, each of which can be exposed. Every $\mathsf{ExposeSS}$ query requires 2 queries at $\mathcal{R}_K^{\mathcal{O}}$. Furthermore, there can be at most one $\mathsf{RevealLTK}$ and one Test query, each of which requires 1 query at $\mathcal{R}_K^{\mathcal{O}}$. Therefore, the total number of queries at $\mathcal{R}_K^{\mathcal{O}}$ is at most $2\sigma + 2$.

- q_{snd} queries at $\mathcal{T}_K^{\mathcal{O}}$, and q_{snd} queries at $\mathcal{V}_K^{\mathcal{O}}$. This is because each message sent may need a tag verification, and a tag generation.

We now pertain to the running-time analysis of $\mathcal{F}$. Notice that $\mathcal{F}$ answers each query of $\mathcal{A}$ with at most a polynomial overhead. Therefore, the running time $T_{\mathcal{F}}(n)$ of $\mathcal{F}$ is $T_{\mathcal{A}}(n) + q(n) \cdot \text{poly}(n)$. ■

It is now easy to prove that FORSAKES is a secure AKE protocol, because we just proved that the adversary has very little chance of delivering messages at the wrong destination, or forging messages.

Theorem 2 (Main Theorem). *In the two-party setting, FORSAKES is a secure AKE protocol, as per Definition 3.*

Proof. Define the following events:

- E_1: Two parties receive the same identifier.
- E_2: The system is colliding.
- E_3: The adversary successfully delivers a second-or-third message $m \in \mathcal{M}$, with a fake source or at a wrong destination.
- E_4: The adversary successfully forges a second-or-third message $m \notin \mathcal{M}$, whose destination session is fresh.

The facts and lemmas in this and previous sections proved that the probability that either of of these events happen is exponentially small in n. Therefore, let us condition the probabilities on the event $\overline{E_1} \wedge \overline{E_2} \wedge \overline{E_3} \wedge \overline{E_4}$.

Based on the conditioning above, the adversary has three choices:

1. Faithfully deliver a message;
2. Delay the delivery of a message beyond a time stage;
3. Delete a message.

In FORSAKES, option (2) causes the destination session to reject, and option (3) prevents the generation of the session key at the destination session. Therefore, the adversary is left with option (1). However, if she delivers messages faithfully, and targets a fresh session using the Test query, she will receive a random and independent value *key*, regardless of the internal coin toss of Test. Therefore, the advantage of $\mathcal{A}$ in guessing the value of the coin is 0. ■

The following corollary is immediate by combining Theorem 2 with Theorem 1.

Corollary 3. *FORSAKES is a secure AKE protocol, regardless of the number of parties in the system.*

6 FORSAKES in Practice

6.1 Replacing the Random Oracle

One of the most important issues regarding FORSAKES is its use of the random oracle model. As in [35], we note that the random oracle was merely a tool used in the proofs, and it can be replaced with pseudorandom functions [31]. This is because we did not use any of the random oracle facilities, such as the ability to intercept the queries or program the response.

There are efficient pseudorandom functions such as [32], which can be used in practice if provable security is desired. However, if even faster implementations are necessary, we suggest the use of encryption functions such as AES, or hash functions such as SHA-1. See [35, Section 6] for more information. It is also possible to use algorithms such as PBKDF2 (Password-Based Key Derivation Function 2), proposed by RFC 2898.

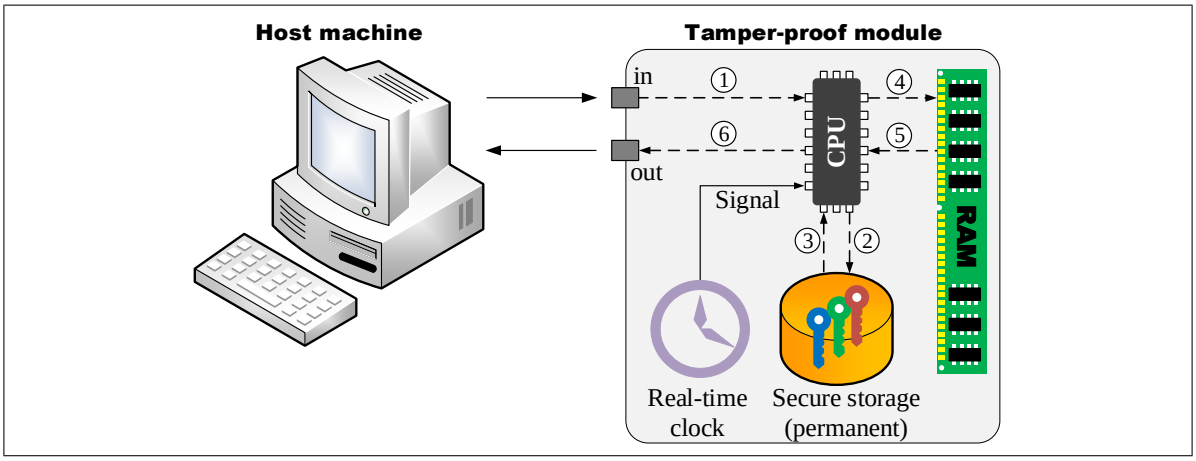


Figure 1. A FORSAKES TPM consists of input/output ports, a CPU, an RTC, a RAM, and a permanent storage. Its main task is to securely host the LTKs, and compute the session and integrity keys, based on the information provided by the host machine. The TPM CPU reads information from the input port (1), addresses the permanent storage (2), retrieves the relevant key (3), communicates with the RAM (4 & 5), and writes information to the output port (6). The real-time clock signals the CPU every τ seconds, upon which the CPU updates every key in the permanent storage.

Remark 4. While attacks such as the *length-extension attack* against keyed hash functions do not seem applicable to our protocol (due to the fixed length of the input), it is wise to use constructions such as HMAC [56] instead of keyed hash functions. ◁

6.2 Implementation on a Constraint Device

FORSAKES is quite efficient, and as described above, can be implemented using hash functions only. Therefore, it is ideal for implementation on constraint devices, such as security tokens and smart cards. The device, also called a *tamper-proof module* (TPM), must be equipped with a *real-time clock* (RTC), and a secure storage for long-term keys. Figure 1 shows the internals of the TPM. Externally, it can be made similar to a SecurID®.

6.3 Implementation Subtleties

We suggest using an *in-memory database* to store session and key information. This approach has several advantages:

- Each session can be stored as a row of a `sessions` table. The table is indexed based in the session ID, which helps in fast retrieval of the session information corresponding to an incoming message. Furthermore, the SQL queries support the `like` keyword, which is ideal for matching with wildcards (a requirement in FORSAKES).
- Databases support the concept of *transactions*. Consider the case where the LTKs should be updated in the middle of the computation of a MAC, or some session key. Using transactions, one can be sure that concurrent accesses to the database are isolated properly. That said, the implementor may need to incorporate proper concurrency controls (such as locks or semaphores) in their code.
- In-memory database have the ability to occasionally save information to some non-volatile memory, to increase the reliability and to perform recovery from a crash. However, notice that it is important to securely wipe the information which is no longer needed.

7 Conclusion and Future Work

In this paper, we formalized a model and definition for *authenticated key exchange* (AKE) protocols, whose long-term keys update regularly. The security definition required *forward security*, meaning that the revelation of long-term keys in later time stages should not compromise the security of previous session keys. We also proposed an AKE protocol called FORSAKES, and rigorously proved its security.

To improve this work, one can consider models where the adversary has the ability to desynchronize the long-term keys between any pair of parties. This attack models the practical scenario when the clocks of either party is skewed. The protocol should then detect the desynchronization, and resynchronize the keys. To this end, each pair of parties can share a non-updating LTK, which is only used for resynchronization, and has no purpose in the actual key exchange.

Another line of work is the analysis of side-channel attacks on the TPM implementations of FORSAKES, and propose improvements which foil such attacks.

References

[1] R. M. Needham, M. D. Schroeder, Using Encryption for Authentication in Large Networks of Computers, Communications of the ACM 21 (12) (1978) 993–999.

[2] D. E. Denning, G. M. Sacco, Timestamps in Key Distribution Protocols, Communications of the ACM 24 (8) (1981) 533–536.

[3] D. Otway, O. Rees, Efficient and Timely Mutual Authentication, ACM SIGOPS Operating Systems Review 21 (1) (1987) 8–10.

[4] C. Boyd, Hidden Assumptions in Cryptographic Protocols, IEE Proceedings of Computers and Digital Techniques 137 (6) (1990) 433–436.

[5] J. Clark, J. Jacob, On the Security of Recent Protocols, Information Processing Letters (IPL) 56 (3) (1995) 151–155.

[6] R. Bird, I. Gopal, A. Herzberg, P. Janson, S. Kutten, R. Molva, M. Yung, Systematic Design of Two-Party Authentication Protocols, in: Advances in Cryptology—CRYPTO '91, Springer, Santa Barbara, California, USA, 1992, pp. 44–61.

[7] D. Basin, C. Cremers, S. Meier, Provably Repairing the ISO/IEC 9798 Standard for Entity Authentication, in: Principles of Security and Trust, Springer, 2012, pp. 129–148.

[8] J. Clark, J. Jacob, A Survey of Authentication Protocol Literature: Version 1.0, available from http://www.cs.york.ac.uk/~jac/PublishedPapers/reviewV1_1997.pdf (November 1997).

[9] C. Boyd, A. Mathuria, Protocols for Authentication and Key Establishment, Springer, 2003.

[10] M. Bellare, P. Rogaway, The Exact Security of Digital Signatures—How to Sign with RSA and Rabin, in: Advances in Cryptology—EUROCRYPT '96, Vol. 1070 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 1996, pp. 399–416.

[11] H. Krawczyk, HMQV: A High-Performance Secure Diffie-Hellman Protocol (Extended Abstract), in: Advances in Cryptology—CRYPTO'05, Springer, Santa Barbara, California, 2005, pp. 546–566, full version is available at http://eprint.iacr.org/2005/176.

[12] A. Menezes, M. Qu, S. Vanstone, Some New Key Agreement Protocols Providing Implicit Authentication, in: Presented at the Workshop on Selected Areas in Cryptography (SAC '95), 1995, pp. 22–32.

[13] L. Law, A. Menezes, M. Qu, J. Solinas, S. Vanstone, An Efficient Protocol for Authenticated Key Agreement, Designs, Codes and Cryptography 28 (2) (2003) 119–134.

[14] A. Menezes, Another Look at HMQV, Journal of Mathematical Cryptology 1 (1) (2007) 47–64, available from http://eprint.iacr.org/2005/205.

[15] C. G. Günther, An Identity-Based Key-Exchange Protocol, in: Advances in Cryptology—EUROCRYPT '89, Springer, Houthalen, Belgium, 1989, pp. 29–37.

[16] W. Diffie, P. C. Oorschot, M. J. Wiener, Authentication and Authenticated Key Exchanges, Designs, Codes and Cryptography 2 (2) (1992) 107–125.

[17] S. Blake-Wilson, D. Johnson, A. Menezes, Key Agreement Protocols and Their Security Analysis, in: Proceedings of the 6th IMA International Conference on Cryptography and Coding (IMACC '97), Springer, Cirencester, UK, 1997, pp. 30–45.

[18] M. Bellare, D. Pointcheval, P. Rogaway, Authenticated Key Exchange Secure against Dictionary Attacks, in: Advances in Cryptology—EUROCRYPT '00, Springer, Bruges, Belgium, 2000, pp. 139–155.

[19] R. Canetti, H. Krawczyk, Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels, in: Advances in Cryptology—EUROCRYPT '01, Springer, Innsbruck, Austria, 2001, pp. 453–474, full version is available at http://eprint.iacr.org/2001/040.

[20] W. Diffie, M. E. Hellman, New Directions in Cryptography, IEEE Transactions on Information Theory IT-22 (6) (1976) 644–654.

[21] H.-Y. Chien, SASI: A New Ultralightweight RFID Authentication Protocol Providing Strong Authentication and Strong Integrity, IEEE Transactions on Dependable and Secure Computing 4 (4) (2007) 337–340.

[22] T. Cao, E. Bertino, H. Lei, Security Analysis of the SASI Protocol, IEEE Transactions on Dependable and Secure Computing 6 (1) (2009) 73–77.

[23] R. C.-W. Phan, Cryptanalysis of a New Ultralightweight RFID Authentication Protocol—SASI, IEEE Transactions on Dependable and Secure Computing 6 (4) (2009) 316–320.

[24] H.-M. Sun, W.-C. Ting, K.-H. Wang, On the Security of Chien's Ultralightweight RFID Authentication Protocol, IEEE Transactions on Dependable and Secure Computing 8 (2) (2011) 315–317.

[25] I. Wiener, Sample SecurID Token Emulator with Token Secret Import, available from http://archives.neohapsis.com/archives/bugtraq/2000-12/0428.html (December 2000).

[26] A. Biryukov, J. Lano, B. Preneel, Cryptanalysis of the Alleged SecurID Hash Function, in: Selected Areas in Cryptography (SAC 2003), Springer, Windsor, Ontario, Canada, 2004, pp. 130–144, extended version available from http://eprint.iacr.org/2003/162.

[27] S. Contini, Y. L. Yin, Fast Software-Based Attacks on SecurID, in: Fast Software Encryption (FSE 2004), Springer, Delhi, India, 2004, pp. 454–471.

[28] A. Biryukov, J. Lano, B. Preneel, Recent Attacks on Alleged SecurID and Their Practical Implications, Computers & Security 24 (5) (2005) 364–370.

[29] R. Canetti, S. Halevi, J. Katz, A Forward-Secure Public-Key Encryption Scheme, Journal of Cryptology 20 (3) (2007) 265–294, see [57] for the conference version.

[30] B. LaMacchia, K. Lauter, A. Mityagin, Stronger Security of Authenticated Key Exchange, in: Proceedings of the 1st International Conference on Provable Security (ProvSec '07), Springer, Wollongong, Australia, 2007, pp. 1–16.

[31] O. Goldreich, S. Goldwasser, S. Micali, How to Construct Random Functions, Journal of the ACM (JACM) 33 (4) (1986) 792–807.

[32] A. Banerjee, C. Peikert, A. Rosen, Pseudorandom Functions and Lattices, in: Advances in Cryptology—EUROCRYPT 2012, Springer, 2012, pp. 719–737.

[33] U. Feige, A. Fiat, A. Shamir, Zero-Knowledge Proofs of Identity, Journal of Cryptology 1 (2) (1988) 77–94.

[34] M. S. Dousti, R. Jalili, Efficient Statistical Zero-Knowledge Authentication Protocols for Smart Cards Secure Against Active & Concurrent Quantum Attacks, submitted to Wiley Security and Communication Networks. Available from http://eprint.iacr.org/2013/709 (2013).

[35] M. Bellare, P. Rogaway, Entity Authentication and Key Distribution, in: Advances in Cryptology—CRYPTO '93, Springer, Santa Barbara, California, USA, 1993, pp. 232–249.

[36] S. Blake-Wilson, A. Menezes, Entity Authentication and Authenticated Key Transport Protocols: Employing Asymmetric Techniques, in: Proceedings of the 5th International Workshop on Security Protocols (SPW '97), Springer, Paris, France, 1998, pp. 137–158.

[37] M. Bellare, P. Rogaway, Provably Secure Session Key Distribution: The Three Party Case, in: Proceedings of the 27th Annual ACM Symposium on Theory of Computing (STOC '95), ACM, Las Vegas, Nevada, USA, 1995, pp. 57–66.

[38] K.-K. R. Choo, C. Boyd, Y. Hitchcock, G. Maitland, On Session Identifiers in Provably Secure Protocols: The Bellare–Rogaway Three-Party Key Distribution Protocol Revisited, in: Security in Communication Networks (SCN 2004), Springer, Amalfi, Italy, 2005, pp. 351–366.

[39] V. Shoup, A. Rubin, Session Key Distribution Using Smart Cards, in: Advances in Cryptology—EUROCRYPT '96, Springer, Saragossa, Spain, 1996, pp. 321–331.

[40] M. Bellare, R. Canetti, H. Krawczyk, A Modular Approach to the Design and Analysis of Authentication and Key Exchange Protocols (Extended Abstract), in: Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC '98), ACM, Dallas, Texas, USA, 1998, pp. 419–428.

[41] V. Shoup, On Formal Models for Secure Key Exchange, Tech. rep., IBM Zurich Research Lab, version 4 is available at http://eprint.iacr.org/1999/012 (1999).

[42] C. Brzuska, M. Fischlin, B. Warinschi, S. C. Williams, Composability of Bellare-Rogaway Key Exchange Protocols, in: Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011), ACM, Chicago, Illinois, USA, 2011, pp. 51–62.

[43] C. Brzuska, M. Fischlin, N. P. Smart, B. Warinschi, S. C. Williams, Less is More: Relaxed yet Composable Security Notions for Key Exchange, International Journal of Information Security 12 (4) (2013) 267–297.

[44] R. Canetti, H. Krawczyk, Universally Composable Notions of Key Exchange and Secure Channels (Extended Abstract), in: Advances in Cryptology—EUROCRYPT '02, Springer, Amsterdam, The Netherlands, 2002, pp. 337–351, full version is available at http://eprint.iacr.org/2002/059.

[45] R. Canetti, Universally Composable Security: A New Paradigm for Cryptographic Protocols (Extended Abstract), in: Proceedings of the 42nd Annual IEEE Symposium on Foundations of Computer Science (FOCS '01), IEEE Computer Society, Washington, DC, USA, 2001, p. 136, see [58] for the full version.

[46] D. Hofheinz, J. Müller-Quade, R. Steinwandt, Initiator-Resilient Universally Composable Key Exchange, in: Proceedings of the 8th European Symposium on Research in Computer Security (ESORICS 2003), Springer, Gjøvik, Norway, 2003, pp. 61–84.

[47] R. Canetti, S. Halevi, J. Katz, Y. Lindell, P. MacKenzie, Universally Composable Password-Based Key Exchange, in: Advances in Cryptology—EUROCRYPT 2005, Springer, Aarhus, Denmark, 2005, pp. 404–421, full version is available from http://eprint.iacr.org/2005/196.

[48] J. Camenisch, A. Lysyanskaya, G. Neven, Practical Yet Universally Composable Two-Server Password-Authenticated Secret Sharing, in: Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS 2012), ACM, Raleigh, NC, USA, 2012, pp. 525–536.

[49] C. J. Cremers, Session-state Reveal Is Stronger Than Ephemeral Key Reveal: Attacking the NAXOS Authenticated Key Exchange Protocol, in: Proceedings of the 7th International Conference on Applied Cryptography and Network Security (ACNS '09), Springer, Paris-Rocquencourt, France, 2009, pp. 20–33.

[50] A. P. Sarr, P. Elbaz-Vincent, J.-C. Bajard, A New Security Model for Authenticated Key Agreement, in: Proceedings of the 7th International Conference on Security and Cryptography for Networks (SCN '10), Springer, Amalfi, Italy, 2010, pp. 219–234.

[51] K. Yoneyama, Y. Zhao, Taxonomical Security Consideration of Authenticated Key Exchange Resilient to Intermediate Computation Leakage, in: Proceedings of the 5th International Conference on Provable Security (ProvSec 2011), Springer, Xi'an, China, 2011, pp. 348–365.

[52] K.-K. R. Choo, C. Boyd, Y. Hitchcock, Examining Indistinguishability-Based Proof Models for Key Establishment Protocols, in: Advances in Cryptology—ASIACRYPT '05, Springer, Chennai, India, 2005, pp. 585–604.

[53] K.-K. R. Choo, Secure Key Establishment, Springer, 2008.

[54] C. Cremers, Examining Indistinguishability-Based Security Models for Key Exchange Protocols: The Case of CK, CK-HMQV, and eCK, in: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS '11), ACM, Hong Kong, China, 2011, pp. 80–91.

[55] M. Bellare, P. Rogaway, Random Oracles are Practical: A Paradigm for Designing Efficient Protocols, in: Proceedings of the 1st Annual ACM Conference on Computer and Communications Security, ACM, 1993, pp. 62–73.

[56] M. Bellare, R. Canetti, H. Krawczyk, Keying Hash Functions for Message Authentication, in: Advances in Cryptology—CRYPTO '96, Springer, Santa Barbara, California, USA, 1996, pp. 1–15.

[57] R. Canetti, S. Halevi, J. Katz, A Forward-Secure Public-Key Encryption Scheme, in: Advances in Cryptology—Eurocrypt 2003, Springer, 2003, pp. 255–271, see [29] for the journal version.

[58] R. Canetti, Universally Composable Security: A New Paradigm for Cryptographic Protocols, Cryptology ePrint Archive, Report 2000/067, available from http://eprint.iacr.org/2000/067. See [45] for the conference version (2005).

[59] J. Katz, Y. Lindell, Introduction to Modern Cryptography: Principles and Protocols, 1st Edition, Chapman & Hall/CRC, 2007.

A Constructing a Secure MAC Using a Random Oracle

In this appendix, we show how to construct a *message authentication code* (MAC) from a random oracle, and prove its security. A MAC is a tag which, in conjunction with a message, proves its authenticity. More formally, a MAC is a triple of efficient algorithms $(\mathsf{Gen}, \mathsf{MAC}, \mathsf{Verif})$, which satisfy the following [59, Section 4.3]:

1. The *key generation algorithm* Gen, which takes 1^n as input, and outputs a key K with $|K| \geq n$.
2. The *tag-generation algorithm* MAC, which takes the key K and a message $m \in \{0,1\}^*$ as input, and outputs a tag t.
3. The *tag-verification algorithm* Verif, which takes the key K, a message $m \in \{0,1\}^*$, and a tag t, and outputs a single bit. It outputs 1 if and only if verification succeeds.

The *syntactic* requirement is that for any key K generated by Gen and any message $m \in \{0,1\}^*$, we have $\mathsf{Verif}(K, m, \mathsf{MAC}(K, m)) = 1$. Informally, the *security* requirement is that no efficient adversary $\mathcal{F}$ can forge a valid tag, even if she can obtain valid tags on messages of her choice. This definition is formalized through the following experiment (called MAC-Forge):

1. Generate a key K by running Gen on the security parameter 1^n.
2. Run the adversary $\mathcal{F}$ on the security parameter 1^n, and give her oracle access to $\mathsf{MAC}(K, \cdot)$. That is, the adversary can query the oracle on messages of her choice, and receive the corresponding tag. Let Q be the list of queries made by $\mathcal{F}$ to the oracle.
3. Eventually, $\mathcal{F}$ outputs a pair (m, t). She wins this experiment if $m \notin Q$ and $\mathsf{Verif}(K, m, t) = 1$. Otherwise, she loses the game.[4]

In a secure MAC, it is required that the probability that any efficient A wins the experiment is negligible in n.

It is possible to construct a secure MAC from a random oracle, and a pre-shared key. One such attempt, called *RO-MAC*, is described in Construction 1.

Construction 1 (RO-MAC). Let $a \stackrel{\text{def}}{=} a(n)$ and $b \stackrel{\text{def}}{=} b(n)$ be two polynomials, and $\mathcal{O}\colon \{0,1\}^* \to \{0,1\}^a$ be a random oracle. The RO-MAC is obtained from $\mathcal{O}$ as follows: $\mathsf{Gen}(1^n)$ picks a random key K from $\{0,1\}^b$. The tag of a message m is $\mathcal{O}(K \,||\, m)$. Finally, $\mathsf{Verif}(K, m, t)$ outputs 1 if and only if $t = \mathcal{O}(K \,||\, m)$. ◁

[4] If the adversary does not output anything, or does not output a pair, she clearly loses. Let us assume, with no loss of generality, that this does not happen.

It is easy to show that RO-MAC is a secure MAC, but we do not prove this fact here. This is because the security of our protocol (FORSAKES) does not depend on the security of RO-MAC. Rather, FORSAKES depends on a more elaborate MAC, which we will construct and prove its security next.

The new construction is called *RO-Multi-MAC*. In this construction, there is a single pre-shared key K, which is used to generate MAC keys used to form MACs. Construction 2 provides a more formal description of RO-Multi-MAC.

Construction 2 (RO-Multi-MAC). Let a, b, and $\mathcal{O}$ be as in Construction 1. The RO-Multi-MAC is obtained from $\mathcal{O}$ as follows: $\mathsf{Gen}(1^n)$ picks a random key K from $\{0,1\}^b$. The tags are *not* obtained from K directly. Rather, any party who knows K can query $\mathcal{O}(K \,||\, \cdot)$ at points of his choice (x's), to obtain one or more MAC keys (k_x's). The tag of a message m under a valid MAC key k_x is a pair $(x, t = \mathcal{O}(k_x \,||\, m))$.

Finally, $\mathsf{Verif}(K, m, x, t)$ outputs 1 if and only if $t = \mathcal{O}(\mathcal{O}(K \,||\, x) \,||\, m)$. ◁

Below, we will prove that RO-Multi-MAC is *strongly secure*. That is, we give the adversary abilities beyond what is required in a MAC-Forge experiment, and show that RO-Multi-MAC is secure even against such strong adversaries.

In a normal MAC-Forge experiment, the adversary obtains tags on messages of her choice. We would like to extend her abilities, and allow her to reveal any number of MAC keys (k's). She wins the experiment if and only if the tag is a valid MAC under any of the *unrevealed* k's, and she has not previously obtained a tag for the message. Further more, the adversary is given access to a $\mathcal{V}_K^{\mathcal{O}}$ oracle, and she can check for the validity of a tag for messages of her choice.

An experiment designed to capture the security of RO-Multi-MAC. Let use formalize the security of RO-Multi-MAC, by designing a new experiment called Multi-MAC-Forge. In this experiment, the adversary $\mathcal{F}$ is given access to four oracles:

1. **The random oracle $\mathcal{O}$:** An ordinary random oracle $\mathcal{O}\colon \{0,1\}^* \to \{0,1\}^a$.
2. **The reveal oracle $\mathcal{R}_K^{\mathcal{O}}$:** On input x, reveals the MAC key $k_x = \mathcal{O}(K \,||\, x)$. This oracle also adds x to the set rev. The set is used when the experiment wants to verify the output of the adversary: She is not allowed to output the tag for a message under a revealed key.
3. **The tag-generation oracle $\mathcal{T}_K^{\mathcal{O}}$:** On input (m, x), returns the MAC t of m under k_x, by computing $k_x \leftarrow \mathcal{O}(K \,||\, x)$ and $t \leftarrow \mathcal{O}(k_x \,||\, m)$. This oracle also adds the pair (m, x) to the set Asked. The set is used when the experiment wants to verify the output of the adversary: She is not allowed to output the tag for a message for which she has already obtained a tag.
4. **The verification oracle $\mathcal{V}_K^{\mathcal{O}}$:** On input (m, x, t), computes the valid tag for m under the key $k_x \leftarrow \mathcal{O}(K \,||\, x)$, and returns true if and only if the tag equals t.

Remark 5. We will assume, without loss of generality, that $\mathcal{F}$ never makes the same query twice to any of her oracles. Moreover, if she obtains a tag for some message, she will not verify the tag. ◁

Algorithm 17. The MULTI-MAC-FORGE experiment for $\mathcal{F}$ against RO-Multi-MAC.

$\mathbf{Exp}^{\text{mmf}}_{\mathcal{F},\text{RO-MULTI-MAC}}(n)$
$K \leftarrow_R \{0,1\}^b$;
$(m,x,t) \leftarrow \mathcal{F}^{\mathcal{O},\mathcal{R}^{\mathcal{O}}_K,\mathcal{T}^{\mathcal{O}}_K,\mathcal{V}^{\mathcal{O}}_K}(1^n)$;
if $\big(x \notin rev$ and $(m,x) \notin$ Asked and $\mathcal{V}^{\mathcal{O}}_K(m,x,t)\big)$
  output 1 and **abort**;
output 0;

The MULTI-MAC-FORGE experiment is described by Algorithm 17. The experiment starts by picking a random long-term key K, and then allowing the adversary to interact with the four oracles described above. When the adversary outputs (m,x,t), this output is verified for validity. The output is considered valid if the key at point x (denoted k_x) is not revealed, a tag for m under k_x is not requested, and t is a valid tag for m under k_x.

Bounding the number of adversarial queries. To prove the security of RO-Multi-MAC, we require a bound on the number of queries the adversary can make. Assume that $\mathcal{F}$ makes at most q_o, q_r, q_t and q_v queries at $\mathcal{O}$, $\mathcal{R}^{\mathcal{O}}_K$, $\mathcal{T}^{\mathcal{O}}_K$, and $\mathcal{V}^{\mathcal{O}}_K$, respectively. In general, q_o, q_r, q_t, and q_v can be functions of the security parameter n. Define $q \stackrel{\text{def}}{=} q_o + q_r + 2q_t + 2q_v + 2$.

Casting the experiment in a different way. Algorithm 17 is rather difficult to analyze. To ease the security analysis, we rewrite Algorithm 17 as Algorithm 18, using a technique called *early sampling*. In this technique, a number of random points are selected prior to the start of the experiment. When the random oracle is queried at some point, one of the random points are returned (with some adjustments). The alternative technique—which we do not use here—is called *lazy sampling*, where the answers of the random oracle are *not* pre-selected. Rather, they are picked randomly and on-the-fly.

Theorem 3. *For any $n \in \mathbb{N}$ and any (q_o,q_r,q_t,q_v)–adversary $\mathcal{F}$ against RO-Multi-MAC (Algorithm 18), the probability that $\mathcal{F}$ wins the* MULTI-MAC-FORGE *experiment is at most $(q^2+1)2^{-a} + q2^{-d} + 2^{-b}$, where $d = \min\{a,b\}$.*

Proof. First, let us count the total number of queries at $\mathcal{O}$: The adversary asks q_o queries directly, plus q_r queries via $\mathcal{R}^{\mathcal{O}}_K$, plus $2q_t$ queries via $\mathcal{T}^{\mathcal{O}}_K$, plus $2q_v$ queries via $\mathcal{V}^{\mathcal{O}}_K$. Furthermore, the challenger asks a verification query, which requires 2 addition queries at $\mathcal{O}$. Totally, there will be $q_o+q_r+2q_t+2q_v+2 = q$ queries from $\mathcal{O}$.

Now consider the following events:

- E_1: Not all o_i's are distinct.
- E_2: The key K has a common prefix of length $d = \min\{a,b\}$ with some o_i.
- E_3: The adversary queries $\mathcal{O}$ on a value whose prefix is K.

The first two events do not depend on the adversary at all, and:

$$\Pr[E_1] \le \tbinom{q}{2}2^{-a} \le q^2 2^{-a}\,,$$
$$\Pr[E_2] = q2^{-d}\,.$$

Furthermore,

$$\Pr[E_3 \mid \overline{E_1} \wedge \overline{E_2}] = 2^{-b}\,.$$

Algorithm 18. The $\mathbf{Exp}^{\mathsf{mmf}}_{\mathcal{F},\mathsf{RO\text{-}Multi\text{-}MAC}}(n)$ experiment (Algorithm 17) with *early sampling*.

Initialization.

$o_i \leftarrow_R \{0,1\}^a$ for $i \in [q]$;
$K \leftarrow_R \{0,1\}^b$;
$O, rev, \mathsf{Asked} \leftarrow \varnothing$ and $i \leftarrow 0$;

Query-response. Respond to $\mathcal{F}$'s query's as follows:

$\mathcal{O}(x)$
  $i \leftarrow i+1$;
  if $\big(x \in \mathrm{dom}(O)\big)$
    $o_i \leftarrow O(x)$;
  $O \leftarrow O \cup \{(x, o_i)\}$;
  return o_i;

$\mathcal{R}^{\mathcal{O}}_K(x)$
  $rev \leftarrow rev \cup \{x\}$;
  $k \leftarrow \mathcal{O}(K \,||\, x)$;
  return k;

$\mathcal{T}^{\mathcal{O}}_K(m,x)$
  $\mathsf{Asked} \leftarrow \mathsf{Asked} \cup \{(m,x)\}$;
  $k_x \leftarrow \mathcal{O}(K \,||\, x)$;
  return $\mathcal{O}(k_x \,||\, m)$;

$\mathcal{V}^{\mathcal{O}}_K(m,x,t)$
  $k_x \leftarrow \mathcal{O}(K \,||\, x)$;
  if $(t = \mathcal{O}(k_x \,||\, m))$
    return 1;
  return 0;

Finalization. A outputs her guess.

$(m,x,t) \leftarrow \mathcal{F}^{\mathcal{O},\mathcal{R}^{\mathcal{O}}_K,\mathcal{T}^{\mathcal{O}}_K,\mathcal{V}^{\mathcal{O}}_K}(1^n)$;
if $\big(x \notin rev$ and $(m,x) \notin \mathsf{Asked}$ and $\mathcal{V}^{\mathcal{O}}_K(m,x,t)\big)$
  output 1 and **abort**;
output 0;

Let us assume with no loss of generality that the adversary asks each query only once, and she never verifies a tag she receives from $\mathcal{T}^{\mathcal{O}}_K$. Conditioned on $E = \overline{E_1} \wedge \overline{E_2} \wedge \overline{E_3}$, all responses from the adversary receives from $\mathcal{O}$ are independent from K, and one can remove lines 6–8 from the algorithm. Therefore, conditioned on E, the the probability that the adversary guesses the correct tag is exactly 2^{-a}. Using a union bound, the theorem follows. ■