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

(Communicated by Joan-Josep Climent)

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.

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 [54], was soon found to have a subtle flaw [37]. Another example is the Otway–Rees [55] protocol, which is shown to be susceptible to typing attacks [15,32]. In another work, Bird et al. [10] 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 [2]. A survey of attacks against entity authentication and AKE protocols can be found in [16,33].

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

2010 Mathematics Subject Classification: Primary: 94A62; Secondary: 68M12, 94A60.
Key words and phrases: Authenticated key exchange protocol, forward security, key evolving schemes, provable security, security model.
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 [9].

A prime example of the first two items is apparent in the work of Krawczyk [48]: 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 [50,53] 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 [52] 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 [48], 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 [45], used famously in the Station-to-Station protocol (STS) [39], and formalized by [5,13,25,48]. 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 [38], 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 [28], provides a good example. SASI attempts to provide forward security without depending on DH, but is found to be flawed by several researchers [27, 56, 60]. 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 $K_{\text{new}}$, she will be unable to infer the previous LTK $K_{\text{old}}$, as doing so requires inverting some one-way function. Since the ephemeral keys depend essentially on $K_{\text{old}}$, it must be impossible for the adversary to obtain the ephemeral keys from $K_{\text{new}}$. 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 [11, 12, 34, 61]). This technique is also used in forward-secure public-key encryption [23], 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. As explained in [23], 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.

1Note 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.
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.

An issue which might arise is the use of the 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 the absolute value of the time. Rather, it is important to keep the time synchrony between the parties in such a way that they perceive the same time stage all the time: When the update frequency decreases (say, once a day), an asynchrony of a few seconds might be acceptable.

To keep parties synchronized, a number of solutions exists:

- The current technology allows us to produce devices, such as the RSA SecurID® (Figure 1(a)), which can keep the time synchrony for a long time. Furthermore, general-purpose hardware such as PCs or mobile phones are quite capable of keeping the time precisely. Solutions such as Google Authenticator™ or HDE OTP (Figure 1(b)) depend on this feature: Once the device time is set, it is rarely skewed due to hardware incompetence.
- Another solution is to incorporate a trusted time source, such as a time server. This solution is applicable to corporate networks as well as the Internet. However, it requires a trust infrastructure, such as PKI, to validate the time returned by the time server.
- The third solution is to provide legitimate parties with a mechanism to dynamically resynchronize their LTKs on demand. We will investigate this option later in Section 7.

Despite these solutions, the clock synchronization problem has not been totally resolved yet [51]. However, many major authentication and AKE protocols are time based. Examples include Kerberos and the Time-Based One-Time Password (TOTP) algorithm, defined in RFC 6238. The former is widely used by Microsoft Active Directory®, while the latter is used in two-factor authentication to Amazon, Dropbox, Evernote, Facebook, Google, Microsoft, and Yahoo, among the others (HDE OTP is a sample implementation of RFC 6238).

The bottom line is that, while time-based security protocols face the potential problem of desynchronized clocks, they are vastly used in everyday networking. It
is therefore beneficial—both in theory and practice—to study time-based authentication and AKE protocols from a theoretical perspective, and formalize a model and definition to capture their security.

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.
- 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, [5, 48, 49] 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 [44], to achieve a secure AKE protocol in the standard model. See Section 6 for more information.
- While there are efficient and provably secure pseudorandom functions [1], 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 [43], 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 [40, 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 [6]. 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 [6] were notable in 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 [6] modeled the two-party authentication/AKE based on symmetric keys. The present paper uses a similar model. Furthermore, [6] uses random oracles in their proofs, similar to the proofs of this paper. However, the AKE definition of [6] 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.* [13, 14] adapted the Bellare–Rogaway model [6] for the case of **asymmetric** keys. Accordingly, their definition was updated to recognize the forward security property, as well as protection against new types of 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 [8] themselves: They proposed a model for the case of three-party AKE, which is the setting used in the Needham–Schroeder protocol [54]. 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 [8] was modeled via an “existentially guaranteed partnering function,” a notion that the authors called **unintuitive** later [5]. Furthermore, the specific partnering function used in the proof of security of 3PKD was later found to be flawed [31].

Shoup and Rubin [59] changed the three-party AKE model of [8] 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 [4,58], 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” [25].

Bellare, Pointcheval, and Rogaway [5] 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. They 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 [25] used a technique introduced in [4] 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 [25] 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 [6] with secure channel protocols was later established [18]. See also [17] 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 [26], this time in the broader context of the Universal Composability (UC) framework [20]. Any protocol proven secure in the UC framework 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 [19, 24] as well.

In his famous paper, Krawczyk [48] proposed an improvement of 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 great flexibility.

LaMacchia, Lauter, and Mityagin [49] 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” [25]. Furthermore, the security definitions in both models do 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 $\alpha$ in a $g^\alpha \mod p$ flow of the Diffie–Hellman protocol. The
security definition of LaMacchia et al. designates a session as 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 [36] 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. [57] 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 [62] showed a flaw in the security proofs of Sarr et al. [57], 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 [16,29,30,35,36].

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, 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 $\ell, r, k, K : \mathbb{N} \rightarrow \mathbb{N}$, 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 $p : \mathbb{N} \rightarrow \mathbb{N}$,
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 binary strings of finite length, 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 \( \lambda \) with any string is the string itself. Another special symbol is ‘\#’. It is called the wildcard symbol, and matches a single bit. We use the \texttt{match} function to compare two binary strings which may include the wildcard symbol. For instance, \texttt{match("1##0","1010")} is true. However, the normal comparison \texttt{"1##0" == "1010"} is false, because the two strings are not literally identical.

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

A function \( f : \mathbb{N} \rightarrow \mathbb{R} \) 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) [7], where all parties, including the adversary, have access to a random function \( \mathcal{O} : \{0,1\}^* \rightarrow \{0,1\}^{p(n)} \) (recall from above that \( p(n) \) is a polynomial upper bound on \( k(n) \) and \( K(n) \), among other things). Once \( \mathcal{O} \) is queried on some value \( x \in \{0,1\}^* \) for the first time, a random \( p(n) \)-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 \( x \). If an entity does not require all \( p(n) \) bits of the returned value from \( \mathcal{O} \), he only keeps the leftmost required bits, and discards the rest.

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 a conditional statement, the conjuncts are evaluated from left to right. As soon as a conjunct is evaluated to false, the remaining conjuncts are ignored, and the whole statement is evaluated to false. A similar convention holds for disjuncts.\(^2\) For instance, consider the statement \texttt{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 nested \texttt{if} statements: The outer \texttt{if} evaluates \( a \), and only if it is true, the inner \texttt{if} will evaluate \( b \).

### 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.

\(^2\)That is, as soon as a disjunct is evaluated to true, the remaining disjuncts are ignored, and the whole statement is evaluated to true.
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, and is incremented every $\tau$ 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 $K^{\theta}_{xy}$ in time stage $T = \theta$. This key is updated whenever the time stage is incremented. Let $K^{\theta+1}_{xy}$ denote the new key. In FORSAKES, we have $K^{\theta+1}_{xy} \leftarrow \mathcal{O}(K^\theta_{xy})$. Notice that the random oracle $\mathcal{O}$ can be considered as an ideal hash function, which in particular, is ideally one-way. Therefore, $K^{\theta+1}_{xy}$ reveals no information about $K^\theta_{xy}$.

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

- $\text{sid}^i_z \in \{\#, 0, 1\}^{2r}$: The session identifier. It is the (ordered) concatenation of two nonces: The nonce sent by the initiator, and the nonce sent by the responder. Let us denote by $\text{rnd}^i_z \in \{0, 1\}^r$ the randomness assigned to session $i$ on party $z$.

- $\text{role}^i_z \in \{I, R\}$: The role of the party in the current session, which is either the initiator (‘I’) or the responder (‘R’).

- $\text{pid}^i_z \in \{0, 1\}^\ell$: The identifier of the session partner.

- $T^i_z \in \{0, 1\}^{64}$: 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.

- $sk^i_z \in \{0, 1\}^k$: 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, it is important to protect

---

3The 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 3.1. Initiator’s handling of session initiation.

**Incoming Message:** 0 || id\_y // Partner ID, set by the caller.

1. \( c \leftarrow c + 1 \);
2. \( \text{pid}^c_x \leftarrow \text{id}_y \);
3. \( \text{if} (x \text{ does not share an LTK with } y) \)
4. \( \text{st}^c_x \leftarrow \lambda \) and \( \text{acc}^c_x \leftarrow 0 \);
5. \( \text{return} (\lambda, \lambda) \);
6. \( \text{sid}^c_x \leftarrow \text{rnd}^x || \# \); // ‘\#’ is a wildcard
7. \( \text{role}^c_x \leftarrow 'I' \);
8. \( T^c_x \leftarrow T \);
9. \( sk^c_x \leftarrow i k^c_x \leftarrow \lambda \);
10. \( \text{acc}^c_x \leftarrow \lambda \); // Session fate is undecided yet.
11. \( \text{st}^c_x \leftarrow \text{sid}^x || \text{role}^x || \text{pid}^x || T^c_x || sk^c_x || ik^c_x || \text{acc}^c_x \);
12. \( \text{Msg}_1 \leftarrow 1 || \text{id}_x || \text{pid}^c_x || T^c_x || \text{rnd}^c_x \);
13. \( \text{return} (\text{Msg}_1, \text{st}^c_x) \);

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.

• \( ik^i \in \{0,1\}^k \): The second part of the ephemeral key, known as the **integrity key**. Once computed, this key protects the integrity of FORSAKES messages.

• \( \text{acc}^i \in \{\lambda, 0, 1\} \): The acceptance decision of the session. In the beginning, the acceptance is undecided (\( \lambda \)). When the session decides it acceptance, \( \text{acc}^c_x \) will be either false (0) or true (1).

Next, we will explain how each message of FORSAKES is handled by the corresponding party. On input an incoming message \( m \) as well as other information (LTKs, session state, session randomness, etc.), the handling algorithms always return a pair \((m', st)\). Here, \( m' \) is the outgoing message, and \( st \) is the updated session state.

**Remark 3.1.** If the handling algorithm does not find a session corresponding to the incoming message, it will return \((\lambda, '?')\). If the session is found but an attack is detected, the handling algorithm will return \((\lambda, \lambda)\). In other cases, \( m \) is processed, and the outgoing message and the updated session state is produced according to FORSAKES rules.

**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 of 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 \( \text{id}_x \), and the ID of the partner (i.e., the responder) is \( \text{id}_y \). The algorithm used by the initiator to handle session initiation, and generation of the first protocol message, is described by Algorithm 3.1. Notice that the zeroth message is denoted by 0 || \( \text{id}_y \).

The initiator first increases the session counter \( c \), and then sets the session partner ID to \( \text{id}_y \). It then checks whether he has shared an LTK with \( y \). If this check fails, \( x \) sets the session state to an empty string, rejects this session by setting \( \text{st}^c_x \leftarrow \lambda \) and \( \text{acc}^c_x \leftarrow 0 \), and returns \((\lambda, \lambda)\) to report the error.
Remark 3.2 (Wiping Sensitive Data). Notice that deleting a sensitive object by freeing the memory allocated to it 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 $\lambda$ to a previously initialized variable denotes the secure wipe of the memory allocated to it.

If no error occurs, the session ID is set to $\text{rnd}_x^{c} \parallel \#^r$. Recall that $\text{rnd}_x^{c}$ 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 role to ‘I’, the session time stage to $T$, the ephemeral keys to empty, and the acceptance state to undecided ($\lambda$). Finally, the session state and $\text{Msg}_1$ are set.

Notice that the session state includes every other state variable:

$$st_x^{c} \leftarrow \text{sid}_x^{c} \parallel \text{role}_x^{c} \parallel \text{pid}_x^{c} \parallel T_x^{c} \parallel \text{sk}_x^{c} \parallel \text{ik}_x^{c} \parallel \text{acc}_x^{c}.$$ 

It will be the only variable that each session stores (by returning it to the caller). That is, there will be no need to store other variables, such as $\text{sid}_x^{c}$, 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 decomposed by the party to its constituent parts.

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

$$\text{Msg}_1 \leftarrow 1 \parallel \text{id}_x \parallel \text{pid}_x^{c} \parallel T_x^{c} \parallel \text{rnd}_x^{c}.$$ 

Notice that the message begins with 1, indicating that this is the first message of the protocol. It helps the responder—who may have participated in multiple concurrent sessions—to distinguish the incoming message. It can prevent typing attacks [15,32], where the adversary uses the syntactical similarity of protocol messages to reorder them and mount an attack.

The message includes the identifier of the sender ($\text{id}_s$), as well as the identifier of the partner ($\text{pid}_r^{c}$). We include these two identifiers, in the order specified, in all messages of FORSAKES. This prevents parallel session attacks [10], where the attacker opens two sessions with a single party in parallel. She then sends every message she receives from the first session in the second session, and vice versa.

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

Finally, $\text{Msg}_1$ includes the nonce of the initiator ($\text{rnd}_x^{c}$). 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.

At the end of handling session initiation, the pair $(\text{Msg}_1, st_x^{c})$ is returned to the caller.

Handling the First Message. The first message of the protocol is specified in Equation 1. Upon receiving this message, the responder verifies the syntax, and if approved, follows Algorithm 3.2. Notice that the incoming message is described as $m = 1 \parallel \text{id}_s \parallel \text{id}_r \parallel T_s \parallel \text{rnd}_s$, where the subscripts $s$ and $r$ denote the sender and receiver, respectively. For instance, $\text{id}_s$ means the sender’s ID, while $\text{id}_r$ means the receiver’s ID.
Algorithm 3.2. Responder’s handling of the first message.

\begin{algorithm}
\textbf{Incoming Message: } \textstyle m = 1 \mid \text{id}_s \mid \text{id}_r \mid T_s \mid \text{rnd}_s
\begin{align*}
1: & \quad d \leftarrow d + 1; \\
2: & \quad \text{if } (\text{id}_r \neq \text{id}_y \text{ or } T_s \neq T \text{ or } (y \text{ does not share an LTK with } x)) \\
3: & \quad \text{return } (\lambda, \lambda); \\
4: & \quad \text{pid}_y^d \leftarrow \text{id}_s; \\
5: & \quad \text{sid}_y^d \leftarrow \text{rnd}_s || \text{rnd}_y^d; \\
6: & \quad \text{role}_y^d \leftarrow \text{‘R’}; \\
7: & \quad \text{T}_y^d \leftarrow T; \\
\end{align*}
\begin{align*}
\triangleright & \text{let } K_{yx}^T \text{ be the LTK shared between } \text{id}_y \text{ and } \text{pid}_y^d; \\
9: & \quad sk_y^d \leftarrow \mathcal{O}(0 || \text{sid}_y^d || K_{yx}^T); \\
10: & \quad ik_y^d \leftarrow \mathcal{O}(1 || \text{sid}_y^d || K_{yx}^T); \\
11: & \quad \text{acc}_y^d \leftarrow \{0 \mid \text{sid}_y^d \mid K_{yx}^T\}; \\
12: & \quad \text{st}_y^d \leftarrow \text{sid}_y^d || \text{role}_y^d || \text{pid}_y^d || \text{T}_y^d || sk_y^d || ik_y^d || \text{acc}_y^d; \\
13: & \quad \text{MSG}_2 \leftarrow 2 || \text{id}_y || \text{id}_s || \text{T}_y^d || \text{sid}_y^d; \\
14: & \quad \text{AUTH}_2 \leftarrow \mathcal{O}(\text{MSG}_2 || ik_y^d); \\
15: & \quad \text{return } (\text{MSG}_2 || \text{AUTH}_2, \text{st}_y^d); \\
\end{align*}
\end{algorithm}

The responder first increments his session counter \(d\). While not necessary, we called the responder’s session counter \(d\), to distinguish it from the initiator’s session counter \(c\). The responder then checks whether the message is intended for him (\(\text{id}_r = \text{id}_y\)), 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 of the conditions fails, or if the receiver does not share an LTK with the sender, the responder rejects this session. To this end, the algorithm sets \(\text{st}_y^d \leftarrow \lambda\) and \(\text{acc}_y^d \leftarrow 0\), and executes the \text{return} \((\lambda, \lambda)\) command.

Remark 3.3. 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 can customize this part, if desired.

Next, the responder sets the partner ID to \(\text{id}_s\), the session ID to \(\text{rnd}_s || \text{rnd}_y^d\), 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.

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, in a specific order):

\[
\begin{align*}
\text{sk}_y^d & \leftarrow \mathcal{O}(0 || \text{sid}_y^d || K_{yx}^T) \\
\text{ik}_y^d & \leftarrow \mathcal{O}(1 || \text{sid}_y^d || K_{yx}^T).
\end{align*}
\]

After setting the ephemeral keys, the acceptance state of the session is set to undecided (\(\lambda\)), 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;
Finally, the responder returns $\text{Auth}_2$ to the caller.

**Handling the Second Message.** After verifying the message syntax, the initiator executes Algorithm 3.3 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 by returning $(\lambda, \text{‘?’})$, meaning that no session state matches the incoming message.

Next, it is verified that exactly one match exists. If there are two or more sessions whose identifiers match the sid of the incoming message, all such sessions will be deleted by setting them to empty string, and rejecting the incoming message.

### Algorithm 3.3. Initiator’s handling of the second message.

**Incoming Message:** $m = 2 || \text{id}_s || \text{id}_r || T_s || \text{sid} || \text{Auth}_2$

1. if (¬match($\text{sid}_s^i, \text{sid}$) for all $i \in [c]$)
   
   return $(\lambda, \text{‘?’})$; // No session matches with $m$.

2. if (match($\text{sid}_s^i, \text{sid}$) for more than one $i \in [c]$)
   
   return $(\lambda, \text{‘?’})$; // Unique match is required.

3. let $i \in [c]$ be the unique value such that match($\text{sid}_s^i, \text{sid}$) = 1;

4. if (acc$_x^i \neq \lambda$)
   
   return $(\lambda, \text{st}_x^i)$; // No incoming message after session is decided.

5. if (id$_s \neq \text{pid}_y^i$ or id$_r \neq \text{id}_x$ or $T_s \neq T$ or $T_s \neq T_x^i$ or role$_x^i \neq \text{‘I’})$
   
   return $(\lambda, \text{sid}_x)$; // No more wildcards.

6. $\text{sid}_x^i \leftarrow \text{sid}$; // sid for more than one

7. $\text{sk}_x^i \leftarrow O(0 || \text{sid}_x^i || K^T_{xy})$;

8. $\text{ik}_y^i \leftarrow O(1 || \text{sid}_x^i || K^T_{xy})$;

9. if ($\text{Auth}_2 \neq O(\text{Auth}_2 || \text{ik}_x^i)$)
   
   return $(\lambda, \text{sid}_x)$;

10. $\text{st}_x^i \leftarrow \lambda$;

11. $\text{acc}_x^i \leftarrow 0$;

12. return $(\lambda, \lambda)$;

13. $\text{sid}_x^i \leftarrow \text{sid}$; // sid for more than one

14. $\text{sk}_x^i \leftarrow O(0 || \text{sid}_x^i || K^T_{xy})$;

15. $\text{ik}_y^i \leftarrow O(1 || \text{sid}_x^i || K^T_{xy})$;

16. if ($\text{Auth}_2 \neq O(\text{Auth}_2 || \text{ik}_x^i)$)
   
   return $(\lambda, \text{sid}_x)$;

17. $\text{st}_x^i \leftarrow \lambda$;

18. $\text{acc}_x^i \leftarrow 0$;

19. return $(\lambda, \lambda)$;

20. $\text{sid}_x^i \leftarrow \text{sid}_x^i$; // sid for more than one

21. $\text{sk}_x^i \leftarrow O(0 || \text{sid}_x^i || K^T_{xy})$;

22. $\text{Auth}_3 \leftarrow O(\text{Auth}_2 || \text{ik}_x^i)$;

23. return $(\text{msg}_3 || \text{Auth}_3, \text{st}_x^i)$;

and (2) the second message is authenticated using the integrity key. The construction of the message and the authenticator is as follows:

\[
\text{msg}_2 \leftarrow 2 || \text{id}_y || \text{id}_x || T_y^d || \text{sid}_y \\
\text{Auth}_2 \leftarrow O(\text{msg}_2 || \text{ik}_y^d).
\]

Finally, the responder returns $(\text{msg}_2 || \text{Auth}_2, \text{st}_y^d)$ to the caller.
Algorithm 3.4. Responder’s handling of the third message.

Incoming Message: \( m = \text{Msg}_3 \mid \text{id}_s \mid \text{id}_r \mid T_s \mid \text{sid} \mid \text{AUTH}_3 \)

1. if ( \( \text{sid}_y \neq \text{sid} \) for all \( i \in [d] \) )
2. \( \text{return} (\lambda, '?' ) \); // No session matches with \( m \).
3. if ( \( \text{sid}_y \neq \text{sid} \) for more than one \( i \in [d] \) )
4. \( \text{return} (\lambda, '?' ) \); // Unique match is required.
5. let \( i \in [d] \) be the unique value such that \( \text{sid}_y \neq \text{sid} \);
6. if ( \( \text{acc}^i_y \neq \lambda \) )
7. \( \text{return} (\lambda, \text{st}^i_y) \); // No incoming message after session is decided.
8. if ( \( \text{id}_s \neq \text{pid}_y^i \) or \( \text{id}_r \neq \text{id}_y^i \) or \( T_s \neq T \) or \( T_s \neq T_y^i \) or \( \text{role}^i_y \neq 'R' \) )
9. \( \text{st}^i_y \leftarrow \lambda \);
10. \( \text{acc}^i_y \leftarrow 0 \);
11. \( \text{return} (\lambda, \lambda) \);
12. if ( \( \text{AUTH}_3 \neq \text{O} (\text{MSG}_3 \mid i_k^y) \) )
13. \( \text{st}^i_y \leftarrow \lambda \);
14. \( \text{acc}^i_y \leftarrow 0 \);
15. \( \text{return} (\lambda, \lambda) \);
16. \( \text{acc}^i_y \leftarrow 1 \);
17. \( \text{st}^i_y \leftarrow \text{sid}_y^i \mid \text{role}^i_y \mid \text{pid}^i_y \mid T_y^i \mid s_k^y \mid i_k^y \mid \text{acc}^i_y \);
18. \( \text{return} (\lambda, \text{st}^i_y) \);

by returning (\( \lambda, '?' \)). This event should occur only with probability exponentially small in \( n \) (see Corollary 5.1 on page 502).

Now that the unique session state matching the incoming message is found, the algorithm checks whether this session is already decided. If so, an empty message as well as the current session state is returned.

Next, five 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, and (5) the role of the current session is ‘I’, meaning that it anticipated the second protocol message. If any of these conditions does not hold, the session state is securely wiped, and the incoming message is rejected by returning (\( \lambda, \lambda \)).

Let us assume that \( K_{xy}^T \) is the current value of the LTK shared between the initiator and the responder. By symmetry, we have \( K_{xy}^T = K_{yx}^T \).

Next, the initiator updates the session ID to \( \text{sid} \). This effectively replaces the wildcard string with the proper value. He will then set the ephemeral keys, and verifies the integrity of the second message by comparing \( \text{AUTH}_2 \) to the value it must be. If the verification fails, the session state is securely wiped, the session acceptance is set to false, and the incoming message is rejected by returning (\( \lambda, \lambda \)).

Otherwise, the acceptance state is set to true, and the session state as well as the third message are computed and returned.
Handling the Third Message. The responder verifies the third message syntactically, and if it is OK, executes Algorithm 3.4. The algorithm first checks whether the sid on the incoming message matches a unique session state. If not, the incoming message is rejected by returning \((\lambda, '?')\), meaning that no session state matches the incoming message.

Next, it is verified that exactly one match exists. If there are two or more sessions whose identifiers match the sid of the incoming message, all such sessions will be deleted by setting them to empty string, and rejecting the incoming message by returning \((\lambda, '?')\). This event should occur only with probability exponentially small in \(n\) (see Corollary 5.1 on page 502).

Now that the unique session state matching the incoming message is found, the algorithm checks whether this session is already decided. If so, an empty message as well as the current session state is returned.

Next, five 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, and (5) the role of the current session is ‘R’, meaning that it anticipated the third protocol message. If any of these conditions does not hold, the session state is securely wiped, and the incoming message is rejected by returning \((\lambda, \lambda)\).

As a final verification, the integrity key is used to verify \(\text{AUTH}_3\) on \(\text{Msg}_3\). If the verification fails, the session state is securely wiped, the session acceptance is set to false, and the incoming message is rejected by returning \((\lambda, \lambda)\).

Ultimately, the responder accepts, updates the session state, and returns \((\lambda, \text{st}_i)\).

Notice that the outgoing message is \(\lambda\), meaning that no outgoing message exists. 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.
### Variable Description

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

### Party-specific variables stored by $C$

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

### Session-specific variables stored by $C$

| Variable | Description |
|----------|-------------|
| $\text{id}^t_{xy}$ | The ordinal identifier of the party to whom the session $(x, y)$ is partnered. |
| $\text{rand}^t_{xy}$ | The randomness used in the session $(x, y)$. |
| role$^t_{xy}$ | The role of the session $(x, y)$, which is either ‘I’ (initiator) or ‘R’ (responder). |
| $sk^t_{xy}$ | The session key of the session $(x, y)$. |
| $sk_{xy}$ | The time stage in which the session key of the session $(x, y)$ is set. |
| $\text{acc}^t_{xy}$ | The acceptance state of the session $(x, y)$. It is $\lambda$ if the decision is yet to be made, 0 if it has rejected, and 1 if it has accepted. |
| $\text{state}^t_{xy}$ | The state of the session $(x, y)$. We assume that this variable encodes, among other information, the value of $\text{id}^t_{xy}$, $\text{rand}^t_{xy}$, $sk^t_{xy}$, and $\text{acc}^t_{xy}$. |
| $\text{exposed}^t_{xy}$ | A Boolean variable, indicating whether the adversary has exposed the session $(x, y)$. |

**Table 1. State variables used by $C$.**

### 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 $\tau$ 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 $C$ and the adversary $A$. The challenger allows the adversary to make a set of predefined queries, to which $C$ 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 $C$.** The input of $C$ is the security parameter $n$, in the unary form $1^n$. Furthermore, $C$ has black-box access to the AKE protocol $\Pi$. We denote this fact by $C^{\Pi(1^n)}$. 
State Information of $C$. Let us discuss the state information which $C$ keeps. As described below, the state changes when either $A$ or the universal clock makes a query to $C$.

The global state variables are detailed in Table 1(a). 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 $A$ and $C$, 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 $C$ also keeps state variables which are specific to one or more parties. These variables are explained in Table 1(b). Finally, $C$ keeps session-specific state variables, which are described in Table 1(c). Notice that $(x, s)$ denotes a session identified by $s$ on party $x$.

Queries Allowed by $C$. Below, we list and describe the queries allowed by $C$. 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 $C$ 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).

$\blacktriangleright$ The TimeEvent() function. The universal clock calls this function at regular intervals (every $\tau$ seconds). Upon receiving this query, $C$ updates all long-term keys in the system by calling UpdateLTK($T, K^{T+1}_{xy}$). In FORSAKES, the function UpdateLTK simply ignores its first argument, and performs a time-independent update: $\text{UpdateLTK}(T, K^{T+1}_{xy}) \equiv O(K^{T}_{xy})$. Immediately after the new key $K^{T+1}_{xy}$ is computed, the old key $K^{T}_{xy}$ is wiped securely (see Remark 3.2).

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

The pseudocode of the TimeEvent() function is described in Algorithm 4.1.

\begin{algorithm}
\caption{The TimeEvent() function.}
1: TimeEvent()
2: \hspace{1em} for ($x \leftarrow 1$ to $N$)
3: \hspace{2em} for ($y \in P_x$)
4: \hspace{3em} $K^{T+1}_{xy} \leftarrow \text{UpdateLTK}(T, K^{T}_{xy})$;
5: \hspace{3em} $K^{T}_{xy} \leftarrow \lambda$; \hspace{1em} // secure wipe of the LTK
6: \hspace{2em} $T \leftarrow T + 1$
7: \hspace{1em} Notify($A$);
\end{algorithm}

Remark 4.1. TimeEvent() can be extended so that it notifies not only the adversary, but also all the parties in the system. In this way, the parties can do tasks such as removing old sessions which have not completed within one time stage.

$\blacktriangleright$ The Register() function. The adversary calls this function to introduce a new party into the system. Upon receiving this query, $C$ increments $N$, and generates
a random $\ell(n)$-bit identifier $\text{id}_N$. The pair $(\text{id}_N, N)$ is added to the set $\mathcal{ID}$, and the set of sessions on $N$ ($\mathcal{sess}_N$) and the set of parties who share an LTK with $N$ ($\mathcal{P}_N$) are set to empty. Finally, $\text{id}_N$ 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 $\text{Register}$ function is described in Algorithm 4.2.

\begin{algorithm}
\caption{The $\text{Register}$ function.}
\begin{algorithmic}[1]
\State $\text{Register}()$
\State $N \leftarrow N + 1$
\State $\text{id}_N \leftarrow R\{0, 1\}^{\ell(n)}$
\State $\mathcal{ID} \leftarrow \mathcal{ID} \cup \{(\text{id}_N, N)\}$
\State $\mathcal{sess}_N \leftarrow \emptyset$
\State $\mathcal{P}_N \leftarrow \emptyset$
\State $\text{return } \text{id}_N$
\end{algorithmic}
\end{algorithm}

\begin{algorithm}
\caption{The $\text{ShareLTK}(x, y)$ function.}
\begin{algorithmic}[1]
\State $\text{ShareLTK}(x \in \mathbb{N}, y \in \mathbb{N})$
\If{$x > N \text{ or } y > N \text{ or } x = y \text{ or } y \in \mathcal{P}_x$}
\State output $0$ and abort;
\EndIf
\State $K_{yx}^T \leftarrow K_{xy}^T \leftarrow R\{0, 1\}^{K(n)}$
\State $\mathcal{P}_x \leftarrow \mathcal{P}_x \cup \{y\}$ and $\mathcal{P}_y \leftarrow \mathcal{P}_y \cup \{x\}$
\State $\text{revealTime}_{xy} \leftarrow \text{revealTime}_{yx} \leftarrow +\infty$
\end{algorithmic}
\end{algorithm}

\begin{algorithm}
\caption{The $\text{Send}(x, s, y, m)$ function.}
\begin{algorithmic}[1]
\State $\text{Send}(x, s, y, m)$
\If{$x = s \text{ or } y = m$}
\State output $0 || \text{id}_y$, where $\text{id}_y$ denotes the ID of the responder.
\EndIf
\end{algorithmic}
\end{algorithm}
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. Furthermore, we syntactically check whether the message $m$ belongs to the valid session $(x, s)$, whether it is sent by $y$, and so on. This check is performed via a call to $\text{SyntaxCheck}^\Pi(x, s, y, m)$. The superscript $\Pi$ means that the check is specific to the syntax of the AKE protocol $\Pi$ at hand.

**Example 4.1.** As an example, consider FORSAKES, which does the following syntax checks (let $c$ be the session counter on $x$):

1. If $m$ starts with 0, then $m = 0 || \text{id}_y$.
2. If $m$ starts with $i \in \{1, 2, 3\}$, then $m$ should start with $i || \text{id}_y || \text{id}_x$.
3. If $m$ starts with 0 or 1, then $s = c + 1$.
4. If $m$ starts with 2 or 3, then $1 \leq s \leq c$.

If any of the above checks fails, $\text{SyntaxCheck}^\Pi(x, s, y, m)$ returns 0. Otherwise, it returns 1.

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

- $\text{sess}_x$: The session $s$ is added to this set.
- $\text{skTime}_x^s$: It is set to 0, which means that the session key is not generated yet.
- $\text{exposed}_x^s$: It is set to false (0), as the session is not exposed yet.
- $\text{rnd}_x^s$: The randomness of the current session is picked randomly from $\{0, 1\}^{\ell(n)}$.
- $\text{sid}_x^s$, $\text{role}_x^s$, $\text{sk}_x^s$, $\text{acc}_x^s$, $\text{state}_x^s$, and $\text{pid}_x^s$: All initialized to default values.

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

- the long-term key ($K_{xy}$);
- the current time stage ($T$);
- the current session state ($\text{state}_x^s$);
- the session randomness ($\text{rnd}_x^s$);
- the identifiers of the current party and its session partner ($\text{id}_x$ and $\text{id}_y$); and
- the incoming message ($m$).

**Remark 4.2** (Possible Extensions to $\text{Send}$). It is possible to extend $\text{Send}$ by allowing the adversary to specify the sender as $y = 0$, meaning that the sender is unspecified. In this case, it is the job the AKE protocol to extract the identity of the sender from the incoming message, and instead of $K_{xy}$, the $\text{Send}$ function should pass $\{K_{xy}\}_{y \in \mathcal{P}_x}$ to $\Pi$.

It is also possible to allow the adversary to specify the receiving session as $s = 0$, meaning that the receiving session is unspecified. Again, it is the job the AKE protocol to extract the receiving session from the incoming message, and instead of $\text{state}_x^s$, the $\text{Send}$ function should pass $\{\text{state}_x^s\}_{s \in \text{sess}_x}$ to $\Pi$.

Notice that FORSAKES is consistent with both extensions, since in FORSAKES, both the sender and the receiving session are extracted from the incoming message.

The output of $\Pi$ is the outgoing message ($m'$), the updated session state ($st$), and the updated randomness ($\text{rnd}_x^s$):

$$
(m', st, \text{rnd}_x^s) \leftarrow \Pi(K_{xy}^T, \text{state}_x^s, \text{state}_x^s, \text{id}_x, \text{id}_y, m).
$$
The reason why the randomness gets updated is explained in the description of the ExposeSS function (see below). As a convention, if a protocol returns just a pair (rather than a triple), the protocol is not interested in updating the session randomness.

Notice that Equation 2 does not update \( \text{state}_x^s \) directly, but introduces the temporary variable \( st \). This is because, as described in Remark 3.1, the protocol \( \Pi \) might not find a unique session corresponding to \( m \), and therefore does not know how to handle the incoming message. In this case, \( \Pi \) will simply return ‘?’ as the updated session state. As \( m \) does not correspond to \((x, s)\), we do not update \( \text{state}_x^s \) in this case.

The second possible case is that \( \Pi \) returns an empty \((\lambda)\) state. We assume (see Assumption 4.1 below) that this corresponds to detecting an error. In this case, only \( \text{acc}_x^s \) is updated to false, but all other session variables (such as partner ID or session key—see Table 1(c)) remain intact.

**Remark 4.3.** The reason for not updating other session variables is as follows: In Section 4.2, we will describe the rules for session partnership. We will see that the session key, session ID, session role, and session partner ID can all affect whether two sessions are partnered. If, upon detecting an error, we set other session variables to their default values, the adversary can easily “un-partner” two sessions by sending an invalid message to one of them. Notice that the session variables (such as the session key) are locally stored by both the challenger \( C \) and the protocol \( \Pi \) itself. Therefore, the protocol can securely wipe the session variables (as per Remark 3.2), without affecting the local variables stored by \( C \).

The third possible case is that \( \Pi \) returns a non-empty session state. As stated in Table 1(c), the session state encodes other values, including the session ID, the partner ID, the session key, and the acceptance state of the session. In this case, these values are obtained by applying some efficient function \( f^\Pi_{ID} \) to \( \text{state}_x^s \):

\[
(\text{sid}_x^s, \text{role}_x^s, \text{pid}_x^s, sk_x^s, \text{acc}_x^s) \leftarrow f^\Pi_{ID}(\text{state}_x^s)
\]

Notice that the subscript \( ID \) in \( f^\Pi_{ID} \) means that \( f \) uses the set \( ID \) to translate the string partner ID to the corresponding ordinal value. As before, the superscript \( \Pi \) means that this function is protocol-specific.

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 \( C \) returns the tuple \((m', \text{sid}_x^s, \text{pid}_x^s, skTime_x^s, \text{acc}_x^s)\) to the adversary. Notice that the returned value does not include the session state \( \text{state}_x^s \) or the session randomness \( \text{rnd}_x^s \).

The pseudocode of the Send \((x, y, s, m)\) function is described in Algorithm 4.4. Before pertaining to the next query, let us mention that in describing the Send query, we assumed that \( \Pi \) satisfies a number of general properties, as listed in Assumption 4.1.

**Assumption 4.1 (Properties of \( \Pi \)).** We assume that \( \Pi \) satisfies the following rules for all \( s, x \in \mathbb{N} \):

- \((x, s)\) returns an empty state if and only if it detects an error. In this case, it returns an empty string as the outgoing message as well.

---

4One such attack is described by an anonymous reviewer on a preliminary version of this paper.
The condition in line 4 of Algorithm 4.4 ensures (among other things) that if session \((x, s)\) exists, \(y\) is its partner. In case \((x, s)\) does not exist, we assume that SyntaxCheck\(^{\Pi}\)(\(x, s, y, m\)) performs this check, meaning that the syntax of \(m\) confirms the claim that \(y\) is its sender. Therefore, we can be sure that \(\text{pid}_x^y = y\) after line 19 of Algorithm 4.4 is executed.

- \(\text{acc}_x^s\) is not set to 1 before \((x, s)\) receives the last incoming message and verifies it.
- \((x, s)\) will no longer accept any incoming message if \(\text{acc}_x^s\) is set to either 1 or 0.
- If \(m\) is the last incoming message of \((x, s)\), the session outputs the empty message \(m' = \lambda\) as the outgoing message.
- \(\text{role}_x^s\) and \(\text{pid}_x^s\) are 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 \(\text{sk}_x^s\) nor \(\text{sid}_x^s\) will change after the session key is set (i.e., after \(\text{skTime}_x^s\) is set to \(T\) in line 21 of Algorithm 4.4).

\[\text{Algo 4.4. The Send}(x, s, y, m)\]

1. Send\((x \in \mathbb{N}, s \in \mathbb{N}, y \in \mathbb{N}, m \in \{0, 1\}^*\))
2. if \((x > N \text{ or } y > N \text{ or } y \notin \mathcal{P}_x \text{ or } \neg\text{SyntaxCheck}^{\Pi}(x, s, y, m))\)
3. output 0 and abort;
4. if \((s \notin \text{sess}_x \text{ and } (\text{acc}_x^s \neq \lambda \text{ or } \text{pid}_x^s \neq y))\)
5. output 0 and abort;
6. if \((s \notin \text{sess}_x)\)
7. sess_x \leftarrow \text{sess}_x \cup \{s\};
8. \text{skTime}_x^s \leftarrow 0;
9. \text{exposed}_x^s \leftarrow 0;
10. \text{rnd}_x^s \leftarrow \mathcal{R}\{(0, 1)^n\};
11. \text{sid}_x^s \leftarrow \text{role}_x^s \leftarrow \text{sk}_x^s \leftarrow \text{state}_x^s \leftarrow \lambda \text{ and } \text{pid}_x^s = -1;
12. \text{sid}_x^s \leftarrow \text{role}_x^s \leftarrow \text{sk}_x^s \leftarrow \text{state}_x^s \leftarrow \lambda \text{ and } \text{pid}_x^s = -1;
13. if \((s = \text{"?"})\)
14. return \((m', \text{"?"});\)
15. state_x^s \leftarrow st;
16. if \((\text{state}_x^s = \lambda)\)
17. \text{acc}_x^s \leftarrow 0; //\text{Other state variables remain intact.}
18. else
19. \((\text{sid}_x^s, \text{role}_x^s, \text{pid}_x^s, \text{sk}_x^s, \text{acc}_x^s) \leftarrow f^{\Pi}_{1D}(\text{state}_x^s);\)
20. if \((\text{skTime}_x^s = 0 \text{ and } \text{sk}_x^s \neq \lambda)\)
21. \text{skTime}_x^s \leftarrow T;
22. return \((m', \text{sid}_x^s, \text{pid}_x^s, \text{sk}_x^s, \text{acc}_x^s);\)

\[\text{ExposureSS}(x, s)\] 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, exposed_x^s is set to 1, and the pair \((\text{state}_x^s, \text{rand}_x^s)\) 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 \( r_{n_d} \) in line 12 of Algorithm 4.4. 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 \texttt{ExposeSS}(x, s) function is described in Algorithm 4.5.

```
Algorithm 4.5. The \texttt{ExposeSS}(x, s) function.
1: \texttt{ExposeSS}(x \in \mathbb{N}, s \in \mathbb{N})
2: \text{if } (x > N \text{ or } s \notin \text{sess}_x)
3: \quad \text{output 0 and abort;}
4: \quad \text{exposed}_x^s \leftarrow 1;
5: \quad \text{return } (\text{state}_x^s, \text{rnd}_x^s);
```

The \texttt{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 \( \text{revealTime}_{xy} \) and \( \text{revealTime}_{yx} \) are set to \( T \), and \( K_{xy}^T \) is returned.

The pseudocode of the \texttt{RevealLTK}(x, y) function is described in Algorithm 4.6.

```
Algorithm 4.6. The \texttt{RevealLTK}(x, y) function.
1: \texttt{RevealLTK}(x \in \mathbb{N}, y \in \mathbb{N})
2: \text{if } (x > N \text{ or } y > N \text{ or } y \notin \mathcal{P}_x)
3: \quad \text{output 0 and abort;}
4: \quad \text{revealTime}_{xy} \leftarrow \text{revealTime}_{yx} \leftarrow T;
5: \quad \text{return } K_{xy}^T;
```

In most papers, the adversary is given access to a \texttt{Corrupt()} query, which returns all long-term keys stored on a single party. \texttt{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 \( \mathcal{P}_x \) of parties to whom \( x \) shares an LTK,\footnote{This is because the parties share an LTK if and only if the adversary uses the \texttt{ShareLTK()} query.} she can effectively corrupt \( x \) by running \texttt{RevealLTK()} on \( x \) and every party in \( \mathcal{P}_x \).

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 4.1, is adopted from [5].

**Definition 4.1** (Session Partnership). Two sessions \((x, s)\) and \((y, t)\) are called partners if the following conditions hold:
1. \( sk_x^s = sk_y^t \neq \lambda \);
2. \( sid_x^s = sid_y^t \neq \lambda \);
3. \( role_x^s \neq role_y^t \);
4. $\text{pid}^x_y = y$ and $\text{pid}^y_x = x$;
5. $\text{sid}^y_z \neq \text{sid}^x_z$ for all $(z, u) \in \mathbb{N}^2 - \{(x, s), (y, t)\}$.

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 very 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 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.

Remark 4.4. We require quite a lot for two sessions to be partnered. It is possible to relax some of these conditions, but the protocol designer should be aware of the effect of such relaxations on the security of the protocol. For instance, let us consider the case where the first condition is removed; i.e., two sessions can be partner even if they have non-identical session keys. This, in turn, means that the adversary can derive the session key of one session by exposing the state information of the partner session. In some (informal) sense, it indicates that one session key can be derived from a non-identical session key, and therefore the session keys are not completely “random looking.”

$C$ uses the function $\text{FindPartnerSession}(x, s)$ in Algorithm 4.7 to find the unique partner of the session $(x, s)$, if one exists according to Definition 4.1. It is assumed that $C$ 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 $y = \text{pid}^x_s$. 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 4.7. The FindPartnerSession$(x, s)$ function.

```
1: FindPartnerSession$(x \in \mathbb{N}, s \in \mathbb{N})$
2: if $(\text{sid}^x_s = \lambda \text{ or } sk^x_s = \lambda)$
3: return $-1$;
4: $c \leftarrow 0$ // number of partners
5: for $z \leftarrow 1$ to $N$
6: for all $(u \in \text{sess}_z)$
7: if $((z, u) \neq (x, s) \text{ and } \text{sid}^u_z = \text{sid}^x_z)$
8: $c \leftarrow c + 1$;
9: $t \leftarrow u$;
10: if $(c \neq 1)$ // no unique partner?
11: return $-1$;
12: $y \leftarrow \text{pid}^s_x$;
13: if $(sk^y_y \neq sk^x_x \text{ or } \text{sid}^y_y \neq \text{sid}^x_x \text{ or } \text{role}^y_y = \text{role}^x_x \text{ or } \text{pid}^y_y \neq x)$
14: return $-1$;
15: return $t$;
```
The idea used in Algorithm 4.7 is as follows: We first count the number of sessions in the system, except \((x, s)\), whose session identifier is \(\text{sid}_x^s\). According to Definition 4.1, there must be a unique session, beyond \((x, s)\), with session identifier \(\text{sid}_x^s\). 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 \(\text{sid}_x^s\) is examined for other conditions in Definition 4.1.

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 will resist a variety of attacks. Our definition of freshness is presented in Definition 4.2.

**Definition 4.2 (Freshness).** A session \((x, s)\) is called fresh if the following conditions hold (here, \(y = \text{pid}_x^s\)):

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

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 \(\text{skTime}_x^s\) and \(\text{revealTime}_{xy}\) 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.

\(C\) uses the function \(\text{fresh}(x, s)\) in Algorithm 4.8 to find whether the session \((x, s)\) is fresh, according to Definition 4.2.

**Algorithm 4.8.** The \(\text{fresh}(x, s)\) function.

```plaintext
1: \(\text{fresh}(x \in \mathbb{N}, s \in \mathbb{N})\)
2: \(\text{if} \ (\text{skTime}_x^s < \text{revealTime}_{xy} \text{ or exposed}_x^s = 1)\)
3: \(\text{return} \ 0;\)
4: \(y \leftarrow \text{pid}_x^s;\)
5: \(t \leftarrow \text{FindPartnerSession}(x, s);\)
6: \(\text{if} \ (t \neq -1)\)
7: \(\text{if} \ (\text{exposed}_y^t = 1 \text{ or skTime}_y^t < \text{revealTime}_{yx})\)
8: \(\text{return} \ 0;\)
9: \(\text{return} \ 1;\)
```
Algorithm 4.9. The Test\((x, s)\) function.

1: \textbf{Test}\((x \in \mathbb{N}, s \in \mathbb{N})\)
2: \hspace{1em} \textbf{if} \((x > N \text{ or } s \notin \text{sess}_x \text{ or } b \neq -1 \text{ or } \text{state}\_x^s = \lambda \text{ or } \neg \text{fresh}(x, s) \text{ or } sk\_x^s = \lambda)\)
3: \hspace{2em} \textbf{output} 0 and abort;
4: \hspace{1em} \(\langle X, S \rangle \leftarrow (x, s)\);
5: \hspace{1em} \(b \leftarrow_R \{0, 1\}\);
6: \hspace{1em} \hspace{1em} \textbf{if} \((b = 0)\)
7: \hspace{2em} \textbf{key} \leftarrow sk\_x^s;
8: \hspace{1em} \textbf{else}
9: \hspace{2em} \textbf{key} \leftarrow_R \{0, 1\}^{k(n)};
10: \hspace{1em} \textbf{return} \textbf{key};

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(a): \(X, S,\) and \(b\). The pair \((X, S)\) denotes the target session, and \(b\) specifies the random coin. The initial value for \(b\) is \(-1\).

\textbf{The Test\((x, s)\) function.} By making this query, the adversary specifies that \((x, s)\) is her target session of choice. She will lose the game if any of the following conditions hold:
1. The party \(x\) or session \((x, s)\) is non-existent;
2. The adversary has already made a Test query (checked by verifying \(b \neq -1\), as the value of \(b\) will be set to either 0 or 1 by this function);
3. The target session state is empty;
4. The target session is unfresh;

\(^6\)This case only happens when the target session has already detected an error and purged its state variables. Notice that as per Remark 4.3, the challenger does not purge the session key. So, testing \(sk\_x^s \neq \lambda\) does not suffice.
5. The session key is not established yet.

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 $\text{Test}(x, s)$ function is described in Algorithm 4.9.

▶ The $\text{Guess}(b')$ function. By calling this function, the adversary announces her guess of the random coin $b$. As a sanity check, the function checks whether a $\text{Test}$ 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 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 $\text{Guess}$ outputs 1. Otherwise, the adversary loses and $\text{Guess}$ outputs 0. In both cases, $D$ aborts the game, as $\text{Guess}$ is the last possible query in the game.

The pseudocode of the $\text{Guess}(b')$ function is described in Algorithm 4.10.

### Algorithm 4.10. The $\text{Guess}(b')$ function.
1: $\text{Guess}(b' \in \{0, 1\}^*)$
2: if ($b = -1$ or $b' \neq b$ or $\neg \text{fresh}(X, S)$)
3: output 0 and abort;
4: output 1 and abort;

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

$$(3) \quad \text{Adv}_{A, D, \Pi}^{\text{ake}}(n) \overset{\text{def}}{=} \Pr[\langle D^\Pi, A \rangle(n) = 1] - \frac{1}{2},$$

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

**Definition 4.3** (Secure AKE). An AKE protocol $\Pi$ is called $(n, T(n), \epsilon(n))$-secure if:

$$(4) \quad \max_A \{ \text{Adv}_{A, D, \Pi}^{\text{ake}}(n) \} < \epsilon(n),$$

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

We call $\Pi$ a secure AKE if for all $c \in \mathbb{N}$, any probabilistic polynomial time adversaries $A$, and all sufficiently large $n \in \mathbb{N}$, we have $\text{Adv}_{A, D, \Pi}^{\text{ake}}(n) < n^{-c}$.

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 setup.
5.1. Reducing a multi-party setting to a two-party setting. Let $q_{\text{reg}}$ denote the upper bound on the number of Register and ShareLTK queries that the adversary makes to $D$. Furthermore, let $q$ denote the upper bound on the total number of queries that the adversary makes to $D$.

Theorem 5.1 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 5.1. Let $\Pi$ be any AKE protocol (not necessarily secure). For all $n \in \mathbb{N}$ and any $(q_{\text{reg}}, q_{\text{lkt}})$-adversary $A$ against $\Pi$ which runs in time at most $T(n)$, there exists a $(2, 1)$-adversary $S$ which runs in time at most $T_S(n) = q(n) \cdot \text{poly}(n) + T_A(n)$, such that:

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

The proof essentially results from three facts in the modeling:

1. The long-term keys ($K^n_{T, y}$) are generated independently from each other;
2. The randomness of sessions ($\text{rnd}^n_x$) are generated independently from each other;
3. The session states ($\text{state}^n_x$) are stored and updated independently from each other.

Since the adversary $A$ creates at most $q_{\text{reg}}$ 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 $S$ randomly picks two distinct integers $\alpha$ and $\beta$ from the set $\{1, \ldots, q_{\text{reg}}\}$. The values $\alpha$ and $\beta$ will denote the ordinal identities of two parties, which we call special parties. Next, $S$ starts to simulate the game between $A$ and $D$. It answers all queries of $A$ by itself, except those queries related to the special parties. The latter type of queries are forwarded to $D$, and the answers are relayed back to $A$. In this way, $S$ will make at most two $\text{Register}(\cdot)$ queries, and at most one $\text{ShareLTK}(\cdot)$ query.

In the final stage, if $A$ picks a session between the special parties as the test session, $S$ will win with the same probability that $A$ wins. Otherwise, $S$ 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 Theorem 5.1 more rigorously.

Proof. Let $S$ be the adversary described by Algorithm 5.1. Since the main purpose of $S$ is to simulate a multi-party AKE setting for $A$, 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 $\text{TimeEvent}(\cdot)$ function (lines 3–4 of Algorithm 5.1), 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 $S$. As an example, when $A$ makes a $\text{ShareLTK}(x, y)$, 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 implicitly by $S$, as specified by the rules in Section 4.

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

- **Initialization**, lines 1–2. $S$ first picks the ordinal identity of the two special parties. This is done by picking two distinct random integers $\alpha$ and $\beta$ from the set $\{q_{\text{reg}}\} = \{1, \ldots, q_{\text{reg}}\}$. To make sure that the numbers are distinct, $\alpha$ is first selected randomly from $\{q_{\text{reg}}\}$, and then $\beta$ 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 $A$. Other state variables ($p, p_\alpha$, and $p_\beta$) keep a mapping between the ordinal identities of the
special parties in the game between $A$ and $S$, and the game between $S$ and $D$. The mapping is detailed in the description of the $\text{Register}(\cdot)$ query below.

- $\text{TimeEvent}()$, lines 3–4. A $\text{TimeEvent}()$ query is made by $D$. The simulator $S$ answers this query by running Algorithm 4.1, which involves incrementing $T$, notifying $A$, and updating all long-term keys.

- $\text{Register}()$, lines 5–12. When $A$ makes a $\text{Register}()$ query, $S$ first checks whether this is either the $\alpha^{th}$ or $\beta^{th}$ query of type $\text{Register}$ (the if statement at line 6). If this is the case, $N$ and $p$ are incremented, and $p_N$ is set to $p$. Example 5.1 illustrates how the mapping works.

Example 5.1. Assume that $A$ is an adversary which makes at most twenty $\text{Register}()$ queries. Therefore, $S$ picks $\alpha$ and $\beta$ randomly and distinctly from the set $\{1, \ldots, 20\}$. Assume that $\alpha = 18$ and $\beta = 13$. When $A$ makes the thirteenth $\text{Register}()$ query, $S$ assigns $p \leftarrow p + 1 = 1$, and $p_{13} \leftarrow p = 1$, and registers the first party with $D$. When $A$ makes the eighteenth $\text{Register}()$ query, $S$ assigns $p \leftarrow p + 1 = 2$, and $p_{18} \leftarrow p = 2$, and registers the second party with $D$.

Next, the query is forwarded to $D$, and the identifier $id_N$ is received.

On the other hand, if this is neither the $\alpha^{th}$ nor $\beta^{th}$ query of type $\text{Register}$, it is treated ordinarily via a call to Algorithm 4.2, where $id_N$ is computed. Finally, $id_N$ is returned to the adversary $A$.

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

- $\text{ShareLTK}(x, y)$, lines 13–17. The simulator $S$ 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 $A$ wants a long-term key to be shared between the special parties. Therefore, the query is forwarded to $D$, with one subtlety: $S$ makes the proper mapping, and sends $\text{ShareLTK}(p_x, p_y)$ to $D$. Otherwise, Algorithm 4.3 is executed.

Nothing is sent back to the adversary as the returned value. Internally, either calling $D$ or calling Algorithm 4.3 creates a random and independent key between the corresponding parties. Therefore, $S$ simulates the $\text{ShareLTK}(\cdot)$ query perfectly.

- $\text{Send}(x, s, y, m)$, lines 18–26. The simulator $S$ first checks whether $\{x, y\} = \{\alpha, \beta\}$.
  - If this is the case, $A$ wants to send a message between two special parties. This is handled by making the proper mapping $(x \times p_x$ and $y \times p_y)$, and forwarding the query to $D$. After receiving the answer $(m', \text{sid}'_x, \text{pid}'_x, \text{skTime}'_x, \text{acc}'_x)$ from $D$, the simulator modifies $\text{pid}'_x$. It is because in the returned value, $\text{pid}'_x = p_y$; whereas $p_y$ should be mapped back to $y$.
  - If $\{x, y\} \neq \{\alpha, \beta\}$, then $A$ wants to send a message between two non-special parties, or between a special and a non-special party. In this case, Algorithm 4.4 is executed, and the result $(m', \text{sid}'_x, \text{pid}'_x, \text{skTime}'_x, \text{acc}'_x)$ is received. In either case, the tuple $(m', \text{sid}'_x, \text{pid}'_x, \text{skTime}'_x, \text{acc}'_x)$ is returned to $A$. This tuple is generated by running the protocol II, and then extracting the information from $\text{state}'_x$ via the function $f_{\Pi}^{\Pi}$ (see lines 12–21 of Algorithm 4.4). Notice that the inputs to II 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 $\text{state}'_x$ is initially empty. The time stage $T$ is incremented.
Forsakes: a forward-secure AKE independently, and the rest of the inputs to $\Pi$ (i.e., $id_x$, $id_y$, and $m$) are determined by the adversary.

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

$\blacktriangleright$ ExposeSS$(x, s)$, lines 27–35. The simulator first finds the partner of the session $(x, s)$ by letting $y \leftarrow \text{pid}_s^x$. If both partners are the special parties, the ExposeSS query is forwarded to $D$ (making proper mappings), and the results are returned to $\mathcal{A}$. Otherwise, Algorithm 4.5 is executed, and the results are returned to $\mathcal{A}$.

In both cases, the returned value is of the form $(\text{state}^s_x, \text{rnd}^s_x)$. As explained above (while describing the way a $\text{Send}$ 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.

$\blacktriangleright$ RevealLTK$(x, y)$, lines 36–42. If this query is made for the key shared between the special parties, $D$ will respond the query ($\mathcal{S}$ makes proper mappings beforehand). Otherwise, Algorithm 4.6 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.

$\blacktriangleright$ 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 $y \leftarrow \text{pid}_s^x$. If the partners of the test session are the special parties, the query is forwarded to $D$ (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 $D$, 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 $q_{\text{reg}}$ parties before a Test query, there are at most $\binom{n_{\text{reg}}}{2}$ pairs of parties in the system. Consequently, the probability that both partners of the test session are special parties is at least $1/q_{\text{reg}}^2$.

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

$\blacktriangleright$ Guess$(b')$, lines 50–51. The simulator simply forwards the guess $b'$ to $D$. This will finish the simulation, as $D$ 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,

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

Notice that $\mathcal{S}$ answers each query of $\mathcal{A}$ with at most a polynomial overhead. Therefore, the running time $T_S(n)$ of $\mathcal{S}$ is $T_A(n) + q(n) \cdot \text{poly}(n)$.
Theorem 5.1 has an important implication: Since for polynomial-time adversaries, the number of $q_{\text{reg}}$ 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 5.1. Let the adversary register at most $q_{\text{reg}} \overset{\text{def}}{=} q_{\text{reg}}(n)$ parties, and assume the system uses random $\ell \overset{\text{def}}{=} \ell(n)$-bit identifiers. Then, the probability that at least two identifiers are equal is at most \( \left( \frac{q_{\text{reg}}}{2} \right)^2 \leq q_{\text{reg}}^2 2^{-\ell}. \)

In the following, we assume that the identifiers are unique. Later, in Theorem 5.2, we account for the probability stated in Fact 5.1.

Fact 5.2. The probability that two particular sessions in FORSAKES output the same nonce is $2^{-r(n)}$.

Notice that Fact 5.2 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 $r \overset{\text{def}}{=} r(n)$ from its random tape, and outputs it.

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

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

Corollary 5.1. In an execution of FORSAKES with $\sigma \overset{\text{def}}{=} \sigma(n)$ 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 $\text{Send}$ query), but not both. Therefore, we obtain the following corollary.

Corollary 5.2. 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 5.1, will be accounted for later in Theorem 5.2. Consequently, we will assume that the fifth condition of Definition 4.1 never holds.

Properties of FORSAKES messages. Let $M$ 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 \text{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 5.1 and Lemma 5.2 (in the next section) consider two separate cases, depending on whether $m$ belongs to $\mathcal{M}$ or not.

**Lemma 5.1.** Let $m \in \mathcal{M}$ be the second or third message of FORSAKES, delivered via \text{Send}(x, s, y, m). Assume that after the delivery, $\text{acc}_x^s = 1$. 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 makes the delivery.

Any second-or-third FORSAKES message $m \in \mathcal{M}$ is authenticated. Since $m$ includes the identifiers of the sender and responder respectively, $(x, s)$ rejects $m$ if the sender is any party other than $y$. Consequently, $\text{acc}_x^s = 1$ 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 $\text{acc}_x^s = 1$. Therefore, case (2) is ruled out as well. \hfill $\square$

**Remark 5.1.** Lemma 5.1 shows, in particular, that parallel session attacks [10] 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 $A$ against an AKE protocol in a multi-party system can be reduced to a an efficient adversary $S$ against an AKE protocol in a two-party system, such that the running times and advantages of $A$ and $S$ 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 $K^T_{xy} = K^T_{yx}$ 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 \text{ExposeSS}(X, S)$ or \text{ExposeSS}(Y, S^*)$.

Advances in Mathematics of Communications Volume 9, No. 4 (2015), 471–514
Algorithm 5.2. The algorithm which $F_{\mathcal{O}, \mathcal{R}_K; T_K; \mathcal{V}_K}(1^n)$ uses to handle II (i.e., protocol FORSAKES). We assume that the input is well formed, and do not check for syntax. Note that in this experiment, $F$ only uses the $T_K$ and $\mathcal{V}_K$ oracles.

| Function II($-, T, \text{state}_k, \text{rd}_y, \text{id}_x, \text{id}_y, m$) |
|---------------------------------------------------------------|
| 1: $m' \leftarrow st \leftarrow \lambda$; |
| Case $m = 0 \parallel \text{id}_y$: |
| 2: $\text{sid}^*_y \leftarrow \text{rdn}_y \parallel \#^\ast$; // $\#^\ast$ is a wildcard |
| 3: $m' \leftarrow 1 \parallel \text{id}_x \parallel \text{id}_y \parallel T \parallel \text{rnd}_y$; |
| 4: $st \leftarrow \text{sid}^*_y \parallel 'I' \parallel \text{id}_y \parallel T \parallel 0^k \parallel 0^k \parallel \lambda$; |
| Case $m = 1 \parallel \text{id}_x \parallel \text{id}_y \parallel T \parallel \text{rd}_y$: |
| 5: If $(\text{id}_x \neq \text{id}_y$ or $\text{id}_x \neq \text{id}_x$) |
| 6: return $(m', st)$; |
| 7: $\text{sid}^*_x \leftarrow \text{nd}_x \parallel \text{rdn}_y$; |
| 8: $\text{msg}_2 \leftarrow 2 \parallel \text{id}_x \parallel \text{id}_y \parallel T \parallel \text{sid}^*_x$; |
| 9: $\text{auth}_2 \leftarrow T_K(\text{msg}_2, 1 \parallel \text{sid}^*_x)$; |
| 10: $m' \leftarrow \text{msg}_2 \parallel \text{auth}_2$; |
| 11: $st \leftarrow \text{sid}^*_y \parallel 'R' \parallel \text{id}_y \parallel T \parallel 0^k \parallel 0^k \parallel \lambda$; |
| Case $m = 2 \parallel \text{id}_x \parallel \text{id}_y \parallel T \parallel \text{id}_y \parallel \text{auth}_2$: |
| 12: if $(\text{id}_x \neq \text{id}_y$ or $\text{id}_x \neq \text{id}_y$ or $\neg$match($\text{sid}_x, \text{sid}_y$)) |
| 13: return $(m', st)$; |
| 14: $\text{sid}_x \leftarrow \text{sid}_y$; // No more wildcards. |
| 15: if $(\neg \mathcal{V}_K(\text{msg}_2, 1 \parallel \text{sid}^*_x, \text{auth}_2))$ |
| 16: return $(m', st)$; |
| 17: if $(m \notin \mathcal{M}$ and fresh($x, s$)) |
| 18: output $(\text{msg}_2, 1 \parallel \text{sid}^*_x, \text{auth}_2)$ and abort; |
| 19: $\text{msg}_3 \leftarrow 3 \parallel \text{id}_x \parallel \text{id}_y \parallel T \parallel \text{sid}^*_y$; |
| 20: $\text{auth}_3 \leftarrow T_K(\text{msg}_3, 1 \parallel \text{sid}^*_y)$; |
| 21: $m' \leftarrow \text{msg}_3 \parallel \text{auth}_3$; |
| 22: $st \leftarrow \text{sid}^*_y \parallel 'R' \parallel \text{id}_y \parallel T \parallel 0^k \parallel 0^k \parallel 1$; |
| Case $m = 3 \parallel \text{id}_x \parallel \text{id}_y \parallel T \parallel \text{id}_y \parallel \text{auth}_3$: |
| 23: if $(\text{id}_x \neq \text{id}_y$ or $\text{id}_x \neq \text{id}_y$ or $\text{sid} \neq \text{sid}^*_y$) |
| 24: return $(m', st)$; |
| 25: if $(\neg \mathcal{V}_K(\text{msg}_3, 1 \parallel \text{sid}^*_y, \text{auth}_3))$ |
| 26: return $(m', st)$; |
| 27: if $(m \notin \mathcal{M}$ and fresh($x, s$)) |
| 28: output $(\text{msg}_3, 1 \parallel \text{sid}^*_y, \text{auth}_3)$ and abort; |
| 29: $m' \leftarrow \lambda$; |
| 30: $st \leftarrow \text{sid}^*_y \parallel 'R' \parallel \text{id}_y \parallel T \parallel 0^k \parallel 0^k \parallel 1$; |
| 31: return $(m', st)$; |

- If the adversary reveals either the long-term keys $K^T_y$ or $K^T_x$ 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 $\text{RevealLTK}(x, y)$ or $\text{RevealLTK}(y, x)$ query, or makes either of these queries in a time stage after the session key of the test session is generated.

Lemma 5.2 pertain to the case $m \notin \mathcal{M}$, where the adversary $A$ successfully forges a second-or-third message of FORSAKES. It is the “dual” of Lemma 5.1 in the previous section, which considered the case $m \in \mathcal{M}$. On a high level, Lemma 5.2 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 Appendix A. Let $q_{ro}(n)$ denote an upper bound on the number of the queries which the adversary makes to the random oracle.
Lemma 5.2. Let $A$ be an adversary against FORSAKES, who succeeds with probability $\epsilon_A(n)$ in delivering a second-or-third message $m \notin M$ in a non-colliding execution, to a fresh session $(x,s)$, after which $\text{acc}^*_A = 1$.

Then, there exists a $(q_{\text{fo}},2\sigma + 2,q_{\text{snd}},q_{\text{rand}})$-forger $F$ against RO-Multi-MAC (Construction A.2) which wins the Multi-MAC-Forge (Algorithm A.1) with probability $\epsilon_F(n) = \epsilon_A(n)$.

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

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 $F$ be the adversary against the RO-Multi-MAC, who takes part in the Multi-MAC-Forge experiment. By definition of this experiment, $F$ has access to four oracles: $O, R^0_K, \mathcal{T}^0_K$, and $V^0_K$. During the experiment, $F$ uses $A$ as a black-box, and simulates for $A$ a two-party execution of FORSAKES, and answers to $A$’s queries as follows (as before, we assume that $A$ does not make an invalid query, which makes her lose the authentication game):

- **Random oracle queries**: The queries are forwarded to $F$’s random oracle, and the answers are returned to $A$.
- **Register()**: A random $\text{id} \in \{0,1\}^\ell$ is selected by $F$. This id is then returned to $A$ by $F$.
- **ShareLTK(x,y)**: Nothing is actually done, since the Multi-MAC-Forge experiment has already picked a random key in the initialization phase.
- **Send(x,y,m)**: This query is handled by $F$ as in Algorithm 4.4; however, when the protocol II (i.e., FORSAKES) is called, $F$ runs Algorithm 5.2. The idea behind this algorithm is simple: It does not have access to the LTK, but uses the tag-generation oracle $\mathcal{T}^0_K$ to generate $\text{Auth}_2$ and $\text{Auth}_3$ (see Protocol 1), and uses the tag-verification oracle $V^0_K$ to verify them. It also checks whether the incoming message belongs to the set $M$. If a second-or-third message has a valid tag, but is not in $M$, it means that $A$ has successfully forged a valid message (assuming the session is fresh). In this case, the forger $F$ outputs the forgery just found, and finishes the game successfully.
- **ExposeSS(x,s)**: First, $F$ uses the $R^0_K$ oracle to get the session and integrity keys. This is done by setting $sk^s_x \leftarrow R^0_K(0||\text{id}^s_x)$ and $ik^s_x \leftarrow R^0_K(1||\text{id}^s_x)$. Next, state$_x^s$ is updated, by setting $sk^s_x$ and $ik^s_x$ into the session and integrity key “placeholders” of state$_x^s$. Finally, the pair $(\text{state}_x^s,\text{rnd}_x^s)$ is returned to $A$.
- **RevealLTK(x,y)**: This query is not allowed at the current time stage, since it makes all sessions unfresh. However, $A$ can be given the value of the long-term key at the next time stage. To this end, $F$ simply queries the $R^0_K$ oracle at $\lambda$, to receive the key $K’ = R^0_K(\lambda) = \mathcal{O}(\lambda||K) = \mathcal{O}(K)$. This is the value of $K$ in the next time stage, and is returned to $A$.
- **Test(x,y)**: First, $F$ flips a random coin $b$. If $b = 1$, a random $k(n)$-bit binary string is returned to $A$. Otherwise, $F$ sets $sk^s_x \leftarrow R^0_K(0||\text{id}^s_x)$, and returns $sk^s_x$ to $A$.
- **Guess(b’)**: This query finishes the game with failure, since it is the ultimate query of $A$, and she has not forge any messages yet.

Since the execution is non-colliding, Lemma 5.1 shows that no messages output by sessions can be delivered with a fake source, or at a fake destination. The
forger $F$ simulates the view of $A$ successfully. $F$ succeeds if and only if $A$ succeeds. Therefore, $\epsilon_F(n) = \epsilon_A(n)$.

The maximum number of queries $F$ makes to its oracles are as follows:

- $q_{ro}$ queries at $O$.
- $2\sigma + 2$ queries at $R_K^{O}$. There are at most $\sigma$ sessions on the system, each of which can be exposed. Every $\text{ExposeSS}$ query requires 2 queries at $R_K^{O}$. Furthermore, there can be at most one $\text{RevealLTK}$ and one $\text{Test}$ query, each of which requires 1 query at $R_K^{O}$. Therefore, the total number of queries at $R_K^{O}$ is at most $2\sigma + 2$.
- $q_{snd}$ queries at $T_K^{O}$, and $q_{snd}$ queries at $V_K^{O}$. This is because each message sent may need a tag verification, and a tag generation.

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

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 5.2 (Main Theorem).** In the two-party setting, FORSAKES is a secure AKE protocol, as per Definition 4.3.

**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 $m \in M$, with a fake source or at a wrong destination.
- $E_4$: The adversary successfully forges a second-or-third message $m \notin M$, 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 $E_1 \land E_2 \land E_3 \land E_4$.

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

1. Faithfully deliver a message;
2. Delay the delivery of a message beyond a time stage;
3. Delete a message.
4. Try to “un-partner” two sessions.

In FORSAKES, option (2) causes the destination session to reject, and option (3) prevents the generation of the session key at the destination session. Option (4) is ruled out because conditioned on the above event, the adversary cannot change the parameters that affect whether two sessions are partnered (session key, session ID, session partner ID, and session role). See also Remark 4.3.

Therefore, the adversary is left with options (1). However, if she delivers messages faithfully, and targets a fresh session using the $\text{Test}$ query, she will receive a random and independent value $\text{key}$, regardless of the internal coin toss of $\text{Test}$. Therefore, the advantage of $A$ in guessing the value of the coin is 0.

The following corollary is immediate by combining Theorem 5.2 with Theorem 5.1.

**Corollary 5.3.** 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 [6], we note that the random oracle was merely a tool used in the proofs, and it can be replaced with pseudorandom functions [44]. 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 [1], 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 [6, 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.

Remark 6.1. 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 [3] 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 2 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 on 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 databases 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 a model where the adversary has the ability to desynchronize the long-term keys between any pair of parties, using a Desync\((x, y)\) query. 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. The security definition should be updated in such a way that it accounts for \textsc{Desync} queries: Entities should be able to detect asynchrony and resynchronize using their non-updating LTKs.

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] A. Banerjee, C. Peikert and A. Rosen, Pseudorandom functions and lattices, in Advances in Cryptology—EUROCRYPT 2012, Springer, Cambridge, United Kingdom, \textbf{7237} (2012), 719–737.

[2] D. Basin, C. Cremers and S. Meier, Provably repairing the ISO/IEC 9798 standard for entity authentication, in Principles of Security and Trust, Springer, \textbf{7215} (2012), 129–148.

[3] M. Bellare, R. Canetti and H. Krawczyk, Keying hash functions for message authentication, in Advances in Cryptology—CRYPTO ’96, Springer, Santa Barbara, CA, USA, \textbf{1109} (1996), 1–15.

[4] M. Bellare, R. Canetti and 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, TX, USA, 1998, 419–428.

[5] M. Bellare, D. Pointcheval and P. Rogaway, Authenticated key exchange secure against dictionary attacks, in Advances in Cryptology—EUROCRYPT ’00, Springer, Bruges, Belgium, \textbf{1807} (2000), 139–155.
Universally composable security: A new paradigm for cryptographic protocols

R. Canetti, R. Bird, I. Gopal, A. Herzberg, P. Janson, S. Kutten, R. Molva and M. Yung, Systematic design of two-party authentication protocols, in Advances in Cryptology—CRYPTO ’91, Springer, Santa Barbara, CA, USA, 576 (1992), 44–61.

A. Biryukov, J. Lano and B. Preneel, Cryptanalysis of the alleged SecurID hash function, in Selected Areas in Cryptography (SAC 2003), Springer, Windsor, ON, Canada, 3006 (2004), 130–144. Extended version available from http://eprint.iacr.org/2003/162.

A. Biryukov, J. Lano and B. Preneel, Recent attacks on alleged SecurID and their practical implications, Computers & Security, 24 (2005), 364–370.

S. Blake-Wilson, D. Johnson and 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, 1355 (1997), 30–45.

S. Blake-Wilson and A. Menezes, Identity authentication and authenticated key transport protocols: Employing asymmetric techniques, in Security Protocols, Lecture Notes in Computer Science, Springer, Paris, France, 1361 (2005), 137–158.

C. Boyd, Hidden assumptions in cryptographic protocols, IEEE Proceedings of Computers and Digital Techniques, 137 (1990), 433–436.

C. Boyd and A. Mathuria, Protocols for authentication and key establishment, Springer, 2003.

C. Brzuska, M. Fischlin, N. P. Smart, B. Warinschi and S. C. Williams, Less is more: Relax yet composable security notions for key exchange, International Journal of Information Security, 12 (2013), 267–297.

C. Brzuska, M. Fischlin, B. Warinschi and 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, IL, USA, 2011, 51–62.

J. Camenisch, A. Lysyanskaya and 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, 525–536.

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, 136–145, See [21] for the full version.

R. Canetti, Universally composable security: A new paradigm for cryptographic protocols, Cryptology ePrint Archive, Report 2000/067, 2000. Available from http://eprint.iacr.org/2000/067. See [20] for the conference version.

R. Canetti, S. Halevi and J. Katz, A forward-secure public-key encryption scheme, in Advances in Cryptology—Eurocrypt 2003, Springer, Warsaw, Poland, 2656 (2003), 255–271, See [23] for the journal version.

R. Canetti, S. Halevi and J. Katz, A forward-secure public-key encryption scheme, Journal of Cryptology, 20 (2007), 265–294, See [22] for the conference version.

R. Canetti, S. Halevi, J. Katz, Y. Lindell and P. MacKenzie, Universally composable password-based key exchange, in Advances in Cryptology—EUROCRYPT 2005, Springer, Aarhus, Denmark, 3494 (2005), 404–421, Full version is available from http://eprint.iacr.org/2005/196.

R. Canetti and H. Krawczyk, Analysis of key-exchange protocols and their use for building secure channels, in Advances in Cryptology—EUROCRYPT ’01, Springer, Innsbruck, Austria, 2045 (2001), 453–474, Full version is available at http://eprint.iacr.org/2001/040.

R. Canetti and H. Krawczyk, Universally composable notions of key exchange and secure channels (extended abstract), in Advances in Cryptology—EUROCRYPT ’02, Springer,
Amsterdam, The Netherlands, 2332 (2002), 337–351, Full version is available at http://eprint.iacr.org/2002/059.

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

[28] H.-Y. Chien, SASI: A new ultralightweight RFID authentication protocol providing strong authentication and strong integrity, IEEE Transactions on Dependable and Secure Computing, 4 (2007), 337–340.

[29] K.-K. R. Choo, Secure key establishment, Springer, 2009.

[30] K.-K. R. Choo, C. Boyd and Y. Hitchcock, Examining indistinguishability-based proof models for key establishment protocols, in Advances in Cryptology—ASIACRYPT ’05, Springer, Chennai, India, 3788 (2005), 585–604.

[31] K.-K. R. Choo, C. Boyd, Y. Hitchcock and 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, Analfi, Italy, 3352 (2005), 351–366.

[32] J. Clark and J. Jacob, On the security of recent protocols, Information Processing Letters (IPL), 56 (1995), 151–155.

[33] J. Clark and J. Jacob, A survey of authentication protocol literature: Version 1.0, 1997, Available from http://www.cs.york.ac.uk/~jac/PublishedPapers/reviewV1_1997.pdf.

[34] S. Contini and Y. L. Yin, Fast software-based attacks on SecurID, in Fast Software Encryption (FSE 2004), Springer, Delhi, India, 3017 (2004), 454–471.

[35] 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, 80–91.

[36] 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, 5536 (2009), 20–33.

[37] D. E. Denning and G. M. Sacco, Timestamps in key distribution protocols, Communications of the ACM, 24 (1981), 533–536.

[38] W. Diffie and M. E. Hellman, New directions in cryptography, IEEE Transactions on Information Theory, IT-22 (1976), 644–654.

[39] W. Diffie, P. C. Oorschot and M. J. Wiener, Authentication and authenticated key exchanges, Designs, Codes and Cryptography, 2 (1992), 107–125.

[40] M. S. Dousti and R. Jalili, Efficient Statistical Zero-Knowledge Authentication Protocols for Smart Cards Secure Against Active & Concurrent Attacks, Cryptology ePrint Archive, 2013, Available from http://eprint.iacr.org/2013/709. See [41] for the journal version.

[41] M. S. Dousti and R. Jalili, An efficient statistical zero-knowledge authentication protocol for smart cards, International Journal of Computer Mathematics, Published online: http://www.tandfonline.com/doi/abs/10.1080/00207160.2015.1011629. See [40] for the full version.

[42] U. Feige, A. Fiat and A. Shamir, Zero-knowledge proofs of identity (extended abstract), in Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC ’87), New York, NY, USA, 1987, 210–217, See [43] for the journal version.

[43] U. Feige, A. Fiat and A. Shamir, Zero-knowledge proofs of identity, Journal of Cryptology, 1 (1988), 77–94, See [42] for the conference version.

[44] O. Goldreich, S. Goldwasser and S. Micali, How to construct random functions, Journal of the ACM (JACM), 33 (1986), 792–807.

[45] C. G. G"unther, An identity-based key-exchange protocol, in Advances in Cryptology—EUROCRYPT ’89, Springer, Houthalen, Belgium, 1989, 29–37.

[46] D. Hofheinz, J. Müller-Quade and 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, 2908 (2003), 61–84.

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

[48] H. Krawczyk, HMQV: A high-performance secure Diffie–Hellman protocol (extended abstract), in Advances in Cryptology—CRYPTO’05, Springer, Santa Barbara, CA, USA, 3621 (2005), 546–566, Full version is available at http://eprint.iacr.org/2005/176.
B. LaMacchia, K. Lauter and A. Mityagin, Stronger security of authenticated key exchange, in Proceedings of the 1st International Conference on Provable Security (ProvSec ’07), Springer, Wollongong, Australia, 4784 (2007), 1–16.

L. Law, A. Menezes, M. Qu, J. Solinas and S. Vanstone, An efficient protocol for authenticated key agreement, Designs, Codes and Cryptography, 28 (2003), 119–134.

C. Lenzen, T. Locher, P. Sommer and R. Wattenhofer, Clock synchronization: Open problems in theory and practice, in International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2010), Springer, 5901 (2010), 61–70.

A. Menezes, Another look at HMQV, Journal of Mathematical Cryptology, 1 (2007), 47–64, Available from http://eprint.iacr.org/2005/205.

A. Menezes, M. Qu and S. Vanstone, Some new key agreement protocols providing implicit authentication, in Presented at the Workshop on Selected Areas in Cryptography (SAC ’95), 1995, 22–32.

R. M. Needham and M. D. Schroeder, Using encryption for authentication in large networks of computers, Communications of the ACM, 21 (1978), 993–999.

D. Otway and O. Rees, Efficient and timely mutual authentication, ACM SIGOPS Operating Systems Review, 21 (1987), 8–10.

R. C.-W. Phan, Cryptanalysis of a new ultralightweight RFID authentication protocol—SASI, IEEE Transactions on Dependable and Secure Computing, 6 (2009), 316–320.

A. P. Sarr, P. Elbaz-Vincent and 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, 6280 (2010), 219–234.

V. Shoup, On formal models for secure key exchange, Technical report, IBM Zurich Research Lab, 1999, Version 4 is available at http://eprint.iacr.org/1999/012.

V. Shoup and A. Rubin, Session key distribution using smart cards, in Advances in Cryptology—EUROCRYPT ’96, Springer, Saragossa, Spain, 1070 (2001), 321–331.

H.-M. Sun, W.-C. Ting and K.-H. Wang, On the security of Chien’s ultralightweight RFID authentication protocol, IEEE Transactions on Dependable and Secure Computing, 8 (2011), 315–317.

I. Wiener, Sample SecurID token emulator with token secret import, 2000, Available from http://archives.neohapsis.com/archives/bugtraq/2000-12/0428.html.

K. Yoneyama and 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, 6980 (2011), 348–365.

APPENDIX 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 (Gen, MAC, Verif), which satisfy the following [47, 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 $\text{Verif}(K, m, \text{MAC}(K, m)) = 1$. Informally, the security requirement is that no efficient adversary $F$ 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 $\text{MAC}(K, \cdot)$. 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 $\text{Verif}(K, m, t) = 1$. Otherwise, she loses the game.\footnote{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.}

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 A.1.

**Construction A.1** (RO-MAC). Let $a \overset{\text{def}}{=} a(n)$ and $b \overset{\text{def}}{=} b(n)$ 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: $\text{Gen}(1^n)$ picks a random key $K$ from $\{0, 1\}^b$. The tag of a message $m$ is $\mathcal{O}(m \mid \mid K)$. Finally, $\text{Verif}(K, m, t)$ outputs 1 if and only if $t = \mathcal{O}(m \mid \mid K)$.

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 A.2** provides a more formal description of RO-Multi-MAC.

**Construction A.2** (RO-Multi-MAC). Let $a$, $b$, and $\mathcal{O}$ be as in Construction A.1. The RO-Multi-MAC is obtained from $\mathcal{O}$ as follows: $\text{Gen}(1^n)$ 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 $\mathcal{O}(\cdot \mid \mid K)$ 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 $(x, t = \mathcal{O}(m \mid \mid k_x))$.

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

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 arbitrary random oracle $\mathcal{O} : \{0, 1\}^* \to \{0, 1\}^a$.
2. **The reveal oracle** $R^\mathcal{O}_K$: On input $x$, reveals the MAC key $k_x = \mathcal{O}(x \mid \mid K)$. This oracle also adds $x$ to the set $\text{rev}$. 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 $\mathcal{T}_K^O$: On input $(m, x)$, returns the MAC $t$ of $m$ under $k_x$, by computing $k_x \leftarrow \mathcal{O}(x || K)$ and $t \leftarrow \mathcal{O}(m || k_x)$. This oracle also adds the pair $(m, x)$ to the set $\text{Asked}$. 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 $\mathcal{V}_K^O$: On input $(m, x, t)$, computes the valid tag for $m$ under $k_x \leftarrow \mathcal{O}(x || K)$, and returns true if and only if the tag equals $t$.

**Remark A.1.** We will assume, without loss of generality, that $F$ 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 A.1.** The Multi-MAC-Forge experiment for $F$ against RO-Multi-MAC.

1. $\text{Exp}_{F, \text{RO-Multi-MAC}}^{\text{mmf}}(n)$
2. $K \leftarrow_R \{0, 1\}^b$;
3. $(m, x, t) \leftarrow F_{\mathcal{O}, \mathcal{R}_K^O, \mathcal{T}_K^O, \mathcal{V}_K^O}(1^n)$;
4. if $(x \notin \text{rev} \text{ and } (m, x) \notin \text{Asked} \text{ and } \mathcal{V}_K^O(m, x, t))$
5. output 1 and abort;
6. output 0;

The Multi-MAC-Forge experiment is described by Algorithm A.1. 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 $F$ makes at most $q_o$, $q_r$, $q_t$, and $q_v$ queries at $\mathcal{O}$, $\mathcal{R}_K^O$, $\mathcal{T}_K^O$, and $\mathcal{V}_K^O$, respectively. In general, $q_o$, $q_r$, $q_t$, and $q_v$ can be functions of the security parameter $n$. Define $q \overset{\text{def}}{=} q_o + q_r + 2q_t + 2q_v + 2$.

**Casting the experiment in a different way.** Algorithm A.1 is rather difficult to analyze. To ease the security analysis, we rewrite Algorithm A.1 as Algorithm A.2, 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 A.1.** For any $n \in \mathbb{N}$ and any $(q_o, q_r, q_t, q_v)$-adversary $F$ against RO-Multi-MAC (Algorithm A.2), the probability that $F$ wins the Multi-MAC-Forge experiment is at most $(q^2 + 1)2^{-n} + q2^{-d} + 2^{-b}$, where $d = \min\{a, b\}$.
Algorithm A.2. The $\text{Exp}_{\mathcal{F}, \text{RO-Multi-MAC}}(n)$ experiment (Algorithm A.1) with early sampling.

**Initialization.**
1: $o_i \leftarrow R \{0, 1\}^a$ for $i \in [q]$;
2: $K \leftarrow R \{0, 1\}^b$;
3: $O, \text{rev}, \text{Asked} \leftarrow \emptyset$ and $i \leftarrow 0$;

**Query-response.** Respond to $\mathcal{F}$’s query’s as follows:
4: $O(x)$
5: $i \leftarrow i + 1$
6: if $(x \in \text{dom}(O))$
7: $o_i \leftarrow O(x)$
8: $O \leftarrow O \cup \{(x, o_i)\}$
9: return $o_i$;
10: $\mathcal{R}_K^O(x)$
11: $\text{rev} \leftarrow \text{rev} \cup \{x\}$
12: $k \leftarrow O(x \| K)$
13: return $k$;
14: $\mathcal{T}_K^O(m, x)$
15: $\text{Asked} \leftarrow \text{Asked} \cup \{(m, x)\}$
16: $k_x \leftarrow O(x \| K)$
17: return $O(m \| k_x)$
18: $\mathcal{V}_K(m, x, t)$
19: $k_x \leftarrow O(x \| K)$
20: if $(t = O(m \| k_x))$
21: return 1
22: return 0

**Finalization.** $A$ outputs her guess.
23: $(m, x, t) \leftarrow \mathcal{F} \circ \mathcal{R}_K^O \circ \mathcal{T}_K^O \circ \mathcal{V}_K^O(1^n)$
24: if $(x \notin \text{rev} \text{ and } (m, x) \notin \text{Asked } \text{ and } \mathcal{V}_K^O(m, x, t))$
25: output 1 and abort
26: output 0

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

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 $O$ on a value whose prefix is $K$.

The first two events do not depend on the adversary at all, and:
\[
\Pr[E_1] \leq \binom{q}{2} 2^{-a} \leq q^2 2^{-a},
\]
\[
\Pr[E_2] = q^2 d.
\]

Furthermore,
\[
\Pr[E_3 | E_1 \land E_2] = 2^{-b}.
\]

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 $\mathcal{T}_K^O$. Conditioned on $E = E_1 \land E_2 \land E_3$, all responses from the adversary receives from $O$ 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.

Received March 2014; revised March 2015.

E-mail address: dousti@ce.sharif.edu
E-mail address: jalili@sharif.edu