SAKE\textsuperscript{+}: Strengthened Symmetric-Key Authenticated Key Exchange with Perfect Forward Secrecy for IoT

Seyed Farhad Aghili\textsuperscript{1}, Amirhossein Adavoudi Jolfaei\textsuperscript{2} and Aysajan Abidin\textsuperscript{1}

\textsuperscript{1} imec-COSIC KU Leuven, Kasteelpark Arenberg 10-bus 2452, 3001 Heverlee, Belgium
\{seyedfarhad.aghili, aysajan\}@kuleuven.be
\textsuperscript{2} Department of Computer Engineering, University of Isfahan, Isfahan, Iran
a_adavoudi@eng.ui.ac.ir

Abstract. Lightweight authenticated key exchange (AKE) protocols based on symmetric-key cryptography is important in securing the Internet of Things (IoT). However, achieving perfect forward secrecy (PFS) is not trivial for AKE based on symmetric-key cryptography, as opposed to AKE based on public-key cryptography. The most recent proposals that provide PFS are SAKE and SAKE-AM. In this paper, we first take a closer look at these protocols and observe that they are vulnerable to a number of attacks, such as, replay attack, denial of service (DoS) attack, tracking attack, etc. We then propose countermeasures to both protocols to restore security. Additionally, our proposed scheme SAKE\textsuperscript{+} enables concurrent execution of the protocol, whereas the original SAKE scheme supports the sequential execution of the protocol. The concurrency provided by our scheme makes it more suitable for IoT applications where a server receives and sends information from a large number of IoT end devices. Finally, we prove the security and soundness of our schemes, and verify using ProVerif tool.

Keywords: Authenticated key exchange, Forward secrecy, IoT, Symmetric-key crypto.

1 Introduction

Key Exchange (KE) is a cryptographic building block that enables two parties to negotiate a shared key securely over an insecure channel. KE protocols are widely used in secure communication protocols, including IPSec, SSH, SSL/TLS, etc., where two parties aim to share a key to securely communicate with each other. The resulted shared key may further be used to provide some cryptographic goals such as authentication, integrity, and confidentiality \[10\]. For instance, the shared key could be used in symmetric cryptographic algorithms such as AES, 3DES, etc., (which are embedded in the mentioned protocols, i.e., IPSec, SSH etc.,) to provide confidentiality \[36\]. Authenticated key exchange (AKE) schemes are a wide selection of KE protocols in which a user and server authenticate each other and establish a session key that is used for transmitting data securely \[10,36\]. The Needham-Schroeder protocol is one of the AKE schemes designed in the earliest publications \[29\]. TLS handshake is also the most widely-used protocol on the Internet, which is an example of AKE protocol \[5,29\]. In \[9\], Blake-Wilson, Johnson, and Menezes have identified several major security attributes of an AKE protocol as below.
known session keys: If an adversary learns the previous session keys, s/he will not compromise the security of the protocol.

(perfect) forward secrecy (PFS): PFS attribute prevents an adversary who has obtained the current long-term secret key from learning the previous session keys.

unknown key-share (UKS): Considering this attack, the party A believes that he has shared a key K with the party B, while the party B mistakenly believes that he has shared the key K with another party C; see also [10].

key-compromise impersonation: Let’s assume that the adversary ξ has compromised the party A’s long-term key (K). It is trivial that ξ can easily impersonate the party A to any other parties using the key K. However, it is not sensible that the adversary ξ masquerades as another party, say B, to establish a valid session with the corrupted party A; see also [33].

loss of information: Comprising any information that is usually not available to an adversary should not impact the security of the protocol.

message independence: The flows of a protocol run between two honest parties need to be independent of each other per session.

AKM protocols could be designed in various settings, including a two-party, three-party, asymmetric, symmetric, group key exchange, and password-based settings. The work [3], for example, made use of two-party and symmetric settings, while the research paper [2] used three-party and symmetric settings. Several studies attempt to achieve PFS among the attributes mentioned above. Researchers tend to take advantage of Diffie-Hellman key agreement (DHKA) schemes to provide PFS, e.g., [16,17,34,35]; however, these protocols are not suitable for use in lightweight applications. In fact, DHKA schemes use two heavy modular exponentiation operations (or elliptic curve operations), which makes it too heavy for resource-limited devices such as the Internet of Things (IoT), the Industrial Internet of Things (IIoT), the Internet of Medical Things (IoMT), etc. Hence, a few proposed schemes [1,3,8,13,19,32] aim at achieving PFS by employing symmetric key settings in which long-term keys (from which the session keys are derived) are modified regularly [19]. Such schemes based on the regular update of the long-term keys are Key-Evolving Schemes [19,20].

In [19] the authors state that the symmetric key based AKE protocols are designed in ad hoc fashion, and such protocols are susceptible to several attacks such as desynchronization. Therefore, design of symmetric key based protocols in a systematic way to ensure the security and privacy issues is crucial. Recently, Avoine, Canard, and Ferreira presented two symmetric key based AKE schemes, namely, SAKE and SAKE-AM, to achieve PFS in [3]. Unlike most of the previous ad hoc designs, SAKE and SAKE-AM are designed in a systematic manner.

In this paper we demonstrate that although the SAKE and SAKE-AM offer several key security features, these schemes suffer from several drawbacks and weaknesses. Finally, we elaborate on our proposed symmetric-based AKE protocols and discuss how our schemes overcome the drawbacks of SAKE and SAKE-AM.
Contributions. In this paper we take a closer look at SAKE and SAKE-AM, observe a few weaknesses of them, and propose fixes. Our specific contributions are as follows.

- We demonstrate that although SAKE and SAKE-AM offer several key security features, these schemes are vulnerable to timeful, replay, DoS, and tracking attacks.
- We propose SAKE+, which not only does inherit the merits of the SAKE, including the PFS and mutual authentication but also mitigates the weaknesses of SAKE.
- We present SAKE+-AM which overcomes the SAKE-AM’s vulnerabilities.
- Our proposed scheme SAKE+ supports concurrent/parallel protocol execution, while SAKE only allows sequential execution. The concurrency of our scheme makes it more appropriate for IoT applications.
- In addition to formally proving security, we demonstrate the properties, such as, soundness, secrecy, and mutual authentication, of our schemes using ProVerif [12].

1.1 Related work

Here we discuss the related work concerning the symmetric key based protocols that ensure PFS. Various research studies have used different terms for PFS [3,19], including forward security [8,13,19,32] and forward secrecy [3,13]. In [3], the authors claim that symmetric key based protocols generally do not ensure security as much as asymmetric key-based protocols do. Particularly, they do not ensure forward secrecy. So far, a few research studies have been conducted with regard to symmetric key based protocols which are discussed in the following. Authors of [19] introduced a definition and model for AKE protocols. They presented an algorithm for their model/definition that can be used by automatic verification tools. They proved that their protocol named FORSAKES is secure within the model. FORSAKES is unconditionally secure in the random oracle model. The notion of FORSAKES is based on the key-evolving scheme in which the long-term keys of the protocol get updated. The proposed scheme assumes a universal clock that is shared between the parties; however, making such an assumption is strong given that achieving perfect time synchronization, in any context, is complicated in practice. Additionally, the presented scheme has not considered an adversary with the capability to desynchronize the long-term keys between two parties [3]. The authors left the side channel attack resistance of their protocol as a future work. The presented scheme in [8] made use of pseudo-random bit generators as the main cryptographic building block to ensure forward security. A pseudo-random generator is secure if its output is computationally indistinguishable from a random string with the same size. Their scheme provides forward-secure symmetric encryption and forward-secure message authentication. Forward-secure message authentication prevents an adversary, who obtained the key used for message authentication, from compromising future uses of the key and from making the previously authenticated data untrustworthy. However, their scheme has not considered the issue of de-synchronization between two parties [3]. The work [32] extended a previous framework for RFID protocols that supports anonymity, authenticity, and availability. This extension attempts to provide forward security when
the long-term keys are compromised by an adversary. The new proposed protocols benefited from a pseudo-random bit generator to address the resource-constraint issues by providing a lightweight mechanism. The schemes have been designed in the Universal Composability framework [15]. To ensure forward security, they assumed that the adversary is able to compromise the activated tag, getting access to keys and memory parameters that are persistent. On the contrary, the server is assumed to be incorruptible. Considering the strong security model where the adversary is capable of corrupting either a tag or a server, the proposed scheme is inherently insecure [3]. The proposed scheme also did not consider attacks that exploit side-channel vulnerabilities of the tags. Authors of [1] posed questions regarding “re-keying” and attempted to answer them. Forward security is one of the benefits gained by re-keying. The authors separated the re-keying process that is dealt with producing sub-keys from the associated application which makes use of these sub-keys. They analyzed different methods of re-keying and their respective applications and demonstrated that re-keying increases the life-time of the master key. However, they did not consider the de-synchronization issue that arises from key-evolving [3]. Recently, authors of [3] proposed a symmetric key based AKE protocol named SAKE that provides PFS for resource-constrained devices. In this protocol, first, an initiator and a server attempt to authenticate each other mutually, and afterward, a session key will be established between them. This protocol makes use of the symmetric-key setting to ensure PFS, whereas many existing schemes use the asymmetric-key setting to do so. However, the resource-constrained devices such as IoT, WSNs, passive RFID tags, and smart cards cannot exploit such schemes since they are too heavy in terms of computation. The authors made several major contributions which makes their scheme distinguishable from their counterparts. In the case of de-synchronization, the parties involved in the protocol can be resynchronized without using a clock or any extra resynchronizing mechanism. The protocol, in fact, benefits from the second chain of master keys to provide synchronization. These master keys provide the tracking the internal state of the protocol and allow resynchronization in case the parties are desynchronized. Moreover, this scheme uses bandwidth efficiently in the sense that it prevents from sending additional data such as large counters. The authors also presented a second scheme called SAKE-AM which is derived from the SAKE protocol. In SAKE-AM, a resource constrained device can be an initiator or a responder. In this case, the end-device does less computation compared to the server.

Relevance. The authors of [3] made use of the security model which is based on the model presented in [14]. This security model [14] is considered for the authenticated key exchange protocols that work using the asymmetric methods (e.g., protocols designed based on the DH scheme with signature). Considering this model, the parties can participate in multiple executions of the protocol either sequentially or concurrently/parallelly; however, the designers of the SAKE in [3] presented a security model that does not support the concurrent/parallel executions. On the other hand, the authors of [3] claim that their scheme is suitable for IoT application, while their presented model conflicts with the way IoT devices act in the real world environment. For exam-
ple, the research study [32] - discussed in the related work section of the reference [3] - states that almost all the RFID systems operate in a concurrent environment. The article [32], more precisely, discusses that a tag reader in a commercialized RFID system should be able to simultaneously identify several tag devices. The work [28] explains that in an IoT application, hundred or thousands of IoT devices send simultaneously their collected data every few seconds to a server. Hence, in our scheme SAKE+, we are motivated by modifying the SAKE protocol such that the parties can take part in concurrent executions of the protocol. This modification not only obviates the need for an exhaustive search which is time-consuming and process-intensive, but also it prevents from the timeful attack. Furthermore, we modify the SAKE-AM scheme to make it resistant against the DoS, replay and traceability attacks.

Outline. We discuss the security of building blocks used in this paper and also list some notation used in the paper’s body in Section 2. In Section 3, we explain the security model used in our proposed protocol. We provide a brief description of the SAKE and SAKE-AM protocols and their security analysis in Sections 4 and 5, respectively. In Section 6 we present our proposed SAKE+ and SAKE+-AM protocols. We evaluate the soundness and security of the proposed SAKE+ protocol in Sections 7 and 8, respectively. Finally, we conclude the paper and discuss future work in Section 9.

2 Security definitions of building blocks

In this section we review the security definition of the building blocks used in this paper. Our definition is based on the research paper [3]. We will further make use of the notations explained in this section in our results. The definition of a negligible function, secure pseudo-random function (PRF), strong unforgeability under chosen-message attacks (SUF-CMA), and matching conversations are discussed as follows:

**Definition 1 (Negligible function):** A function $\epsilon : \mathbb{N} \rightarrow \mathbb{N}$ is said to be a negligible function of $k$ if for every positive polynomial $p(.)$ we have $\epsilon(k) < \frac{1}{p(k)}$ for all sufficiently large $k$ which is a security parameter of a cryptographic building block.

**PRF:** A PRF $F : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^\gamma$, where $\lambda, \gamma$ are positive integers, is a family of functions that takes one input of length $\lambda$ and another input of arbitrary length, and returns an output of length $\gamma$. For any $k \xleftarrow{} \{0,1\}^\lambda$, one can define $f_k : \{0,1\}^* \rightarrow \{0,1\}^\gamma$ by $f_k(x) = f(k, x)$. Then, $f_k$ is an instance of $F$.

We describe the security of $F$ using an experiment between an Adversary $A$ and a challenger below. Note that $x \xleftarrow{} X$ denotes sampling $x$ uniformly at random from $X$.  

- Let $f_k$ is an instance of the PRF family function $F$, and the challenger samples the following values uniformly at random: $K \xleftarrow{} \{0,1\}^\lambda$, $G \xleftarrow{} F$, and $b \xleftarrow{} \{0,1\}$;  
- The adversary adaptively sends the values $x$ to the oracles $OPRF(\cdot)$ and $O-Test(\cdot)$. The responses are as follows: the oracle $O-Test(\cdot)$ either returns $y = f(k, x)$ for $b = 0$, or returns $y = G(x)$ for $b = 1$; the oracle $OPRF(\cdot)$ returns $f(k, x)$ if $x \notin X$;  
- Finally, the adversary makes a guess $b' \in \{0,1\}$ of $b$.  

5
The experiment between the adversary and the challenger described above is summarized as the following pseudo codes.

<table>
<thead>
<tr>
<th>EXP PRF:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K \overset{$}{\leftarrow} \text{KeyGen}(1^\lambda)$</td>
</tr>
<tr>
<td>$b \overset{$}{\leftarrow} {0, 1}$</td>
</tr>
<tr>
<td>$X = \emptyset; c \leftarrow 0$</td>
</tr>
<tr>
<td>$y = \mathcal{A}^{\text{O-CMc2}(), \text{O-PRF}()}$</td>
</tr>
<tr>
<td>return $b = b'$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>O-PRF(x):</th>
</tr>
</thead>
<tbody>
<tr>
<td>if $x \in X$:</td>
</tr>
<tr>
<td>return ⊥</td>
</tr>
<tr>
<td>else:</td>
</tr>
<tr>
<td>$X \leftarrow X \cup {x}$</td>
</tr>
<tr>
<td>return $f(k, x)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>O-Test(x):</th>
</tr>
</thead>
<tbody>
<tr>
<td>if $(c = 1) \lor (x \in X)$:</td>
</tr>
<tr>
<td>return ⊥</td>
</tr>
<tr>
<td>$c \leftarrow 1; X \leftarrow X \cup {x}$</td>
</tr>
<tr>
<td>if $(b = 0)$:</td>
</tr>
<tr>
<td>$y \leftarrow f(k, x)$</td>
</tr>
<tr>
<td>else:</td>
</tr>
<tr>
<td>$y \overset{$}{\leftarrow} \mathcal{G}(x)$</td>
</tr>
<tr>
<td>return $y$</td>
</tr>
</tbody>
</table>

Based on the experiment EXP PRF, we have the advantage of $\mathcal{A}$ as

$$\text{adv}_{\mathcal{A}}^{\text{PRF}} = |\Pr[b = b'] - \frac{1}{2}|.$$  

**Definition 2 (Secure PRF)** If for all probabilistic polynomial time (PPT) adversary $\mathcal{A}$, $\text{adv}_{\mathcal{A}}^{\text{PRF}}(\mathcal{A})$ is a negligible in $\lambda$, then $f_k : \{0, 1\}^* \rightarrow \{0, 1\}^\gamma$ is a secure PRF.

**Secure Message authentication code (MAC):** It consists of three different algorithms named KeyGen, MAC$(\cdot, \cdot)$ and $\text{Vrf}((\cdot, \cdot), \cdot)$ [22]. These algorithms are described as follows:

- **KeyGen**: This randomized algorithm generates a $k$–bit key $K$ that is used as a parameter in the algorithms MAC$(\cdot, \cdot)$ and Vrf$(\cdot, \cdot, \cdot)$.
- **MAC$(\cdot, \cdot)$**: This algorithm (tagging) takes as input a key $K \in \{0, 1\}^k$ and a message $m \in \{0, 1\}^\gamma$ and returns an output named tag $\tau \in \{0, 1\}^\gamma$.
- **Vrf$(\cdot, \cdot, \cdot)$**: This verification algorithm takes as input a key $K$, a message $m$, and a corresponding tag $\tau$, and outputs $1$ if a tag on message $m$ is valid, otherwise $0$.

**Strong unforgeability under chosen-message attacks (SUF-CMA)**

We define the notion of SUF-CMA for a MAC $G = (\text{KeyGen}, \text{MAC}(\cdot, \cdot), \text{Vrf}(\cdot, \cdot, \cdot))$ with the help of the experiment between a challenger and an adversary $\mathcal{A}$ as follows:

- The challenger sets $S \leftarrow \emptyset$ and then samples $K \overset{\$}{\leftarrow} \{0, 1\}^k$.
- The adversary may send values $m$ to the challenger, and it sends back the respective values $\tau = \text{MAC}(K, m)$ and then saves $(m, \tau) : S \leftarrow S \cup \{(m, \tau)\}$. The adversary, additionally, may send the values $(m', \tau')$ to the challenger, and it returns the respective values $\text{Vrf}(K, m', \tau')$ to the challenger.
- At the end, $\mathcal{A}$ sends to the challenger the pair $(m^*, \tau^*)$.

The experiment between the adversary and the challenger described above is summarized as the following pseudo codes.

<table>
<thead>
<tr>
<th>EXP SUF-CMA:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K \overset{$}{\leftarrow} \text{KeyGen}(1^\lambda)$</td>
</tr>
<tr>
<td>$S = \emptyset$</td>
</tr>
<tr>
<td>$(m^<em>, \tau^</em>) \leftarrow \mathcal{A}^{\text{MAC}(\cdot, \cdot), \text{Vrf}(\cdot, \cdot, \cdot)}$</td>
</tr>
<tr>
<td>return $(m^<em>, \tau^</em>)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MAC(K, m):</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau = \text{MAC}(K, m)$</td>
</tr>
<tr>
<td>saves $(m, \tau)$</td>
</tr>
<tr>
<td>$S \leftarrow S \cup {(m, \tau)}$</td>
</tr>
<tr>
<td>return MAC(K, m)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vrf(K, m', \tau')</th>
</tr>
</thead>
<tbody>
<tr>
<td>if $(m', \tau') \notin S$:</td>
</tr>
<tr>
<td>return $\text{Vrf}(K, m', \tau')$</td>
</tr>
<tr>
<td>else:</td>
</tr>
<tr>
<td>return ⊥</td>
</tr>
</tbody>
</table>

The advantage of $\mathcal{A}$ is as $\text{adv}_{\mathcal{A}}^{\text{SUf-CMA}}(\mathcal{A}) = \Pr[\text{Vrf}(K, m^*, \tau^*) = 1 \land (m^*, \tau^*) \notin S]$.  

6
Definition 3 (SUF-CMA): If for all PPT adversary $A$, $adv^{suf-cma}_G(A)$ is a negligible function in $k$, then $G = (\text{KeyGen}, \text{MAC}(\cdot, \cdot), Vrf(\cdot, \cdot, \cdot))$ with $\text{MAC} : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^\gamma$ is secure against SUF-CMA.

Sessions and Instances: Before explaining the matching conversation, we discuss the session as follows. A session refers to one run of the protocol, and each party can create at most $q$ sessions. We associate an instance $\pi_i$ to the party $P_i$’s $s$th session, which has access to the master keys $K$ and $K'$.

Definition 4 (Matching Conversations): The authors of [4] discussed the definition of matching conversations proposed by [6, 24]. We assume that $T_{i,s}$ is the sequence of all messages in chronological order which are sent and received by an instance $\pi_i$. Considering two transcripts $T_{j,t}$ and $T_{i,s}$, if $T_{i,s}$ includes one message at least, and the messages in $T_{i,s}$ are the same as the first $|T_{i,s}|$ messages of $T_{j,t}$ then we say that $T_{i,s}$ is a prefix of $T_{j,t}$. The instance $\pi_i$ has a matching conversation to $\pi_j$, if $|T_{j,t}|$ is a prefix of $T_{i,s}$, and $\pi_i$ has sent all protocol messages, or $|T_{i,s} = T_{j,t}|$, and $\pi_j$ has sent all protocol messages.

2.1 Preliminaries

In this section the notations (Table 1) and preliminaries used in the paper are described.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A, B$</td>
<td>Identities of an initiator and a responder, respectively.</td>
</tr>
<tr>
<td>$K$</td>
<td>The master key used for authentication purpose.</td>
</tr>
<tr>
<td>$K'$</td>
<td>The master key used for session key generation.</td>
</tr>
<tr>
<td>$K_{j-1}^i$, $K_j^i$, $K_{j+1}^i$</td>
<td>These values refer to the protocol states $j-1$, $j$, and $j+1$, respectively.</td>
</tr>
<tr>
<td>$r_A, r_B$</td>
<td>Random numbers generated by the entities $A$ and $B$, respectively.</td>
</tr>
<tr>
<td>$kdf(\cdot, \cdot)$</td>
<td>This function updates the session key $sk$ such that $sk \leftarrow kdf(K, g(r_A, r_B))$ the function $g(\cdot)$ can be the bit string concatenation.</td>
</tr>
<tr>
<td>$\text{upd}_i(\cdot)$, $\text{upd}_i^*(\cdot)$</td>
<td>These functions are used for updating the master keys of entity $i$.</td>
</tr>
<tr>
<td>$Vrf(K, m, \tau)$</td>
<td>This function returns true if the tag $\tau$ on message $m$ is true, otherwise it returns false.</td>
</tr>
<tr>
<td>$| |$</td>
<td>Concatenation operation</td>
</tr>
</tbody>
</table>

These master keys $K$, $K'$, $K_{j-1}^i$, $K_j^i$, and $K_{j+1}^i$ are initialized such that $K$ and $K'$ are random values, $K_{j-1}^i \leftarrow \perp$, $K_j^i \leftarrow K'$, and $K_{j+1}^i \leftarrow \text{update}(K')$.

The initiator $A$ and the responder $B$ update their master keys using, respectively, $\text{upd}_A(\cdot)$ and $\text{upd}_B(\cdot)$ functions as follows:

<table>
<thead>
<tr>
<th>upd_A(·)</th>
<th>upd_B(·)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K \leftarrow \text{update}(K)$</td>
<td>$K' \leftarrow \text{update}(K')$</td>
</tr>
<tr>
<td>$K_{j-1}^i \leftarrow K_j^i$</td>
<td>$K_j^i \leftarrow K_{j+1}^i$</td>
</tr>
<tr>
<td>$K_{j+1}^i \leftarrow \text{update}(K_{j+1}^i)$</td>
<td>$K_{j+1}^i \leftarrow \text{update}(K_{j+1}^i)$</td>
</tr>
</tbody>
</table>

Session Key generation: It is based on the key-evolving method. Using this method, both the initiator and responder update the master key $K$ regularly per session. This protects against computing the past session keys by an adversary who has
corrupted either the initiator or responder, and gained the current session key \( sk \). Considering the session key generation process, the key point is that the session key must be computed after the mutual authentication, not during the session. In fact, if the session key \( sk \) is computed in the session, this can cause a de-synchronization problem.

### 3 Security model

We profit from the model explained in [3] which itself is based on the work [14]. The adversary in this model has a full control over the channel in that s/he is capable of forwarding, altering, dropping any messages sent by honest parties, or even s/he is able to create new messages. It is worth noting that we adopted the concurrent execution approach which is discussed in [14]. We later discuss in detail the concurrent approach versus of sequential approach in the Section 3.2.

**Parties:** We assume that all the parties in our protocol constitute the set \( P = \{ P_0, P_1, \ldots, P_{n-1} \} \). Each party has its own unique master keys \( K, K' \). The key \( K \) is used for the purpose of session key creation, while the key \( K' \) is used for the authentication purpose. Each party in our scheme can take part in multiple concurrent executions of the protocol. On the contrary, a party in SAKE [3] can participate in multiple sequential executions of the protocol. Each instance \( \pi_s^i \) comprises the following states:

Table 2. The States of an Instance

<table>
<thead>
<tr>
<th>States Description</th>
<th>States Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \rho )</td>
<td>This state denotes the role of the session that belongs to the set ( \rho \subset { \text{init, resp} } ). The init and resp values refer to the roles initiator and responder, respectively.</td>
</tr>
<tr>
<td>( \text{pid} )</td>
<td>This identity is associated with the intended communication partner of ( \pi_s^i ), and ( \text{pid} \in P ).</td>
</tr>
<tr>
<td>( \alpha )</td>
<td>It shows the state of the instance which could be one of the following elements ( { \bot, \text{running, accepted, rejected} } ).</td>
</tr>
<tr>
<td>( \text{sk} )</td>
<td>It denotes the session key that is derived by ( \pi_s^i ).</td>
</tr>
<tr>
<td>( \kappa )</td>
<td>It indicates the status of the session key ( \pi_s^i \cdot \text{sk} ), and ( \kappa \in { \bot, \text{revealed} } ).</td>
</tr>
<tr>
<td>( \text{sid} )</td>
<td>It refers to the the identifier of the session.</td>
</tr>
<tr>
<td>( b )</td>
<td>It indicates a random bit ( b \in { 0, 1 } ) that is sampled at initialization of ( \pi_s^i ).</td>
</tr>
</tbody>
</table>

Below we define two correctness requirements with the help of variables \( \alpha, \text{sk}, \text{sid}, \) and \( \text{pid} \) as follows:

\[
(\pi_s^i \cdot \alpha = \text{accepted}) \Rightarrow (\pi_s^i \cdot \text{sk} \neq \bot \land \pi_s^i \cdot \text{sid} \neq \bot) \tag{1}
\]

\[
(\pi_s^i \cdot \alpha = \pi_j^i \cdot \alpha = \text{accepted} \land \pi_s^i \cdot \text{sid} = \pi_j^i \cdot \text{sid}) \Rightarrow \begin{cases} 
\pi_s^i \cdot \text{sk} = \pi_j^i \cdot \text{sk} \\
\pi_s^i \cdot \text{pid} = P_j \\
\pi_j^i \cdot \text{pid} = P_i 
\end{cases} \tag{2}
\]

**Adversarial Queries:** The adversary interacts with the instances by means of the following queries:
\begin{itemize}
  \item \textbf{NewSession}(P_i, \rho, pid): This query creates an instance \( \pi_i^s \) with the role \( \rho \) and the intended partner \( pid \).
  \item \textbf{Send}(\( \pi_i^s \), \( m \)): Using this query, the adversary can send a message \( m \) to the instance \( \pi_i^s \) and the instance responds as follows: it returns \( \perp \), if \( \pi_i^s \cdot \alpha \neq \text{running} \); otherwise, \( \pi_i^s \) responds according to the protocol specification.
  \item \textbf{Corrupt}(P_i): The adversary, using this query, can obtain the long-term key \( P_i \cdot \ltk \) of \( P_i \). We say that \( P_i \) is \( v \)-corrupted if \( \text{Corrupt}(P_i) \) is the \( v \)-th query sent by the adversary. If \( v = +\infty \), it means that the party has not been corrupted.
  \item \textbf{Reveal}(\( \pi_i^s \)): It returns the session key \( \pi_i^s \cdot sk \), and then the value \text{revealed} will be assigned to \( \pi_i^s \cdot \kappa \).
  \item \textbf{Test}(\( \pi_i^s \)): Through the game, this query can be called only once. This query returns \( \perp \) if \( \pi_i^s \cdot \alpha \neq \text{accepted} \). Otherwise, it creates an independent key \( sk_0 \leftarrow \kappa \), and returns \( sk_0 \) (Test-challenge), where \( sk_1 = \pi_i^s \cdot sk \).
\end{itemize}

\textbf{Definition 5 (Partnership)}: If \( \pi_i^s \cdot sid = \pi_j^t \cdot sid \), we say that these two instances are partners.

\textbf{Definition 6 (Freshness)}: An instance \( \pi_i^s \) is fresh with intended partner \( P_j \), if
\begin{itemize}
  \item \( \pi_i^s \cdot \alpha = \text{accepted} \) and \( \pi_i^s \cdot pid = P_j \) when the adversary sends its \( v_0 \)-th query,
  \item \( \pi_i^s \cdot \kappa \neq \text{revealed} \) and \( P_i \) is uncorrupted (resp. \( v \)-corrupted with \( v_0 < v \)), and
  \item for any partner instance \( \pi_j^t \) of \( \pi_i^s \), we have that \( \pi_j^t \cdot \kappa \neq \text{revealed} \) and \( P_j \) is \( v' \)-corrupted with \( v_0 < v' \).
\end{itemize}

An AKE protocol satisfies the two aforementioned correctness requirements (1) and (2), and its security is defined using the AKE experiment that is played between a challenger and an adversary \( \mathcal{A} \). Following definitions 7 and 8, \( \mathcal{A} \) can win this experiment.

\textbf{Definition 7 (Entity Authentication (EA))}: The adversary can win the AKE experiment by making an instance accepts maliciously. An instance \( \pi_i^s \) of a protocol \( \Pi \) accepts maliciously with intended partner \( P_j \), if
\begin{itemize}
  \item \( \pi_i^s \cdot \alpha = \text{accepted} \) and \( \pi_i^s \cdot pid = P_j \) when the adversary \( \mathcal{A} \) sends its \( v_0 \)-th query,
  \item \( P_i \) and \( P_j \) are uncorrupted (resp. \( v \)- and \( v' \)-corrupted with \( v_0 < v, v' \)), and
  \item there is no unique instance \( \pi_j^t \) such that \( \pi_i^s \) and \( \pi_j^t \) are partners.
\end{itemize}

The adversary’s advantage is defined as \( \text{adv}^\text{ent-auth}_\Pi(\mathcal{A}) = \Pr[\mathcal{A} \text{ wins the EA game}] \).

\textbf{Definition 8 (Key Indistinguishability)}: An adversary \( \mathcal{A} \) can win the AKE experiment by guessing the secret bit of the Test-instance. The adversary \( \mathcal{A} \) sends the Test-query to the instance \( \pi_i^s \) during this experiment and answers the Test-challenge correctly if the output \( b' \) and the instance \( \pi_i^s \) are as follows:
\begin{itemize}
  \item \( \pi_i^s \) is fresh with some intended partner \( P_j \), and
  \item \( \pi_i^s \cdot b = b' \).
\end{itemize}

The adversary’s advantage is defined as \( \text{adv}^\text{key-ind}_\Pi(\mathcal{A}) = \Pr[\pi_i^s \cdot b = b'] - \frac{1}{2} \).

It is worth mentioning that an adversary can make use of the definitions 7 and 8 in order to corrupt an instance involved in the experiment.
Definition 9 (AKE Security): A two-party AKE protocol $\Pi$ is secure if it meets the correctness requirements (1) and (2), and the advantages $\text{adv}_{\Pi}^{\text{ent-auth}}(A)$ and $\text{adv}_{\Pi}^{\text{key-ind}}(A)$ are negligible.

3.1 Data search operation

Our proposed protocols SAKE and SAKE-AM benefit from a technique named content-addressable memory (CAM, associative memory or associative storage) [27]. The merit of this technique is that we can find the desired data in a CAM memory just in a single clock cycle compared to traditional memories. The function $\text{fetch}(\cdot)$ used in our schemes makes use of this technique. In fact, the initiator uses the $\text{fetch}(\cdot)$ to search the digest of a responder’s identity against all the digests stored in the database and obtain the corresponding address. This search takes constant time; that is, with the complexity $O(1)$ which makes our protocol independent of the number of the responders involved. By means of this method, not only the initiator can search within a constant time and independent of number of end devices involved in the protocol, but also this technique prevents from the timeful attack in that the server spends the same amount of time to look for all responders’ data and authentication at every execution of the protocol (see Section 5 for details).

3.2 Concurrent vs. sequential

Let $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ denote a set of parties participating in a two-party protocol. In a sequential approach, two parties, e.g., $P_1$ and $P_2$ each one executes just one instance of a protocol, while in a concurrent approach the mentioned parties run parallel executions of the protocol [3][14]. The AKE protocols that operate based on the DH scheme support unlimited number of concurrent executions, whereas the AKE symmetric-based protocols do not allow parallel executions of the protocol in the sense that the shared master keys that evolves regularly cause abortion problem to some sessions [3]. However, we overcome this problem using separate master keys associated with parties’ identity.

4 Description of the SAKE and the SAKE-AM protocols

In this section, a brief review of the SAKE and its variant SAKE-AM is presented. In these schemes, with the help of two type of pre-shared master keys $K$ and $K'$, both parties, the end device and back end device, not only mutually authenticate each other but also establish a session key to support confidentiality. More precisely, the key $K'$ along with two other pseudo-random values $r_A$, and $r_B$ are used in $\text{MAC}(\cdot, \cdot)$ as $\text{MAC}(K', B || A || r_B || r_A)$ (on the responder’s side) and $\text{MAC}(K', A || B || r_A || r_B)$ (on the initiator’s side) for mutual authentication purpose, whereas the master key $K$ together with $r_A$, and $r_B$ values are utilized in the pseudo-random generator $\text{kdf}(\cdot, \cdot)$ as $\text{kdf}(K, g(r_A, r_B))$ for the session key generation purpose.
\[
(A, K, K'_j + 1, K'_j, K'_{j-1})
\]

**SAKE**

\[
r_A \xleftarrow{\$} \{0,1\}^\lambda
\]

\[
\text{if } (\text{Vrf}(K'_j, B||A||r_A||r_B) = \text{true})
\]

\[
\delta_{AB} \leftarrow 0
\]

\[
K' \leftarrow K'_j; \text{kdf}; \text{upd}_A; \ E \leftarrow 0
\]

\[
\text{else if } (\text{Vrf}(K'_j+1, B||A||r_A||r_B) = \text{true})
\]

\[
\delta_{AB} \leftarrow -1
\]

\[
K' \leftarrow K'_j+1; \text{kdf}; \text{upd}_A; \ E \leftarrow 0
\]

\[
\text{else}
\]

\[
\text{abort}
\]

\[
\tau_A \leftarrow \text{MAC}(K', E||A||B||r_A||r_B)
\]

\[
m_A \leftarrow E||\tau_A
\]

\[
\tau_B \leftarrow \text{MAC}(K', r_B||E||A||B||r_A||r_B)
\]

\[
m_B \leftarrow r_B||r_B
\]

\[
\text{if } (\text{Vrf}(K'_j, B||A||r_A||r_B) = \text{false})
\]

\[
\text{abort}
\]

\[
\text{if } (E = 0)
\]

\[
K' \leftarrow K'_j
\]

\[
\text{if } (\text{Vrf}(K'_j, r_A||r_A, r_A') = \text{false})
\]

\[
\text{abort}
\]

\[
kdf; \text{upd}_A
\]

\[
\tau_A \leftarrow \text{MAC}(K', r_A||r_B)
\]

\[
\text{[Same as the SAKE]}
\]

\[
\text{[Same as the SAKE]}
\]

\[\text{Fig. 1. SAKE and SAKE-AM Protocols.}\]

\[
(B, K, K')
\]

\[
r_B \xleftarrow{\$} \{0,1\}^\lambda
\]

\[
\tau_B \leftarrow \text{MAC}(K', B||A||r_B)
\]

\[
m_B \leftarrow r_B||r_B
\]

\[
\text{[Same as the SAKE]}
\]

\[
\text{[Same as the SAKE]}
\]

\[
\text{[Same as the SAKE]}
\]
4.1 SAKE protocol

As depicted in Figure 1, the SAKE protocol starts with initiator A sending the fresh random value $r_A$ along with its identity $A$. In response, the responder B sends the message $m_B = r_B || \tau_B$ in which $r_B$ is the new random value generated by B and $\tau_B$ generated using the MAC function to inform the A about the current state of the B securely. Using the DRSP (see Section 6.1), A distinguishes in which state the party B is and generates the session key in the case of $\epsilon = 0$. Then, A responds with the message $m_A$ that includes the current state of the party B. At this point, B syncs itself and generates the session key after the verification and informs A by sending the message $\tau'_B$. After receiving the message, A verifies the message and generates the session key in the case $\epsilon = 1$. Then, A sends the confirmation message $\tau'_A$ to B. Finally, B finishes the session if and only if the received message is valid.

4.2 SAKE-AM protocol

The authors of [3], presented a variant of the SAKE scheme named SAKE-AM in which the end device can be an initiator (see Figure 1). Due to this variant, the authors claim that SAKE-AM is suitable for IoT applications where a resource-constrained end device establishes a connection with a central server. The SAKE and the SAKE-AM protocols differ as follows: the initiator A in the SAKE has the master keys $(K, K'_{j+1}, K'_j, K'_{j-1})$, whereas B in the SAKE-AM scheme has those master keys. The initiator A in the SAKE does most of the operation, while the initiator A in the SAKE-AM does the lease calculations.

5 Security analysis of SAKE and SAKE-AM

In this section, we perform security analysis of the SAKE and SAKE-AM protocols.

5.1 Exhaustive search problem

The SAKE protocol suffers from the exhaustive search, which makes it unsuitable for an IoT application, particularly an application with many IoT sensors. In the SAKE protocol, the initiator must perform a complete search in its database upon receiving the message $m_B$ from the responder. This is due to the fact that the responder does not send its identity along with $m_B$, therefore the initiator does not know which end device has sent the $m_B$ message. The initiator, hence, has to check all the conditional if statements (four if statements) against all the different identities exist in the database until it finds the identity associated with the message $m_B$. For example, if there are $n$ number of end devices in a real scenario, the initiator must at least execute $4n/2$ number of if statements on average along with the computation of $\tau_A$ and $m_A$. (In case of $\delta_{AB} = 0$ and $\delta_{AB} = -1$, the PRF function is executed at least twice within the if statements) to find the identity corresponding to the message $m_B$. To address this problem, in our scheme SAKE$^+$ the responder sends its identity, which is random,
along with the message $m_B$ to the initiator. The initiator, then, benefits from an efficient search technique called CAM to find the corresponding responder’s information in its database.

5.2 Timeful attack

The authors of [21, 23] described an attack named timeful. This type of attack can be performed in the AKE protocols in which the server (reader) must perform exhaustive search to find a tag’s related information. The adversary, in fact, can exploit this fact that the same amount of time is always spent by the server to authenticate and accordingly respond to a particular tag for every execution of the protocol. This leads an adversary to detect which tag has been authenticated by the server (reader). The SAKE protocol is vulnerable to this attack as the initiator do exhaustive search to authenticate a responder. In our scheme SAKE$^+$, we profit from random identities to prevent from this attack.

5.3 DoS and replay attacks

An adversary is able to lunch a DoS attack by means of the message $m_B = B||r_B||\tau_B$ sent by the initiator B in the SAKE-AM protocol. To do so, the attacker first capture the valid message $m_B$ related to the last session between the initiator and the responder A. Then, the adversary resends the captured message $m_B$ to A, repeatedly. After receiving the message $m_B$ by A, it will use the $Vrf(\cdot, \cdot, \cdot)$ function to check the validity of the value $\tau_B$ attached to $m_B$. Whatever the master key $K'$ is (it could be either $(K'_j+1$ or $K'_j$ or $K'_j-1$)), the value $\tau_B$ computed by one of the if statements will equal true. This is due to the assumption that the captured message $m_B$ by the adversary is valid. After the verification of $\tau_B$, A will compute $\tau_A = MAC(K', \epsilon||A||B||r_A||r_B)$ and create the message $m_A = \epsilon||\tau_A||r_A$. Now, A sends the created message to the initiator B. Hence, the adversary succeeded in forcing the party A to perform calculations by calling the following functions and eventually creating and sending $m_A$ for each received message $m_B$. If $K' = K'_j$, A computes $(Vrf(\cdot, \cdot, \cdot), kdf(\cdot, \cdot), upd_A(\cdot), MAC(\cdot, \cdot))$, if $K' = K'_j-1$, A computes $(Vrf(\cdot, \cdot, \cdot), MAC(\cdot, \cdot))$, and If $K' = K'_j+1$, A computes $(Vrf(\cdot, \cdot, \cdot), kdf(\cdot, \cdot), upd_A(\cdot), MAC(\cdot, \cdot))$.

It is worth mentioning that the described attack includes the replay attack as the adversary can replay the message $m_B$, which belongs to the last valid session, and accordingly receive $m_A$ as a response from A. This scenario holds until the initiator establishes a new session with the responder. Later we propose SAKE$^+$-AM which is resilient against these attacks with the help of a random value and performing a comparison on the responder’s side.

5.4 Traceability (tracking) attack

Considering the SAKE-AM protocol, an adversary can easily relate all the messages $m_B$ that has captured through the valid sessions between B and A. This attack is based
on the fact that the initiator B attaches its identity, B, to $m_B$ in plaintext; hence, the adversary is able to eavesdrop this identity and thus tracking all the eavesdropped messages $m_B$. This traceability can be done using a simple comparison among the identities attached to $m_B$ messages. Based on the privacy model presented in [26], this attack formally captured by the following queries. The untraceability (UNT) is the fact that an adversary $A$ cannot tell between two responders.

- $\text{Execute}(A, B, j)$ query. The adversary $A$ requests for access to the messages of the $j$-the session between $A$ and $B$.
- $\text{Test}(B_0, B_1, j + 1)$ query. For a random $b \in \{0, 1\}$, $A$ is challenged by the messages exchanged between $B_b$ and $A$ in their $j + 1$-th session, and has to guess $b$.

Following the above queries, $A$ performs the traceability attack as follows.

- In round $j$, $A$ sends an $\text{Execute}(A, B_0, j)$ query and obtains $B_{0,j}$;
- The attacker $A$ selects two responders $B_0$ and $B_1$, sends a $\text{Test}(B_1, B_0, j + 1)$ query, and obtains $B_b$, where $b \in \{0, 1\}$;
- Then, $A$ sends an $\text{Execute}(A, B_b, j + 1)$ query and obtains $B_{b,j+1}$;
- $A$ learns that $b = 0$ if $B_{0,j} = B_{b,j+1}$, otherwise $b = 1$;
- Since the value of $B$ in the message $m_B$ is constant (i.e., the responder’s identity is fixed), $B_{b,j+1} = B_{0,j}$ implies that $B_b = B_0$.

As a result, $Adv_A^{\text{UNT}}(k) = (\Pr[A \text{ guesses } b \text{ correctly}] - 1/2) = 1 - 1/2 = 1/2$. In our scheme SAKE$^+$-AM, to mitigate this attack, the parties’ identities are updated per session.

6 Description of our proposed protocols SAKE$^+$ and SAKE$^+$-AM

This section presents our enhanced protocols SAKE$^+$ and SAKE$^+$-AM, addressing the aforementioned security issues associated with SAKE and SAKE-AM. We begin by describing the intended security properties of our improved protocols.

6.1 Intended properties of our protocols

To enhance the SAKE protocol the (PFSP), (SP) and (DRSP) solutions are inherited from SAKE and the rest are proposed in the current article.

**Perfect forward secrecy property (PFSP):** Considering that the SAKE scheme guarantees the synchronization of the master key $K$ and benefits from the key-evolving technique, it ensures the perfect forward secrecy, which is the goal of SAKE$^+$ protocol.

**Synchronization property (SS):** The SAKE$^+$ scheme takes advantage of $K'$ used for mutual authentication to prevent from synchronization problem. The initiator makes use of three keys, namely $\text{update}(K'_j), K'_j, K'_{j-1}$ to track the session key. This tracking is guaranteed by ensuring that the master keys, $K$ and $K'$, will be updated simultaneously. It is the initiator, in fact, that makes the responder (with the help of $\epsilon$) how to behave. This behaviour is discussed as follows: if $\epsilon = 0$, it means that the
initiator and responder are synchronized, so B will update its session key $sk$ along with its master keys $(K, K')$ (using $udp_B^*(\cdot)$), and if $\epsilon = 1$, it means that the initiator and responder are not synchronized and B must first update its master keys $(K, K')$ and then update its session key $sk$ together with the master keys $(K, K')$ for the second time.

**Distinguishing the responder’s state property (DRSP):** The initiator (A) uses the message $m_B$, sent by the responder (B), to distinguish in which state the party B is. The value $K'$ which is used in $m_B$ indicates the state of the party B. The party A makes use of the parameters $update(K'_j), K'_j, K'_{j-1}$ and the message $m_B$ to compute the value $\delta_{AB}$. The value $\delta_{AB}$ is interpreted as follows:

- $\delta_{AB} = 0$: This means that A and B both are synchronized, thus updating their session key $sk$ using $kdf(\cdot, \cdot)$ function as well as updating their corresponding master keys with the help of $udp_A^*(\cdot)$ and $udp_B^*(\cdot)$ functions, respectively.
- $\delta_{AB} = 1$: This means that A is one step further, and, therefore, B needs first to resynchronize its master keys $(K, K')$. Then B behaves just the same as the case where $\delta_{AB} = 0$, i.e., to compute the new session key $sk$ and its master keys $(K, K')$.
- $\delta_{AB} = -1$: This means that A is one step behind; hence, A must resynchronize its master keys $(K, update(K'_j), K'_j, K'_{j-1})$. Then, A follows the protocol to update the session key $sk$ and its master keys $(K, update(K'_j), K'_j, K'_{j-1})$.

**Tracking resistance (TR):** The identity of a responder $B$ is updated per session on both initiator and responder sides. This update is intended to prevent tracking attacks. It is noteworthy that our architecture includes several responders communicating concurrently with one initiator. So, employing the fixed identity for the typical responder makes it vulnerable to following tracking attack. An attacker creates a message $A\|r_A$, sends it continually to B, and receives a response containing the responder’s identity ($B$) for each message. By linking the identities in the responses, the attacker can successfully trace the target responder. To overcome this privacy threat, the responder B saves a one-bit flag $\varphi$ in it’s memory. This flag is used to prevent the tracking attack. Let us say that in SAKE$^+$, the user has set the flag $\varphi$ to one at the beginning of the protocol, and the value of the flag has reset to zero (after updating the identity $B$), then, the responder can easily recognize whether the attacker is trying to track:

- $\varphi = 1$: This means that B has received the message $A\|r_A$, responded with the message $m_B$ and is waiting to receive the corresponding message $m_A$.
- $\varphi = 0$: This means that B has received the message $m_A$, verified this message and successfully executed the $udp_B^*(\cdot)$ and $kdf(\cdot, \cdot)$ functions.

**Security Against Replay and DoS (SRD):** The initiator A stores $(K, K'_{j-1}, K'_j, B_{j-1}, B_j, r_{B_{j-1}})$ in which $B_j$ and $B_{j-1}$ are the identity of the responder in a current and a previous sessions, respectively. Using an identity $B$ in the responder’s messages makes it possible for the initiator A to search with $O(1)$ (we refer the reader to the subsection 3.1). The value of the $r_{B_{j-1}}$ is stored on the initiator’s side to protect the protocol against the replay and DoS attacks. The following comparisons are made
by the initiator using the $equal(\cdot, \cdot)$ function. Note that by $equal(a, b)$ or $equal(a, c)$ we mean $equal(a, b)$ or $equal(a, c)$.

- $r_{B_{j-1}} = r_{B_j}$: This means that the attacker is trying to replay the message $m_B$.
- $r_{B_{j-1}} \neq r_{B_j}$: This means that A has received the fresh message $m_B$. In this case, A starts to verify this message.

Moreover, A does not need to store $K'_{j+1}$; it can compute this value using $K'_j$. However, A retains the value $K'_{j}$, although it can compute the value $K_{j}$ using $K'_{j-1}$; in fact, it needs to run $upd^*_A(\cdot)$ function per session, and thus causing the computation overhead. The same logic applies to the B value (i.e., A does not need to store $B_{j+1}$).

It is worth mentioning that in SAKE+, there are some changes with regard to $upd^*_A(\cdot)$ and $upd^*_B(\cdot)$ functions as follows (the new changes are shown in blue):

<table>
<thead>
<tr>
<th>$upd^*_A(\cdot)$:</th>
<th>$upd^*_B(\cdot)$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K \leftarrow update(K)$</td>
<td>$K \leftarrow update(K)$</td>
</tr>
<tr>
<td>$B_{j-1} \leftarrow B_j$</td>
<td>$B \leftarrow update(B</td>
</tr>
<tr>
<td>$B_j \leftarrow update(B_j</td>
<td></td>
</tr>
<tr>
<td>$K'_{j-1} \leftarrow K'_j$</td>
<td></td>
</tr>
<tr>
<td>$K'_j \leftarrow update(K'_j)$</td>
<td></td>
</tr>
</tbody>
</table>

### 6.2 Description of the SAKE+ protocol

In this subsection, we propose our solutions to overcome the drawbacks of the SAKE protocol. To enhance this protocol, we just made several changes illustrated in Figure 2 with blue color. Considering the intended properties discussed in Section 6.1, the protocol runs as below. The initiator A generates a random number $r_A$ and starts the new session by broadcasting a challenge $A||r_A$ to all the responders.\(^3\) Upon receiving the message, the responder B generates a random number $r_B$ and computes $\tau_B$. At this point, the responder runs the $equal(\cdot, \cdot)$ function.\(^4\) In case of $\varphi$ equal to 0, the responder learns that the protocol is well done in the last session and computes $m_B$ as $B||r_B||\tau_B$ and sets $\varphi = 1$. Conversely, if the value of the bit $\varphi$ equals to 1, the responder concludes that the attacker may be trying to execute the tracking attack. Then, it generates the random number $r_\alpha$ and computes $m_B$ as $r_\alpha||r_B||\tau_B$. Finally, the responder sends the message $m_B$ to the initiator. Once the initiator receives the message $m_B$, it runs the $fetch(\cdot)$ function to obtain the corresponding values related to one of the identities $B_{j-1}$, $B_j$ or $update(B||K'_j)$ (i.e., $(K, K'_{j-1}, K'_j, r_{B_{j-1}})$). If none of the values $B_{j-1}$, $B_j$, or $update(B||K'_j)$ are present in the memory, A will find the responder’s identity with the help of the if–else statements. Then, A runs the $equal(\cdot, \cdot)$ function for the received value $r_B$ and the fetched value $r_{B_{j-1}}$. If it is not $true$, A will run the next three if–else conditions as the same as the SAKE protocol does to distinguish the responder’s state and to set the $\delta_{AB}$ value. Finally, A replaces the value of $r_{B_{j-1}}$ with the received $r_B$ and sends $m_A$ to the responder. At this point, the responder B authenticates the initiator A.

---

\(^3\) As we discussed in subsection 3.2, the SAKE+ is a concurrent protocol. For the sake of simplification, when explaining a protocol, we only consider one responder.

\(^4\) This function takes two inputs and performs as follows: if the given inputs are equal to each other, it returns $true$, otherwise it returns $false$. 
\[
(A, K, K', K'_{-1}, B_{-1}, B_i, r_{B_{i-1}}) \quad \text{SAKE}^+ \quad (B, K, K', \varphi)
\]

\[
\begin{array}{c}
\begin{aligned}
&A \parallel r_A \\
m_B & \rightarrow
\end{aligned}
\end{array}
\]

\[
\begin{array}{c}
\begin{aligned}
&\delta_{AB} \leftarrow 0 \\
&K' \leftarrow K'_j \text{kdf}; \text{upd}^*_A; E \leftarrow 0 \\
&\text{else if } (\text{Vrf}(K'_j, B||A||r_A, \tau_A) = \text{true}) \\
&\delta_{AB} \leftarrow 1 \\
&K' \leftarrow K'_j; E \leftarrow 1 \\
&\text{else if } (\text{Vrf}(K'_j, B||A||r_A, \tau_A) = \text{true}) \\
&\delta_{AB} \leftarrow -1 \\
&K' \leftarrow K'_{j-1}; \text{upd}^*_A; \text{kdf}; \text{upd}^*_A; E \leftarrow 0 \\
&\text{else}
\end{aligned}
\end{array}
\]

\[
\begin{array}{c}
\begin{aligned}
&\tau_A \leftarrow \text{MAC}(K', E||A||B||r_A||r_B) \\
&m_A \leftarrow E||\tau_A \\
&\text{[Same as the SAKE]}
\end{aligned}
\end{array}
\]

\[
\begin{array}{c}
\begin{aligned}
&\text{[Same as the SAKE]}
\end{aligned}
\end{array}
\]

\[
\begin{array}{c}
\begin{aligned}
&\text{if (equal}(B, B_i \text{ or } B_{i-1} \text{ or update}(B_j||K'_j)) = \text{true}) \\
&B, K'_j, K'_{j-1} \leftarrow \text{fetch}()
\end{aligned}
\end{array}
\]

\[
\begin{array}{c}
\begin{aligned}
&\text{if (equal}(r_A, r_{B_{j-1}}) = \text{true}) \\
&\text{abort}
\end{aligned}
\end{array}
\]

\[
\begin{array}{c}
\begin{aligned}
&\text{if (Vrf}(K'_j, B||A||r_A, \tau_A) = \text{true}) \\
&\delta_{AB} \leftarrow 0 \\
&K' \leftarrow K'_j \text{kdf}; \text{upd}^*_A; E \leftarrow 0 \\
&\text{else if } (\text{Vrf}(K'_j, B||A||r_A, \tau_A) = \text{true}) \\
&\delta_{AB} \leftarrow 1 \\
&K' \leftarrow K'_j; E \leftarrow 1 \\
&\text{else if } (\text{Vrf}(K'_j, B||A||r_A, \tau_A) = \text{true}) \\
&\delta_{AB} \leftarrow -1 \\
&K' \leftarrow K'_{j-1}; \text{upd}^*_A; \text{kdf}; \text{upd}^*_A; E \leftarrow 0 \\
&\text{else}
\end{aligned}
\end{array}
\]

\[
\begin{array}{c}
\begin{aligned}
&\tau_A \leftarrow \text{MAC}(K', E||A||B||r_A||r_B) \\
&m_A \leftarrow E||\tau_A \\
&\text{[Same as the SAKE']} \\
&\text{[Same as the SAKE']}
\end{aligned}
\end{array}
\]

\[
\begin{array}{c}
\begin{aligned}
&\text{[Same as the SAKE']}
\end{aligned}
\end{array}
\]

\[
\begin{array}{c}
\begin{aligned}
&\text{[Same as the SAKE]}
\end{aligned}
\end{array}
\]

\[
\begin{array}{c}
\begin{aligned}
&\text{[Same as the SAKE]}
\end{aligned}
\end{array}
\]

Fig. 2. SAKE$^+$ and SAKE$^+$-AM Protocols.
by verification of the received message $m_A$ and runs the $kdf(\cdot, \cdot)$ and $upd^*_B(\cdot)$ functions based on the $\epsilon$ value. Finally, $B$ resets the value of the $\varphi$ to '0' and sends the message $B||\tau^*_B$ to the initiator. By resetting the value of the $\varphi$, $B$ indicates for the next session that the message $m_A$ was received successfully in the last session and it was valid. The rest flows and computations of the protocol are the same as those in the SAKE protocol. The only difference is with regard to the identity of $B$. It is concatenated with the protocol messages in order to preserve the concurrency property.

6.3 Description of the SAKE$^+$-AM protocol

In this subsection, we describe our proposed protocol SAKE$^+$-AM. The SAKE$^+$-AM is based on the SAKE$^+$ with some modifications. These modifications are shown in Figure 2 (in blue color). These changes are related to the message $m_A$, input of the $MAC(\cdot, \cdot)$, and $Vrf(\cdot, \cdot, \cdot)$ functions because of the absence of the party $A$’s random number $r_A$. The SAKE$^+$-AM renders it possible that a party involved in the protocol becomes either an initiator or a responder. The SAKE$^+$-AM scheme is resistant against the vulnerabilities of SAKE-AM discussed in Section 5. It is appropriate for IoT applications in that it is a lightweight protocol in terms of computation and communication; additionally, using the SAKE$^+$-AM scheme, resource-constrained devices establishing a connection to a server perform the least computation as the initiator, in this scheme, does lightweight calculations compared to the responder.

7 Soundness of SAKE$^+$

In this section we discuss that our proposed scheme is sound, which means once a correct session is finished, both parties have shared the same new session key and the same new identities, updated their respective internal states, and are synchronized successfully. For showing the soundness of our scheme, we make use of the similar notions used in [3]. We first define a lemma and then try to prove the corresponding items. We use the following notations in our proof:

- $c_A, c_B$: These are the monotonically increasing counters that are initialized with 0. The $c_A$ follows the master keys $K, K'_j, K'_{j-1}, B_j, B_{j-1}$, while $c_B$ follows the master keys $K, K'$.  
- $\delta_{AB}$: As we mentioned earlier, the $\delta_{AB}$ denotes the gap between $A$ and $B$, and it is computed as $\delta_{AB} = c_A - c_B$.  
- $(i_A, i_B)$: This notation means that the last valid message received by $A$ is the value $i_A$, and similarly the last valid message received by $B$ is $i_B$ (we assume that the session is completed).
- $(i_A, i_B)$-session: This is a session where the last message received by $A$ is $i_A$, and the last message received by $B$ is $i_B$.  
- $i_A = 0$: It means that $A$ has received no message.  
- $A$ and $B$ send back and forth the messages which are numbered from 1 to 5.
We consider that by default the value \((4, 5)\) is set for \((i_A, i_B)\), and the only possible valid values for \((i_A, i_B)\) are as follows: \((0, 1)\), \((2, 1)\), \((2, 3)\), \((4, 3)\), \((4, 5)\).

**Lemma 1:** We assume that the initiator \(A\) and the responder \(B\) are participating in a session of the SAKE\(^+\) protocol. Considering this assumption, we can conclude that \(\delta_{AB} \in \{-1, 0, 1\}\), and after termination of the session between \(A\) and \(B\), independent of the value of \(\delta_{AB}\), the following conditions holds: \(A\) and \(B\) have updated their master keys at least once, \(A\) and \(B\) are synchronized which means \(\delta_{AB}\) is set to 0, \(A\) and \(B\) share the same session key \(K\), and finally \(A\) and \(B\) share the same identity.

The proof of this lemma is given in Appendix A.

### 8 Security of SAKE\(^+\)

We make use of the security model described earlier in Section 3 and the sequence of games \([7, 31]\) approach to prove the security of the SAKE\(^+\) protocol. With the help of matching conversations, we define the partnering between two instances. Additionally, we benefit from the ProVerif tool \([12]\) to automatically analyze the security of our proposed scheme.

#### 8.1 Security of SAKE\(^+\) using the sequence of games

We use the sequence of games approach in which an attack game is played between an adversary and a challenger, and the security is linked to an event \(S\). Security means that considering every efficient adversary, the probability that event \(S\) happens is very close to some specified target probability. The target probability is either 0, \(\frac{1}{2}\), or the probability of some event \(T\) in some other game \([31]\). To prove using this approach, one builds a sequence of games, i.e., Game 0, Game 1, . . . , Game \(n\), where Game 0 is the original attack game regarding a cryptographic primitive and a given adversary \([31]\). We use the notations presented in Table 3 in our proof.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n)</td>
<td>a number of parties</td>
</tr>
<tr>
<td>(\lambda)</td>
<td>the size of values ((r_A, r_B))</td>
</tr>
<tr>
<td>(q)</td>
<td>a maximum number of instances per party</td>
</tr>
<tr>
<td>(\pi)</td>
<td>an instance that is targeted by (A)</td>
</tr>
<tr>
<td>(update(\cdot))</td>
<td>(K \leftarrow f(k, x)), for some (constant) value (x)</td>
</tr>
<tr>
<td>(E_i)</td>
<td>an event that (A) wins experiment in Game (i)</td>
</tr>
<tr>
<td>(B)</td>
<td>an adversary against PRF-security of (update(\cdot))</td>
</tr>
<tr>
<td>(C)</td>
<td>an adversary against SUF-CMA-security</td>
</tr>
<tr>
<td>(D)</td>
<td>an adversary against PRF-security of (kdf(\cdot, \cdot))</td>
</tr>
</tbody>
</table>

**Theorem 1:** The protocol SAKE\(^+\) is a secure AKE protocol, and for any PPT adversary \(\mathcal{A}\) in the AKE security experiment against protocol SAKE\(^+\), the following conditions hold:

\[
adv_{SAKE^+}^{ent-auth}(\mathcal{A}) \leq nq((nq - 1)2^{-\lambda} + (q + 1)adv^{PRF}_{update}(\mathcal{B}) + 2adv^{suf-cma}_{MAC}(\mathcal{C}))
\]

\[
adv_{SAKE^+}^{key-ind}(\mathcal{A}) \leq nq((q - 1)adv^{PRF}_{update}(\mathcal{B}) + adv^{PRF}_{kdf}(\mathcal{D}) + adv_{SAKE^+}^{ent-auth}(\mathcal{A}))
\]
The proof can be found in Appendix B.

8.2 Formal verification using ProVerif

We verify that \( \text{SAKE}^+ \) is robust and that it ensures forward secrecy, synchronisation, and mutual authentication using ProVerif tool [12], which is an automated formal verification tool. The source code and its description are presented in Appendix C.

9 Conclusion

In this paper we reviewed the symmetric-based AKE protocols that attempt to ensure an important security feature, i.e., the perfect forward secrecy. We focused on the recent state-of-the-art work [3] that presented two symmetric-based AKE schemes named SAKE and SAKE-AM. The authors of [3] claimed that these schemes are lightweight and secure to be employed in IoT applications consisting of resource-constrained IoT devices. However, we explained in detail how SAKE and SAKE-AM are vulnerable to several attacks, including the timeful, tracking, denial of service and replay attacks, and we discussed how the SAKE scheme suffers from the exhaustive search that makes it unsuitable for the context of the IoT. Considering the SAKE and SAKE-AM schemes, we presented two enhanced symmetric-based AKE protocols, i.e. \( \text{SAKE}^+ \) and \( \text{SAKE}^{+-\text{AM}} \). This enhancement is achieved by making as less changes as possible in the original protocols, and the improvements include PFSP, SP, DRSP, TS, and SRD. We claim that our protocols \( \text{SAKE}^+ \) and \( \text{SAKE}^{+-\text{AM}} \) not only maintain the security features of the original protocols, but also our schemes are resilient against the mentioned vulnerabilities and drawbacks. Additionally, in our schemes parties can take part in concurrent executions unlike the SAKE and SAKE-AM protocols. This concurrency makes our schemes far applicable to IoT applications in the sense that hundred or thousands of IoT devices simultaneously can send their collected data every few seconds to a server. We benefited from the sequence of games approach and the Proverif tool as well to prove the security and soundness of our schemes.

Considering that \( \text{SAKE}^+ \) and \( \text{SAKE}^{+-\text{AM}} \) are lightweight and ensure vital security features, they can be used in IoT applications. However, entities such as sensors and RFID tags in IoT application need to access their related information, not all the data stored on the server. Hence, providing lightweight access control in such applications plays a crucial role. In our future work, we aim to design symmetric-based AKE protocols that provide the lightweight access control mechanism for entities while maintaining the security features.

References

Proof of Lemma 1

Proof of Lemma 1 (item1) We first prove the item 1 of Lemma 1 and consider the following three different cases:

- When A and B both are synchronized, i.e., $\delta_{AB} = c_A - c_B = 0$. After all valid $(i_A, i_B)$-sessions, the values for $(c_A, c_B)$ and $\delta_{AB}$ are as follows:
  - After a $(0, 1)$-session $\Rightarrow (c_A, c_B) = (i, i)$ and $\delta_{AB} = 0$
  - After a $(2, 1)$-session $\Rightarrow (c_A, c_B) = (i + 1, i)$ and $\delta_{AB} = 1$
  - After a $(2, 3)$-session $\Rightarrow (c_A, c_B) = (i + 1, i + 1)$ and $\delta_{AB} = 0$
  - After a $(4, 3)$-session $\Rightarrow (c_A, c_B) = (i + 1, i + 1)$ and $\delta_{AB} = 0$
  - After a $(4, 5)$-session $\Rightarrow (c_A, c_B) = (i + 1, i + 1)$ and $\delta_{AB} = 0$

It is obvious that, in this case, the possible values for $\delta_{AB}$ are 0 and 1.

- When A is one step further, i.e., $\delta_{AB} = c_A - c_B = 1$. After all valid $(i_A, i_B)$-sessions, the values for $(c_A, c_B)$ and $\delta_{AB}$ are as follows:
  - After a $(0, 1)$-session $\Rightarrow (c_A, c_B) = (i + 1, i)$ and $\delta_{AB} = 1$
  - After a $(2, 1)$-session $\Rightarrow (c_A, c_B) = (i + 1, i)$ and $\delta_{AB} = 1$
  - After a $(2, 3)$-session $\Rightarrow (c_A, c_B) = (i + 1, i + 2)$ and $\delta_{AB} = -1$
  - After a $(4, 3)$-session $\Rightarrow (c_A, c_B) = (i + 2, i + 2)$ and $\delta_{AB} = 0$
  - After a $(4, 5)$-session $\Rightarrow (c_A, c_B) = (i + 2, i + 2)$ and $\delta_{AB} = 0$

We see that all possible values for $\delta_{AB}$ are 0, 1, and -1.

- When A is one step behind, i.e., $\delta_{AB} = c_A - c_B = -1$. After all valid $(i_A, i_B)$-sessions, the values for $(c_A, c_B)$ and $\delta_{AB}$ are as follows:
  - After a $(0, 1)$-session $\Rightarrow (c_A, c_B) = (i, i + 1)$ and $\delta_{AB} = -1$
  - After a $(2, 1)$-session $\Rightarrow (c_A, c_B) = (i, i + 1)$ and $\delta_{AB} = -1$
  - After a $(2, 3)$-session $\Rightarrow (c_A, c_B) = (i + 2, i + 2)$ and $\delta_{AB} = 0$
  - After a $(4, 3)$-session $\Rightarrow (c_A, c_B) = (i + 2, i + 2)$ and $\delta_{AB} = 0$
  - After a $(4, 5)$-session $\Rightarrow (c_A, c_B) = (i + 2, i + 2)$ and $\delta_{AB} = 0$

Similar to the previous case, the only possible values for $\delta_{AB}$ are 0, 1, and -1.
Based on the discussed cases, we can conclude that the only possible values for $\delta_{AB}$ are 0, 1, and -1, i.e., $\delta_{AB} \in \{0, 1, -1\}$.

**Proof of Lemma 1 (item2)** As we can see, the item 2 of Lemma 1 has four different cases. We discuss each case as follows:

- $A$ and $B$ have updated their master keys at least once: Whatever the value of $\delta_{AB}$ at the beginning of each session, after the $(4, 5)$-session, the value of the tuple $(c_A, c_B)$ is incremented at least by one (shown in bold). This means that $A$ and $B$ have updated their internal states at least once.

- $A$ and $B$ are synchronized: Whatever the value of $\delta_{AB}$ at the beginning of each session, after the $(4, 5)$-session, the value of $\delta_{AB}$ equals 0 (shown in bold). This means that when the session is completed, the initiator $A$ and responder $B$ will eventually be synchronized.

- $A$ and $B$ share the same session key $K$: Considering that the master keys $K$ and $K'$ are updated simultaneously on the $B$ side (resp $K, K'_{j-1}$, and $K'_j$ on the $A$ side), and the function $kdf(K, g(r_A, r_B))$ updates the session key immediately after $K$ has been updated, we can guarantee that $A$ and $B$ share the same session key $sk$. More precisely, after a correct and complete session ($(4, 5)$-session), and using the same values $r_A$ and $r_B$, the initiator and responder share the same session key $sk$.

- $A$ and $B$ share the same identity value: Considering that the master keys $K', B_{j-1}$, and $B_j$ are updated simultaneously on the $A$ side within $upd^*_A(\cdot)$, and the master keys $K'$ and $B$ are updated together on the $B$ side within $upd^*_B(\cdot)$, we claim that $A$ and $B$ share the same identity $B$ after a correct and complete session ($(4, 5)$-session).

**B Proof of Theorem 1**

Before starting our proof, we discuss the following notes that could be helpful during the proof of the security of SAKE$^+$. **Note:** An initiator instance $\pi^*_i$ at some party $P_i$ accepts, if two valid messages $m_B$ and $r'_B$ (valid MAC tags) are received by $\pi^*_i$. We can reduce the security of the MAC function to the ability to forge a valid output. We assume that the value $K'$, used during the first session between the initiator and the responder, is uniformly chosen at random. Considering that $K'$ is random, we can rely on the pseudo-randomness of the function $update(\cdot) = PRF(\cdot, \cdot)$. On the other hand, since $f(k', \cdot)$ can be replaced with a truly random function, the updated $K'$ accordingly is random. Hence, we can conclude that we can rely on the pseudo-randomness of the function $update(\cdot)$ with the new updated key $K'$, and so forth. Each update of $K'$ means a loss $(adv^{PRF}_{update}(B))$ that corresponds to the ability of an adversary $B$ to distinguish between the $update(\cdot)$ function and a random function.

**Note:** $P_i$ updates its keys at most once per session on average. This is due to the following facts:

- $\delta_{AB} = 0$: In this case $P_i$ updates its master keys only once.
- $\delta_{AB} = 1$: $P_i$ updates its keys at most once.
- $\delta_{AB} = -1$: $P_i$ updates its keys twice.
**Note:** $P_i$ updates its keys at most once per session on average. Where, in the case $\delta_{AB} = 0$, $P_i$ updates its master keys only once; in the case $\delta_{AB} = 1$, $P_i$ updates its keys at most once; and in the case $\delta_{AB} = -1$, $P_i$ updates its keys twice.

We conclude that when $P_i$ starts the $u$-session, it has updated its keys at most $u - 1$ times on average, and $P_i$ updates its keys at most two times upon receiving the message $\tau_B$.

**Note:** The previous note also applies to the responder. It means that it updates its keys at most once per session on average. This is due to the following facts:

- $\epsilon = 0$: In this case (upon reception of the message $m_A$), the responders updates its keys only once.
- $\epsilon = 1$: Upon reception of the message $m_A$, the responders updates its keys twice.

**Note:** The previous note also applies to the responder, meaning that it updates its keys at most once per session on average. Where, in the case $\epsilon = 0$, the responder updates its keys only once, and in the case $\epsilon = 1$, the responder updates its keys twice.

The responder $P_j$ has updated its keys at most $u - 1$ times on average, when it starts the $u$-session. It updates its keys twice when receiving the message $m_A$.

**Proof of entity authentication:** In our proof, each consecutive game aims at reducing the challenger’s dependency on the functions $MAC(\cdot, \cdot)$, $update(\cdot)$ and $kdf(\cdot, \cdot)$.

- Game 0: This game is associated with the experiment between the adversary and the challenger defined in definition 7 (Entity Authentication). The probability that the adversary wins the entity authentication is:

$$Pr[E_0] = adv_{SAKE}^{ent-auth}(A).$$

- Game 1: There is at most $n \times q$ random values $r_A$ or $r_B$ chosen uniformly at random in $\{0, 1\}^\lambda$. Hence, the probability that at least two random values be equal is at most $\frac{nq(nq-1)}{2^{\lambda}}$. If there exists any instance that chooses a random value $r_A$ or $r_B$ that is not unique, then the challenger will abort. Therefore

$$Pr[E_0] \leq Pr[E_1] + \frac{nq(nq-1)}{2^{\lambda}}.$$

- Game 2: The challenger tries to guess which instance will be the first to accept maliciously. The game is aborted if the guess is wrong. The number of instances is at most $nq$. Hence

$$Pr[E_2] = Pr[E_1] \times \frac{1}{nq}.$$

- Game 3: We define an abort rule as follows:

  If $\pi$ receives a valid message $m_B$ (resp. $m_A$), the challenger aborts the experiment. We reduce the probability of this event to the security of the function $MAC(\cdot, \cdot)$ and $update(\cdot)$.

$$Pr[E_2] \leq Pr[E_3] + (q - 1)adv_{update}^{PRF}(B) + adv_{MAC}^{\text{cma}}(C)$$

24
– Game 4: In this game, we reduce the probability of the adversary to win the game to the security of the MAC(·, ·) function (for $\tau'_B$ resp. $\tau'_A$ computation). In fact, we must rely on the randomness of the MAC(·, ·) key and accordingly to the security of the function update(·) which is used for updating $K'$ (i.e., used as an input for the function MAC(·, ·)). Considering that the master keys are updated at most twice between the message $m_B$ (resp. $m_A$) being received by $\pi$ and the message $\tau'_B$ (resp. $\tau'_A$) being received by $\pi$, we conclude that

$$Pr[E_3] \leq Pr[E_4] + 2adv^{PRF}_{update}(B) + adv^{suf-cma}_{MAC}(C).$$

By adding up all the probabilities from Game 0 to 4, we reach the mentioned bound

$$adv^{ent-auth}_{SAKE+}(A) \leq nq((nq - 1)2^{-\lambda} + (q + 1)adv^{PRF}_{update}(B) + 2adv^{suf-cma}_{MAC}(C)).$$

**Proof of the key indistinguishability security**: We assume that $E'_i$ is an event that an adversary wins the key indistinguishability experiment in Game $i$, and

$$adv_i = Pr[E'_i] - \frac{1}{2}.$$

– Game 0: This game is concerned with the experiment between the adversary and the challenger defined in definition 8 (Key Indistinguishability). The probability that the adversary wins the key indistinguishability experiment is computed as

$$Pr[E'_0] = \frac{1}{2} + adv^{key-ind}_{SAKE+}(A) = \frac{1}{2} + adv_0.$$

– Game 1: If there exists an instance that accepts maliciously, the challenger aborts the experiment and chooses $b' \in \{0, 1\}$ uniformly at random. Hence

$$adv_0 \leq adv_1 + adv^{ent-auth}_{SAKE+}(A).$$

– Game 2: The challenger tries to guess which instance is targeted by the adversary. If the guess is wrong, the game is aborted. Hence

$$adv_2 = adv_1 \times \frac{1}{nq}.$$

– Game 3: We reduce the advantage of the adversary to win this game to the security of the pseudo-random function $kdf(\cdot, \cdot)$. By assumption, the value of $K$ used by this function is uniformly chosen. On average, the key $K$ is updated at most once per session as discussed earlier. Hence, $K$ has been updated at most $u-1$ times when the $u$-th session starts. We must, therefore, consider the successive losses caused by the key update using the update(·) function. This loss is at most $(q - 1)adv^{PRF}_{update}(B)$ as there is at most $q$ sessions per party. Therefore, we consider truly random functions $G'_{update}, \ldots, G'_{update,q-2}$ instead of each function update($K$) = $f(k, x)$. Additionally, to key update, if an instance uses the same key $K = K_i, 0 \leq i \leq q - 1$, then we use $G'_{update,i}$ instead of the update(·). Hence, we reduce the ability of $A$ to win the security of the function $kdf(\cdot, \cdot)$ as

25
\[ \text{adv}_2 \leq \text{adv}_3 + (q - 1) \text{adv}_{\text{update}}^{\text{PRF}}(B) + \text{adv}_{kdf}^{\text{PRF}}(D). \]

The value of \( \text{adv}_3 \) is zero as to that point the session key is random. Indeed, the adversary has no advantage in guessing whether \( \pi \cdot b = b' \).

By summation of all the probabilities from Game 0 to 3, we reach the mentioned bound

\[ \text{adv}_{\text{SAKE}^+}^{\text{key-ind}}(A) \leq nq((q - 1) \text{adv}_{\text{update}}^{\text{PRF}}(B) + \text{adv}_{kdf}^{\text{PRF}}(D)) + \text{adv}_{\text{SAKE}^+}^{\text{ent-auth}}(A). \]

### C Security verification of the SAKE\(^+\) with ProVerif

We use the ProVerif tool \[12\] to formally prove the SAKE\(^+\) protocol. The ProVerif enables us to verify the concurrent execution of our protocol and to make sure whether our protocol achieves the desired security objectives or not. Parties involved in the protocol use a channel to communicate with each other. This channel is assumed to be controlled by an adversary that is able to read, change, delete, and create messages, and the model in which the attacker operates is called “Dolev-Yao” \[18\]. The attacker is capable of the modification of the protocol’s messages in that s/he can decrypt messages (if s/he gets access to the related keys) and can even compute the \( i \)th element of a tuple. We can encode our desired protocol and its objectives using the ProVerif’s input language formally, enabling the ProVerif tool to verify our claimed security properties. The cryptographic primitives used in ProVerif is assumed to be perfect, i.e., the adversary is not able to make use of any polynomial-time algorithm and s/he can only makes use of the cryptographic primitives defined by the user. With the help of rewrite rules and/or an equational theory, the cryptographic primitives are associated with each other.

A protocol that is written in the ProVerif tool’s input language (the typed pi calculus \[30\]) includes the following components: the \textit{declarations}, the \textit{processmacros} and the \textit{mainprocess}. These components are discussed as follows:

- **Declarations:** The declarations consists of the user types, the functions describing the cryptographic primitives, and the security properties as well.
- **Process macros:** The process macros include sub-process definitions; each sub-process is a sequence of events.
- **Main process:** It is defined with the help of macros. In our particular SAKE\(^+\) protocol, the main process is defined as the parallel composition (denoted by \( | \)) of the unbounded replication (denoted by \(!\)) of two process macros representing the \textit{processInitiator}, and \textit{processResponder} nodes.

ProVerif can prove both reachability property and correspondence assertions \[11\]. The Reachability property allows us to check which information an attacker can access, i.e. secrecy. Correspondence property is of the form “if some event is executed, then another event has been executed previously”, and could be used for checking various types of authentication \[25\]. We encoded the SAKE\(^+\) AKE protocol in the ProVerif language. In general, a protocol model can be divided into three different parts: the
declarations (lines 1-49), the process macros (lines 49-255), and the main process (lines 256-264).

```plaintext
(* - SAKE + channel -*)
free c: channel.

(* - SAKE + types -*)
type key.
type nonce.
type host.

(* - SAKE + keys -*)
free SKa, SKb, K, Kj0, Kj1, Kj2: bitstring [private].
free Bj0, Bj1, Bj2, Rb0, B: bitstring.
free A: host.

(* - SAKE + constants -*)
const f0, f1, Dhel0, Dhel1, DhelN, Epsi0, Epsi1, X: bitstring.
table TA(host, key, bitstring, bitstring, bitstring, bitstring, nonce).
table TB(bitstring, key, bitstring, bitstring).

(* - SAKE + functions -*)
fun nonce_to_bitstring(nonce): bitstring [data, typeConverter].
fun bitstring_to_key(bitstring): key [data, typeConverter].
fun host_to_bitstring(host): bitstring [data, typeConverter].
fun bitstring_to_nonce(bitstring): nonce [data, typeConverter].
fun mac(bitstring, key): bitstring.
reduc forall m: bitstring, k: key; get_message(mac(m, k)) = m.
fun PRF(bitstring, key): bitstring.
fun con(bitstring, bitstring): bitstring.

(* - SAKE + events -*)
event beginAparam(host, nonce).
event endAAuth(host, nonce).
event beginBparam(bitstring, nonce).
event endBAuth(bitstring, nonce).
event beginsyncBkey(bitstring, host, nonce, key).
event endsyncBkey(bitstring, host, nonce, key).

(* - SAKE + queries -*)
query attacker(SKa).
query attacker(SKb).
query attacker(K).
query attacker(Kj0).
query attacker(Kj1).
query attacker(Kj2).
query x: host, y: nonce; inj-event(endAAuth(x, y)) ==> inj-event(
  beginAparam(x, y)).
query x: bitstring, y: nonce; inj-event(endBAuth(x, y)) ==> inj-event(
  beginBparam(x, y)).
query x: bitstring, y: host, z: nonce, t: key; inj-event(endsyncBkey(x, y, z, t)) ==> inj-event(beginsyncBkey(x, y, z, t)).
```

C.1 Declarations

The declarations include the user types, the functions that describe the cryptographic primitives, and the security properties. Additional user types can be declared as in lines 4-6 apart from the built-in types: channel and bitstring. Free names are defined as in
lines 2 and 8-10 where the channel with names $c$ is declared. By default, the free names are accessible to the attacker unless qualified by [Private]. Finally, constant values are declared by const. The language supports tables for persistent storage. In lines 13 and 14, tables that model the subscribers database is declared.

Constructors are functions used to build terms. A constructor is declared by defining its names, the types of its arguments and the return value (see lines 16-20, 22-23). Functions, by default, are one-way; i.e., the attacker cannot infer the arguments from the return value, unless qualified by [data]. Destructors (line 21) are special functions that are used to manipulate terms. Constructors and destructors jointly are used to capture the relationship between cryptographic primitives.

Message authentication codes (MAC) can be declared by a constructor (with no associated destructor or equation), much like a keyed hash function as follow:

```plaintext
type key.
fun mac (bitstring,key):bitstring.
```

This model is strong in the sense that it considers the MAC as a random oracle. If the MAC is considered to be a pseudo-random function (PRF), it is probably the best possible model (in line 22, it is presented as

```plaintext
fun PRF (bitstring,key):bitstring.
```

Considering that the MAC is unforgeable (UF-CMA), one can declare a destructor which leaks the MACed message as follow:

```plaintext
reduc for all m:bitstring, k: key; getmessage (mac(m,k)) = m.
```

A sequences of events presented in lines 25-30, are defined as follows:

- The `beginAparam` event declares that the initiator A starts the authentication protocol with its identity A and a fresh nonce.
- The `endAAuth` event declares that the initiator A will authenticated with the responder that received the fresh nonce generated by A.
- The events `beginBparam` and `endBAuth` for the entity B are the same as `beginAparam` and `endAAuth` events for the entity A, respectively.
- The `eginsyncBkey` and `endsyncBkey` events declare that the responder B is in the synchronize state with the initiator A.

We model correspondence assertions of the form: “if an event end has been executed, then event begin has been previously executed.” with the queries presented in lines 38-40 that the first two queries (lines 38 and 39) are for the mutual authentication and the last one is for the synchronization. The rest of the queries which are presented in lines 32-37 is base on a built-in predicate attacker used to check which terms are compromised.

### C.2 Process macros

The process macros consist of sub-process definitions that are a sequence of events. Messages are represented by terms, i.e., a name, a variable, a tuple of terms, a constructor or destructor application. The language, additionally, supports some common Boolean functions ($=, &&, ||, <>$) with the infix notation.
There are term evaluation, restriction, communication and condition events defined as follows:

- The pattern \( x : t \) matches any term of type \( t \) and binds it to \( x \).
- The \texttt{let x = M in} binds the term \( M \) to \( x \).
- The name restriction event \texttt{new} declares a fresh name of a specific type and binds it inside the events. For instance, line 43 binds the type \texttt{nonce} to the fresh name \( R_a \).
- The communication event \texttt{in (c,(x:host,y:nonce))} listen from a channel \( c \) and binds the received terms to \( x \) and \( y \) where the first one has type \texttt{host} and the second one has type \texttt{nonce}.
- The communication event \texttt{textbf{out (c,(x:host,y:nonce))}}, sends the terms \( x \) and \( y \) on channel \( c \).
- The conditional \texttt{if M else P then Q} continues as the process \( P \) if the term \( M \) evaluates to true, continues as the process \( Q \) if \( M \) evaluates to another value.

```plaintext
41 (* Role of the initiator *)
42 let processInitiator =
43     new Ra: nonce;
44     new aDhel: bitstring;
45     new aEpsi: bitstring;
46     new akj: bitstring;
47     get TA(aA,aK,aKj0,aKj1,aBj0,aBj1,aRb0) in
48     let A = aA in
49     let m0 = (A,Ra) in
50     event beginAparam(A, Ra);
51     (* ---->A||r_A *)
52     out(c,m0);
53     (* m_B<---- *)
54     in(c,(aB:bitstring,aRb:nonce,aTb:bitstring));
55     let aBj2 = PRF(X, bitstring_to_key(aBj1)) in
56     if aB = aBj0 then let B = aBj0 in
57     if aB = aBj1 then let B = aBj1 in
58     if aB = aBj2 then let B = aBj2 in
59     if aB <> aBj0 && aB <> aBj1 && aB <> aBj2 then let B = aBj1 in
60     if aRb0 <> aRb then
61         (;
62         let Tbin = con(B,con(host_to_bitstring(A),con(nonce_to_bitstring(aRb),nonce_to_bitstring(Ra)))) in
63         let Tb = mac(Tbin, bitstring_to_key(aKj1)) in
64         if Tb = aTb then
65             (;
66             let aDhel = Dhel0 in
67             let akj = aKj1 in
68             let SKa = PRF(con(nonce_to_bitstring(Ra),nonce_to_bitstring(aRb)), aK) in
69             let aK = PRF(X, aK) in
70             let aBj1 = PRF(X, bitstring_to_key(con(aBj1,aKj1))) in
71             let aKj0 = aKj1 in
72             let aKj1 = PRF(X, bitstring_to_key(aKj1)) in
73             let aBj0 = aBj1 in
74             let aEpsi = Epsi0 in
```

29
let aRb0 = aRb in
insert TA(A, bitstring_to_key(aK), aKj0, aKj1, aBj0, aBj1, aRb0);
    let Tain = con(aEpsi, con(host_to_bitstring(A), con(B, (nonce_to_bitstring(Ra), nonce_to_bitstring(aRb)))))) in
let Ta = mac(Tain, bitstring_to_key(akj)) in
let ma = (B, aEpsi, Ta) in
out(c, ma)
    (* ----> m_A *)
)
else
let Tbin = con(B, con(host_to_bitstring(A), con(nonce_to_bitstring(aRb), nonce_to_bitstring(Ra)))) in
let Tb = mac(Tbin, bitstring_to_key(aKj0)) in
if Tb = aTb then
    (let aDhel = Dhel1 in
let akj = aKj0 in
let aEpsi = Epsi1 in
let aRb0 = aRb in
insert TA(A, aK, aKj0, aKj1, aBj0, aBj1, aRb0);
    let Tain = con(aEpsi, con(host_to_bitstring(A), con(B, (nonce_to_bitstring(Ra), nonce_to_bitstring(aRb)))))) in
let Ta = mac(Tain, bitstring_to_key(akj)) in
let ma = (B, aEpsi, Ta) in
out(c, ma)
    (* ----> m_A *)
)
else
let Tbin = con(B, con(host_to_bitstring(A), con(nonce_to_bitstring(aRb), nonce_to_bitstring(Ra)))) in
let Tb = mac(Tbin, bitstring_to_key(PRFX, bitstring_to_key(aKj1)))
in
if Tb = aTb then
    (let aDhel = DhelN in
let akj = PRFX, bitstring_to_key(aKj1)) in
let aK = PRFX(X, aK) in
let aBj1 = PRFX(X, bitstring_to_key(con(aBj1, aKj1))) in
let aKj0 = aKj1 in
let aKj1 = PRFX(X, bitstring_to_key(aKj1)) in
let aBj0 = aBj1 in
let SKa = PRFX(con(nonce_to_bitstring(Ra), nonce_to_bitstring(aRb)), bitstring_to_key(aK)) in
let aK = PRFX(X, bitstring_to_key(aK)) in
let aBj1 = PRFX(X, bitstring_to_key(con(aBj1, aKj1))) in
let aKj0 = aKj1 in
let aKj1 = PRFX(X, bitstring_to_key(aKj1)) in
let aBj0 = aBj1 in
let aEpsi = Epsi0 in
let aRb0 = aRb in
insert TA(A, bitstring_to_key(aK), aKj0, aKj1, aBj0, aBj1, aRb0);
    let Tain = con(aEpsi, con(host_to_bitstring(A), con(B, (nonce_to_bitstring(Ra), nonce_to_bitstring(aRb)))))) in
let Ta = mac(Tain, bitstring_to_key(akj)) in
let ma = (B, aEpsi, Ta) in
out(c, ma)
(* ---> m_A *)

else
yield

else
tmp(b) <------ *
in(c, (b: bitstring, atp: bitstring));
let B = aB in
if aEpsi = Epsi0 then

let akj = aKj1 in
let Tpbin = con(nonce_to_bitstring(aRb), nonce_to_bitstring(Ra)) in
let Tpb = mac(Tpbin, bitstring_to_key(akj)) in
if Tpb = atpb then

let Tpain = con(nonce_to_bitstring(Ra), nonce_to_bitstring(aRb)) in
let Tpa = mac(Tpain, bitstring_to_key(akj)) in
let m3 = (B, Tpa) in
event beginsyncBkey(B, A, aRb, bitstring_to_key(akj));
out(c, m3)
(* ---> B||T^p_A *)

else
yield

else
if aEpsi = Epsi1 then

let akj = PRF(X, bitstring_to_key(aKj1)) in
let Tpbin = con(nonce_to_bitstring(aRb), nonce_to_bitstring(Ra)) in
let Tpb = mac(Tpbin, bitstring_to_key(akj)) in
if Tpb = atpb then

let SKa = PRF(con(nonce_to_bitstring(Ra), nonce_to_bitstring(aRb)), aK) in
let aK = PRF(X, aK) in
let aBj1 = PRF(X, bitstring_to_key(con(aBj1, aKj1))) in
let aKj0 = aKj1 in
let aKj1 = PRF(X, bitstring_to_key(aKj1)) in
let aBj0 = aBj1 in
let Tpain = con(nonce_to_bitstring(Ra), nonce_to_bitstring(aRb)) in
let Tpa = mac(Tpain, bitstring_to_key(akj)) in
let m3 = (B, Tpa) in
event beginsyncBkey(B, A, aRb, bitstring_to_key(akj));
event endBAuth(B, aRb);
out(c, m3)
(* ---> B||T^p_A *)

else


yield
)
else
0.
(* Role of the responder *)
let processResponder =
(* A||r_A<---- *)
in(c,(bA:host , bRa:nonce));
get TB(bB,bK,bKj,bf) in
new Rb: nonce;
let bTbin = con(bB,con(host_to_bitstring(bA),con(nonce_to_bitstring(Rb),
    nonce_to_bitstring(bRa)))) in
let bTb = mac(bTbin, bitstring_to_key(bKj)) in
new Ralfa: nonce;
let mb = (if bf= f1 then (nonce_to_bitstring(Ralfa),Rb,bTb) else (bB,Rb,
bTb)) in
let bf= f1 in
event beginBparam(bB, Rb);
(* ---->m_B *)
out(c,(mb));
(* m_A<---- *)
in(c,(bBp: bitstring,bEpsi: bitstring,bTa: bitstring));
let Tapin = con(bEpsi,con(host_to_bitstring(bA),con(bB,con(
    nonce_to_bitstring(bRa),nonce_to_bitstring(Rb)))))) in
let Tap = mac(Tapin, bitstring_to_key(bKj)) in
if Tap = bTa then
if bEpsi = Epsi1 then
(
let bK = PRF(X, bK) in
let bB = PRF(X, bitstring_to_key(con(bB,bKj))) in
let bKj = PRF(X, bitstring_to_key(bKj)) in
let SKb = PRF(con(nonce_to_bitstring(bRa),nonce_to_bitstring(Rb)),
    bitstring_to_key(bK)) in
let bK = PRF(X, bitstring_to_key(bK)) in
let bB = PRF(X, bitstring_to_key(con(bB,bKj))) in
let bKj = PRF(X, bitstring_to_key(bKj)) in
let bf= f0 in
insert TB(bB,bitstring_to_key(bK),bKj,bf);
let Tpbbin = con(nonce_to_bitstring(Rb),nonce_to_bitstring(bRa)) in
let Tpb = mac(Tpbbin, bitstring_to_key(bKj)) in
let m2 = (bB,Tpb) in
(* ---->B||T^p_B *)
out(c,m2)
)
else
let SKb = PRF(con(nonce_to_bitstring(bRa),nonce_to_bitstring(Rb)), bK) in
let bK = PRF(X, bK) in
let bB = PRF(X, bitstring_to_key(con(bB,bKj))) in
let bKj = PRF(X, bitstring_to_key(bKj)) in
let bf= f0 in
insert TB(bB,bitstring_to_key(bK),bKj,bf);
let Tpbbin = con(nonce_to_bitstring(Rb),nonce_to_bitstring(bRa)) in
let Tpbp = mac(Tpbpin, bitstring_to_key(bKj)) in
let m2 = (bB, Tpbp) in
(* ---->B||T^ p_B *)
out(c, m2);
(* B||T^p_A<----- *)
in(c,(bBpp:bitstring,bTpa:bitstring));
let bTpapin = con(nonce_to_bitstring(bRa), nonce_to_bitstring(Rb)) in
let bTpap = mac(bTpapin, bitstring_to_key(bKj)) in
if bTpap = bTpa then
  (event endsyncBkey(bBpp, bA, Rb, bitstring_to_key(bKj));
  event endAAuth(bA, bRa)
  )
else
  0.

C.3 Main process

Finally, the main process is defined by means of two process macros that represent the
processInitiator (line 242) and processInitiator (line 243) nodes. The initialization
phase of the scheme is presented in lines 237-240 for an initiator A and a responder B
in line 237, and lines 238-240, respectively. Finally, in lines 240-243, the parallel com-
positions of processInitiator and processResponder denoted by | with the unbounded
replication (denoted by !).

process
  insert TA(A, bitstring_to_key(K), Kj0, Kj1, Bj0, Bj1, bitstring_to_nonce(Rb0 ));
  new f: bitstring;
  let f = f0 in
  insert TB(B, bitstring_to_key(K), Kj0, f);
  ( !processInitiator ) |
  ( !processResponder )
)

C.4 Security properties

Security properties are declared with the keyword. In our example of SAKE⁺, the
goal is to establish the shared session key $SK_a = SK_b$ between A and B after mutual
authentication by preserving the forward secrecy. The protocol should be robust against
the traceability and de-synchronization attacks. In order to check this, we consider the
following queries.

(*-SAKE+ queries-* )
query attacker(SKa).
query attacker(SKb).
query attacker(K).
query attacker(Kj0).
query attacker(Kj1).
query attacker(Kj2).
query x: host, y: nonce; inj-event(endAAuth(x, y)) ==> inj-event(beginAParam(x, y)).
query x: bitstring, y: nonce; inj-event(endBAuth(x, y)) ==> inj-event(beginBParam(x, y)).
query x: bitstring, y: host, z: nonce, t: key; inj-event(endsyncBKey(x,y,z,t)) ==> inj-event(beginsyncBKey(x,y,z,t)).

- The first six queries presented in lines 32-37 is base on a built-in predicate attacker used to check which terms are compromised.
- The query presented in line 38 proves that B successfully authenticates A, if ProVerif returns true. The event beginAParam is called in line 50 on the new nonce Ra generated by the initiator A and the event endAAuth is called by the responder B in line 23 after successful authentication of the initiator and establishing the session key.
- The query presented in line 39 proves that A successfully authenticates B, if ProVerif returns true. The events of this query are presented as beginBParam and endBAuth in lines 188 and 168, respectively.
- The query presented in line 40 proves that the responder B will be successful in the synchronization state using the events beginsyncBkey and endsyncBkey on the master key K' (see lines 143,167 and 231) in case that true is resulted from ProVerif.

The results are illustrated bellow and show that all the events result in true, which prove that the SAKE+ can preserve all the mentioned security queries.

Verification summary:
1. Query not attacker(SKa[]) is true.
2. Query not attacker(SKb[]) is true.
3. Query not attacker(K[]) is true.
4. Query not attacker(Kj0[]) is true.
5. Query not attacker(Kj1[]) is true.
6. Query not attacker(Kj2[]) is true.
7. Query inj-event(endAAuth(x,y)) ==> inj-event(beginAParam(x,y)) is true.
8. Query inj-event(endBAuth(x,y)) ==> inj-event(beginBParam(x,y)) is true.
9. Query inj-event(endsyncBkey(x,y,z,t)) ==> inj-event(beginsyncBkey(x,y,z,t)) is true.

C.5 Analysis and discussion

As mentioned earlier, all the queries are solved as expected, that is, the correspondence and secrecy ones are proved. We encode our security goals using the ProVerif queries as follows:

Secrecy for a message, such as m2, encoded using MAC function that asks the adversary to guess the value of Kj'; if the adversary succeeds, the ProVerif issues false to the query query attacker(Kj0). We have the same discussion for the other terms free X:bitstring [private], in our scheme. For all of these queries, we received the result true from ProVerif, which means that the protocol satisfies secrecy.
Forward secrecy for a master key $K'_j$, we use the `free Kj1:bitstring`, instead of `free Kj1:bitstring [private]`, which gives the attacker knowledge of the current master key $K'_j$ and asks the adversary to guess the value of $K'_{j-1}$ using the query `query attacker(Kj0)`; 
if the adversary succeeds, the ProVerif issues `false` to the query. We received the result `true` from ProVerif meaning that the protocol satisfies forward secrecy.

Authentication. Considering the query presented in line 38, if ProVerif returns `true`, it proves that B successfully authenticates A and considering the query presented in line 39, if ProVerif returns `true`, it proves that A successfully authenticates B. By satisfying these two queries, we can ensure that our scheme provides mutual authentication successfully.

Replay. The events `beginAparam`, `endAAuth`, `beginBparam` and `endBAuth` used for the authentication are based on the fresh nonces $R_a$ and $R_b$. With regard to the result `true` from both queries presented in lines 38 and 39, it proves that the scheme is secure against the replay attack.

synchronization. The query presented in line 40 proves that the responder B will be successful in the synchronization state using the events `eginsyncBkey` and `endsyncBkey` on the master key $K'_j$ (see lines 143, 167 and 231) in case that ProVerif shows the `true` results.

In addition to the above queries, our scripts also include built-in predicate attacker used to check which terms are compromised (presented in lines 32-37).