Note on Subversion-Resilient Key Exchange

Magnus Ringerud

Department of Mathematical Sciences
NTNU – Norwegian University of Science and Technology, Trondheim, Norway,
magnus.ringerud@ntnu.no

Abstract. In this work, we set out to create a subversion resilient authenticated key exchange protocol. The first step was to design a meaningful security model for this primitive, and our goal was to avoid using building blocks like reverse firewalls and public watchdogs. We wanted to exclude these kinds of tools because we desired that our protocols to be self contained in the sense that we could prove security without relying on some outside, tamper-proof party. To define the model, we began by extending models for regular authenticated key exchange, as we wanted our model to retain all the properties from regular AKE.
While trying to design protocols that would be secure in this model, we discovered that security depended on more than just the protocol, but also on engineering questions like how keys are stored and accessed in memory. Moreover, even if we assume that we can find solutions to these engineering challenges, other problems arise when trying to develop a secure protocol, partly because it's hard to define what secure means in this setting. It is in particular not clear how a subverted algorithm should affect the freshness predicate inherited from trivial attacks in regular AKE. The attack variety is large, and it is not intuitive how one should treat or classify the different attacks.
In the end, we were unable to find a satisfying solution for our model, and hence we could not prove any meaningful security of the protocols we studied. This work is a summary of our attempt, and the challenges we faced before concluding it.

Keywords: Authenticated key exchange, subversion-resilient protocols, reverse firewalls, unique signatures.

1 Introduction

The goal of this work was to create a model for subversion resilient authenticated key exchange protocols. We desired for it to retain all the properties for regular AKE, while adding some form of subversion resilience. We initially based our model on the AKE model in [HJK$^+$21], and treated subversions as an oracle an adversary can query, analogous to how it's done in [BCJ21, AMV15, BJK15, BPR14]. Our goal was to avoid using primitives like reverse firewalls [AMV15, DMS16] or watchdogs [BCJ21], keeping the already complicated model as self contained as possible.

One of the most significant challenges is to determine how subversions should affect the freshness predicate inherited from the trivial attacks in regular AKE given in Table 1.

In short, we say that a session is fresh if the adversary cannot trivially learn the session key from queries it made, such as through corrupting the long term secret key before creating the session key, or revealing the session key of a partnered session.

Our naive model is given in Section 3. We knew from the start that the subversion oracle should have some influence on the freshness predicate, but we did not know exactly how. Hence we went forward with analyzing protocols without changing the predicate, aiming to revisit the issue when we had developed more knowledge of the properties we desired. When we later on acknowledged that it might not be possible to properly define a model with the desired properties, we stopped working on this and hence never went back to change anything in this section.

In Sections 4 and 5, we discuss and construct attacks and countermeasures for a signed Diffie-Hellman protocol. Without the proper countermeasures, several of the attacks make the protocol insecure, but again it is not clear how one should treat the freshness predicate with regards to the attacks. In regular AKE, once a user has been corrupted and given its long term secret key to the adversary, a session involving this user cannot be used to break the AKE game. While some of the subversion attacks we present require multiple sessions to perform, the end result is the same in that the adversary learns the secret key, and hence it's not clear how or if one should treat these attacks differently from regular corruptions. If one decides to treat these types of subversions like regular corruptions, the question then becomes whether the extended model provides any useful security guarantees that regular AKE models do not.

Regardless of how one treats the freshness predicate discussed above, the model should also capture the attacks that do not outright break the protocol by leaking secret information, but simply makes it less secure. Examples of such are attacks in which an adversary is able to choose randomness such that the final session key has low or high weight, specific weight, starts with a certain amount of zeros, and so forth. One could also consider subversions that remove countermeasures to side channel attacks, like swapping a constant time algorithm for a non-constant time one. Formally treating this is challenging, as the attack surface for such subversions is huge, and it's hard to quantify them in a meaningful manner.

At this point we concluded our work, as we could not see any clear way to define a useful freshness predicate, nor how or if one could distinguish certain subversions from regular corruptions, or in general how to classify which types of subversions should be allowed or not in the model.

We note that the paper by Dodis et al. [DMS16] solves many of the challenges we face by using a reverse firewall. While we set out to explicitly avoid this, we were unable to do so in a satisfactory manner, and hence we refer readers to their paper for what we believe to be the currently best model and practical protocol for subversion resilient authenticated key exchange.

2 Preliminaries

For a randomized algorithm Π and some input X, we write $Y \leftarrow \Pi(X)$ as the process of randomly generating Y by running Π with input X. In some cases where we want to specify the randomness r, we will write $Y := \Pi(X; r)$. When we want to sample an element r uniformly at random from a set $\mathcal{R}$, we write this as $r \leftarrow_{\$} \mathcal{R}$. For an algorithm $\mathcal{A}$ returning a bit b after running in some game, we write $\mathcal{A} \Rightarrow b$.

Definition 1 (Signature Scheme). *A signature scheme* $\mathsf{SIG} = (\mathsf{SIG.Setup}, \mathsf{SIG.Gen}, \mathsf{Sign}, \mathsf{Ver})$ *is defined by the following algorithms:*

- $\mathsf{SIG.Setup}$*: The setup algorithm takes a security parameter* 1^λ *as input and outputs public parameters* $\mathsf{pp_{SIG}}$ *which defines a message space* $\mathcal{M}$*, signature space* Σ *and public key and secret key spaces* $\mathcal{PK} \times \mathcal{SK}$*. If the scheme is probabilistic, it also defines a randomness space* $\mathcal{R}$*.*
- $\mathsf{SIG.Gen}(\mathsf{pp_{SIG}})$*: The key generation inputs* $\mathsf{pp_{SIG}}$ *and outputs a key pair* $(\mathsf{pk}, \mathsf{sk}) \in \mathcal{PK} \times \mathcal{SK}$*.*
- $\mathsf{Sign}(\mathsf{sk}, m)$*: The signing algorithm inputs the secret (signing) key* sk *and a message* m *and outputs a signature* $\sigma \in \Sigma$*.*
- $\mathsf{Ver}(\mathsf{pk}, m, \sigma)$*: The deterministic verification algorithm inputs a public key* pk*, a message* m *and a signature* σ *on the message, and outputs 1 or 0 indicating whether* σ *is a valid signature on* m *under* sk*.*

We require that for all $\mathsf{pp_{SIG}} \in \mathsf{SIG.Setup}(1^\lambda), (\mathsf{pk}, \mathsf{sk}) \in \mathsf{SIG.Gen}(\mathsf{pp_{SIG}})$*, we have* $\mathsf{Ver}(\mathsf{pk}, m, \mathsf{Sign}(\mathsf{sk}, m)) = 1$.

Definition 2 (Min-entropy of signatures). *We say that a signature scheme has* η *bits of min-entropy if the following holds:*

$$2^{-\eta} = \max_{\mathsf{sk} \in \mathcal{SK}, m \in \mathcal{M}, \sigma \in \Sigma} \Pr[\mathsf{Sign}(\mathsf{sk}, m; r) := \sigma], \tag{1}$$

where the probability is taken over the random coins $r \in \mathcal{R}$*.*

Definition 3. *A signature scheme* SIG *has unique signatures if for all* pk *output by* Gen *and all* $m \in \mathcal{M}$*, there exists a unique value* $\sigma \in \Sigma$ *such that* $\mathsf{Ver}(\mathsf{pk}, m, \sigma) = 1$.

3 Security Model for Subversion-Resilient Authenticated Key Exchange

In this section, we propose a model for Subversion-Resilient Authenticated Key Exchange (SRAKE) which is similar to regular models for AKE [BHJ$^+$15, LS17, GJ18, HJK$^+$21], with the addition of subversions. We model subversions by following the approaches in [BCJ21, AMV15, BJK15, BPR14], where an adversary supplies a subverted implementation $\widetilde{\Pi}$ of protocol Π to the challenger. Such an attack is called an Algorithm-Substitution Attack (ASA). In our case, the adversary supplies subverted algorithms to the parties in the AKE-game. In the security game, this behavior is modeled by an oracle $\mathsf{Subv}(n, \Pi, \widetilde{\Pi})$ that replaces P_n's protocol Π with $\widetilde{\Pi}$. This might be a whole other protocol, or just changes to one of the internal algorithms.

We build our security model by extending and adjusting the model in [HJK$^+$21] to define the security of subversion-resilient authenticated key exchange protocols, and hence this section closely resembles Section 4 of [HJK$^+$21]. In this work, we do not consider state reveal and replay attacks as is done in [HJK$^+$21]. In their work, replay attacks are handled by including a nonce as the first message, which we could also do here, but that would only complicate the analysis. Since our main focus is subversions, we aim at creating a functional model for this setting, which can later be extended.

3.1 Definition of Subversion Resilient Authenticated Key Exchange

We will often write SR instead of SRAKE to save space.

Definition 4 (Subversion resilient authenticated key exchange scheme). *A subversion resilient authenticated key exchange scheme* $\mathsf{SRAKE} = (\mathsf{SR.Setup}, \mathsf{SR.Gen}, \mathsf{SR.Protocol})$ *consists of two probabilistic algorithms and an interactive protocol.*

- $\mathsf{SR.Setup}$*: The setup algorithm outputs the public parameter* $\mathsf{pp}_{\mathsf{SR}}$.
- $\mathsf{SR.Gen}(\mathsf{pp}_{\mathsf{SR}}, \mathsf{P}_i)$*: The generation algorithm takes as input* $\mathsf{pp}_{\mathsf{SR}}$ *and a party* P_i*, and outputs a key pair* $(\mathsf{pk}_i, \mathsf{sk}_i)$*.*
- $\mathsf{SR.Protocol}(\mathsf{P}_i(\mathsf{res}_i) \rightleftharpoons \mathsf{P}_j(\mathsf{res}_j))$*: The protocol involves two parties* P_i *and* P_j*, who have access to their own resources,* $\mathsf{res}_i := (\mathsf{sk}_i, \mathsf{pp}_{\mathsf{SR}}, \{\mathsf{pk}_u\}_{u\in[\mu]}, \Pi)$ *and* $\mathsf{res}_j := (\mathsf{sk}_j, \mathsf{pp}_{\mathsf{SR}}, \{\mathsf{pk}_u\}_{u\in[\mu]}, \Pi)$*, respectively. Here* Π *contains the protocol specifications, and* μ *is the total number of users. After execution,* P_i *outputs a flag* $\Psi_i \in \{\emptyset, \mathbf{accept}, \mathbf{reject}\}$*, and a session key* k_i *(*k_i *might be the empty string* $\emptyset$*), and* P_j *outputs* (Ψ_j, k_j) *similarly.*

Correctness of SRAKE. Any pair of distinct and honest parties P_i and P_j should share the same session key after the execution of $\mathsf{SR.Protocol}(\mathsf{P}_i(\mathsf{res}_i) \rightleftharpoons \mathsf{P}_j(\mathsf{res}_j))$, which implies that $\Psi_i = \Psi_j = \mathbf{accept}$ and $k_i = k_j \neq \emptyset$.

We note that these algorithms are identical to those in a regular AKE protocol $\mathsf{AKE} = (\mathsf{AKE.Setup}, \mathsf{AKE.Gen}, \mathsf{AKE.Protocol})$, and hence we don't provide a separate definition for AKE.

3.2 Security Model of SRAKE

We will adapt the security model formalized by [BHJ+15, LS17, GJ18], which in turn followed the model proposed by Bellare and Rogaway [BR94]. We use multiple test queries with respect to the same random bit [JKRS21].

First, we will define oracles and their static variables in the model. Then we describe the security experiment and the corresponding security notions.

Oracles. Suppose there are at most μ users $\mathsf{P}_1, \mathsf{P}_2, ..., \mathsf{P}_\mu$, and each user will involve at most ℓ instances. P_i is formalized by a series of oracles, $\pi_i^1, \pi_i^2, ..., \pi_i^\ell$. Oracle π_i^s formalizes P_i's execution of the s-th protocol instance. Each oracle π_i^s has access to P_i's resource $\mathsf{res}_i := (\mathsf{sk}_i, \mathsf{pp}_{\mathsf{SR}}, \mathsf{PKList} := \{\mathsf{pk}_u\}_{u\in[\mu]}, \Pi)$, and it has its own variables $\mathsf{var}_i^s := (\mathsf{st}_i^s, \mathsf{Pid}_i^s, k_i^s, \Psi_i^s)$.

- st_i^s: State information that has to be stored between two rounds in order to execute the protocol.
- Pid_i^s: The intended communication peer's identity.
- $k_i^s \in \mathcal{K}$: The session key computed by π_i^s. Here $\mathcal{K}$ is the session key space. We assume that $\emptyset \in \mathcal{K}$.
- $\Psi_i^s \in \{\emptyset, \mathbf{accept}, \mathbf{reject}\}$: Ψ_i^s indicates whether π_i^s has completed the protocol execution and accepted k_i^s.

At the beginning, $(\mathsf{st}_i^s, \mathsf{Pid}_i^s, k_i^s, \Psi_i^s)$ are initialized to $(\emptyset, \emptyset, \emptyset, \emptyset)$. We declare that $k_i^s \neq \emptyset$ if and only if $\Psi_i^s = \mathbf{accept}$.

Security Experiment. To define the security notion of SRAKE, we first formalize the security experiment $\mathsf{Exp}_{\mathsf{SRAKE},\mu,\ell,\mathcal{A}}$ with the help of the oracles defined above.

$\mathsf{Exp}_{\mathsf{SRAKE},\mu,\ell,\mathcal{A}}$ is a game played between a SRAKE challenger $\mathcal{C}$ and an adversary $\mathcal{A}$. $\mathcal{C}$ will simulate the executions of the ℓ protocol instances for each of the μ users with oracles π_i^s. We give a formal description in Figure 1.

```
$\mathsf{Exp}_{\mathsf{SRAKE},\mu,\ell,\mathcal{A}}$:
$\mathsf{pp}_{\mathsf{SR}} \leftarrow \mathsf{SR.Setup}$
**for** $i \in [\mu]$:
    $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{SR.Gen}(\mathsf{pp}_{\mathsf{SR}}, \mathsf{P}_i)$;
    $crp_i :=$ **false**                                  //Corruption variable
$\mathsf{PKList} := \{\mathsf{pk}_i\}_{i \in [\mu]}$; $b \leftarrow\!\$ \{0,1\}$
**for** $(i,s) \in [\mu] \times [\ell]$:
    $\mathsf{var}_i^s := (\mathsf{Pid}_i^s, k_i^s, \Psi_i^s) := (\varnothing, \varnothing, \varnothing)$;
    $\mathsf{Aflag}_i^s :=$ **false**   //Whether $\mathsf{Pid}_i^s$ is corrupted when $\pi_i^s$ accepts
    $T_i^s :=$ **false**; $kRev_i^s :=$ **false**          // Test, Key Reveal variables
$b^* \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{SR}}(\cdot)}(\mathsf{pp}_{\mathsf{SR}}, \mathsf{PKList})$

$\mathsf{Win}_{\mathsf{Auth}} :=$ **false**
$\mathsf{Win}_{\mathsf{Auth}} :=$ **true**, If $\exists (i,s) \in [\mu] \times [\ell]$ s.t.
(1) $\Psi_i^s =$ **accept**                                  //$\pi_i^s$ is $\tau$-accepted
(2) $\mathsf{Aflag}_i^s =$ **false** //$\mathsf{P}_j$ is $\hat{\tau}$-corrupted with $j := \mathsf{Pid}_i^s$ and $\hat{\tau} > \tau$
(3) (3.1) $\vee$ (3.2) $\vee$. Let $j := \mathsf{Pid}_i^s$
    (3.1) $\nexists\, t \in [\ell]$ s.t. $\mathsf{Partner}(\pi_i^s \leftarrow \pi_j^t)$
    (3.2) $\exists\, t \in [\ell], (j', t') \in [\mu] \times [\ell]$ with $(j,t) \neq (j',t')$ s.t.
        $\mathsf{Partner}(\pi_i^s \leftarrow \pi_j^t) \wedge \mathsf{Partner}(\pi_i^s \leftarrow \pi_{j'}^{t'})$

$\mathsf{Win}_{\mathsf{Ind}} :=$ **false**
**if** $b^* = b$:
    $\mathsf{Win}_{\mathsf{Ind}} :=$ **true**; **return** 1
**else** : **return** 0

$\mathsf{Partner}(\pi_i^s \leftarrow \pi_j^t)$:            //Checking whether $\mathsf{Partner}(\pi_i^s \leftarrow \pi_j^t)$
**if** $\pi_i^s$ sent the first message and $k_i^s = \mathsf{K}(\pi_i^s, \pi_j^t) \neq \varnothing$:
    **return** 1
**if** $\pi_i^s$ received the first message and $k_i^s = \mathsf{K}(\pi_j^t, \pi_i^s) \neq \varnothing$:
    **return** 1
**return** 0

$\pi_i^s(\mathsf{msg}, j)$:  // $\pi_i^s$ executes the protocol according to the specification
**if** $\mathsf{Pid}_i^s = \varnothing$: $\mathsf{Pid}_i^s := j$
**if** $\mathsf{Pid}_i^s = j$:
    $\pi_i^s$ receives $\mathsf{msg}$ and uses $\mathsf{res}_i$, $\mathsf{var}_i^s$ to generate the next
    message $\mathsf{msg}'$ of SRAKE, and updates $(\mathsf{Pid}_i^s, k_i^s, \Psi_i^s)$;
    **if** $\mathsf{msg} = \top$: $\pi_i^s$ generates the first message $\mathsf{msg}'$ as initiator;
    **if** $\mathsf{msg}$ is the last message of SRAKE: $\mathsf{msg}' := \varnothing$;
    **return** $\mathsf{msg}'$
**if** $\mathsf{Pid}_i^s \neq j$: **return** $\perp$

$\mathcal{O}_{\mathsf{SR}}(\mathsf{query})$:
**if** query=$\mathsf{Send}(i, s, j, \mathsf{msg})$:
    **if** $\Psi_i^s =$ **accept**: **return** $\perp$
    $\mathsf{msg}' \leftarrow \pi_i^s(\mathsf{msg}, j)$
    **if** $\Psi_i^s =$ **accept**:
        **if** $crp_j =$ **true**: $\mathsf{Aflag}_i^s :=$ **true**;
    **return** $\mathsf{msg}'$

**if** query=$\mathsf{Corrupt}(i)$:
    **if** $i \notin [\mu]$: **return** $\perp$
    $crp_i :=$ **true**
    **return** $\mathsf{sk}_i$

**if** query=$\mathsf{RegisterCorrupt}(u, \mathsf{pk}_u)$:
    **if** $u \in [\mu]$: **return** $\perp$
    $\mathsf{PKList} := \mathsf{PKList} \cup \{\mathsf{pk}_u\}$
    $crp_u :=$ **true**
    **return** $\mathsf{PKList}$

**if** query=$\mathsf{SessionKeyReveal}(i, s)$:
    **if** $\Psi_i^s \neq$ **accept**: **return** $\perp$
    **if** $T_i^s =$ **true**: **return** $\perp$      //avoid **TA2**
    Let $j := \mathsf{Pid}_i^s$
    **if** $\exists t \in [\ell]$ s.t. $\mathsf{Partner}(\pi_i^s \leftrightarrow \pi_j^t)$:
        **if** $T_j^t =$ **true**: **return** $\perp$  //avoid **TA4**
    $kRev_i^s :=$ **true**; **return** $k_i^s$

**if** query=$\mathsf{Subv}(i, \mathit{\Pi}, \widetilde{\mathit{\Pi}})$:
    $\mathsf{res}_i := (\mathsf{sk}_i, \mathsf{pp}_{\mathsf{SR}}, \mathsf{PKList}, \widetilde{\mathit{\Pi}})$
    **return** $\perp$          //Nothing gets returned

**if** query=$\mathsf{Test}(i, s)$:
    **if** $\Psi_i^s \neq$ **accept** $\vee$ $\mathsf{Aflag}_i^s =$ **true**
     $\vee$ $kRev_i^s =$ **true** $\vee$ $T_i^s =$ **true**:
        **return** $\perp$  //avoid **TA1**, **TA2**, **TA3**
    Let $j := \mathsf{Pid}_i^s$
    **if** $\exists t \in [\ell]$ s.t. $\mathsf{Partner}(\pi_i^s \leftrightarrow \pi_j^t)$ :
        **if** $kRev_j^t =$ **true** $\vee$ $T_j^t =$ **true**:
            **return** $\perp$          //avoid **TA4**, **TA5**
    $T_i^s :=$ **true**; $k_0 := k_i^s$; $k_1 \leftarrow\!\$ \mathcal{K}$;
    **return** $k_b$
```

Fig. 1. The security experiment $\mathsf{Exp}_{\mathsf{SRAKE},\mu,\ell,\mathcal{A}}$. The list of trivial attacks is given in Table 1.

Adversary $\mathcal{A}$ may copy, delay, erase, and interpolate the messages transmitted in the network. This is formalized by the query Send to oracle π_i^s. With Send, $\mathcal{A}$ can send arbitrary messages to any oracle π_i^s. Then π_i^s will execute the SRAKE protocol according to the protocol specification $\mathit{\Pi}$ for P_i.

We also allow the adversary to observe session keys of its choices. This is reflected by a $\mathsf{SessionKeyReveal}$ query to oracle π_i^s.

A $\mathsf{Corrupt}$ query allows $\mathcal{A}$ to corrupt a party P_i and get its long-term secret key sk_i. With a $\mathsf{RegisterCorrupt}$ query, $\mathcal{A}$ can register a new party without public key certification. The public key is then known to all other users.

A query to Subv allows $\mathcal{A}$ to subvert a user and substitute an algorithm or the entire protocol specification.

We introduce a Test query to formalize the pseudorandomness of k_i^s. Therefore, the challenger chooses a bit $b \leftarrow_{\$} \{0,1\}$ at the beginning of the experiment. When $\mathcal{A}$ issues a Test query for π_i^s, the oracle will return $\perp$ if the session key k_i^s is not generated yet. Otherwise, π_i^s will return k_i^s or a random key, depending on b. The task of $\mathcal{A}$ is to tell whether the key is the true session key or a random key. The adversary is allowed to make multiple test queries.

Formally, the queries by $\mathcal{A}$ are described as follows.

- $\mathsf{Send}(i,s,j,\mathsf{msg})$: If $\mathsf{msg} = \top$, it means that $\mathcal{A}$ asks oracle π_i^s to send the first protocol message to P_j. Otherwise, $\mathcal{A}$ impersonates P_j to send message msg to π_i^s. Then π_i^s executes the SRAKE protocol with msg as P_i does, computes a message msg', and updates its own variables $\mathsf{var}_i^s = (\mathsf{Pid}_i^s, k_i^s, \Psi_i^s)$. The output message msg' is returned to $\mathcal{A}$.
  If $\mathsf{Send}(i,s,j,\mathsf{msg})$ is the τ-th query asked by $\mathcal{A}$ and π_i^s changes Ψ_i^s to **accept** after that, then we say that π_i^s is *τ-accepted.*
- $\mathsf{Corrupt}(i)$: $\mathcal{C}$ reveals party P_i's long-term secret key sk_i to $\mathcal{A}$. After corruption, $\pi_i^1, ..., \pi_i^\ell$ will stop answering queries from $\mathcal{A}$.
  If $\mathsf{Corrupt}(i)$ is the τ-th query asked by $\mathcal{A}$, we say that P_i is *τ-corrupted.*
  If $\mathcal{A}$ has never asked $\mathsf{Corrupt}(i)$, we say that P_i is *∞-corrupted.*
- $\mathsf{RegisterCorrupt}(i, \mathsf{pk}_i)$: It means that $\mathcal{A}$ registers a new party P_i $(i > \mu)$. $\mathcal{C}$ distributes $(\mathsf{P}_i, \mathsf{pk}_i)$ to all users. In this case, we say that P_i is *0-corrupted.*
- $\mathsf{SessionKeyReveal}(i,s)$: The query means that $\mathcal{A}$ asks $\mathcal{C}$ to reveal π_i^s's session key. If $\Psi_i^s \neq$ **accept**, $\mathcal{C}$ returns $\perp$. Otherwise, $\mathcal{C}$ returns the session key k_i^s of π_i^s.
- $\mathsf{Subv}(i, \Pi, \widetilde{\Pi})$: replaces P_i's algorithm or protocol specification Π with $\widetilde{\Pi}$.
- $\mathsf{Test}(i,s)$: If $\Psi_i^s \neq$ **accept**, $\mathcal{C}$ returns $\perp$. Otherwise, $\mathcal{C}$ sets $k_0 = k_i^s$, samples $k_1 \leftarrow_{\$} \mathcal{K}$, and returns k_b to $\mathcal{A}$. We require that $\mathcal{A}$ can ask $\mathsf{Test}(i,s)$ to each oracle π_i^s only once.

Informally, the pseudorandomness of k_i^s asks that any PPT adversary $\mathcal{A}$ with access to $\mathsf{Test}(i,s)$ cannot distinguish k_i^s from a uniformly random key. Yet, we have to exclude some trivial attacks. We will define them later and first introduce partnering.

Definition 5 (Original Key [LS17]). *For two oracles π_i^s and π_j^t, the original key, denoted as $\mathsf{K}(\pi_i^s, \pi_j^t)$, is the session key computed by the two peers of the protocol under a passive adversary only, where π_i^s is the initiator.*

Remark 1. We note that $\mathsf{K}(\pi_i^s, \pi_j^t)$ is determined by the identities of P_i and P_j and the internal randomness.

Types	Trivial attacks	Explanation
TA1	$T_i^s = \textbf{true} \wedge \mathsf{Aflag}_i^s = \textbf{true}$	π_i^s is tested but π_i^s's partner is corrupted when π_i^s accepts session key k_i^s
TA2	$T_i^s = \textbf{true} \wedge kRev_i^s = \textbf{true}$	π_i^s is tested and its session key k_i^s is revealed
TA3	$T_i^s = \textbf{true}$ when $\mathsf{Test}(i, s)$ is queried	$\mathsf{Test}(i, s)$ is queried at least twice
TA4	$T_i^s = \textbf{true} \wedge \mathsf{Partner}(\pi_i^s \leftrightarrow \pi_j^t) \wedge kRev_j^t = \textbf{true}$	π_i^s is tested, π_i^s and π_j^t are partnered to each other, and π_j^t's session key k_j^t is revealed
TA5	$T_i^s = \textbf{true} \wedge \mathsf{Partner}(\pi_i^s \leftrightarrow \pi_j^t) \wedge T_j^t = \textbf{true}$	π_i^s is tested, π_i^s and π_j^t are partnered to each other, and π_j^t is tested

Table 1. Trivial attacks **TA1**-**TA5** for security experiment $\mathsf{Exp}_{\mathsf{SRAKE},\mu,\ell,\mathcal{A}}$. Note that "$\mathsf{Aflag}_i^s = \textbf{false}$" is implicitly contained in **TA2**-**TA5** because of **TA1**.

Definition 6 (Partner [LS17]). *Let $\mathsf{K}(\cdot, \cdot)$ denote the original key function. We say that an oracle π_i^s is partnered to π_j^t, denoted as $\mathsf{Partner}(\pi_i^s \leftarrow \pi_j^t)$*[3]*, if one of the following requirements holds:*

- *π_i^s has sent the first message and $k_i^s = \mathsf{K}(\pi_i^s, \pi_j^t) \neq \varnothing$, or*
- *π_i^s has received the first message and $k_i^s = \mathsf{K}(\pi_j^t, \pi_i^s) \neq \varnothing$.*

We write $\mathsf{Partner}(\pi_i^s \leftrightarrow \pi_j^t)$ if $\mathsf{Partner}(\pi_i^s \leftarrow \pi_j^t)$ and $\mathsf{Partner}(\pi_j^t \leftarrow \pi_i^s)$.

Trivial Attacks. In order to prevent the adversary from trivial attacks, we keep track of the following variables for each party P_i and oracle π_i^s:

- crp_i: whether P_i is corrupted.
- Aflag_i^s: whether the intended partner is corrupted when π_i^s accepts.
- T_i^s: whether π_i^s was tested.
- $kRev_i^s$: whether the session key k_i^s was revealed.

Based on that we give a list of trivial attacks **TA1**-**TA5** in Table 1.

Definition 7 (Security of SRAKE). *Let μ be the number of users and ℓ the maximum number of protocol executions per user. The security experiment $\mathsf{Exp}_{\mathsf{SRAKE},\mu,\ell,\mathcal{A}}$ (see Fig. 1) is played between the challenger $\mathcal{C}$ and the adversary $\mathcal{A}$ in the following way:*

1. *$\mathcal{C}$ runs $\mathsf{SR.Setup}$ to get SRAKE public parameter $\mathsf{pp}_{\mathsf{SR}}$.*
2. *For each party P_i, $\mathcal{C}$ runs $\mathsf{SR.Gen}(\mathsf{pp}_{\mathsf{SR}}, \mathsf{P}_i)$ to get the long-term key pair $(\mathsf{pk}_i, \mathsf{sk}_i)$. Next it chooses a random bit $b \leftarrow_{\$} \{0, 1\}$ and provides $\mathcal{A}$ with the public parameter $\mathsf{pp}_{\mathsf{SR}}$ and the list of public keys $\mathsf{PKList} := \{\mathsf{pk}_i\}_{i \in [\mu]}$.*
3. *$\mathcal{A}$ queries Send, $\mathsf{Corrupt}$, $\mathsf{RegisterCorrupt}$, $\mathsf{SessionKeyReveal}$, Subv and Test to $\mathcal{C}$ adaptively.*
4. *At the end of the experiment, $\mathcal{A}$ terminates with an output b^*.*

- **Strong Authentication.** *Let $\mathsf{Win}_{\mathsf{Auth}}$ denote the event that $\mathcal{A}$ breaks authentication in the security experiment. $\mathsf{Win}_{\mathsf{Auth}}$ happens iff $\exists (i, s) \in [\mu] \times [\ell]$ s.t.*
  (1) *π_i^s is τ-accepted.*
  (2) *P_j is $\hat{\tau}$-corrupted with $j := \mathsf{Pid}_i^s$ and $\hat{\tau} > \tau$.*

[3] The arrow notion $\pi_i^s \leftarrow \pi_j^t$ means π_i^s (not necessarily π_j^t) has computed and accepted the original key.

(3) *Either (3.1) or (3.2)happens*[1]*Let* $j := \mathsf{Pid}_i^s$.

  (3.1) *There is no oracle* π_j^t *that* π_i^s *is partnered to.*

  (3.2) *There exist two distinct oracles* π_j^t *and* $\pi_{j'}^{t'}$, *to which* π_i^s *is partnered.*

- **Indistinguishability.** *Let* $\mathsf{Win}_{\mathsf{Ind}}$ *denote the event that* $\mathcal{A}$ *breaks indistinguishability in the experiment* $\mathsf{Exp}_{\mathsf{SRAKE},\mu,\ell,\mathcal{A}}$ *above. Let* b^* *be* $\mathcal{A}$*'s output. Then* $\mathsf{Win}_{\mathsf{Ind}}$ *happens iff* $b^* = b$. *Trivial attacks are already considered during the execution of the experiment. A list of trivial attacks is given in Table 1.*

Note that $\mathsf{Exp}_{\mathsf{SRAKE},\mu,\ell,\mathcal{A}} \Rightarrow 1$ *iff* $\mathsf{Win}_{\mathsf{Ind}}$ *happens. Hence, the advantage of* $\mathcal{A}$ *is defined as*

$$\begin{aligned}\mathsf{Adv}_{\mathsf{SRAKE},\mu,\ell}(\mathcal{A}) :&= \max\{\Pr[\mathsf{Win}_{\mathsf{Auth}}], |\Pr[\mathsf{Win}_{\mathsf{Ind}}] - 1/2|\}\\ &= \max\{\Pr[\mathsf{Win}_{\mathsf{Auth}}], |\Pr[\mathsf{Exp}_{\mathsf{SRAKE},\mu,\ell,\mathcal{A}} \Rightarrow 1] - 1/2|\}.\end{aligned}$$

Remark 2 (Perfect Forward Security and KCI Resistance). The security model of AKE supports (perfect) forward security (a.k.a. forward secrecy [Gün90]). That is, if P_i or its partner P_j has been corrupted at some moment, then the exchanged session keys computed before the corruption remain hidden from the adversary. Meanwhile, π_i^s may be corrupted before $\mathsf{Test}(i,s)$, which provides resistance to key-compromise impersonation (KCI) attacks [Kra05].

3.3 Goals for SAs

We continue by defining *undetectability* of subverted algorithms. Note that we don't specify the inputs to the algorithms, as we want to play the same game for various algorithms which take different inputs. Recall that we write generating an output by running an algorithm Π on some input as $Y \leftarrow \Pi(X)$. We assume that a subverted algorithm has access to the same or sometimes additional inputs as the original algorithm. In the case where some specific input, say a secret key sk, is important, we write this as $Y \leftarrow \Pi(\mathsf{sk}, X)$.

Definition 8 (Undetectability). *A subverted algorithm* $\widetilde{\Pi}$ *is* (t, ε, q)*-undetectable iff for all detection algorithms* $\mathcal{D}$ *playing the game in Figure 2 and running in time t, we have*

$$\left|\Pr[\mathsf{Detect}^{\mathcal{D}}_{\Pi,\widetilde{\Pi}} \Rightarrow 1] - \frac{1}{2}\right| \leq \varepsilon.$$

Other works [AMV15, BJK15] specify that a successful adversary should also recover a key from the subverted party. We don't set this as a required goal for a successful subversion in this work, as a key recovery will eventually break the security of, and hence is already captured by, the standard definitions for AKE security.

One could also argue that such a subversion closely mirrors the intended functionality of the corruption oracle already included in AKE models, and as such should not be of any more concern than regular corruptions. However, due to the effectiveness and simplicity of the attack, it could enable corruptions on a large scale, which is clearly something we would like to avoid, even if the model allows it.

[1] Given $(1) \wedge (2)$, (3.1) indicates a successful impersonation of P_j, (3.2) suggests one instance of P_i has multiple partners.

```
GAME $\mathsf{Detect}^{\mathcal{D}}_{\Pi,\widetilde{\Pi}}$
$b \leftarrow_\$ \{0,1\}$
$\mathsf{cnt} := 0$
$b' \leftarrow \mathcal{D}^{\textsc{Run}}$
return $[\![ b = b' ]\!] \wedge \mathsf{cnt} \le q$

$\textsc{Run}(X)$
$\mathsf{cnt}$ ++
if $b = 1$
    $Y \leftarrow \Pi(X)$
else $Y \leftarrow \widetilde{\Pi}(X)$
return $Y$
```

Fig. 2. Game Detect for algorithm Π, where X is some unspecified input.

We will still formally define key recovery under subversion for two message AKE, as it will make discussions easier. Note that in this game we only let our adversary act passively, and he is thus unable to edit the messages being sent. This is because the fewer abilities the adversary has, the stronger a successful attack becomes. Hence, the adversary is effectively limited to subverting algorithms and creating protocol transcripts, which extends to multi-round protocols in a natural way.

The relevant oracles in the security game for AKE are defined analogously to those for SRAKE in Figure 1, i.e. $\mathcal{O}_{\mathsf{AKE}}(\mathsf{query}) = \mathcal{O}_{\mathsf{SR}}(\mathsf{query})$. We write $\mathcal{O}_{\mathsf{AKE}}(\mathsf{Send}')$ for the adjusted send oracle where the adversary is not able to edit the messages being sent, but can redirect, interleave, and otherwise change the flow of the protocol.

Definition 9 (Private key recovery). *Let $\mathcal{A}$ be a passive adversary against an AKE protocol* $\mathsf{AKE} = (\mathsf{AKE.Setup}, \mathsf{AKE.Gen}, \mathsf{AKE.Protocol})$ *with μ parties $\mathsf{P}_1, \ldots, \mathsf{P}_\mu$. We say that $\mathcal{A}$ (t, ε, μ, S)-recovers the private key of a party P_i if $\mathcal{A}$ runs in time t with μ users, creates at most S session oracles π_i^s, and $\Pr[\mathsf{KR}^{\mathcal{A}} \Rightarrow 1] \ge \varepsilon$ in the* KR *game described in Figure 3.*

```
GAME $\mathsf{KR}^{\mathcal{A}}$
$\mathsf{pp}_{\mathsf{AKE}} \leftarrow \mathsf{AKE.Setup}$
for $i \in [\mu]$:
    $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{AKE.Gen}(\mathsf{pp}_{\mathsf{SR}}, \mathsf{P}_i)$
$\mathsf{PKList} := \{\mathsf{pk}_i\}_{i \in [\mu]}$
$(i, \mathsf{sk}') \leftarrow \mathcal{A}^{\mathrm{O}}(\mathsf{pp}_{\mathsf{AKE}}, \mathsf{PKList})$
return $[\![ \mathsf{sk}_i = \mathsf{sk}' ]\!]$
```

Fig. 3. Game KR for AKE. Adversary $\mathcal{A}$ has access to oracles $\mathrm{O} := \{\mathcal{O}_{\mathsf{AKE}}(\mathsf{Send}'), \mathsf{Subv}\}$ defined in Figure 1 and the text above.

4 Breaking Probabilistic Explicitly Authenticated Key Exchange

Recall that for implicit authentication, we require that only the owner of a corresponding private key can obtain the shared session key, while for explicit authentication we additionally require *key confirmation*, i.e. "the property whereby one party is assured that a second (possibly unidentified) party actually has possession of a particular secret key" [MVOV18].

We argue that for any protocol achieving explicit authentication, at least one message in the transcript must depend on the secret key sk of the authenticated party. If this was not the case, then one could obtain explicit authentication using only public/random information. This further implies that anyone could impersonate the party, as proving key confirmation could be done without using, and therefore also without access to, said key.

Note that this is not a problem for implicit authentication, as Eve can initiate a session with Bob claiming to be Alice, in which the implicit authentication then, from Bobs point of view, implies that only the owner of Alice's private key can obtain the session key. Hence, Eve never actually obtains the key.

We adapt the stateless attacks described in [BPR14, BJK15, AMV15] to our setting to break the security of any protocol which inputs the long term secret key of a party when randomly generating parts of the transcript. With the above discussion in mind, we then argue that the attack breaks any one-round, probabilistic, explicit authentication protocol.

4.1 Pseudorandom Functions

We recall the definition and security goals of a pseudorandom function (PRF). Let $F\colon \{0,1\}^\kappa \times \mathcal{X} \to \mathcal{Y}$ be a function which inputs a key $k \in \{0,1\}^\kappa$ and an element $x \in \mathcal{X}$, and outputs an element $y \in \mathcal{Y}$. We will often use the compressed notation $F_k(x) := F(k,x)$.

Definition 10 (Pseudorandom Function). *A function $F\colon \{0,1\}^\kappa \times \mathcal{X} \to \mathcal{Y}$ is a (t,ε,q)-secure pseudorandom function if for all adversaries $\mathcal{A}$ running in time t we have*

$$\left| \Pr\left[\mathcal{A}^{F_k(\cdot)}(1^\kappa) \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{f(\cdot)}(1^\kappa) \Rightarrow 1\right] \right| \leq \varepsilon \qquad (2)$$

where $f(\cdot)$ is a random function $f\colon \mathcal{X} \to \mathcal{Y}$, and the adversary $\mathcal{A}$ queries the oracle at most q times.

4.2 Generic attack description

To construct the attack, let $\ell = |\mathsf{sk}|$ be the length of the secret key, and let $F_k\colon \{0,1\}^* \to \{0,1\} \times [\ell]$ be a PRF with a pre-specified key k chosen by an adversary. Finally let $\widetilde{\Pi_{k,\tau}}$ with $\tau \in \mathbb{N}$ be the subverted implementation described in Figure 4.

The intuition behind the attack is that when $\widetilde{\Pi_{k,\tau}}$ returns Y, evaluating the PRF F_k on Y (or parts of it) returns (v,t) such that v is the t'th bit of sk. Given that the adversary knows F_k and Y is a public ciphertext, the adversary hence learns the t'th bit of sk.

4.3 Attack on one-round AKE

From the paragraphs at the start of this section, it follows that the attack can be mounted against any one-round, probabilistic, explicitly authenticated key exchange protocol $\mathsf{AKE} = (\mathsf{AKE.Setup}, \mathsf{AKE.Gen}, \mathsf{AKE.Protocol})$ and with high probability win the key

```
$\widetilde{\Pi_{k,\tau}}(\mathsf{sk})$
00 $X \leftarrow \mathcal{M}$      //$X$ can also be given as input to the algorithm.
01 $j := 0$
02 while $j < \tau$
03     $r \leftarrow_\$ \mathcal{R}$
04     $Y := \Pi(\mathsf{sk}, X; r)$
05     $(v, t) := F_k(Y)$
06     if $\mathsf{sk}[t] = v$
07         endwhile
08     $j$ ++
09 return $Y$
```

Fig. 4. Description of subverted algorithm $\widetilde{\Pi_{k,\tau}}$, where k is a PRF-key, $\tau \in \mathbb{N}$, and Π is a probabilistic algorithm with message space $\mathcal{M}$ and randomness space $\mathcal{R}$.

recovery game in the subversion model as described in Figure 3. Note that by the argument at the beginning of Section 4, we know that the protocol must use the target key sk_i at some point, and hence we highlight this required access. For protocols which use a signature scheme this follows naturally, but we note that our approach also works for protocols which may not. This access implies that the check in line 06 of Figure 4 is meaningful.

To illustrate the attack and provide a concrete analysis, we run it against the signed Diffie-Hellman protocol in [PQR22], where the signing algorithm is probabilistic. The protocol is presented in Figure 5, where $\mathsf{pp}_{\mathsf{AKE}}$ is computed as in lines 01 to 03 in Figure 6. The subversion will explicitly target a party's initiate algorithm Init by subverting the underlying Sign algorithm, but as we will discuss in Section 5, there are several different choices and combinations that would work.

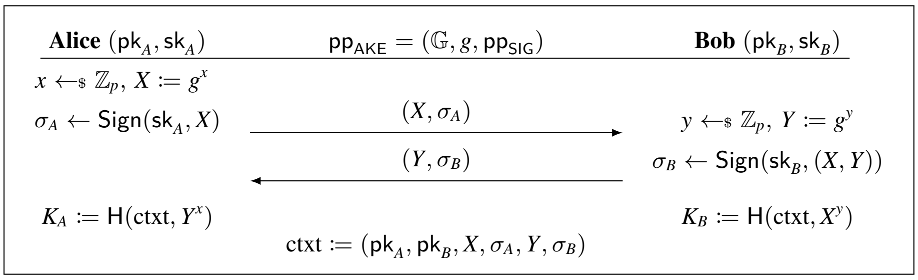
Alice $(\mathsf{pk}_A, \mathsf{sk}_A)$
$\mathsf{pp}_{\mathsf{AKE}} = (\mathbb{G}, g, \mathsf{pp}_{\mathsf{SIG}})$
Bob $(\mathsf{pk}_B, \mathsf{sk}_B)$
$x \leftarrow_\$ \mathbb{Z}_p,\ X := g^x$
$\sigma_A \leftarrow \mathsf{Sign}(\mathsf{sk}_A, X)$
(X, σ_A)
$y \leftarrow_\$ \mathbb{Z}_p,\ Y := g^y$
(Y, σ_B)
$\sigma_B \leftarrow \mathsf{Sign}(\mathsf{sk}_B, (X, Y))$
$K_A := \mathsf{H}(\mathrm{ctxt}, Y^x)$
$K_B := \mathsf{H}(\mathrm{ctxt}, X^y)$
$\mathrm{ctxt} := (\mathsf{pk}_A, \mathsf{pk}_B, X, \sigma_A, Y, \sigma_B)$

Fig. 5. Regular signed Diffie-Hellman key exchange protocol. All signatures are verified upon arrival with the corresponding messages, and the protocol aborts if any verification fails.

In Figure 6 we concretely define the algorithms in the protocol and specify the protocol flow $\mathsf{AKE.Protocol} = (\mathsf{Init}, \mathsf{Der_R}, \mathsf{Der_I})$, to illustrate how the oracle $\pi_i^s(\mathsf{msg}, j)$ behaves with respect to different $\mathcal{O}_{\mathsf{SR}}(\mathsf{Send})$ queries.

The analysis is very similar to that of Theorem 1 in [AMV15], in that their attack on signature schemes naturally carries over to when they are deployed in an AKE.

```
AKE.Setup():
    Choose $\mathbb{G} = \langle g \rangle$ with $|\mathbb{G}| = p$ for prime $p$
    $\mathsf{pp_{SIG}} \leftarrow \mathsf{SIG.Setup}$
    return $\mathsf{pp_{AKE}} = (\mathbb{G}, g, \mathsf{pp_{SIG}})$

AKE.Gen($\mathsf{pp_{AKE}}, \mathsf{P}_i$):
    return $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{SIG.Gen}(\mathsf{pp_{SIG}})$

Init($\mathsf{sk}_i, \top$):
    $\mathsf{st} := x \leftarrow_\$ \mathbb{Z}_p$
    $X := g^x$
    $\sigma_i \leftarrow \mathsf{Sign}(\mathsf{sk}_i, X)$
    return $(X, \sigma_i, \mathsf{st})$

$\mathsf{Der_R}(\mathsf{sk}_r, X, \sigma_i)$
    if $\mathsf{Ver}(\mathsf{pk}_i, X, \sigma_i) = 0$
        return $\perp$
    $y \leftarrow_\$ \mathbb{Z}_p$
    $Y := g^y$
    $\sigma_r \leftarrow \mathsf{Sign}(\mathsf{sk}_r, (X, Y))$
    $K := \mathsf{H}(\mathrm{ctxt}, X^y)$
    return $(Y, \sigma_r, K)$

$\mathsf{Der_I}(X, Y, \sigma_r, \mathsf{st})$
    if $\mathsf{Ver}(\mathsf{pk}_r, (X, Y), \sigma_r) = 0$
        return $\perp$
    $K := \mathsf{H}(\mathrm{ctxt}, Y^{\mathsf{st}})$
    return $K$
```

Fig. 6. A signed Diffie-Hellman protocol $\mathsf{AKE_{DH}} = (\mathsf{AKE.Setup}, \mathsf{AKE.Gen}, \mathsf{AKE.Protocol})$, with the specification $\mathsf{AKE.Protocol} = (\mathsf{Init}, \mathsf{Der_R}, \mathsf{Der_I})$ and $\mathrm{ctxt} := (\mathsf{pk}_i, \mathsf{pk}_r, X, \sigma_i, Y, \sigma_r)$.

Theorem 1. *Let* $\mathsf{AKE_{DH}} = (\mathsf{AKE.Setup}, \mathsf{AKE.Gen}, \mathsf{AKE.Protocol})$ *be a signed Diffie-Hellman protocol like in Figure 5, with* $\mathsf{AKE.Protocol}$ *specified by the algorithms* $\mathsf{Init}, \mathsf{Der_R}$, *and* $\mathsf{Der_I}$. *Let* SIG *be the probabilistic signature scheme used in* $\mathsf{AKE_{DH}}$, *with signature space* Σ, *randomness space* $\mathcal{R}$, *signing key space* $\mathcal{SK} = \{0,1\}^\ell$ *and min-entropy* η. *Let* $F\colon \{0,1\}^\kappa \times \Sigma \to \{0,1\} \times [\ell]$ *be a* $(t_{\mathsf{PRF}}, \varepsilon_{\mathsf{PRF}}, q_{\mathsf{PRF}})$*-secure PRF.*

(i) Then there exists an adversary $\mathcal{A}$ *that* (t, ε, μ, S)*-recovers the private key of a party* P_i *with*

$$\varepsilon \geq 1 - \varepsilon_{\mathsf{PRF}} - \frac{S^2\tau^2}{2} \cdot 2^{-\eta} - S \cdot 2^{-\tau} - \ell \cdot e^{-S/\ell}, \qquad (3)$$

when $q_{\mathsf{PRF}} \geq S \cdot \tau$ *and* t_{PRF} *is at least* S *times the running time of computing* $X = g^x$ *plus* $S \cdot \tau$ *times the running time of running* Sign *on* X. *The running time of* $\mathcal{A}$ *is at most* S *times the running time of computing* F_k. *The running time of the subverted party* P_i *is at most* τ *times the running time of* Sign *and* F_k *for each of the* S *sessions.*

(ii) The algorithm $\widetilde{\mathsf{Init}_{k,\tau}}$ *in Figure 7 is* $(t_{\mathsf{PRF}}, \varepsilon_{\mathsf{PRF}} + S^2\tau^2 \cdot 2^{-(\eta+1)}, q_{\mathsf{PRF}})$*-undetectable for* $q_{\mathsf{PRF}}, t_{\mathsf{PRF}}$ *as above.*

Proof. (Part i) The adversary interacts with a challenger as in the KR game in Figure 3. The KR challenger begins by running the setup and key generation algorithms to create public parameters $\mathsf{pp_{AKE}}$ and key-pairs $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{AKE.Gen}(\mathsf{pp_{AKE}}, \mathsf{P}_i)$ for all $i \in [\mu]$. Then it sends the public parameters and keys to the adversary.

The adversary starts by choosing a target P_i, and creates a subversion of the initiate algorithm Init by sampling a PRF-key $k \leftarrow_\$ \{0,1\}^\kappa$, and defining $\widetilde{\mathsf{Init}_{k,\tau}}$ as in Figure 7. Note that while we here restrict ourself to one target, it is possible to extend the approach to multiple targets.

Now $\mathcal{A}$ initializes $\mathsf{sk}'_i := 0^\ell$, and then queries $\mathsf{Subv}(i, \mathsf{Init}, \widetilde{\mathsf{Init}_{k,\tau}})$. Given that $\mathcal{A}$ only wishes to recover P_i's private key, it has no need to create session oracles that does not involve P_i, or to complete any session in which P_i has already computed a signature. For simplicity, we therefore assume that all S session oracles are with P_i as the initiator. Note that this means that we never actually complete any sessions, as this

```
$\widetilde{\mathsf{Init}_{k,\tau}}(\mathsf{sk}, \top)$
$x \leftarrow_\$ \mathbb{Z}_p$
$X := g^x$
$j := 0$
while $j < \tau$
    $r \leftarrow_\$ \mathcal{R}$
    $\sigma := \mathsf{Sign}(\mathsf{sk}, X; r)$
    $(v, t) := F_k(\sigma)$
    if $\mathsf{sk}_i[t] = v$
        endwhile
    $j$ ++
return $(X, \sigma)$
```

Fig. 7. Description of subverted algorithm $\widetilde{\mathsf{Init}_{k,\tau}}$, where k is a PRF-key, $\tau \in \mathbb{N}$, for the proof of Theorem 1.

would require us to create an additional session oracle for the intended partner Pid_i^s. For a realistic model, where sessions are completed and keys derived, it would be natural to restrict us to $S/2$ sessions for P_i and use the remaining $S/2$ for the intended partners.

Now, whenever a query to Send makes P_i output something, the output will be of the form $\mathsf{msg}' = (X_i, \sigma_i)$. When $\mathcal{A}$ receives (X_i, σ_i), he runs $(v, t) := F_k(\sigma_i)$, and sets $\mathsf{sk}'[t] = v$. After creating all S sessions, $\mathcal{A}$ returns (i, sk'). The running time of $\mathcal{A}$ is roughly that of S evaluations of F_k.

We analyze the success probability through a sequence of games in Figure 8.

```
GAMES $G_0$, $\boxed{G_1}$, $\boxed{G_2}$
$\mathsf{pp}_{\mathsf{AKE}} \leftarrow \mathsf{AKE.Setup}$
$(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{AKE.Gen}(\mathsf{pp}_{\mathsf{AKE}})$
$k \leftarrow_\$ \{0,1\}^\kappa$
$\boxed{\mathcal{L} := \varnothing}$
for $S$ times:
    $x \leftarrow_\$ \mathbb{Z}_p$,    $X := g^x$        //$p$ and $\mathbb{G} = \langle g \rangle$ implicitly defined by $\mathsf{pp}_{\mathsf{AKE}}$
    $j := 0$
    while $j < \tau$:
        $r \leftarrow_\$ \mathcal{R}$
        $\sigma := \mathsf{Sign}(\mathsf{sk}_i, X; r)$
        $(v, t) := F_k(\sigma)$
        $\boxed{\text{if } \mathcal{L}[\sigma] \text{ undefined: } (v,t) \leftarrow_\$ \{0,1\} \times [\ell] \text{ and set } \mathcal{L}[\sigma] := (v,t)}$
        $\boxed{(v,t) \leftarrow_\$ \{0,1\} \times [\ell]}$
        if $\mathsf{sk}_i[t] = v$
            endwhile
        $j$++
```

Fig. 8. Games G_0-G_2 for the proof of Theorem 1.

GAME G_0: In this game a challenger will run the computations that the subverted algorithm would do in the original KR-game. We define the event $E := E' \vee E''$, where E' and E'' are defined in the probability space of G_0:

- Event E': The event is true if for at least one of the S repetitions, the counting index j reaches τ, which means $\mathsf{sk}_i[t] \neq v$ for all the (v, t), in particular for the last signature generated.
- Event E'': The event is true if at the end of the S repetitions, the values v do not cover the entire set $[\ell]$, i.e. there exists an index $\hat{t}$ such that for all pairs (v, t), we have $t \neq \hat{t}$.

Note that the distribution of the pairs (v, t) in G_0 is identical to the one induced by running the described subversion adversary against the game in Definition 9. [2] Since E only depends on these pairs, we have that $\Pr_{G_0}[E] = \Pr_{\mathsf{KR}}[E]$. If the event E does not happen, each repetition terminates successfully by line 14 in Figure 8, and there are no empty indexes in $[\ell]$. Hence, in the KR-game, $\mathcal{A}$ recovers the signing key of P_i with probability one, and we therefore have $\Pr[\mathsf{KR}^{\mathcal{A}} = 1] \geq 1 - \Pr_{\mathsf{KR}}[E]$, and it remains to bound $\Pr_{\mathsf{KR}}[E] = \Pr_{G_0}[E]$.

GAME G_1: In this game, instead of computing the pairs as $(v, t) = F_k(\sigma)$, we sample them uniformly at random from $\{0, 1\} \times [\ell]$, unless σ was previously generated, in which case we return the previously sampled pair. Again following [AMV15], we claim that

$$\left| \Pr_{G_0}[E] - \Pr_{G_1}[E] \right| \leq \varepsilon_{\mathsf{PRF}}. \tag{4}$$

To sketch a proof, we construct a distinguisher $\mathcal{D}$ for the statement in Definition 10. The distinguisher creates its own parameters $\mathsf{pp}_{\mathsf{AKE}} \leftarrow \mathsf{AKE.Setup}$ and a key pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{AKE.Gen}(\mathsf{pp}_{\mathsf{AKE}})$, and then computes signatures as in lines 04 to 09 in Figure 8. For each repetition it forwards the computed signature to the challenge oracle and receives a pair (v, t). Finally it uses the received pairs to check whether event E has happened or not. If E happened it outputs 1, otherwise it outputs 0. The distinguisher thus queries the oracle $S \cdot \tau \leq q_{\mathsf{PRF}}$ times, and runs in time roughly S times the time it takes to compute a message $X = g^x$ plus $S \cdot \tau$ times the time it takes to compute a signature σ. If the PRF-oracle is an actual PRF $F_k(\cdot)$, the probability that $\mathcal{D}$ outputs 1 is the same as the probability of E happening in game G_0. If the oracle is a random function $f(\cdot)$, the probability is the same as the probability of E happening in game G_1. Hence we get

$$\left| \Pr_{G_0}[E] - \Pr_{G_1}[E] \right| = \left| \Pr[\mathcal{D}^{F_k(\cdot)} \Rightarrow 1] - \Pr[\mathcal{D}^{f(\cdot)} \Rightarrow 1] \right| \leq \varepsilon_{\mathsf{PRF}}. \tag{5}$$

GAME G_2: In this game, the pair (v, t) is sampled uniformly at random for each repetition, regardless of previous queries (line 12). Hence, the only way to distinguish between games G_1 and G_2 is if there for some input is a collision in the computation of the

[2] This is another reason for only considering sessions where P_i is the initiator, otherwise we either need to describe a message sampler which for each repetition samples either X or (X, Y) exactly as in the KR-game, or somehow otherwise argue that the distinction between such messages does not impact the distribution of (v, t).

signatures. Let W denote the event that a collision happens, and note that games G_1 and G_2 are identical if W does not happen. This means that $\Pr_{G_1}[E \wedge \neg W] = \Pr_{G_2}[E \wedge \neg W]$. Since the computation of signatures is independent of the pairs (v,t), we also have that $\Pr_{G_1}[W] = \Pr_{G_2}[W] = \Pr[W]$. By the difference lemma, we therefore have that

$$\left| \Pr_{G_1}[E] - \Pr_{G_2}[E] \right| \leq \Pr[W]. \tag{6}$$

We now want to bound $\Pr[W]$. By the min-entropy of SIG, we know that the worst case probability of a particular σ being hit is at most $2^{-\eta}$. There are at most $S \cdot \tau$ signatures generated, so a union bound gives

$$\Pr[W] \leq \binom{S \cdot \tau}{2} \cdot 2^{-\eta} \leq \frac{S^2\tau^2}{2} \cdot 2^{-\eta}, \tag{7}$$

and hence we get

$$\left| \Pr_{G_1}[E] - \Pr_{G_2}[E] \right| \leq \frac{S^2\tau^2}{2} \cdot 2^{-\eta}. \tag{8}$$

Finally, it remains to bound $\Pr_{G_2}[E]$. Note that in G_2, all pairs (v,t) are now drawn independently and uniformly at random. We analyze $E = E' \vee E''$ by looking at the sub-events:

- Event E': For each repetition, we have at most τ trials where in each the value v hits $\mathsf{sk}_i[t]$ with probability $1/2$. The probability of all τ trials failing is $2^{-\tau}$, and since there are S repetitions, we get $\Pr_{G_2}[E'] \leq S \cdot 2^{-\tau}$.
- Event E'': Each index in $[\ell]$ is hit with probability $1/\ell$ in each repetition, so the probability of never hitting a particular index at the end of S repetitions is $(1 - 1/\ell)^S \leq e^{-S/\ell}$. There are ℓ indexes, so a union bound gives $\Pr_{G_2}[E''] \leq \ell \cdot e^{-S/\ell}$.

By a union bound we get $\Pr_{G_2}[E] \leq S \cdot 2^{-\tau} + \ell \cdot e^{-S/\ell}$. Putting everything together, we get

$$\Pr[\mathsf{KR}^{\mathcal{A}} = 1] \geq 1 - \Pr_{\mathsf{KR}}[E] \geq 1 - \varepsilon_{\mathsf{PRF}} - \frac{S^2\tau^2}{2} \cdot 2^{-\eta} - S \cdot 2^{-\tau} - \ell \cdot e^{-S/\ell}, \tag{9}$$

which concludes part (i.)

(Part ii.) By part (ii) of Theorem 1 in [AMV15], it also follows that $\widetilde{\mathsf{Init}}_{k,\tau}$ is $(t_{\mathsf{PRF}}, \varepsilon_{\mathsf{PRF}} + S^2\tau^2 \cdot 2^{-(\eta+1)}, q_{\mathsf{PRF}})$-undetectable for $q_{\mathsf{PRF}} \geq S \cdot \tau$ and t_{PRF} is at least S times the running time of computing $X = g^x$ plus $S \cdot \tau$ times the running time of running Sign on X.

An adversary $\mathcal{A}$ in the Detect-game in Figure 2 has to distinguish whether he is interacting with a real or a subverted algorithm. We define games G_0 and G_1 as copies of the Detect-game with $b = 0$ and $b = 1$, and we need to show that G_0 and G_1 are computationally indistinguishable. The idea is to create intermediate games H_0-H_2 where we change the subversion as in Figure 8. Game H_0 is identical to G_0, in H_1 we draw $(v,t) \leftarrow_{\$} \{0,1\} \times [\ell]$ in a consistent way, and in H_2 we draw at random in each repetition.

For the adversary to win in H_0 and H_1, he needs to output $b' = 0$. We create a distinguisher $\mathcal{D}$ which creates parameters and computes signatures like the one used to derive Equation (4), and which on input b' outputs $1 - b'$. Depending on whether we are in G_0 or G_1, the challenge oracle is either a PRF $F_k(\cdot)$ or a random function $f(\cdot)$, and we get

$$\Pr_{H_0}[\mathcal{A} \text{ wins}] = \Pr_{H_0}[\mathcal{A} \Rightarrow 0] = \Pr[\mathcal{D}^{F_k(\cdot)} \Rightarrow 1]$$
$$\Pr_{H_1}[\mathcal{A} \text{ wins}] = \Pr_{H_1}[\mathcal{A} \Rightarrow 0] = \Pr[\mathcal{D}^{f(\cdot)} \Rightarrow 1],$$

which together imply

$$\left|\Pr_{H_0}[\mathcal{A} \text{ wins}] - \Pr_{H_1}[\mathcal{A} \text{ wins}]\right| = \left|\Pr[\mathcal{D}^{F_k(\cdot)} \Rightarrow 1] - \Pr[\mathcal{D}^{f(\cdot)} \Rightarrow 1]\right| \leq \varepsilon_{\mathsf{PRF}}. \quad (10)$$

Analogously to before, the games H_1 and H_2 are identical unless there is a collision in the computation of signatures, so we let W be this event and we get the same result as before:

$$\left|\Pr_{H_1}[\mathcal{A} \text{ wins}] - \Pr_{H_2}[\mathcal{A} \text{ wins}]\right| \leq \Pr[W] \leq \frac{S^2\tau^2}{2} \cdot 2^{-\eta}. \quad (11)$$

Finally we note that in H_2, the pair (v, t) is independent of the signature σ for each repetition, and hence does not affect the distribution. We therefore get that from the perspective of $\mathcal{A}$, the games are identical and therefore

$$\Pr_{H_2}[\mathcal{A} \text{ wins}] = \Pr_{G_1}[\mathcal{A} \text{ wins}], \quad (12)$$

and by combining the above equations we get

$$\left|\Pr_{G_0}[\mathcal{A} \text{ wins}] - \Pr_{G_1}[\mathcal{A} \text{ wins}]\right| \leq \varepsilon_{\mathsf{PRF}} + \frac{S^2\tau^2}{2} \cdot 2^{-\eta}. \quad (13)$$

□

5 Countermeasures and discussion

In the above attack, for easier analysis we only focused on one user and only on the initiation algorithm. Given that there are μ parties in the key exchange game, in a practical setting one could subvert both the Init and $\mathsf{Der_R}$ algorithm of several parties. Note also that while we required the entire key to be leaked for a successful attack, each leaked bit halves the search space for a brute force attack on the key.

It turns out that defending against the attack is more difficult than one might think. In Theorem 1, we targeted the Init algorithm through the signature algorithm Sign. Hence, one might think that the problem could be solved by using a subversion resilient signature scheme. As demonstrated in [AMV15], unique signatures (Definition 3) are subversion resilient under some fairly weak additional assumptions on the signatures generated. Using unique signatures, line 04 of Figure 7 becomes meaningless, which turns this particular attack impossible.

However, this is not quite enough to fix our problems. The attack relied on exploiting the randomness of the signature scheme, but we can modify this to instead exploit the randomization of the underlying key exchange. Such variants of the attack are harder to defend against, as non-randomized, i.e. static, key exchange does not provide features like forward secrecy.

In the first such modification, the subverted algorithm picks an $x \leftarrow_{\$} \mathbb{Z}_p$, compute $X = g^x$ and a signature $\sigma = \mathsf{Sign}(\mathsf{sk}, X)$, run the PRF to get $(v,t) = F_k(\sigma)$ and finally check if $\mathsf{sk}[t] = v$. If not, we start again with a new x. This makes the attack work again, at the cost of computing $S \cdot \tau$ instead of S key-shares X. We can modify this to get a more efficient version of the attack by noting that since X is sent along with the signature, we can just run the PRF on X instead to get $(v,t) = F_k(X)$. Like the previous one, this attack creates at most $S \cdot \tau$ random key-shares X, but only a single signature σ for each of the S sessions.

While the original attack focused on protocols using signatures, this change also means that we can apply the attack to any randomized explicitly authenticated key exchange protocol, due to the necessity of using the secret key at some point.

Hence, it seems like a required condition is to somehow separate the randomized part of an algorithm from the part where it has access to the secret key. But even if we could somehow generate an X in a secure way, i.e. making sure that the algorithm does not have access to the secret key sk when doing the computation, it is still not enough. Given that sk must be used at some point, regardless of whether the "honest" algorithm has access to it when computing X, the subverted algorithm can do the bit-check when, or after, using it. If the check fails, it will just restart and compute a new X in an "honest" manner. For the signed Diffie-Hellman above, the subversion creates a maximum of $S \cdot \tau$ key-shares X and signatures σ, as a new signature will have to be computed for each repetition.

5.1 Commitments

From the last argument, it becomes apparent that we need a way to make sure that once the random values are computed, the subverted algorithm cannot simply restart the process if the end result is not satisfactory.

This naturally leads us towards commitment schemes, and in general to "commit" to a value by splitting the protocol into multiple rounds, where we make sure that there is no randomization taking place in the round where the secret key is used.

Using commitments was also suggested by Dodis et al. [DMS16] in the context of reverse firewalls. A reverse firewall is a third party that sits between a user and the outside world which cannot be tampered with, who obtains and "sanitizes" the transcript in the protocol. These were also used in [AMV15], to provide subversion resilient signature schemes beyond the unique signatures discussed earlier. See [DMS16, AMV15] for extensive analysis of reverse firewalls. The analysis in [DMS16] also brings up several issues which makes our goal of creating subversion resilient authenticated key exchange without additional trust assumptions much harder.

Note that while we so far have mostly considered the initiator, the arguments above also apply to the responder in the protocol, and furthermore it becomes clear that we need even more countermeasures than simply "committing" by splitting the process into multiple rounds. This is due to a different class of attacks than above, namely: If the key is simply computed as $K = \mathsf{H}(g^{xy})$, at the moment when the responder picks $Y := g^y$, he has all the required information to compute K, and hence he can keep randomizing until he gets a key that for instance has low or hight weight, a specific target weight, starts with a set amount of zero's, and so on. In general, he is able to pick a weak key, and hence we require that neither party has full information of the different components going into the key during their final randomization.

For the signed Diffie-Hellman case, one way to solve this is to include both signatures in the final hash, so that $K = \mathsf{H}(\mathrm{ctxt}, g^{xy})$ where $\mathrm{ctxt} := (\mathsf{pk}_i, \mathsf{pk}_r, X, \sigma_i, Y, \sigma_r)$. The final randomization happens before the signature is computed, and even though the signature algorithm is deterministic, either party would have to forge their peer's signature to use the above attacks. A more general solution is to use commitment schemes as is done in [DMS16].

5.2 Final problems

In all of the examples above, we also encounter a problem with how keys are stored in memory. Even if we are able to force the subverted algorithm to do "honest" computations, where the key is stored securely during computation of X, if the subversion can copy and store it somewhere else after reading from the secure location, then the above attacks are valid again even though the initial X is computed securely and the signature scheme is subversion resilient. Hence, subversions are often distinguished by whether they keep a state or not. In our case, a subversion keeping the secret key as part of an internal state naturally requires even more countermeasures than before.

All of our examples and solutions above rely on being able to hide the secret key from an algorithm at appropriate times. But is this a realistic property? Simply assuming that such a property is achievable seems like a fairly strong claim.It is also unclear if one can increase trust in this claim by proving that one has acted honestly. While proving that you know, i.e have access to, a key is central to many cryptographic primitives like identification and signature schemes, public key encryption, commitment schemes and more, proving that one does not have knowledge of something is probably impossible. For such an interaction, a prover can simply choose to never use their secret information, and hence whether they actually possess it or not becomes irrelevant, and the outcome provides no trust.

This combined with the dependency on that the subverted algorithm does not keep the secret key in its state, suggests that unless everything is deterministic and verifiable (like unique signatures), it is extremely hard to design protocols that provide trust in that your device has not been tampered with, based on the output alone. One possible way to counter, or at least detect, the attacks in this work is to set up internal tracking of power and time consumption to register unusual behavior, but this is beyond the scope of this work.

The remarks above suggest that reverse firewalls might be the best solution to our problems, as their ability to sanitize transcripts means that any hidden message in the algorithm output will be removed by the firewall. This means that we no longer require "honest" computations, and hence both the problem of securely storing the key and the subverted algorithm keeping state disappears. The only thing we need is a subversion resilient signature scheme, for instance unique signatures, or one designed with a reverse firewall.

Dodis et al. provide such a protocol and firewall in Figures 12 and 13 of [DMS16]. Their solution builds on commitment schemes which are *malleable* and rerandomizable, which means that any commitment C to a value $A := g^a$ can be publicly turned into a commitment C' to A^α, and then further rerandomized to a uniformly random commitment to the same value. These properties ensure that nothing leaks from either the randomness used in the original message, or the randomness used when committing. Since neither operation requires a secret key, both can be performed by the reverse firewall, which hides any underlying information in the original ciphertext.

An open problem would be to design post quantum alternatives to their protocols.

5.3 Comments on the model

So far in this section, we have discussed different attack strategies which require different solutions. While our original goal was to create a meaningful model for subversion resilient AKE without resorting to primitives like reverse firewalls, in the end it seems like this is the solution that best satisfies the desired properties.

From the perspective of the model, we again want to highlight our problems with classifying different types of subversions, and how they should be treated. To illustrate, the remarks in Subsection 5.1 indicate that we need to separate randomization from secret key usage. While we in this case can argue that this can be fixed by using commitment schemes, the problem is that we currently do not have a model which captures the attack, and hence we only end up developing a specific solution against a specific attack. Building upon this, we expect that there are other attacks which require different countermeasures, which also isn't covered by the model, and since we're able to find attacks which aren't covered, proving that something is secure in it is of little to no value.

If one could find a way to classify subversions effectively, we might be able to fix this. But where should we draw the line? From the example above, a key with a very high weight could probably be detected and might be disregarded, but in Section 4 we demonstrated an undetectable attack that breaks security. The attack leaked the secret key, and hence could possibly be discarded as a trivial attack. There's also the question of what to do with subversions which simply introduces side channels, like swapping a constant time implementation for a non-constant one.

We could argue that we should only care about subversions that do something to the output of an algorithm, but given that the attack in Section 4 only uses output created by the underlying honest algorithm, it is not clear how one should define this properly.

To summarize, it seems like for every problem, we get more questions than answers, and we just keep kicking the can down the road to the next issue, instead of finding a consistent way to deal with the problems. At this point, we conceded that we were unable to define a meaningful model for subversions which could handle all the issues we explained above, and be integrated into a key exchange model. Without a working security model, any further analysis came down to creating countermeasures against different attacks on an ad-hoc basis, and hence we stopped working on the project.

References

AMV15. Giuseppe Ateniese, Bernardo Magri, and Daniele Venturi. Subversion-resilient signature schemes. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 364–375. ACM Press, October 2015.

BCJ21. Pascal Bemmann, Rongmao Chen, and Tibor Jager. Subversion-resilient public key encryption with practical watchdogs. In Juan Garay, editor, *PKC 2021, Part I*, volume 12710 of *LNCS*, pages 627–658. Springer, Heidelberg, May 2021.

BHJ$^+$15. Christoph Bader, Dennis Hofheinz, Tibor Jager, Eike Kiltz, and Yong Li. Tightly-secure authenticated key exchange. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, *TCC 2015, Part I*, volume 9014 of *LNCS*, pages 629–658. Springer, Heidelberg, March 2015.

BJK15. Mihir Bellare, Joseph Jaeger, and Daniel Kane. Mass-surveillance without the state: Strongly undetectable algorithm-substitution attacks. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 1431–1440. ACM Press, October 2015.

BPR14. Mihir Bellare, Kenneth G. Paterson, and Phillip Rogaway. Security of symmetric encryption against mass surveillance. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 1–19. Springer, Heidelberg, August 2014.

BR94. Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In Douglas R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 232–249. Springer, Heidelberg, August 1994.

DMS16. Yevgeniy Dodis, Ilya Mironov, and Noah Stephens-Davidowitz. Message transmission with reverse firewalls—secure communication on corrupted machines. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 341–372. Springer, Heidelberg, August 2016.

GJ18. Kristian Gjøsteen and Tibor Jager. Practical and tightly-secure digital signatures and authenticated key exchange. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 95–125. Springer, Heidelberg, August 2018.

Gün90. Christoph G. Günther. An identity-based key-exchange protocol. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *EUROCRYPT'89*, volume 434 of *LNCS*, pages 29–37. Springer, Heidelberg, April 1990.

HJK$^+$21. Shuai Han, Tibor Jager, Eike Kiltz, Shengli Liu, Jiaxin Pan, Doreen Riepel, and Sven Schäge. Authenticated key exchange and signatures with tight security in the standard model. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 670–700, Virtual Event, August 2021. Springer, Heidelberg.

JKRS21. Tibor Jager, Eike Kiltz, Doreen Riepel, and Sven Schäge. Tightly-secure authenticated key exchange, revisited. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part I*, volume 12696 of *LNCS*, pages 117–146. Springer, Heidelberg, October 2021.

Kra05. Hugo Krawczyk. HMQV: A high-performance secure Diffie-Hellman protocol. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 546–566. Springer, Heidelberg, August 2005.

LS17. Yong Li and Sven Schäge. No-match attacks and robust partnering definitions: Defining trivial attacks for security protocols is not trivial. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1343–1360. ACM Press, October / November 2017.

MVOV18. Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 2018.

PQR22. Jiaxin Pan, Chen Qian, and Magnus Ringerud. Signed (group) Diffie-Hellman key exchange with tight security. *Journal of Cryptology*, 35(4):1–42, 2022.