Des. Codes Cryptogr. (2018) 86:481–516
https://doi.org/10.1007/s10623-017-0337-5

Strengthening the security of authenticated key exchange against bad randomness

Michèle Feltz[1] · Cas Cremers[2]

Received: 17 June 2016 / Revised: 28 November 2016 / Accepted: 24 January 2017 /
Published online: 13 February 2017
© Springer Science+Business Media New York 2017

Abstract Recent history has revealed that many random number generators (RNGs) used in cryptographic algorithms and protocols were not providing appropriate randomness, either by accident or on purpose. Subsequently, researchers have proposed new algorithms and protocols that are less dependent on the RNG. One exception is that all prominent authenticated key exchange (AKE) protocols are insecure given bad randomness, even when using good long-term keying material. We analyse the security of AKE protocols in the presence of adversaries that can perform attacks based on chosen randomness, i.e., attacks in which the adversary controls the randomness used in protocol sessions. We propose novel stateful protocols, which modify memory shared among a user's sessions, and show in what sense they are secure against this worst case randomness failure. We develop a stronger security notion for AKE protocols that captures the security that we can achieve under such failures, and prove that our main protocol is correct in this model. Our protocols make substantially weaker assumptions on the RNG than existing protocols.

Keywords Authenticated key exchange (AKE) · Security models · Stateless protocols · Stateful protocols · Chosen randomness

Mathematics Subject Classification 94A60

1 Introduction

Authenticated key exchange (AKE) protocols are a critical building block in most security infrastructures. They provide the glue between asymmetric cryptography (e.g., for addressing

Communicated by C. Boyd.

✉ Michèle Feltz
mmc.feltz@gmail.com

Cas Cremers
cas.cremers@cs.ox.ac.uk

1 Institute of Information Security, ETH Zurich, Zurich, Switzerland

2 University of Oxford, Oxford, UK

the key distribution problem) and symmetric cryptography (e.g., for efficient encryption of large amounts of data). Since the proposal of the Diffie–Hellman key exchange protocol, much research effort has gone into improving AKE protocol designs, achieving ever stronger notions of security. These include protocols such as the TLS and IKE handshakes, as well as pure key exchange protocols such as MQV, HMQV, and NAXOS. A common factor among these designs is that they explicitly rely on the existence of another building block: a random number generator (RNG).

Constructing a good RNG is hard, as indicated by a large number of security vulnerabilities that involve either flawed or weakened RNGs. Recently there have been a surprising number of examples in which bad RNGs were substantially weakening the security guarantees of cryptographic systems. We recall some instances. In 2008, Bello discovered a randomness vulnerability in Debian's OpenSSL package; keys generated by the RNG of this package were predictable [1]. Affected keys included SSH keys, OpenVPN keys, DNSSEC keys, key material for use in X.509 certificates and ephemeral Diffie–Hellman keys leading to compromised session keys in SSL/TLS connections [1]. In 2012 it was shown that the RSA keys of many users can be easily factored because the prime factors from which they are derived were not sufficiently random [2]. In August 2013 attackers took control of Bitcoin transactions due to flaws in Android's Java and OpenSSL RNG [3]. As multiple transactions were signed using the same randomness in the ECDSA signature generation, the attacker was able to recover the long-term secret signing key of the user initiating the transactions, enabling the attacker to perform transactions on the user's behalf. After the Snowden revelations, it has become clear that the NSA was involved in the backdoor found in a NIST-standardized RNG [4–6]. This RNG was the default in RSA's BSAFE toolkit and also made its way into ISO/IEC standards. This backdoor can be effectively used to break the security of deployed internet protocols, such as TLS [6].

As a result, researchers have worked on providing cryptographic mechanisms that depend less on the RNG (e. g., for digital signatures [5,7] or in the public key encryption setting [8,9]). Unfortunately, the security of state-of-the-art key exchange protocols still depends critically on the RNG.

One might intuitively think that a faulty/broken/malicious RNG is not a protocol problem, and should be solved by the RNG designers instead. There are two main arguments against this view.

First, designing for security implies reducing single points of failure and reducing the assumptions under which the protocol is secure. In many AKE protocols, the RNG is such a single point of failure. Our techniques show that this can be avoided: if the RNG becomes insecure, we may still retain some security at the AKE level.

Second, there are solutions to bad randomness that can be naturally implemented at the higher (AKE) level, but not at the RNG level. In particular, our work confirms earlier results. For example, the approach from [10] involves replacing the randomness x in the exponent of the ephemeral public Diffie–Hellman key g^x by a combination of the randomness and a static secret a, namely $g^{H(a,x)}$. This forces the attacker to learn both the randomness and the static secret if he wants to recompute the exponent. If one would like to provide a solution at the level of the RNG's design, there is no secret to leverage. At the level of the AKE protocol, we have natural access to a long-term secret that can be leveraged to reduce the dependency on good randomness, enabling security under weaker assumptions.

Contributions

In this work, we propose new protocols for AKE that offer stronger security against bad randomness than previous protocols. Thus, our protocols are secure under significantly weaker assumptions (on the RNG) than previous protocols.

To achieve and prove this stronger notion of security, we design a strong eCK-like security model that additionally incorporates the adversarial ability of choosing session-specific randomness, which corresponds to the worst-case RNG scenario. Previous works that considered such a capability concluded that AKE security cannot be achieved against such an adversary. In contrast, we show that these works implicitly assumed that protocols are stateless, i.e., that different sessions of the same user do not write into shared memory. Our protocols leverage state shared among sessions to achieve a stronger notion of security.

Related work

Randomness failures The first models addressing the leakage of session-specific information include the eCK model [10] and the CK model [11]. The eCK model considers an information-leaking RNG that leaks values after they have been generated, which is modelled via the query ephemeral-key. Intermediate protocol computations are assumed to be outside of the adversary's control. In contrast, the CK model considers long-term keys stored in secure memory (e.g., an HSM), whereas protocol computations are (partly) done in less-protected memory. The adversary has read-only access to the less-protected memory through a query session-state. However, in the vast majority of proofs in the CK model, the less-protected memory has been defined to contain exactly the randomness, thereby effectively modeling an information-leaking RNG. Unlike our work, the CK and eCK models do not consider predictable, failing, or compromised RNGs.

Yang et al. [12] first analyzed AKE security w.r.t. adversaries who can manipulate random values. They define two security models: Reset-1 and Reset-2. In the Reset-1 model the adversary controls the randomness of each session, with the restrictions that the adversary (a) does not issue corrupt queries to the actor and peer of the test session, and (b) the randomness used in the test and partner session is not used in any other session. The Reset-2 model captures repeated secret randomness in multiple sessions due to reset attacks, but no chosen-randomness attacks. Critically, the Reset-1 model does not capture weak perfect forward secrecy and both models do not allow the adversary to perform reset attacks against the target session or its partner session. Both models are based on their impossibility result that no protocol can be secure against *reset-and-replay attacks* on the target session [12, p. 120]. In a reset-and-replay attack the adversary first sets the randomness of a session to the same randomness as used in a previous session of the same user and then replays messages to the session so that both sessions compute the same session key [12]. Yang et al. [12] propose a transformation that turns any stateless protocol secure in the Reset-2 model into a *stateless* protocol secure in the Reset-1 model. We show that *stateful* protocols achieve stronger security guarantees against attacks based on bad randomness; in particular, they can achieve security against reset-and-replay attacks on the target session.

Ristenpart and Yilek [13] show that virtual machine (VM) snapshots can lead to VM reset attacks. As a countermeasure, they propose a framework for *hedging* cryptographic operations based on preprocessing potentially bad RNG-supplied randomness together with additional inputs with HMAC to provide pseudorandomness for the cryptographic operation; their framework uses hedging techniques for public-key encryption of Bellare et al. [8]. Hedging a cryptographic operation means designing it in such a way that, given good randomness, the operation provably (in the random oracle model) achieves strong security goals, and, given bad randomness, the operation achieves weaker, but still meaningful, security goals [8]. Our models cover VM reset attacks on stateless protocols, i.e., resettable randomness, and we thereby address some of their future work.

Kamara and Katz [14] provide security notions for private-key encryption schemes that incorporate chosen-randomness attacks modelling the ability of the adversary to completely control the randomness used in the encryption process. In their definitions, the adversary has

complete control over the random coins used by the encryption oracle, but has no control over the randomness involved in the encryption of the challenge ciphertext, which is supposed to be secret (i.e. unknown to the adversary) and chosen uniformly at random by the challenger.

Bellare and Tackmann [15] investigate randomness failures (and, in particular, subversion of RNGs) in the context of public-key encryption (PKE) and introduce *nonce-based public-key encryption*. The differences between nonce-based PKE schemes and standard PKE schemes are that (a) the sender needs to run a seed-generation algorithm and (b) the encryption algorithm is deterministic, taking as input a nonce n and a seed s in addition to an encryption key k and a message $m \in \{0, 1\}^*$. They also provide two new security definitions for nonce-based PKE schemes and construct a nonce-based PKE scheme based on a hedged extractor that achieves both security notions, that is, the scheme guarantees security if either the sender's seed is secret and message-nonce pairs do not repeat, or even if the sender's seed is compromised and the nonces are unpredictable. Bellare and Tackmann give constructions of a hedged extractor in the random oracle model and in the standard model yielding concrete nonce-based PKE schemes in the random oracle model and in the standard model, respectively.

Stateless and stateful AKE protocols Most AKE protocols (e. g., HMQV [16], NAXOS [10], CMQV [17]) are stateless, i.e., they only modify session-specific memory, whereas the memory that is shared among sessions is invariant under protocol execution. Furthermore, the security of stateful AKE protocols, which update the memory that is shared among sessions during execution of the protocol, has not been considered in the context of randomness failures.

A few stateful protocols have been suggested. For example, in [18], Blake-Wilson et al. propose to modify their Protocol 2 by concatenating the secret value that is used as the secret material to derive the session key with the value of a counter. We denote this new protocol by Protocol 2C. Instead of running the protocol each time a session key is required, a new session key is obtained by simply incrementing the counter and computing a new hash value [18]. The idea of using a counter variable is presented in the context of special applications for which it might not be desirable to run the protocol whenever a new session key has to be established. However, no security proof of Protocol 2C has been given. In Sect. 4 we present a new protocol, which we call CNX, and prove its security in a model capturing chosen-randomness attacks. The CNX protocol includes a global counter value, which is shared across the sessions of a user, as input to the hash function H_1 used in the computation of the outgoing messages.

Overview

In Sect. 2 we define a generic AKE framework that includes queries for the adversary to reveal randomness and to choose the randomness used by sessions. In Sect. 3 we introduce the notions of stateful and stateless protocols, and show that no stateless protocol can achieve security in a model that permits the adversary to choose the randomness of sessions and to reveal certain session keys. The proof sketch of this statement is as follows. The adversary can make two sessions of the same user, say $\hat{B}$, accept the same session key without being partnered (or *matching sessions*) by making them use the same randomness and sending the messages of the partner session to both sessions. As the two sessions of user $\hat{B}$ are not partnered, the adversary can learn the session key of either of them by revealing the session key of the other, non-partnered, session. We then proceed by showing an initial attempt at addressing this issue with a novel stateful protocol in Sect. 4. In Sect. 5 we step back and consider the question: what is the optimal security that stateful protocols can achieve under chosen randomness? We use our findings in Sect. 6 to develop a stronger security model

Table 1 Elements of the session state

actor	The session's actor (the user running the session)
peer	The session's peer (the intended communication partner)
role	Taken role; either $\mathcal{I}$ (initiator) or $\mathcal{R}$ (responder)
sent, *recv*	Concatenation of all messages sent, respectively received, in the session
status	Session status; either `active`, `accepted`, or `rejected`
key	Key established in the session
rand	Randomness used in the session
data	Any additional session-specific or protocol-specific data
step	Protocol step to be executed (in the session)

against chosen randomness. We propose a protocol that is provably secure in our stronger security model. We conclude in Sect. 7.

2 AKE framework

We first define a framework to reason about the security of different classes of AKE protocols against adversaries with diverse capabilities. This framework allows to express existing models such as the eCK model [10], the eCK^w model [19] and the eCK-PFS model [19] as well as extensions of these models that permit the adversary to choose the randomness used in protocol sessions.

2.1 Security model

Sessions and session-specific memory Let $\mathcal{P}$ be a finite set of N binary strings representing user identifiers. Each user can execute multiple instances of an AKE protocol, called sessions, concurrently. We can uniquely identify specific sessions of a user by referring to the order in which they are created. Thus, the i'th session of user $\hat{P}$ is denoted by the tuple $(\hat{P}, i) \in \mathcal{P} \times \mathbb{N}$. These tuples are not used by the protocol, but allow the adversary to identify the sessions he created. We model each user by a probabilistic Turing machine. For each user $\hat{P}$, the state of its Turing machine consists of the memory contents of the user, where we differentiate between session-specific memory and user memory, which is shared among different sessions. We take an abstract view on the session-specific memory and assume that it can be separated into distinct named fields, referred to as variables and listed in Table 1. Some of these variables are set upon session creation, whereas others are set or updated during execution of the protocol. The next step to be executed by the protocol is stored in the variable $step$. Alternatively, this value could be stored in the variable $data$. We choose to store it in a separate variable for clarity. We say that a session s has accepted (or is completed) if the value of its $status$ variable taking values in the set {`active`, `accepted`, `rejected`} is `accepted`. We denote by st_s the session-specific memory related to session s. The session-specific memory contains the session-specific variables of Table 1. Initially we assume that each session-specific variable is undefined, denoted by $\perp$.

User memory The user memory of some user stores the user's long-term public/secret key pair, the public key of all other users $\hat{Q} \in \mathcal{P}$ as well as additional variables that might be required by the protocol. The information stored in the user memory is accessed and possibly updated by sessions of the user according to the protocol specification. In contrast to session-

specific information, data stored in the user memory of some user $\hat{P}$ is shared among different sessions of the user $\hat{P}$. We denote by $st_{\hat{P}}$ the user memory of user $\hat{P} \in \mathcal{P}$.

Game state and game behaviour (see also [20]) The adversary, modeled as a probabilistic polynomial-time algorithm, interacts with the users in the set $\mathcal{P}$ within a game through queries in a set Q. The state of the game (or game state) contains session-specific state information st_s for all sessions s, user-specific information $st_{\hat{P}}$ for each user $\hat{P} \in \mathcal{P}$ as well as other information related to the game such as some bit that the adversary attempts to guess. The game behaviour, which we denote by Φ, describes how the game processes the queries in Q. More precisely, the game behaviour Φ is an algorithm taking as input the current state of the game GST, a query $q \in Q$, a protocol π, and a security parameter k, and returning a new state GST' as well as a response $\mathsf{response} \in \{0, 1\}^* \cup \{\perp, \star\}$ to the adversary's query q.

Definition 1 (*h-message protocol*) Let k be a security parameter. An *h-message protocol* π, where h is the sum of the number of messages sent and received during a protocol session that accepts, consists of

- a set of domain parameters,
- a probabilistic polynomial-time key generation algorithm KeyGen, which takes as input 1^k and outputs a public/secret key pair, and
- a deterministic polynomial-time algorithm Ψ executed by a user in a session. This algorithm takes as input 1^k, the session-specific memory st_s of a session s, the user memory $st_{\hat{P}}$ of the actor $\hat{P}$ of session s, and a message $m \in \{0, 1\}^*$, and outputs a triple $(m', st'_s, st'_{\hat{P}})$, where $m' \in \{0, 1\}^* \cup \{\star\}$ is a message, st'_s is an updated internal session state, and $st'_{\hat{P}}$ is an updated state of the user memory of user $\hat{P}$.

If h is even, then the number of messages $m' \neq \star$ output by Ψ during a protocol session is $\frac{h}{2}$ for both roles initiator and responder. If h is odd, then the number of messages $m' \neq \star$ output by Ψ during a protocol session is $\frac{h+1}{2}$ for the initiator role and $\frac{h-1}{2}$ for the responder role.

The output of the key exchange algorithm Ψ (see Definition 1) may include the value $\star$ to indicate that the session does not generate an outgoing message.

Setup of the game A setup algorithm *SetupG* is used to generate a set of a fixed number N of user identifiers, to set all session-specific variables to $\perp$, and to initialize the user memory of each user. The algorithm *SetupG* takes as input the protocol π and the security parameter 1^k, and outputs an initial game state GST_{init}. More precisely, the setup algorithm proceeds as follows:

1. generate a set $\mathcal{P} = \{\hat{P}_1, \ldots, \hat{P}_N\}$ of N distinct binary strings (representing user identifiers),
2. for all users $\hat{P} \in \mathcal{P}$: generate a long-term public/secret key pair $(\mathsf{pk}_{\hat{P}}, \mathsf{sk}_{\hat{P}})$ using algorithm KeyGen,
3. for all users $\hat{P} \in \mathcal{P}$: store the key pair $(\mathsf{pk}_{\hat{P}}, \mathsf{sk}_{\hat{P}})$ together with the set $\{(\hat{P}, \mathsf{pk}_{\hat{P}}) \mid \hat{P} \in \mathcal{P} \backslash \{\hat{P}\}\}$ in the user memory $st_{\hat{P}}$, and
4. store and initialize with public values all other user-specific variables used by the protocol.

Queries The specification of some of the queries that we define below is similar to queries defined in the framework of Boyd et al. [21]. The $\mathsf{public\text{-}info}$ query, which was informally introduced in [20, p. 4], allows the adversary to obtain information that was generated during the setup phase of the game such as the users' identifiers and their public keys.

– public-info(). The query returns a set $\mathcal{L}$ of information, which contains the set $\{(\hat{P}, \mathsf{pk}_{\hat{P}}) \mid \hat{P} \in \mathcal{P}\}$ as well as the initial values of all other variables stored in the user memory of each user, except for the users' long-term secret key, if such variables are used by the protocol.

The queries in the set $Q_{\mathsf{R}} = \{\mathsf{create}, \mathsf{send}\}$ model regular execution of the protocol.

– create($\hat{P}, r[, \hat{Q}]$). The query models the creation of a new session s for the user with identifier $\hat{P}$. It requires that $\hat{P} \in \mathcal{P}$, $\hat{Q} \in \mathcal{P}$, and that $r \in \{\mathcal{I}, \mathcal{R}\}$; otherwise, it returns $\perp$. Session variables are initialized as

$$(s_{actor}, s_{role}, s_{sent}, s_{recv}, s_{status}, s_{key}, s_{step}) \leftarrow (\hat{P}, r, \epsilon, \epsilon, \texttt{active}, \perp, 1)\ .$$

A bit string in $\{0, 1\}^k$ is sampled uniformly at random and assigned to s_{rand}.[1] If the optional peer identifier $\hat{Q}$ is provided, the variable s_{peer} is set to $\hat{Q}$. The key exchange algorithm Ψ is executed on input $(1^k, st_s, st_{\hat{P}}, \epsilon)$. The algorithm returns a triple $(m', st'_s, st'_{\hat{P}})$. We set $st_s \leftarrow st'_s$ and $st_{\hat{P}} \leftarrow st'_{\hat{P}}$. The query returns m'.

– send($\hat{P}, i, m$). The query models sending message m to the i'th session of user $\hat{P}$, which we denote by s. It requires that $s_{status} = \texttt{active}$; otherwise it returns $\perp$. The algorithm Ψ is run on input $(1^k, st_s, st_{\hat{P}}, m)$, and outputs a triple $(m', st'_s, st'_{\hat{P}})$. We set $st_s \leftarrow st'_s$ and $st_{\hat{P}} \leftarrow st'_{\hat{P}}$. The query returns m'.

We next define the queries in the set $Q_{\mathsf{C}} = \{\mathsf{session\text{-}key}, \mathsf{corrupt}, \mathsf{randomness}, \mathsf{cr\text{-}create}\}$, which model the corruption of a user's secrets. The randomness query models the adversary's capability of revealing the randomness s_{rand} of a particular session s. In contrast, the cr-create query models the adversary's capability of choosing the randomness used within a session. We do not explicitly model repeated randomness, i.e., secret uniform bits that have been used in previous key exchange sessions; as we show in Sect. 6.2, security in a model that permits the adversary to reveal the randomness used in sessions and to choose the randomness of sessions implies security in a model capturing repeated randomness.

In the definition of the queries session-key and randomness we denote the i'th session of user $\hat{P}$ by s.

– session-key($\hat{P}, i$). The query requires that $s_{status} = \texttt{accepted}$; otherwise, it returns $\perp$. The query returns the session key s_{key} of session s.
– corrupt($\hat{P}$). If $\hat{P} \notin \mathcal{P}$, then S returns $\perp$. Otherwise the query returns the long-term secret key $\mathsf{sk}_{\hat{P}}$ of user $\hat{P}$.
– randomness($\hat{P}, i$). If $s_{status} \neq \perp$, then the randomness s_{rand} used in session s is returned. Otherwise, the query returns $\perp$.
– cr-create($\hat{P}, r, rnd[, \hat{Q}]$). The query models the creation of a new session s, using randomness rnd chosen by the adversary, for the user $\hat{P}$. The query requires that $\hat{P} \in \mathcal{P}$, $\hat{Q} \in \mathcal{P}$, $rnd \in \{0, 1\}^k$, and that $r \in \{\mathcal{I}, \mathcal{R}\}$; otherwise, it returns $\perp$. The session variables are initialized as

$$(s_{actor}, s_{role}, s_{sent}, s_{recv}, s_{status}, s_{key}, s_{rand}, s_{step}) \leftarrow (\hat{P}, r, \epsilon, \epsilon, \texttt{active}, \perp, rnd, 1)\ .$$

If the optional peer identifier $\hat{Q}$ is provided, the variable s_{peer} is set to $\hat{Q}$.
The key exchange algorithm Ψ is executed on input $(1^k, st_s, st_{\hat{P}}, \epsilon)$. The algorithm returns a triple $(m', st'_s, st'_{\hat{P}})$. We set $st_s \leftarrow st'_s$ and $st_{\hat{P}} \leftarrow st'_{\hat{P}}$. The query returns m'.

[1] Note that our syntax implies that all randomness required during the execution of session s is deterministically derived from s_{rand}.

The set $Q_{\mathsf{noCR}} = Q_{\mathsf{R}} \cup (Q_{\mathsf{C}} \backslash \{\textsf{cr-create}\})$ contains all execution and corruption queries, except the query cr-create.

The notion of matching sessions specifies when two sessions are supposed to be intended communication partners. It is formalized below via matching conversations as in [10,19].

Definition 2 (*Matching sessions*) Let π be an h-message protocol. We say that two sessions s and s' of π are *matching* if $s_{status} = s'_{status} = \texttt{accepted}$ and $s_{actor} = s'_{peer} \wedge s_{peer} = s'_{actor} \wedge s_{sent} = s'_{recv} \wedge s_{recv} = s'_{sent} \wedge s_{role} \neq s'_{role}$.

We next define a parameterized family of AKE security models. The parameters for each model consist of a subset Q of the above adversary queries and a freshness predicate F, which restricts the adversary from performing certain combinations of queries.

Definition 3 (*AKE security model*) Let π be an h-message protocol. Let Q be a set of adversary queries such that $Q_{\mathsf{R}} \subseteq Q \subseteq Q_{\mathsf{R}} \cup Q_{\mathsf{C}}$. Let F be a freshness predicate, that is, a predicate that takes a session of protocol π and a sequence of queries (including arguments and results) in Q. We call (Q, F) an *AKE security model*.

Remark 1 In this work we fix a particular definition for matching sessions (namely, Definition 2) and construct strong security models with respect to this definition. It is straightforward to adapt these models to other definitions of matching sessions that are suitable for analyzing protocols such as (H)MQV that allow two sessions performing the same role to compute the same session key.

2.2 Security experiment

We associate to each AKE security model $X = (Q, F)$ a security experiment $W(X)$, defined below, played by an adversary E against a challenger. To win the experiment, the adversary aims to distinguish a real session key from a random key, modelled through the following query.

- test-session(s). This query requires that $s_{status} = \texttt{accepted}$; otherwise, it returns $\perp$. A bit b is chosen at random. If $b = 0$, then s_{key} is returned. If $b = 1$, then a random key is returned according to the probability distribution of keys generated by the protocol.

Definition 4 (*Security experiment*) Let k be a security parameter and π be an h-message protocol. Let $X = (Q, F)$ be an AKE security model. We define experiment $W(X)$, between an adversary E and a challenger who implements all the users, as follows:

1. The game is initialized with domain parameters for security parameter k and the setup algorithm *SetupG* is executed.
2. The adversary E first issues the query public-info, and then performs any sequence of queries from the set Q.
3. At some point in the experiment, E issues a test-session query to a session s that has accepted and satisfies F at the time the query is issued.
4. The adversary may continue with queries from Q, under the condition that the test session must continue to satisfy F.
5. Finally, E outputs a bit b' as his guess for b.

The adversary E wins the security experiment $W(X)$ if he correctly guesses the bit b chosen by the challenger during the test-session query (i.e., if $b = b'$, where b' is E's guess). Success of E in the experiment is expressed in terms of E's advantage in distinguishing

whether he received the real or a random session key in response to the test-session query. The advantage of adversary E in the above security experiment against a key exchange protocol π for security parameter k is defined as $Adv^{\pi,E}_{W(X)}(k) = |2P(b = b') - 1|$.

Definition 5 (*AKE security*) A key exchange protocol π is said to be *secure in AKE security model* $X = (Q, F)$ if, for all PPT adversaries E, it holds that

- if two users successfully complete matching sessions, then they compute the same session key,
- the probability of event Multiple-Match$^{W(X)}_{\pi,E}(k)$ is negligible, where Multiple-Match$^{W(X)}_{\pi,E}(k)$ denotes the event that there exists a session that has accepted with at least two matching sessions, and
- E has no more than a negligible advantage in winning the $W(X)$ security experiment, that is, there exists a negligible function $negl$ in the security parameter k such that $Adv^{\pi,E}_{W(X)}(k) \leq negl(k)$.

Informally, the second requirement in Definition 5 (see also [22]) states that, for a given session of protocol π that has accepted, it holds that its matching session, if it exists, is unique.

3 Chosen randomness and stateless protocols

3.1 Stateless and stateful protocols

We start by defining a global class of AKE protocols. Such protocols are required to be executable, i.e., if the messages of two users $\hat{A}$ and $\hat{B}$ are faithfully relayed to each other, then both users end up with a shared session key [22–24]. A second requirement ensures that protocol messages depend on session-specific randomness.

Definition 6 (*Protocol class AKE*) We define AKE as the class of all h-message protocols, where h is the sum of the number of messages sent and received during a protocol session that accepts, that meet the following requirements: In the presence of an eavesdropping adversary,

- if the messages of two sessions s and s' with $s_{actor} = s'_{peer} \wedge s_{peer} = s'_{actor} \wedge s_{role} \neq s'_{role}$ are faithfully relayed to one another (that is, such that $s_{sent} = s'_{recv} \wedge s_{recv} = s'_{sent}$), then $s_{status} = s'_{status} = \texttt{accepted}$ and $s_{key} = s'_{key}$.
- the probability that two sessions of the same user output in all protocol steps identical messages is negligible in the security parameter.

We distinguish between stateless and stateful protocols. Stateless protocols leave the state of a user's memory $st_{\hat{P}}$ (where $\hat{P} \in \mathcal{P}$), that is, the memory that is shared among sessions, invariant under execution of the protocol. In contrast, the state of a user's memory $st_{\hat{P}}$ is modified when executing a protocol that is not stateless. The vast majority of two-message AKE protocols proposed in the last two decades (e. g. NAXOS, HMQV, CMQV) are stateless.

Definition 7 (*Stateless protocol, SL*) Let $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ be sets. Let $\text{proj}_3 : \mathcal{A} \times \mathcal{B} \times \mathcal{C} \to \mathcal{C}$ be the map given by $\text{proj}_3(a, b, c) = c$ for all $(a, b, c) \in \mathcal{A} \times \mathcal{B} \times \mathcal{C}$. Let π be a protocol in the class AKE. We say that π is a *stateless protocol* if

$$\text{proj}_3\big(\Psi(1^k, st_s, st_{\hat{P}}, m)\big) = st_{\hat{P}},$$

for all $(k, st_s, st_{\hat{P}}, m) \in \mathbb{N} \times \{st_s \mid s \in \mathcal{P} \times \mathbb{N}\} \times \{st_{\hat{P}} \mid \hat{P} \in \mathcal{P}\} \times \{0,1\}^*$. We denote by SL the class of all stateless protocols.

If a protocol is not stateless, we say that it is *stateful*.

Remark 2 Stateless protocols cannot provide message replay detection to reject messages that have been received in earlier sessions of the same user as this would require storing all previously received messages in a table in the user memory and, upon receipt of a valid message in a session, accessing the table in the user memory and checking whether the message corresponds to a message in the table.

3.2 Insecurity of stateless protocols against chosen-randomness attacks

Yang et al. state that no protocol can be secure against *reset-and-replay attacks* on the target session [12, p. 120]. However their statement only holds because of an implicit assumption on the protocol class that they consider. In particular, their definition of the protocols' execution model implies that they only consider stateless protocols, according to our Definition 7.[2]

The following proposition states that no stateless protocol is secure in a model that permits the adversary to choose the randomness of sessions and to reveal certain session keys. This model gives the adversary access to the minimal set of queries for the proposition to hold.

Proposition 1 (Impossibility result for AKE ∩ SL) *Let $X = (Q, F)$ be an AKE security model, where $\left(Q_R \cup \{\mathsf{cr\text{-}create}, \mathsf{session\text{-}key}\}\right) \subseteq Q \subseteq Q_R \cup Q_C$ and F is defined as follows. A session s is said to satisfy F if no $\mathsf{session\text{-}key}(s)$ query has been issued and, for all sessions s^* such that s^* matches s, no $\mathsf{session\text{-}key}(s^*)$ query has been issued. No protocol in the class $\mathsf{AKE} \cap \mathsf{SL}$ can satisfy security in the model X.*

Proof of Proposition 1 The proof is based on an attack sketched by Yang et al. [12]. Let π be an arbitrary protocol in SL. There exists an adversary E that wins the $W(X)$ game against the challenger with non-negligible probability as follows. The adversary E chooses randomness $r_1 \in \{0,1\}^k$ and creates a responder session s' of user $\hat{B}$ via the query $\mathsf{cr\text{-}create}(\hat{B}, \mathcal{R}, r_1, \hat{A})$. E then completes a protocol execution between the users $\hat{A}$ and $\hat{B}$; $\hat{A}$ and $\hat{B}$ complete sessions s and s', respectively. E creates another responder session s'' of user $\hat{B}$ via the query $\mathsf{cr\text{-}create}(\hat{B}, \mathcal{R}, r_1, \hat{A})$, where r_1 is the same randomness as used to create session s', and replays the messages from session s to session s''. As the randomness used in session s'' is identical to the randomness used in session s' and $\pi \in \mathsf{SL}$, the messages that E receives from session s'' are the same as the messages sent by session s'. Now, E chooses the completed session s' as the test session, and reveals the session key computed in session s'' via a $\mathsf{session\text{-}key}(s'')$ query. As the session keys computed in sessions s' and s'' are the same and both sessions are non-matching, the adversary learns the session key of the test session. □

4 CNX: preventing key repetition

In this section we first describe a general method of how to prevent replay attacks combined with chosen-randomness attacks, where two sessions of the same user accept the same

[2] The crucial observation is that the protocol execution algorithm P in [12] uses abstract session-specific state information for a user U's session i, denoted by St^i_U. Additionally, the framework includes user-specific information: the identity U, and public/private keys pk_U, sk_U. It follows from their definition of the protocol execution algorithm that a protocol can only update the session-specific state St^i_U, but cannot change any state that can be accessed by other sessions of the same user. Hence, stateful protocols are not modeled in their framework.

session key without being matching sessions (see, for example, the attack in the proof of Proposition 1). We then apply our method to a concrete protocol, namely to the NAXOS protocol.

Let $\pi \in \Lambda \cap \mathsf{SL}$, where the protocol class Λ is defined in Definition 10. Let $H : \{0,1\}^* \to \mathbb{Z}_q$ (with $||q|| = k$, where k denotes the security parameter) be a hash function. We propose the following security-strengthening method to build a protocol π' with the goal to achieve security against eCK-like adversaries who, in addition, can perform replay attacks combined with chosen-randomness attacks on different sessions of the same user. The resulting protocol π' uses the user's state to prevent key repetition by implementing a counter. More precisely, protocol π' is defined in the same way as protocol π except that in protocol π':

- Each user $\hat{P} \in \mathcal{P}$ maintains a counter l, taking values in $\mathbb{N}$, initialized with 0 and incremented by one upon creation of a new session. This counter variable is stored in the user memory $st_{\hat{P}}$. We write $st_{\hat{P}}.l$ to access the counter variable l of user $\hat{P}$. We assume that the "read and increment" is atomic, i.e., different sessions are guaranteed to obtain different values.
- The user memory of each user $\hat{P} \in \mathcal{P}$ is given by $st^{\pi'}_{\hat{P}} = (st^{\pi}_{\hat{P}}, l)$.
- The functions $f_{\mathcal{I}}$ and $f_{\mathcal{R}}$ (of Definition 10) are instantiated as follows in protocol π':

$$f_{\mathcal{I}}(r, v, st^{\pi'}_{\hat{A}}) = H(r, a, v)$$
$$f_{\mathcal{R}}(r, v, st^{\pi'}_{\hat{B}}) = H(r, b, v).$$

- The key exchange algorithm for protocol π' is defined in a similar way as the one for protocol π except that if $s_{step} = 1$, the algorithm for π' first executes the instructions

$$st^{\pi'}_{\hat{P}}.l \leftarrow st^{\pi'}_{\hat{P}}.l + 1;$$
$$s_{data} \leftarrow st^{\pi'}_{\hat{P}}.l$$

  before executing the remaining instructions of protocol π (with $f_{\mathcal{I}}$, $f_{\mathcal{R}}$ as defined in the previous point).

We next apply our security-strengthening method to the NAXOS protocol. NAXOS is a modern Diffie–Hellman type protocol that is less efficient than HMQV, but enjoys a simpler security proof in the eCK^w model [25]. Similarly, our method can be applied to other protocols such as CMQV.

Let $H_1 : \{0,1\}^* \to \mathbb{Z}_q$ and $H_2 : \{0,1\}^* \to \{0,1\}^k$ be two hash functions. For user $\hat{A}$, we write a as a shorthand for $\mathsf{sk}_{\hat{A}}$ and write A as a shorthand for $\mathsf{pk}_{\hat{A}} = g^a$, and define (b, B) similarly for user $\hat{B}$. Applying our security-strengthening method to the NAXOS protocol yields a new protocol, which we call the CNX ("Counter-NaXos") protocol, shown in Fig. 1. In contrast to the NAXOS protocol, the CNX protocol is a stateful protocol belonging to the class $\mathsf{AKE} \setminus \mathsf{SL}$.

The model CR-eCK^w, which we define below, is given by the set of queries $Q = Q_{\mathsf{noCR}} \cup \{\mathsf{cr\text{-}create}\}$ and its freshness predicate is obtained from the freshness predicate of the eCK^w model [25] by adding two conditions taking into account combinations of $\mathsf{corrupt}$ and $\mathsf{cr\text{-}create}$ queries. We start by recalling the notion of an *origin session* [25], which is used to relate a received message that was not constructed by the adversary to the session it originates from.

Definition 8 (*Origin session* [25]) We say that a session s' with $s'_{status} \neq \perp$ is an origin session for a session s with $s_{status} = \mathtt{accepted}$ if $s'_{send} = s_{recv}$.

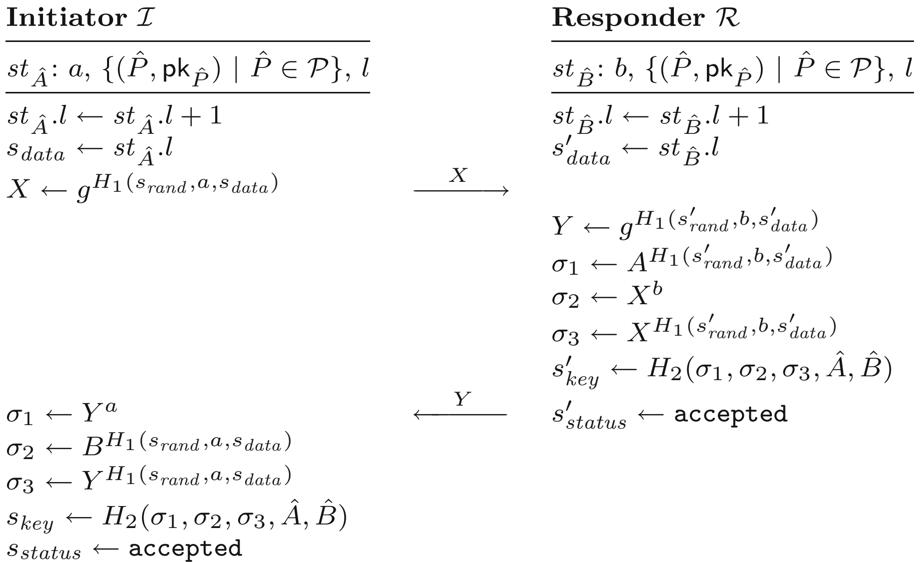

Fig. 1 The CNX protocol

Definition 9 (*CR-eCK^w*) The CR-eCK^w model is defined by (Q, F), where $Q = Q_{\mathsf{noCR}} \cup \{\mathsf{cr\text{-}create}\}$ and a session s is said to satisfy F if all of the following conditions hold:

1. no session-key(s) query has been issued, and
2. for all sessions s^* such that s^* matches s, no session-key(s^*) query has been issued, and
3. not both queries corrupt(s_{actor}) and (randomness(s) or cr-create$(.)$ creating session s) have been issued, and
4. for all sessions s' such that s' is an origin session for session s, not both queries corrupt(s_{peer}) and (randomness(s') or cr-create$(.)$ creating session s') have been issued, and
5. if there exists no origin session for session s, then no corrupt(s_{peer}) query has been issued.

The following proposition states that the CNX protocol is secure in model CR-eCK^w.

Proposition 2 *Let k be a security parameter. Under the GAP-CDH assumption [26] in the cyclic group G of prime order q with $||q|| = k$, the* CNX *protocol is secure in model CR-eCK^w, when the hash functions $H_1 : \{0, 1\}^* \rightarrow \mathbb{Z}_q$ and $H_2 : \{0, 1\}^* \rightarrow \{0, 1\}^k$ are modeled as independent random oracles.*

We refer the reader to Appendix 1 for the proof of Proposition 2.

Remark 3 (comparison with [12]) Yang et al. [12] argue that whenever the randomness of one session is identical to the randomness of another session of the same user, the adversary can learn the session key of either of the two sessions by performing a replay attack combined with a session-key query (as both sessions compute the same session key, but are non-matching). While Proposition 1 confirms that this statement holds for all protocols in the class AKE ∩ SL, we have shown that there exists a protocol in AKE, namely CNX, that achieves security even under such reset-and-replay attacks against the target session.

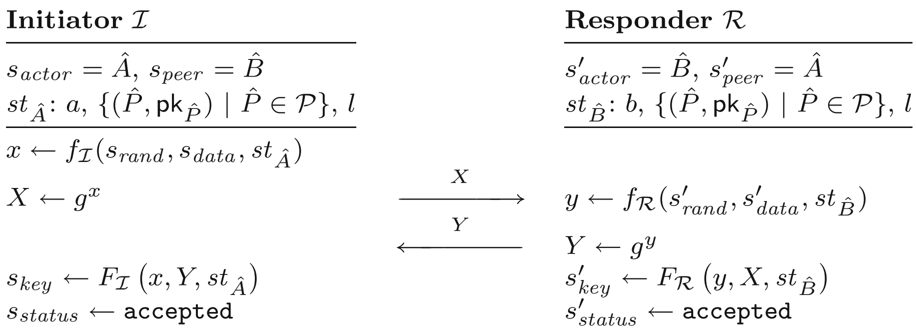

Fig. 2 Messages for generic two-message Diffie–Hellman type protocol in the class Λ

5 Impossibility result for chosen randomness

The CNX protocol prevents key repetition even in the presence of chosen randomness. The natural next question is: can we offer even more guarantees in the presence of chosen randomness? We tackle this question in three steps. In this section we prove an impossibility result for a class of protocols with respect to chosen randomness. We use this impossibility result to construct a stronger security model against chosen randomness in Sect. 6.1. Then, in Sect. 6.3, we construct a protocol that is secure in this model, thereby offering even stronger guarantees than CNX.

In this paper we restrict ourselves to the subclass Λ of the class AKE. The class Λ contains all two-message Diffie–Hellman type protocols that may only access and update user memory upon creation of sessions. Our results also apply to larger classes of AKE protocols, as we show in [27].

Definition 10 (*Protocol class* Λ) Let k be a security parameter. The protocol class Λ consists of all two-message protocols in the class AKE of the following form, specified by polynomial-time (in the security parameter k) computable functions $f_{\mathcal{I}}$, $f_{\mathcal{R}}$, $F_{\mathcal{I}}$, $F_{\mathcal{R}}$:

- Domain parameters (G, g, q), where $G = \langle g \rangle$ is a group of prime order q with $||q|| = k$, generated by g.
- $\mathsf{KeyGen}(1^k)$: Choose $a \in_R [0, q-1]$. Set $A \leftarrow g^a$. Return secret key $\mathsf{sk} = a$ and public key $\mathsf{pk} = A$.
- The specification of how users respond to create and send queries as well as how the session key is computed is given in Fig. 2.[3]
- The protocol can *only* access and update user memory upon creation of sessions.

The class Λ contains, e. g., the protocols NAXOS, HMQV, CMQV, CNX, and NXPR (presented in Sect. 6.3).[4] However, it does not contain, e. g., protocols providing message replay detection as such protocols need to access and update the list of received messages stored in the user memory upon receipt of a message.

We now provide an impossibility result for protocols in the class Λ. Theorem 1 shows the impossibility of achieving certain security guarantees in protocol class Λ. That is, given an

[3] Note that the ephemeral secret keys x and y can either be stored in a session-specific variable and reused in the key derivation phase or recomputed in the key derivation phase.

[4] In the long version of this paper, the class Λ is referred to as $\mathsf{INDP\text{-}DH} \cap \mathsf{ISM}$.

arbitrary protocol from class Λ, our impossibility result indicates attacks that are applicable to this protocol. From Theorem 1 we then derive a strong security model in Sect. 6.1 and show that there exists a protocol that is secure in this model in Sect. 6.3.

Theorem 1 *Let π be an arbitrary protocol in the class Λ. Let $X = (Q_{\mathsf{R}} \cup Q_{\mathsf{C}}, F)$ be the AKE security model with F being true for all sessions s and all sequences of queries. Let s^* denote the test session. There exist adversaries who win the security experiment $W(X)$ against protocol π with non-negligible probability by issuing either*

1. *a query* $\mathsf{session\text{-}key}(s^*)$*, or*
2. *a query* $\mathsf{session\text{-}key}(s')$*, where s' and s^* are matching sessions, or*
3. *a query* $\mathsf{corrupt}(s^*_{actor})$ *in combination with queries (*$\mathsf{randomness}(\tilde{s})$ *or* $\mathsf{cr\text{-}create}(\tilde{s})$*) on all sessions $\tilde{s}$ for which $\tilde{s}_{actor} = s^*_{actor}$ that were started not later than session s^* (and therefore includes s^*), or*
4. *for any session $s^\sharp$, such that $s^\sharp$ is an origin session for session s^*: a query* $\mathsf{corrupt}(s^\sharp_{actor})$ *in combination with queries (*$\mathsf{randomness}(\tilde{s})$ *or* $\mathsf{cr\text{-}create}(\tilde{s})$*) on all sessions $\tilde{s}$ for which $\tilde{s}_{actor} = s^\sharp_{actor}$ before the start of session $s^\sharp$, or*
5. *a query* $\mathsf{corrupt}(s^*_{peer})$ before *completion of session s^* and impersonating the peer s^*_{peer} to session s^*, or*
6. *a query* $\mathsf{corrupt}(s^*_{peer})$ after *completion of session s^* and impersonating the peer s^*_{peer} to session s^*.*

Proof Let $\pi \in \Lambda$. There exist PPT adversaries who win the security game $W(X)$ against protocol π with non-negligible probability, as follows.

Scenario 1 and 2: Since $\pi \in \Lambda$, π is also a member of AKE. By the definition of AKE protocols, an adversary E_1 can establish via a sequence of create and send queries two sessions s and s' that are matching according to Definition 2. He next issues the $\mathsf{test\text{-}session}$ query to one of the two sessions, say to session s. He then issues a $\mathsf{session\text{-}key}$ query to either session s (the test session) or s' (the matching session). Since by the definition of AKE, the matching sessions compute the same key, E_1 thereby learns the session key of session s.
Scenario 3: We consider a sequence of queries in which the adversary E_3 creates sessions by using create or $\mathsf{cr\text{-}create}$, and which contain at least two sessions s^* and s' that are matching. Such a sequence exists because $\pi \in \mathsf{AKE}$. E_3 now chooses session s^* as the test session. We use $\hat{A}$ to denote the actor of the test session, i.e., $\hat{A} = s^*_{actor}$. E_3 issues the query $\mathsf{corrupt}(\hat{A})$. E_3 also issues $\mathsf{randomness}(\tilde{s})$ for all sessions $\tilde{s}$ that (a) were started no later than the test session, (b) for which $\tilde{s}_{actor} = \hat{A}$, and (c) were not created using $\mathsf{cr\text{-}create}$. The adversary now has all public and secret information available to $\hat{A}$ for executing these sessions. In particular, E_3 also has the randomness of all sessions up to the test session, because he either chose it (by $\mathsf{cr\text{-}create}$) or revealed it (by $\mathsf{randomness}$). He can thus emulate the first created session of $\hat{A}$, denoted by s_1, by executing the algorithm Ψ on input $(1^k, st_{s_1}, st_{\hat{A}}, m')$. By the definition of Λ, the intermediate computations of the protocol only depend on values known to E_3, and it can therefore also compute the new contents of the state. By induction, E_3 can compute this for all subsequent sessions up to and including the test session s^*, which implies that it can compute the session key of s^*.
Scenario 4: We consider a sequence of queries in which the adversary E_4 creates sessions by using create or $\mathsf{cr\text{-}create}$, and which contain at least two sessions s^* and s' such that s' is an origin session for session s^* and $s^*_{status} = \mathtt{accepted}$. Such a sequence

exists because $\pi \in \Lambda$. E_4 now chooses session s^* as the test session. E_4 issues the query corrupt(s^*_{peer}). E_4 also issues randomness($\tilde{s}$) for all sessions $\tilde{s}$ that (a) were started no later than the origin session s', (b) for which $\tilde{s}_{actor} = s^*_{peer}$, and (c) were not created using cr-create. The adversary now has all public and secret information available to s^*_{peer} for executing these sessions. In particular, E_4 also has the randomness of all sessions up to the origin session, because he either chose it (by cr-create) or revealed it (by randomness). He can thus emulate the first created session of s^*_{peer}, denoted by s_1, by executing the algorithm Ψ on input $(1^k, st_{s_1}, st_{s^*_{peer}}, \epsilon)$. By the definition of Λ, the intermediate computations of the protocol only depend on values known to E_4, and it can therefore also compute the new contents of the state. By induction, E_4 can compute this for all subsequent sessions up to and including the origin session s'. The adversary now emulates the session key computation of a matching session and computes the session key of session s^* by executing Ψ on input $(1^k, st_{s'}, st_{s^*_{peer}}, m)$, where m denotes the outgoing message of session s^*.
Scenario 5: Adversary E_5 issues a corrupt query to some user, say user $\hat{Q}$. He then creates a responder session s by issuing the query create($\hat{P}, \mathcal{R}, \hat{Q}$). The adversary now impersonates user $\hat{Q}$ to s_{actor} as follows. E_5 chooses randomness $r \in_R \{0,1\}^k$ and runs the protocol with $\hat{P}$ on behalf of $\hat{Q}$ by executing the algorithm Ψ. The algorithm Ψ executed by the adversary takes as input, among others, the user state of user $\hat{Q}$ containing its long-term secret key $\mathsf{sk}_{\hat{Q}}$ and the set $\mathcal{L}$ returned as response to the public-info query. Once session s has accepted, he chooses the latter as the test session. The adversary can compute the session key of session s, for which no origin session exists, by emulating a matching session.
Scenario 6: Adversary E_6 first creates an initiator session s at $\hat{A}$ with peer $\hat{B}$ via the query create($\hat{A}, \mathcal{I}, \hat{B}$) and receives as a response the message $m = X$, where X is the Diffie–Hellman exponential generated in session s. E_6 chooses a value $z \in_R \mathbb{Z}_q$, computes $Z = g^z$, and sends message $\tilde{m} = Z$ to session s. Upon receiving message $\tilde{m}$ in session s, $\hat{A}$ executes $\Psi(1^k, st_s, st_{\hat{A}}, \tilde{m})$. E_6 then chooses the completed session s as the test session and reveals the long-term secret key of user $\hat{B}$ via the query corrupt($\hat{B}$). This enables him to compute the session key of the test session as $F_{\mathcal{R}}(f_{\mathcal{R}}(z, st_{\hat{B}}), st_{\hat{B}}, X)$. Note that the query public-info returned the initial values of additional variables stored in the user memory. This is a generalisation of Krawczyk's attack [16, p. 15]. □

6 Stronger security against chosen randomness

6.1 Security model

Theorem 1 gives rise to the model Ω_Λ below. In contrast to previous AKE security models (including eCK, CR-eCK^w, and Reset-1), the Ω_Λ model permits the adversary to compromise the randomness of the target session and the long-term secret key of the actor of that session as long as the randomness of at least one of the previous sessions of the same user has not been compromised. A similar statement holds for sessions that are origin sessions for the target session.

Definition 11 The model Ω_Λ is defined by (Q, F), where $Q = Q_\mathsf{R} \cup Q_\mathsf{C}$ and a session s is said to satisfy F if all of the following conditions hold:

1. no session-key(s) has been issued,

2. for all sessions s^* such that s^* matches s, no $\mathsf{session\text{-}key}(s^*)$ query has been issued,
3. not all queries $\mathsf{corrupt}(s_{actor})$ as well as ($\mathsf{randomness}$ or $\mathsf{cr\text{-}create}$) on all sessions $\tilde{s}$ with $\tilde{s}_{actor} = s_{actor}$, where the query create or $\mathsf{cr\text{-}create}$ creating session $\tilde{s}$ occurred before or at creation of session s, have been issued,
4. for all sessions s' such that s' is an origin session for session s, not all queries $\mathsf{corrupt}(s_{peer})$ as well as ($\mathsf{randomness}$ or $\mathsf{cr\text{-}create}$) on all sessions $\tilde{s}$ with $\tilde{s}_{actor} = s'_{actor}$, where the query create or $\mathsf{cr\text{-}create}$ creating session $\tilde{s}$ occurred before or at creation of session s', have been issued, and
5. if there exists no origin session for session s, then no $\mathsf{corrupt}(s_{peer})$ query has been issued.

Proposition 3 *The* CNX *protocol is insecure in model Ω_Λ.*

Proof of Proposition 3. The following attack shows that the CNX protocol is insecure in Ω_Λ. The adversary creates an initiator session s of user $\hat{A}$ via the query $\mathsf{create}(\hat{A}, \mathcal{I}, \hat{B})$ and an initiator session s” of user $\hat{B}$ by issuing the query $\mathsf{create}(\hat{B}, \mathcal{I}, \hat{C})$. He then creates a responder session s' via the query $\mathsf{create}(\hat{B}, \mathcal{R}, \hat{A})$ and activates session s' by sending the message $X = g^x$ sent by session s to session s'. The adversary then sends message Y sent by session s' to session s. Session s accepts the key $s_{key} = H_2(Y^a, B^{H_1(s_{rand}, a, s_{data})}, Y^{H_1(s_{rand}, a, s_{data})}, \hat{A}, \hat{B})$ as the session key, while session s' accepts as its key $s'_{key} = H_2(A^{H_1(s'_{rand}, b, s'_{data})}, X^b, X^{H_1(s'_{rand}, b, s'_{data})}, \hat{A}, \hat{B})$. The completed session s is chosen as the test session. Now a $\mathsf{randomness}$ query to session s' revealing the randomness of session s' followed by a $\mathsf{corrupt}(\hat{B})$ query revealing the long-term secret key of user $\hat{B}$, allows the adversary to compute the session key of the test session s (as he knows the counter value used in session s'). Note that the test session is fresh in Ω_Λ since the adversary did issue neither the query $\mathsf{randomness}$ nor the query $\mathsf{cr\text{-}create}$ to session s” of user $\hat{B}$, where the query $\mathsf{create}(\hat{B}, \mathcal{I}, \hat{C})$ creating session s'' occurred before creation of session s'. □

Proposition 4 *The model Ω_Λ is stronger than the model CR-eCK^w with respect to* AKE.

Proof We first show that model Ω_Λ is at least as strong as model CR-eCK^w. The first condition of Definition 5 is satisfied as matching is defined in the same way for both models Ω_Λ and CR-eCK^w. To see that the second condition of Definition 5 holds, it suffices to show that if there exists an adversary E such that the probability of event Multiple-Match$_{\pi,E}^{W(\text{CR-eCK}^w)}(k)$ is non-negligible, then there exists an adversary E' such that the probability of event Multiple-Match$_{\pi,E'}^{W(\Omega_\Lambda)}(k)$ is non-negligible. This is straightforward. Let $\pi \in \Pi$. To show that the third condition of Definition 5 holds, we construct an adversary E' attacking protocol π in model Ω_Λ using an adversary E attacking π in model CR-eCK^w, where $\pi \in \mathsf{AKE}$. Whenever E issues a query $q \in Q_{\mathsf{noCR}} \cup \{\mathsf{cr\text{-}create}, \mathsf{test\text{-}session}\}$, adversary E' issues the same query and forwards the answer received to E. Note that if the freshness condition of CR-eCK^w holds for the test session, then the freshness condition of Ω_Λ is also satisfied.

The CNX protocol provides an example of a protocol that is secure in model CR-eCK^w, but insecure in model Ω_Λ (see Proposition 3). □

6.2 Security against repeated randomness failures

In this section we show that security in our model Ω_Λ implies security against repeated randomness. To this end, we introduce the query $\mathsf{reset\text{-}create}$, defined below, which allows the adversary to create a session that uses the same randomness as used in a previous session of the same user.

The query reset-create below creates a new session with the same randomness as used in a previous session of the same user. Practically, this query models a flawed RNG that produces the same value more than once. A similar query is found in the Reset-2 model of Yang et al. [12].

- reset-create$(\hat{P}, r, i[, \hat{Q}])$. The query models the creation of a new session s, using the same randomness as in session $s' = (\hat{P}, i)$, for the user $\hat{P}$. The query requires that $\hat{P} \in \mathcal{P}$, $\hat{Q} \in \mathcal{P}$, $r \in \{\mathcal{I}, \mathcal{R}\}$, and that $s'_{status} \neq \perp$; otherwise, it returns $\perp$. The session variables are initialized as

  $$(s_{actor}, s_{role}, s_{sent}, s_{recv}, s_{status}, s_{key}, s_{rand}, s_{step}) \leftarrow (\hat{P}, r, \epsilon, \epsilon, \texttt{active}, \perp, s'_{rand}, 1)\ .$$

  If the optional peer identifier $\hat{Q}$ is provided, the variable s_{peer} is set to $\hat{Q}$. The key exchange algorithm Ψ is executed on input $(1^k, st_s, st_{\hat{P}}, \epsilon)$. The algorithm returns a triple $(m', st'_s, st'_{\hat{P}})$. We set $st_s \leftarrow st'_s$ and $st_{\hat{P}} \leftarrow st'_{\hat{P}}$. The query returns m'.

Proposition 5 below states that security in a model that permits the adversary to reveal the randomness used in sessions and to control the randomness of sessions implies security against repeated randomness failures.

The relative strengths of security between game-based security models was investigated by Choo et al. [28], and formally defined by Cremers and Feltz [19] as follows. Let $secure(X, \Pi)$ be a predicate that is true if and only if the protocol Π is secure in security model X.

Definition 12 [19] Let π be a class of AKE protocols. Let X and Y be two security models. We say that model Y is *at least as strong as* model X with respect to π, denoted by $X \leq^{\pi}_{Sec} Y$, if

$$\forall\, \Pi \in \pi.\ secure(Y, \Pi) \Rightarrow secure(X, \Pi). \tag{1}$$

We say that model Y is *stronger* than model X with respect to protocol class π, if $X \leq^{\pi}_{Sec} Y$ and not $Y \leq^{\pi}_{Sec} X$.

Proposition 5 *Let F be defined as follows: a session s is said to satisfy F if no* session-key(s) *query has been issued and, for all sessions s^* such that s^* matches s, no* session-key(s^*) *query has been issued. Let $X = (Q, F)$ with $Q = Q_{\mathsf{R}} \cup \{$*reset-create, session-key*$\}$. Let $Y = (Q', F)$ with $Q' = Q_{\mathsf{R}} \cup \{$*cr-create, randomness, session-key*$\}$. The model Y is at least as strong as the model X w.r.t.* AKE.

Proof The first condition of Definition 5 is satisfied since matching is defined in the same way for both models X and Y. Let $\pi \in \Pi$. To show that the second and third condition of Definition 5 hold, we construct an adversary E' attacking protocol π in model Y using an adversary E attacking π in model X. Adversary E' proceeds as follows. Whenever E issues a query create, send, session-key or test-session, adversary E' issues the same query and forwards the answer received to E. Whenever E issues a query reset-create$(\hat{P}, r, i[, \hat{Q}])$ to create a new session of user $\hat{P}$, adversary E' first checks whether the status of session $s = (\hat{P}, i)$ is different from $\perp$. If this is the case, then E' issues the following sequence of queries: 1. randomness$(\hat{P}, i)$, and 2. cr-create$(\hat{P}, r, s_{rand}[, \hat{Q}])$. At the end of E's execution, i.e. after it has output its guess bit b, E' outputs b as well. Hence, it holds that $Adv^{\pi,E}_{W(X)}(k) \leq Adv^{\pi,E'}_{W(Y)}(k)$, where k denotes the security parameter. Since by assumption protocol π is secure in Y, there is a negligible function g such that $Adv^{\pi,E'}_{W(Y)}(k) \leq g(k)$. It follows that protocol π is secure in X. □

Corollary 1 follows from Proposition 5 and from the fact that Implication (1) is transitive.

Corollary 1 *Let X and Y be defined as in Proposition 5. Let* $Z = (Q'', F'')$ *with* $Q_{\mathsf{R}} \subseteq Q'' \subseteq Q_{\mathsf{R}} \cup Q_{\mathsf{C}}$. *If model Z is at least as strong as model Y with respect to* AKE, *then model Z is also at least as strong as model X with respect to* AKE.

We show in Proposition 6 that security in Ω_Λ implies security against repeated randomness, modelled by the query reset-create. Hence, security in Ω_Λ also implies security against reset-and-replay attacks on the target session or its partner session.

Proposition 6 *Let F be defined as follows: a session s is said to satisfy F if no* session-key(s) *query has been issued and, for all sessions* s^* *such that* s^* *matches s, no* session-key(s^*) *query has been issued. Let* $X = (Q, F)$ *with* $Q = Q_{\mathsf{R}} \cup \{$reset-create, session-key$\}$. *The model* Ω_Λ *is at least as strong as the model X with respect to* AKE.

Proof Let Y be defined as in Proposition 5. The proof that model Ω_Λ is at least as strong as the model Y with respect to AKE proceeds in a similar way as the proof of Proposition 5. The proposition then follows from Corollary 1. □

6.3 NXPR: achieving strong AKE security against chosen randomness

In this section we first describe a general method of how to prevent attacks based on the compromise of random values that are used in sessions of a user and the long-term secret keys of that user. We then apply our method to a concrete protocol, namely to the NAXOS protocol.

Let $\pi \in \Lambda \cap \mathsf{SL}$, where the protocol class Λ is defined in Definition 10. Let $H : \{0, 1\}^* \to \mathbb{Z}_q$ (with $||q|| = k$, where k denotes the security parameter) be a hash function. We propose the following security-strengthening method to build a protocol π' with the goal to achieve security even under compromise of the randomness of the target session and the long-term secret key of the actor of that session as long as the randomness of at least one of the previous sessions of the same user has not been compromised. The resulting protocol π' includes the user's long-term private key, the current session's randomness, and the randomness of all sessions (of the same user) that have been previously created, as input to the hash function H. The protocol uses the user's state to store the concatenation of all previously generated random values of the user. More precisely, protocol π' is defined in the same way as protocol π except that in protocol π':

- Each user $\hat{P} \in \mathcal{P}$ maintains a variable $l \in \{0, 1\}^*$ initialized with the empty string ϵ. This variable l stores the concatenation of all previously generated random values in sessions of user $\hat{P}$. It is stored in the user memory $st_{\hat{P}}$. We write $st_{\hat{P}}.l$ to access the variable l of user $\hat{P}$.
- The user memory of each user $\hat{P} \in \mathcal{P}$ is given by $st_{\hat{P}}^{\pi'} = (st_{\hat{P}}^{\pi}, l)$.
- The functions $f_{\mathcal{I}}$ and $f_{\mathcal{R}}$ (of Definition 10) are instantiated as follows in protocol π':

$$f_{\mathcal{I}}(r, v, st_{\hat{A}}^{\pi'}) = H(r, v, a),$$

where v denotes the concatenation of the random values that have been generated in all the previous sessions of user $\hat{A}$;

$$f_{\mathcal{R}}(r, v, st_{\hat{B}}^{\pi'}) = H(r, v, b),$$

where v denotes the concatenation of the random values that have been generated in all the previous sessions of user $\hat{B}$;

Initiator $\mathcal{I}$		Responder $\mathcal{R}$
$st_{\hat{A}}$: a, $\{(\hat{P}, \mathsf{pk}_{\hat{P}}) \mid \hat{P} \in \mathcal{P}\}$, l		$st_{\hat{B}}$: b, $\{(\hat{P}, \mathsf{pk}_{\hat{P}}) \mid \hat{P} \in \mathcal{P}\}$, l
$s_{data} \leftarrow st_{\hat{A}}.l$		$s'_{data} \leftarrow st_{\hat{B}}.l$
$st_{\hat{A}}.l \leftarrow (s_{rand}, s_{data})$		$st_{\hat{B}}.l \leftarrow (s'_{rand}, s'_{data})$
$X \leftarrow g^{H_1(s_{rand}, s_{data}, a)}$	$\xrightarrow{X}$	
		$Y \leftarrow g^{H_1(s'_{rand}, s'_{data}, b)}$
		$\sigma_1 \leftarrow A^{H_1(s'_{rand}, s'_{data}, b)}$
		$\sigma_2 \leftarrow X^b$
		$\sigma_3 \leftarrow X^{H_1(s'_{rand}, s'_{data}, b)}$
		$s'_{key} \leftarrow H_2(\sigma_1, \sigma_2, \sigma_3, \hat{A}, \hat{B})$
$\sigma_1 \leftarrow Y^a$	$\xleftarrow{Y}$	$s'_{status} \leftarrow \mathtt{accepted}$
$\sigma_2 \leftarrow B^{H_1(s_{rand}, s_{data}, a)}$		
$\sigma_3 \leftarrow Y^{H_1(s_{rand}, s_{data}, a)}$		
$s_{key} \leftarrow H_2(\sigma_1, \sigma_2, \sigma_3, \hat{A}, \hat{B})$		
$s_{status} \leftarrow \mathtt{accepted}$		

Fig. 3 The NXPR protocol

– The key exchange algorithm for protocol π' is defined in a similar way as the one for protocol π except that if $s_{step} = 1$, the algorithm for π' executes the instructions

$$s_{data} \leftarrow st^{\pi'}_{\hat{P}}.l;$$
$$st^{\pi'}_{\hat{P}}.l \leftarrow (s_{rand}, s_{data})$$

before executing the remaining instructions of protocol π (with $f_{\mathcal{I}}$, $f_{\mathcal{R}}$ as defined in the previous point).
In other words, upon creation of a session of user $\hat{P}$, the value of the variable l, which stores the concatenation of all previously generated random values of the user, is assigned to the variable s_{data}. The variable l is then updated and assigned the concatenation of the randomness that has been generated in the current session and the randomness that has been generated in all the previous sessions of the user.

We next apply our security-strengthening method to the NAXOS protocol. Let H_1, H_2, (a, A), and (b, B) be defined as in Sect. 4. Applying our security-strengthening method to the NAXOS protocol yields a new stateful protocol, which we call the NXPR ("NaXos with Previous Randomness") protocol, shown in Fig. 3. We show in Proposition 7 that the NXPR protocol is secure in the Ω_Λ model. It thus provides even stronger security guarantees than the CNX protocol.

If we compare NXPR to the CNX protocol, we see that CNX used the user's state to prevent key repetition, essentially by implementing a counter. NXPR has an implicit counter (i.e., the number of previously generated random values) and incorporates all randomness previously generated by the user. This construction ensures that partially leaking or choosing randomness does not imply loss of the exponent $H_1(\ldots)$, even if the adversary knows a user's long-term secret key.

Proposition 7 *Let k be a security parameter. Under the GAP-CDH assumption* [26] *in the cyclic group G of prime order q with $||q|| = k$, the* NXPR *protocol is secure in the Ω_Λ*

model, when the hash functions $H_1 : \{0,1\}^* \rightarrow \mathbb{Z}_q$ *and* $H_2 : \{0,1\}^* \rightarrow \{0,1\}^k$ *are modeled as independent random oracles.*

We refer the reader to Appendix 2 for the proof of Proposition 7.

Intuitively, the NXPR protocol is resilient against the attack described in the proof of Proposition 1 since if the adversary creates a responder session s'' at $\hat{B}$ via the query cr-create$(\hat{B}, \mathcal{R}, rnd, \hat{A})$, where rnd are the random coins used in the previous responder session s' of user $\hat{B}$, then the session key computed in session s'' is different (and independent via the RO assumption) from the session key computed in session s' since the outgoing message of session s'' is different to the message sent in session s'.

Remark 4 (user state comparison of NXPR to CNX) In contrast to the CNX protocol, the NXPR protocol requires each user to store the concatenation of the randomness generated in all his sessions in the user memory. Thus, before the n'th session of user $\hat{P}$ is created (where $n > 1$), the user memory $st_{\hat{P}}$ contains its long-term secret key, the long-term public key of all users $\hat{Q} \in \mathcal{P}$, and the concatenation of $n-1$ bit strings of length k corresponding to the randomness generated in all previous sessions of user $\hat{P}$.

Remark 5 Instead of concatenating the randomness used in all previous sessions with the current randomness, one could determine the randomness of the n'th session of user $\hat{A}$, with long-term secret key a, as $H_1(s^n_{rand}, H_1(s^{n-1}_{rand}, H_1(\ldots, H_1(s^1_{rand}, a))))$, i.e., the hash of the concatenation of the current session's randomness together with a previously stored value of fixed length k. The advantage of this minor modification would be that each user needs to store its long-term secret key and a bit string of fixed length k, which is updated upon activation of new session of the user.

7 Conclusions

In this paper we explored the limits of AKE security with respect to adversaries who can perform chosen-randomness attacks, whereby they control the randomness used in protocol sessions. While stateless protocols fail to achieve security against attacks based on this worst case randomness failure, we constructed stateful variants of the NAXOS protocol that provide security even against such attacks. We use the user's state only locally: our protocols do not require any form of state synchronisation between communicating users. Our new protocols allow us to weaken the assumptions made on the security of the RNG used to generate session-specific randomness.

Stepping back, we see that our analysis identifies the double purpose of randomness in AKE protocols. The first purpose of randomness in AKE is to ensure that the session keys of subsequent sessions are different. Both the CNX and the NXPR protocol leverage the users' state to ensure this goal is met even when the RNG fails. The second purpose of randomness in AKE is to provide a session-specific secret. In state-of-the-art protocols this session-specific secret is combined with the user's long-term secret to serve as a combined secret for the session key. For these protocols this means that if the randomness of a specific session is bad (e.g., chosen or predictable for the adversary) then the session-key is only protected by the long-term key. In contrast, our NXPR protocol reduces the impact of some bad randomness by combining all previously generated randomness with the long-term key to protect the session key.

It is intriguing that all broken/flawed/subverted RNGs that we mentioned in the introduction already maintain local state as part of their design. However, the local state of the

RNG serves a different purpose than what we need for AKE protocols. History has made abundantly clear that the presence of local state in the RNG's design does not guarantee that it has not been subverted [6,29].

Moving forward, we hope that future practical protocols can be made less dependent on their complex, and often external, RNG libraries. For example, in TLS 1.2, the RNG is a single point of failure: if an adversary observes a DH-based TLS handshake and learns/predicts/chooses the randomness used by one user in this session, he can directly compute the session keys, even without knowing any long-term secrets. This is a weakness in the design of TLS that enables an adversary to turn the NIST RNG backdoor into an actual attack on Internet traffic, as described in [6]. Given the history surrounding RNG subversion and its role in Internet communications, we argue that it is prudent to make the next versions of protocols such as TLS and IKE more resilient against RNG problems. The technical means are available.

Acknowledgements Funding was provided by ETH Research Grant ETH-30 09-3.

Appendix 1: Proof of Proposition 2

Proof It is straightforward to verify the first condition of Definition 5. We next verify that the second condition of Definition 5 holds. Let E denote a PPT adversary against protocol $\pi :=$ CNX. We show that the probability of event $\text{Multiple-Match}_{\pi,E}^{W(\text{CR-eCK}^w)}(k)$ is bounded above by a negligible function in the security parameter k, where $\text{Multiple-Match}_{\pi,E}^{W(\text{CR-eCK}^w)}(k)$ denotes the event that, in the security experiment, there exist a session s with $s_{status} =$ `accepted` and at least two distinct sessions s' and s'' that are matching session s. Note that, if both sessions s' and s'' are matching session s, then it must hold that $s''_{actor} = s'_{actor}$ and $s''_{role} = s'_{role}$). In addition, the counter value in two different sessions of the same user are distinct. For some fixed session s that has accepted, let Ev denote the event that there exist two distinct sessions s' and s'' such that s and s' are matching as well as s and s''. We have:

$$\begin{aligned} P(Ev) &\leq P\Big(\bigcup_{\substack{s',s'' \\ s'\neq s''}} \{H_1(s''_{rand}, \mathsf{sk}_{\hat{P}}, i) = H_1(s'_{rand}, \mathsf{sk}_{\hat{P}}, j)\}\Big) \\ &\leq \sum_{\substack{s',s'' \\ s'\neq s''}} P(\{H_1(s''_{rand}, \mathsf{sk}_{\hat{P}}, i) = H_1(s'_{rand}, \mathsf{sk}_{\hat{P}}, j)\}) \\ &\leq q_s^2 \frac{1}{p}, \end{aligned}$$

where $\hat{P} = s''_{actor} = s'_{actor}$, $i \neq j$ and q_s denotes the number of created sessions (either via the create or the $\mathsf{cr\text{-}create}$ query) by the adversary. Therefore, $P(\text{Multiple-Match}_{\pi,E}^{W(\text{CR-eCK}^w)}(k)) \leq q_s^3 \frac{1}{p}$.

The third condition of Definition 5 is implied by an adaptation of the security proof of NAXOS in the eCK^w model from [19]. Let s^* denote the test session. Consider first the event K^c where the adversary M wins the security experiment against π with non-negligible advantage and does not query H_2 with $(\sigma_1, \sigma_2, \sigma_3, \hat{A}, \hat{B})$, where $\sigma_1 = CDH(Y, A), \sigma_2 = CDH(B, X)$ and $\sigma_3 = CDH(X, Y)$.

Event K^c

If event K^c occurs, then the adversary M must have issued a session-key query to some session s such that $K_s = K_{s^*}$ (where K_s and K_{s^*} denote the session keys computed in sessions s and s^*, respectively) and s does not match s^*. We consider the following four events:

1. A_1: there exist two distinct sessions s', s'' created via a create query such that $s'_{rand} = s''_{rand}$.
2. A_2: there exists a session $s \neq s^*$ such that $H_1(s_{rand}, \mathsf{sk}_{s_{actor}}, i) = H_1(s^*_{rand}, \mathsf{sk}_{s^*_{actor}}, j)$.
3. A_3: there exists a session $s' \neq s^*$ such that $H_2(\text{input}_{s'}) = H_2(\text{input}_{s^*})$ with $\text{input}_{s'} \neq \text{input}_{s^*}$.
4. A_4: there exists an adversarial query input_M to the oracle H_2 such that $H_2(\text{input}_M) = H_2(\text{input}_{s^*})$ with $\text{input}_M \neq \text{input}_{s^*}$.

In contrast to the NAXOS protocol with respect to model CR-eCK^w, the adversary cannot force two sessions of protocol π of the same user with the same role to compute the same session key via a chosen-randomness replay attack, as the H_1 values in both sessions will be different with overwhelming probability due to different counter values. The latter event is included in event A_2.

Analysis of event K^c

We denote by q_s the number of created sessions (either via the create or the cr-create query) by the adversary and by q_{ro2} the number of queries to the random oracle H_2. We have that

$$P(K^c) \leq P(A_1 \vee A_2 \vee A_3 \vee A_4) \leq P(A_1) + P(A_2) + P(A_3) + P(A_4)$$
$$\leq \frac{q_s^2}{2}\frac{1}{2^k} + \frac{q_s}{p} + \frac{q_s + q_{\text{ro2}}}{2^k},$$

which is a negligible function of the security parameter k.

In the subsequent events (and their analyses) we assume that no collisions in the queries to the oracle H_1 occur and that none of the events $A_1, \ldots, A_4$ occurs. As in the proof of [19, Proposition 7], we next consider the following three events:

1. $DL \wedge K$,
2. $T_O \wedge DL^c \wedge K$, and
3. $(T_O)^c \wedge DL^c \wedge K$, where

T_O denotes the event that there exists an origin-session for the test session, DL denotes the event where there exists a user $\hat{C} \in \mathcal{P}$ such that the adversary M, during its execution, queries H_1 with $(*, c, *)$ before issuing a corrupt$(\hat{C})$ query and K denotes the event that M wins the security experiment against NAXOS by querying H_2 with $(\sigma_1, \sigma_2, \sigma_3, \hat{A}, \hat{B})$, where $\sigma_1 = CDH(Y, A)$, $\sigma_2 = CDH(B, X)$ and $\sigma_3 = CDH(X, Y)$.

Event $DL \wedge K$

Let the input to the GAP-DLog challenge be C. Suppose that event $DL \wedge K$ occurs with non-negligible probability. In this case, the simulator S chooses one user $\hat{C} \in \mathcal{P}$ at random and sets its long-term public key to C. S chooses long-term secret/public key pairs for the remaining honest parties and stores the associated long-term secret keys. Additionally S chooses a random value $m \in_R \{1, 2, \ldots, q_s\}$. We denote the m'th activated session by adversary M by s^*. Suppose further that $s^*_{actor} = \hat{A}$, $s^*_{peer} = \hat{B}$ and $s^*_{role} = \mathcal{I}$, w.l.o.g. We now define S's responses to M's queries for the pre-specified peer setting; the post-specified peer case proceeds similarly. Algorithm S maintains tables Q, J, T and L, all of which are initially empty. S also maintains a variable ω initialized with 1 and a table CV maintaining for each user the current counter value. Initially, table CV contains an entry $(\hat{P}, 0)$ for each user $\hat{P} \in \mathcal{P}$.

1. $\mathsf{create}\left(\hat{P}, r, \hat{Q}\right)$ to create session s: S checks whether $\hat{P} \in \mathcal{P}$, $\hat{Q} \in \mathcal{P}$, and $r \in \{\mathcal{I}, \mathcal{R}\}$. If one of the checks fails, then S returns $\bot$. Else, S initializes the session variables according to the protocol specification, and stores an entry of the form $\left(s, s_{rand}, \mathsf{sk}_{s_{actor}}, l_s, \kappa\right) \in (\mathcal{P} \times \mathbb{N}) \times \{0, 1\}^k \times (\mathbb{Z}_p \cup \{*\}) \times \mathbb{N} \times \mathbb{Z}_p$ in table Q as follows:
   - S retrieves the counter value c for the user with identifier $\hat{P}$ from table CV, increments c by 1, and updates the counter value for $\hat{P}$ stored in table CV with $c + 1$,
   - S chooses $s_{rand} \in_R \{0, 1\}^k$ (i.e. the randomness of session s),
   - S chooses $\kappa \in_R \mathbb{Z}_p$,
   - if $s_{actor} \neq \hat{C}$, then S stores the entry $\left(s, s_{rand}, \mathsf{sk}_{s_{actor}}, c + 1, \kappa\right)$ in Q, else S stores the entry $(s, s_{rand}, *, c + 1, \kappa)$ in Q,[5] and
   - if $r = \mathcal{I}$, then S returns the Diffie–Hellman exponential g^κ to M, else S returns $\star$.
2. $\mathsf{cr\text{-}create}\left(\hat{P}, r, str, \hat{Q}\right)$ to create session s: S checks whether $\hat{P} \in \mathcal{P}$, $\hat{Q} \in \mathcal{P}$, and $r \in \{\mathcal{I}, \mathcal{R}\}$. If one of the checks fails, then S returns $\bot$. Else, S initializes the session variables according to the protocol specification, and stores an entry of the form $\left(s, s_{rand}, \mathsf{sk}_{s_{actor}}, l_s, \kappa\right) \in (\mathcal{P} \times \mathbb{N}) \times \{0, 1\}^k \times (\mathbb{Z}_p \cup \{*\}) \times \mathbb{N} \times \mathbb{Z}_p$ in table Q as follows:
   - S retrieves the counter value c for the user with identifier $\hat{P}$ from table CV, increments c by 1, and updates the counter value for $\hat{P}$ stored in table CV with $c + 1$,
   - if there is an entry $(r_i, h_i, l_i, \kappa_i)$ in table J such that $r_i = str$, $h_i = \mathsf{sk}_{\hat{P}}$, and $l_i = c + 1$, then S sets $\omega \leftarrow \kappa_i$, else S chooses $\kappa \in_R \mathbb{Z}_p$, and sets $\omega \leftarrow \kappa$.[6]
   - if $s_{actor} \neq \hat{C}$, then S stores the entry $\left(s, s_{rand}, \mathsf{sk}_{s_{actor}}, c + 1, x_5\right)$ in Q, else S stores the entry $(s, s_{rand}, *, c + 1, x_5)$ in Q, where x_5 denotes the value of variable ω,
   - if $r = \mathcal{I}$, then S returns the Diffie–Hellman exponential g^κ to M, else S returns $\star$.
3. S stores entries of the form $(r, h, l, \kappa) \in \{0, 1\}^k \times \mathbb{Z}_p \times \mathbb{N} \times \mathbb{Z}_p$ in table J. When M makes a query of the form (r, h, l) to the random oracle for H_1, answer it as follows:
   - If $C = g^h$, then S aborts M and is successful by outputting $\mathrm{DLog}_g(C) = h$.
   - Else if $(r, h, l, \kappa) \in J$ for some $\kappa \in \mathbb{Z}_p$, then S returns κ to M.
   - Else if there exists an entry $\left(s, s_{rand}, \mathsf{sk}_{s_{actor}}, l_s, \kappa\right)$ in Q, for some $s \in \mathcal{P} \times \mathbb{N}$, $s_{rand} \in \{0, 1\}^k$, $\mathsf{sk}_{s_{actor}} \in \mathbb{Z}_p$, $l_s \in \mathbb{N}$ and $\kappa \in \mathbb{Z}_p$, such that $s_{rand} = r$, $\mathsf{sk}_{s_{actor}} = h$ and $l_s = l$, then S returns κ to M and stores the entry (r, h, l, κ) in table J.
   - Else, S chooses $\kappa \in_R \mathbb{Z}_p$, returns it to M and stores the entry (r, h, l, κ) in table J.
4. $\mathsf{send}(\hat{P}, i, V)$ to send message V to session $s = (\hat{P}, i)$: If $s_{status} \neq \texttt{active}$, then S returns $\bot$. Else if $s_{role} = \mathcal{I}$, then S does the following. If $V \notin G$, then the status of session s is set to `rejected`. Else, the status of session s is set to `accepted`, the variable $recv$ is updated to $s_{recv} \leftarrow (s_{recv}, V)$ and
   - If there exists an entry $\left(s_{peer}, s_{actor}, \mathcal{R}, s_{recv}, s_{sent}, \lambda\right)$ in table T, then S stores the entry $\left(s_{actor}, s_{peer}, \mathcal{I}, s_{sent}, s_{recv}, \lambda\right)$ in table T.

[5] We do not need to keep consistency with H_1 queries via lookup in table J since the probability that the adversary guesses the randomness of a session created via a query create is negligible.

[6] Here we need to keep consistency with H_1 queries via lookup in table J to be able to consistently answer all possible combinations of queries. Consider, e. g., the following scenario. The adversary first issues a query $(x, \mathsf{sk}_{\hat{P}}, i)$ to H_1 and then issues the query $\mathsf{cr\text{-}create}(\hat{P}, r, x, \hat{Q})$, which increments the current counter value $i - 1$ by 1 so that the counter value used in session $s = (\hat{P}, i)$ is i. So, in contrast to the NAXOS proof with respect to model eCK^w, we need to additionally keep consistency between $\mathsf{cr\text{-}create}$ queries and queries to the random oracle for H_1.

- Else if there exists an entry $\left(\sigma_1, \sigma_2, \sigma_3, s_{actor}, s_{peer}, \lambda\right)$ in table L, for some $\lambda \in \{0,1\}^k$, such that $\mathrm{DDH}(s_{recv}, s_{sent}, \sigma_3) = 1$, $\mathrm{DDH}(s_{sent}, \mathsf{pk}_{s_{peer}}, \sigma_2) = 1$ and $\mathrm{DDH}(s_{recv}, \mathsf{pk}_{s_{actor}}, \sigma_1) = 1$, then S stores $\left(s_{actor}, s_{peer}, \mathcal{I}, s_{sent}, s_{recv}, \lambda\right)$ in table T.
- Else, S chooses $\mu \in_R \{0,1\}^k$ and stores the entry $(s_{actor}, s_{peer}, \mathcal{I}, s_{sent}, s_{recv}, \mu)$ in T.

Else if $s_{role} = \mathcal{R}$, then S does the following. If $V \notin G$, then the status of session s is set to `rejected`. Else, S sets the status of session s to `accepted`, and the variable $recv$ to (s_{recv}, V). S returns g^{κ} to M, where κ denotes the last element of the entry $\left(s, r, \mathsf{sk}_{s_{actor}}, l, \kappa\right)$ in table Q, and proceeds in a similar way as in the previous case.

5. When M makes a query of the form $\left(\sigma_1, \sigma_2, \sigma_3, \hat{P}_i, \hat{P}_j\right)$ to the random oracle for H_2, answer it as follows:
   - If $\left(\sigma_1, \sigma_2, \sigma_3, \hat{P}_i, \hat{P}_j, \lambda\right) \in L$ for some $\lambda \in \{0,1\}^k$, then S returns λ to M.
   - Else if there exist entries $\left(\hat{P}_i, \hat{P}_j, \mathcal{I}, U, V, \lambda\right)$ or $\left(\hat{P}_j, \hat{P}_i, \mathcal{R}, V, U, \lambda\right)$ in table T, for some $\lambda \in \{0,1\}^k$ and $U, V \in G$, such that $\mathrm{DDH}(V, U, \sigma_3) = 1$, $\mathrm{DDH}(V, \mathsf{pk}_{\hat{P}_i}, \sigma_1) = 1$ and $\mathrm{DDH}(U, \mathsf{pk}_{\hat{P}_j}, \sigma_2) = 1$, then S returns λ to M and stores the entry $\left(\sigma_1, \sigma_2, \sigma_3, \hat{P}_i, \hat{P}_j, \lambda\right)$ in table L.
   - Else, S chooses $\mu \in_R \{0,1\}^k$, returns it to M and stores the entry $\left(\sigma_1, \sigma_2, \sigma_3, \hat{P}_i, \hat{P}_j, \mu\right)$ in L.
6. $\mathsf{randomness}(s)$: If $s_{status} = \perp$, then S returns $\perp$. Otherwise, S returns s_{rand} (via lookup in table Q).
7. $\mathsf{session\text{-}key}(s)$: If $s_{status} \neq$ `accepted`, then S returns $\perp$. Otherwise, S answers this query by lookup in table T.
8. $\mathsf{test\text{-}session}(s)$: If $s \neq s^*$, then S aborts; otherwise S answers the query in the appropriate way.
9. $\mathsf{corrupt}(\hat{P})$: If $\hat{P} \notin \mathcal{P}$, then S returns $\perp$. Else if $\hat{P} = \hat{C}$, then S aborts. Else S returns $\mathsf{sk}_{\hat{P}}$.
10. M outputs a guess: S aborts.

Analysis of event $DL \wedge K$

Similar to the analysis of the related event $DL \wedge K$ in the proof of [19, Proposition 7].

Event $T_O \wedge DL^c \wedge K$

Let s^* and s' denote the test session and the origin-session for the test session, respectively. We split event $Evt := T_O \wedge DL^c \wedge K$ into the following events $B_1, \ldots, B_3$ so that $Evt = B_1 \vee B_2 \vee B_3$:

1. B_1: Evt occurs and $s^*_{peer} = s'_{actor}$.
2. B_2: Evt occurs and $s^*_{peer} \neq s'_{actor}$ and M does issue neither a $\mathsf{randomness}(s')$ query nor a $\mathsf{cr\text{-}create}(s', \times)$ query to the origin-session s' of s^*, but may issue a $\mathsf{corrupt}(s^*_{peer})$ query.
3. B_3: Evt occurs and $s^*_{peer} \neq s'_{actor}$ and M does not issue a $\mathsf{corrupt}(s^*_{peer})$ query, but may issue either a $\mathsf{randomness}(s')$ query or a $\mathsf{cr\text{-}create}(s', \times)$ query to the origin-session s' of s^*.

Event B_1

Let the input to the GDH challenge be (X_0, Y_0). Suppose that event B_1 occurs with non-negligible probability. In this case S chooses long-term secret/public key pairs for all the honest parties and stores the associated long-term secret keys. Additionally S chooses two random values $m, n \in_R \{1, 2, \ldots, q_s\}$. The m'th activated session by adversary M will be called s^* and the n'th activated session will be called s'. Suppose further that $s^*_{actor} = \hat{A}$, $s^*_{peer} = \hat{B}$ and $s^*_{role} = \mathcal{I}$, w.l.o.g. The simulation of M's environment proceeds as follows:

1. $\mathsf{create}(\hat{A}, \mathcal{I}, \hat{B})$ or $\mathsf{cr\text{-}create}(\hat{A}, \mathcal{I}, str, \hat{B})$ to create session s^*: If create is issued, then S chooses $s^*_{rand} \in_R \{0,1\}^k$. Else, S sets $s^*_{rand} \leftarrow str$. Then, S (a) returns the message X_0, where (X_0, Y_0) is the GDH challenge, (b) increments by 1 the counter value c for the user with identifier $\hat{A}$ (stored in table CV), and (c) stores the updated counter value $c+1$ for $\hat{A}$ in table CV.[7]
2. $\mathsf{create}(\hat{B}, r, \hat{Q})$ or $\mathsf{cr\text{-}create}(\hat{B}, r, str, \hat{Q})$ to create session s': If create is issued, then S chooses $s'_{rand} \in_R \{0,1\}^k$. Else, S sets $s'_{rand} \leftarrow str$. S then increments by 1 the counter value c for the user with identifier $\hat{B}$ (stored in table CV), and stores the updated counter value $c+1$ for $\hat{B}$ in table CV.If $r = \mathcal{I}$, then S returns message Y_0 to M, where (X_0, Y_0) is the GDH challenge. Else, $\star$ is returned.
3. $\mathsf{send}(\hat{B}, i, Z)$ with $(\hat{B}, i) = s'$: If $s'_{status} \neq \texttt{active}$, then S returns $\perp$. Else if $s'_{role} = \mathcal{R}$ and $Z \in G$, then S returns message Y_0 to M, where (X_0, Y_0) is the GDH challenge, sets the status of session s' to `accepted`, and proceeds as in the previous simulation for completing the session. Else, S proceeds as in the previous simulation.
4. $\mathsf{send}(\hat{A}, i, Y_0)$ with $(\hat{A}, i) = s^*$: S proceeds as in the previous simulation for completing the session.
5. Other create, $\mathsf{cr\text{-}create}$ and send queries are answered as in the previous simulation.
6. $\mathsf{randomness}(s)$: If $s_{status} = \perp$, then S returns $\perp$. Else, S returns s_{rand}.
7. $\mathsf{session\text{-}key}(s)$: If $s_{status} \neq \texttt{accepted}$, then S returns $\perp$. Otherwise, S answers this query by lookup in table T.
8. $\mathsf{test\text{-}session}(s)$: If $s \neq s^*$ or if s' is not the origin-session for session s^*, then S aborts; otherwise S answers the query in the appropriate way.
9. $H_1(r, h, *)$: If $h = a$ and $r = s^*_{rand}$ or if $h = b$ and $r = s'_{rand}$, then S aborts. Otherwise S simulates a random oracle as in the previous simulation.
10. $\mathsf{corrupt}(\hat{P})$: If $\hat{P} \notin \mathcal{P}$, then S returns $\perp$. Else, S returns $\mathsf{sk}_{\hat{P}}$.
11. When M makes a query of the form $\left(\sigma_1, \sigma_2, \sigma_3, \hat{P}_i, \hat{P}_j\right)$ to the random oracle for H_2, answer it as follows:
    - If $\left\{\hat{P}_i, \hat{P}_j\right\} = \left\{\hat{A}, \hat{B}\right\}$, $\sigma_1 = Y_0^a$, $\sigma_2 = X_0^b$ and $\mathrm{DDH}(X_0, Y_0, \sigma_3) = 1$, then S aborts M and is successful by outputting $CDH(X_0, Y_0) = \sigma_3$.
    - Else if $\left(\sigma_1, \sigma_2, \sigma_3, \hat{P}_i, \hat{P}_j, \lambda\right) \in L$ for some $\lambda \in \{0,1\}^k$, then S returns λ to M.
    - Else if there exist entries $\left(\hat{P}_i, \hat{P}_j, \mathcal{I}, U, V, \lambda\right)$ or $\left(\hat{P}_j, \hat{P}_i, \mathcal{R}, V, U, \lambda\right)$ in table T, for some $\lambda \in \{0,1\}^k$ and $U, V \in G$, such that $\mathrm{DDH}(V, U, \sigma_3) = 1$, $\mathrm{DDH}(V, \mathsf{pk}_{\hat{P}_i}, \sigma_1) = 1$ and $\mathrm{DDH}(U, \mathsf{pk}_{\hat{P}_j}, \sigma_2) = 1$, then S returns λ to M and stores the entry $\left(\sigma_1, \sigma_2, \sigma_3, \hat{P}_i, \hat{P}_j, \lambda\right)$ in table L.
    - Else, S chooses $\mu \in_R \{0,1\}^k$, returns it to M and stores the entry $\left(\sigma_1, \sigma_2, \sigma_3, \hat{P}_i, \hat{P}_j, \mu\right)$ in L.

[7] Note that s^*_{rand} is not used in the calculation.

12. M outputs a guess: S aborts.

Analysis of event B_1
Similar to the analysis of the related event B_1 in the proof of [19, Proposition 7].
Event B_2
Let the input to the GDH challenge be (X_0, Y_0). Suppose that event B_2 occurs with non-negligible probability. The simulation of S proceeds in the same way as for event B_1 with the following changes:

- $\mathsf{create}(\hat{B}, r, \hat{Q})$ or $\mathsf{cr\text{-}create}(\hat{B}, r, str, \hat{Q})$ to create session s': If $\mathsf{cr\text{-}create}$ is issued, then S aborts. Else, S proceeds as described before.
- $\mathsf{randomness}(s)$: If $s_{status} = \perp$, then S returns $\perp$. Else if $s = s'$, then S aborts. Else, S returns s_{rand}.
- $H_1(r, h, *)$: If $h = a$ and $r = s^*_{rand}$, then S aborts. Otherwise S simulates a random oracle as in the previous simulation.

Analysis of event B_2
Similar to the analysis of the related event B_2 in the proof of [19, Proposition 7].
Event B_3
Let the input to the GDH challenge be (X_0, B). Suppose that event B_3 occurs with non-negligible probability. In this case, S chooses one user $\hat{B} \in \mathcal{P}$ at random from the set $\mathcal{P}$ and sets its long-term public key to B. S chooses long-term secret/public key pairs for the remaining parties in $\mathcal{P}$ and stores the associated long-term secret keys. Additionally S chooses two random values $m, n \in_R \{1, 2, \ldots, q_s\}$. We denote the m'th activated session by adversary M by s^* and the n'th activated session by s'. Suppose further that $s^*_{actor} = \hat{A}$, $s^*_{peer} = \hat{B}$ and $s^*_{role} = \mathcal{I}$, w.l.o.g. Algorithm S maintains tables Q, J, T and L, all of which are initially empty. S also maintains a variable ω initialized with 1 and a table CV maintaining for each user the current counter value. Initially, table CV contains an entry $(\hat{P}, 0)$ for each user $\hat{P} \in \mathcal{P}$. The simulation of M's environment proceeds as follows:

1. $\mathsf{create}(\hat{A}, \mathcal{I}, \hat{B})$ or $\mathsf{cr\text{-}create}(\hat{A}, \mathcal{I}, str, \hat{B})$ to create session s^*: If create is issued, then S chooses $s^*_{rand} \in_R \{0, 1\}^k$. Else, S sets $s^*_{rand} \leftarrow str$. Then, S (a) returns the message X_0, (b) increments by 1 the counter value c for the user with identifier $\hat{A}$ (stored in table CV), and (c) stores the updated counter value $c + 1$ for $\hat{A}$ in table CV.
2. $\mathsf{create}\left(\hat{P}, r, \hat{Q}\right)$ to create session s: S checks whether $\hat{P} \in \mathcal{P}$, $\hat{Q} \in \mathcal{P}$, and $r \in \{\mathcal{I}, \mathcal{R}\}$. If one of the checks fails, then S returns $\perp$. Else, S initializes the session variables according to the protocol specification, and stores an entry of the form $\left(s, s_{rand}, \mathsf{sk}_{s_{actor}}, l_s, \kappa\right) \in (\mathcal{P} \times \mathbb{N}) \times \{0, 1\}^k \times (\mathbb{Z}_p \cup \{*\}) \times \mathbb{N} \times \mathbb{Z}_p$ in table Q as follows:
   - S retrieves the counter value c for the user with identifier $\hat{P}$ from table CV, increments c by 1, and updates the counter value for $\hat{P}$ stored in table CV with $c + 1$,
   - S chooses $s_{rand} \in_R \{0, 1\}^k$ (i.e. the randomness of session s),
   - S chooses $\kappa \in_R \mathbb{Z}_p$,
   - if $s_{actor} \neq \hat{B}$, then S stores the entry $\left(s, s_{rand}, \mathsf{sk}_{s_{actor}}, c + 1, \kappa\right)$ in Q, else S stores the entry $(s, s_{rand}, *, c + 1, \kappa)$ in Q, and
   - if $r = \mathcal{I}$, then S returns the Diffie–Hellman exponential g^{κ} to M, else S returns $\star$.
3. $\mathsf{cr\text{-}create}\left(\hat{P}, r, str, \hat{Q}\right)$ to create session s: S checks whether $\hat{P} \in \mathcal{P}$, $\hat{Q} \in \mathcal{P}$, and $r \in \{\mathcal{I}, \mathcal{R}\}$. If one of the checks fails, then S returns $\perp$. Else, S initializes the session variables according to the protocol specification, and stores an entry of the form $\left(s, s_{rand}, \mathsf{sk}_{s_{actor}}, l_s, \kappa\right) \in (\mathcal{P} \times \mathbb{N}) \times \{0, 1\}^k \times (\mathbb{Z}_p \cup \{*\}) \times \mathbb{N} \times \mathbb{Z}_p$ in table Q as follows:

- S retrieves the counter value c for the user with identifier $\hat{P}$ from table CV, increments c by 1, and updates the counter value for $\hat{P}$ stored in table CV with $c+1$,
- if there is an entry $(r_i, h_i, l_i, \kappa_i)$ in table J such that $r_i = str$, $h_i = \mathsf{sk}_{\hat{P}}$, and $l_i = c+1$, then S sets $\omega \leftarrow \kappa_i$, else S chooses $\kappa \in_R \mathbb{Z}_p$, and sets $\omega \leftarrow \kappa$.
- if $s_{actor} \neq \hat{B}$, then S stores the entry $\left(s, s_{rand}, \mathsf{sk}_{s_{actor}}, c+1, x_5\right)$ in Q, else S stores the entry $(s, s_{rand}, *, c+1, x_5)$ in Q, where x_5 denotes the value of variable ω,
- if $r = \mathcal{I}$, then S returns the Diffie–Hellman exponential g^κ to M, else S returns $\star$.

4. S stores entries of the form $(r, h, l, \kappa) \in \{0,1\}^k \times \mathbb{Z}_p \times \mathbb{N} \times \mathbb{Z}_p$ in table J. When M makes a query of the form (r, h, l) to the random oracle for H_1, answer it as follows:
   - If $r = s^*_{rand}$ and $h = a$, then S aborts,
   - Else if $(r, h, l, \kappa) \in J$ for some $\kappa \in \mathbb{Z}_p$, then S returns κ to M.
   - Else if there exists an entry $\left(s, s_{rand}, \mathsf{sk}_{s_{actor}}, l_s, \kappa\right)$ in Q, for some $s \in \mathcal{P} \times \mathbb{N}$, $s_{rand} \in \{0,1\}^k$, $\mathsf{sk}_{s_{actor}} \in \mathbb{Z}_p$, $l_s \in \mathbb{N}$ and $\kappa \in \mathbb{Z}_p$, such that $s_{rand} = r$, $\mathsf{sk}_{s_{actor}} = h$ and $l_s = l$, then S returns κ to M and stores the entry (r, h, l, κ) in table J.
   - Else, S chooses $\kappa \in_R \mathbb{Z}_p$, returns it to M and stores the entry (r, h, l, κ) in table J.

5. $\mathsf{send}(\hat{P}, i, V)$ to send message V to session $s = (\hat{P}, i)$: If $s_{status} \neq$ `active`, then S returns $\perp$. Else if $s_{role} = \mathcal{I}$, then S does the following. If $V \notin G$, then the status of session s is set to `rejected`. Else, the status of session s is set to `accepted`, the variable $recv$ is updated to $s_{recv} \leftarrow (s_{recv}, V)$ and
   - If there exists an entry $\left(s_{peer}, s_{actor}, \mathcal{R}, s_{recv}, s_{sent}, \lambda\right)$ in table T, then S stores the entry $\left(s_{actor}, s_{peer}, \mathcal{I}, s_{sent}, s_{recv}, \lambda\right)$ in table T.
   - Else if there exists an entry $\left(\sigma_1, \sigma_2, \sigma_3, s_{actor}, s_{peer}, \lambda\right)$ in table L, for some $\lambda \in \{0,1\}^k$, such that $\mathrm{DDH}(s_{recv}, s_{sent}, \sigma_3) = 1$, $\mathrm{DDH}(s_{sent}, \mathsf{pk}_{s_{peer}}, \sigma_2) = 1$ and $\mathrm{DDH}(s_{recv}, \mathsf{pk}_{s_{actor}}, \sigma_1) = 1$, then S stores $\left(s_{actor}, s_{peer}, \mathcal{I}, s_{sent}, s_{recv}, \lambda\right)$ in table T.
   - Else, S chooses $\mu \in_R \{0,1\}^k$ and stores the entry $(s_{actor}, s_{peer}, \mathcal{I}, s_{sent}, s_{recv}, \mu)$ in T.

   Else if $s_{role} = \mathcal{R}$, then S does the following. If $V \notin G$, then the status of session s is set to `rejected`. Else, S sets the status of session s to `accepted`, and the variable $recv$ to (s_{recv}, V). S returns g^κ to M, where κ denotes the last element of the entry $\left(s, r, \mathsf{sk}_{s_{actor}}, l, \kappa\right)$ in table Q, and proceeds in a similar way as in the previous case.

6. When M makes a query of the form $\left(\sigma_1, \sigma_2, \sigma_3, \hat{P}_i, \hat{P}_j\right)$ to the random oracle for H_2, answer it as follows:
   - If $s'_{status} \neq \perp$, $\left\{\hat{P}_i, \hat{P}_j\right\} = \left\{\hat{A}, \hat{B}\right\}$, $\sigma_1 = A^\kappa$, $\mathrm{DDH}(X_0, B, \sigma_2) = 1$, and $\sigma_3 = X_0^\kappa$, where κ denotes the last element of the entry $(s', s'_{rand}, \mathsf{sk}_{s'_{actor}}, l, \kappa)$ in table Q,[8] then S aborts M and is successful by outputting $CDH(X_0, B) = \sigma_2$.
   - Else if $\left(\sigma_1, \sigma_2, \sigma_3, \hat{P}_i, \hat{P}_j, \lambda\right) \in L$ for some $\lambda \in \{0,1\}^k$, then S returns λ to M.
   - Else if there exist entries $\left(\hat{P}_i, \hat{P}_j, \mathcal{I}, U, V, \lambda\right)$ or $\left(\hat{P}_j, \hat{P}_i, \mathcal{R}, V, U, \lambda\right)$ in table T, for some $\lambda \in \{0,1\}^k$ and $U, V \in G$, such that $\mathrm{DDH}(V, U, \sigma_3) = 1$, $\mathrm{DDH}(V, \mathsf{pk}_{\hat{P}_i}, \sigma_1) = 1$ and $\mathrm{DDH}(U, \mathsf{pk}_{\hat{P}_j}, \sigma_2) = 1$, then S returns λ to M and stores the entry $\left(\sigma_1, \sigma_2, \sigma_3, \hat{P}_i, \hat{P}_j, \lambda\right)$ in table L.

[8] This entry exists in table Q since the status of the session is different to $\perp$.

– Else, S chooses $\mu \in_R \{0, 1\}^k$, returns it to M and stores the entry $\left(\sigma_1, \sigma_2, \sigma_3, \hat{P}_i, \hat{P}_j, \mu\right)$ in L.

7. randomness(s): If $s_{status} = \perp$, then S returns $\perp$. Else, S returns s_{rand}.
8. session-key(s): If $s_{status} \neq \texttt{accepted}$, then S returns $\perp$. Otherwise, S answers this query by lookup in table T.
9. test-session(s): If $s \neq s^*$ or if s' is not the origin-session for session s^*, then S aborts; otherwise S answers the query in the appropriate way.
10. corrupt($\hat{P}$): If $\hat{P} \notin \mathcal{P}$, then S returns $\perp$. Else if $\hat{P} = \hat{B}$, then S aborts. Else, S returns $\mathsf{sk}_{\hat{P}}$.
11. M outputs a guess: S aborts.

Analysis of event B_3
Similar to the analysis of the related event B_3 in the proof of [19, Proposition 7].
Event $(T_O)^c \wedge DL^c \wedge K$
The simulation and analysis are very similar to the simulation and analysis related to event B_3. □

Appendix 2: Proof of Proposition 7

Proof It is straightforward to verify the first condition of Definition 5. We next verify that the second condition of Definition 5 holds. Let E denote a PPT adversary against protocol $\pi :=$ NXPR. We show that the probability of event Multiple-Match$_{\pi,E}^{W(\Omega_\Lambda)}(k)$ is bounded above by a negligible function in the security parameter k, where Multiple-Match$_{\pi,E}^{W(\Omega_\Lambda)}(k)$ denotes the event that, in the security experiment, there exist a session s with $s_{status} = \texttt{accepted}$ and at least two distinct sessions s' and s'' that are matching session s. Note that, if both sessions s' and s'' are matching session s, then it must hold that $s''_{actor} = s'_{actor}$ and $s''_{role} = s'_{role}$. In addition, it is easy to see that the value of the variable $data$ in two different sessions of the same user are distinct (since of different length). For some fixed session s that has accepted, let Ev denote the event that there exist two distinct sessions s' and s'' such that s and s' are matching as well as s and s''. We have:

$$
\begin{aligned}
P(Ev) &\leq P\left(\bigcup_{\substack{s', s'' \\ s' \neq s''}} \{H_1(s''_{rand}, s''_{data}, \mathsf{sk}_{\hat{P}}) = H_1(s'_{rand}, s'_{data}, \mathsf{sk}_{\hat{P}})\}\right) \\
&\leq \sum_{\substack{s', s'' \\ s' \neq s''}} P\big(\{H_1(s''_{rand}, s''_{data}, \mathsf{sk}_{\hat{P}}) = H_1(s'_{rand}, s'_{data}, \mathsf{sk}_{\hat{P}})\}\big) \\
&\leq \frac{q_s^2}{p},
\end{aligned}
$$

where $\hat{P} = s''_{actor} = s'_{actor}$ and q_s denotes the number of created sessions (either via the create or the cr-create query) by the adversary.

In the above computation, we distinguished between the following two events:

1. $D_1 := \{s''_{rand} \neq s'_{rand} \wedge s''_{data} \neq s'_{data}\}$; the probability that the two hash values are identical given D_1 is the probability of a collision in the hash function, and

2. $D_2 := \{s''_{rand} = s'_{rand} \wedge s''_{data} \neq s'_{data}\}$; the probability that the two hash values are identical given D_2 is the probability of a collision in the hash function.

The events $D_3 := \{s''_{rand} = s'_{rand} \wedge s''_{data} = s'_{data}\}$ and $D_4 := \{s''_{rand} \neq s'_{rand} \wedge s''_{data} = s'_{data}\}$ both occur with probability zero.

Even though the value of the variable $rand$ can be the same for two different session of the same user due to the queries cr-create and randomness, the value of the variable $data$ of two different sessions s' and s'' of the same user is always different since the bit strings s'_{data} and s''_{data} differ in length. Given a created session s, the length of the bit string s_{data} depends on the number of sessions of user s_{actor} that have already been created either via create or cr-create.

Finally, $P(\text{Multiple-Match}_{\pi,E}^{W(\Omega_A)}(k)) \leq q_s^3 \frac{1}{p}$.

The third condition of Definition 5 is implied by an adaptation of the security proof of protocol CNX in the $\Omega^-_{\mathsf{INDP\text{-}DH}}$ model (see Appendix 1). Let s^* denote the test session. Consider first the event K^c where the adversary M wins the security experiment against π with non-negligible advantage and does not query H_2 with $(\sigma_1, \sigma_2, \sigma_3, \hat{A}, \hat{B})$, where $\sigma_1 = CDH(Y, A), \sigma_2 = CDH(B, X)$ and $\sigma_3 = CDH(X, Y)$.

Event K^c

If event K^c occurs, then the adversary M must have issued a session-key query to some session s such that $K_s = K_{s^*}$ (where K_s and K_{s^*} denote the session keys computed in sessions s and s^*, respectively) and s does not match s^*. We consider the following four events:

1. A_1 : there exist two distinct sessions s', s'' created via a create query such that $s'_{rand} = s''_{rand}$.[9]
2. A_2 : there exists a session $s \neq s^*$ such that $H_1(s_{rand}, s_{data}) = H_1(s^*_{rand}, s^*_{data})$.
3. A_3 : there exists a session $s' \neq s^*$ such that $H_2(\text{input}_{s'}) = H_2(\text{input}_{s^*})$ with $\text{input}_{s'} \neq \text{input}_{s^*}$.
4. A_4 : there exists an adversarial query input_M to the oracle H_2 such that $H_2(\text{input}_M) = H_2(\text{input}_{s^*})$ with $\text{input}_M \neq \text{input}_{s^*}$.

Analysis of event K^c

We denote by q_s the number of created sessions (either via the query create or the query cr-create) by the adversary and by q_{ro2} the number of queries to the random oracle H_2. We have that

$$P(K^c) \leq P(A_1 \vee A_2 \vee A_3 \vee A_4) \leq P(A_1) + P(A_2) + P(A_3) + P(A_4)$$
$$\leq \frac{q_s^2}{2}\frac{1}{2^k} + \frac{q_s}{p} + \frac{q_s + q_{\text{ro2}}}{2^k},$$

which is a negligible function of the security parameter k.

In contrast to the NAXOS protocol analyzed with respect to model $\Omega_{\mathsf{INDP\text{-}DH}}$, the adversary cannot force two sessions of protocol π of the same user with the same role to compute the same session key via a chosen-randomness replay attack since the H_1 values in both sessions will be different with overwhelming probability. The latter event is included in event A_2.

In the subsequent events (and their analyses) we assume that no collisions in the queries to the oracle H_1 occur and that none of the events $A_1, \ldots, A_4$ occurs. As in the proof of [19, Proposition 7], we next consider the following three events:

[9] Under event A_1 the query randomness (e.g., for two sessions of different users) together with other queries might enable the adversary to learn all the information necessary to compute the session key of the target session without violating the freshness condition.

1. $DL \wedge K$,
2. $T_O \wedge DL^c \wedge K$, and
3. $(T_O)^c \wedge DL^c \wedge K$, where

T_O denotes the event that there exists an origin-session for the test session, DL denotes the event where there exists a user $\hat{C} \in \mathcal{P}$ such that the adversary M, during its execution, queries H_1 with $(*, c)$ before issuing a $\mathsf{corrupt}(\hat{C})$ query and K denotes the event that M wins the security experiment against NXPR by querying H_2 with $(\sigma_1, \sigma_2, \sigma_3, \hat{A}, \hat{B})$, where $\sigma_1 = CDH(Y, A)$, $\sigma_2 = CDH(B, X)$ and $\sigma_3 = CDH(X, Y)$.

Event $DL \wedge K$

Let the input to the GAP-DLog challenge be C. Suppose that event $DL \wedge K$ occurs with non-negligible probability. In this case, the simulator S chooses one user $\hat{C} \in \mathcal{P}$ at random and sets its long-term public key to C. S chooses long-term secret/public key pairs for the remaining honest parties and stores the associated long-term secret keys. Additionally S chooses a random value $m \in_R \{1, 2, \ldots, q_s\}$. We denote the m'th activated session by adversary M by s^*. Suppose further that $s^*_{actor} = \hat{A}$, $s^*_{peer} = \hat{B}$ and $s^*_{role} = \mathcal{I}$, w.l.o.g. We now define S's responses to M's queries for the pre-specified peer setting; the post-specified peer case proceeds similarly. Algorithm S maintains tables Q, J, T and L, all of which are initially empty. S also maintains a variable ω initialized with 1.

1. $\mathsf{create}\left(\hat{P}, r, \hat{Q}\right)$ to create session s: S checks whether $\hat{P} \in \mathcal{P}$, $\hat{Q} \in \mathcal{P}$, and $r \in \{\mathcal{I}, \mathcal{R}\}$. If one of the checks fails, then S returns $\perp$. Else, S initializes the session variables according to the protocol specification, and stores an entry of the form $\left(s, s_{rand}, l_s, \mathsf{sk}_{s_{actor}}, \kappa\right) \in (\mathcal{P} \times \mathbb{N}) \times \{0,1\}^k \times \{0,1\}^* \times (\mathbb{Z}_p \cup \{*\}) \times \mathbb{Z}_p$ in table Q as follows:
   - S chooses $s_{rand} \in_R \{0,1\}^k$ (i.e. the randomness of session s),
   - S chooses $\kappa \in_R \mathbb{Z}_p$,
   - if there is no entry $\left(s, s_{rand}, l_s, \mathsf{sk}_{s_{actor}}, \kappa\right)$ in table Q such that $s_{actor} = \hat{P}$, then S sets the value of l_s to s_{rand}, else S sets the value of l_s to $(s_{rand}, l_{s'})$, where s' is the previous session with $s'_{actor} = s_{actor}$ for which an entry in table Q has been made.[10]
   - if $s_{actor} \neq \hat{C}$, then S stores the entry $\left(s, s_{rand}, s_{data}, \mathsf{sk}_{s_{actor}}, \kappa\right)$ in Q, else S stores the entry $(s, s_{rand}, s_{data}, *, \kappa)$ in Q, and
   - if $r = \mathcal{I}$, then S returns the Diffie–Hellman exponential g^κ to M, else S returns $\star$.

2. $\mathsf{cr\text{-}create}\left(\hat{P}, r, str, \hat{Q}\right)$ to create session s: S checks whether $\hat{P} \in \mathcal{P}$, $\hat{Q} \in \mathcal{P}$, and $r \in \{\mathcal{I}, \mathcal{R}\}$. If one of the checks fails, then S returns $\perp$. Else, S initializes the session variables according to the protocol specification, and stores an entry of the form $\left(s, s_{rand}, l_s, \mathsf{sk}_{s_{actor}}, \kappa\right) \in (\mathcal{P} \times \mathbb{N}) \times \{0,1\}^k \times \{0,1\}^* \times (\mathbb{Z}_p \cup \{*\}) \times \mathbb{Z}_p$ in table Q as follows:
   - if there is an entry (r_i, h_i, κ_i) in table J such that $r_i = (str, l_{s'})$, and $h_i = \mathsf{sk}_{\hat{P}}$, where s' is the previous session with $s'_{actor} = s_{actor}$ for which an entry in table Q has been made, then S sets $\omega \leftarrow \kappa_i$, else S chooses $\kappa \in_R \mathbb{Z}_p$ and sets $\omega \leftarrow \kappa$.
   - if $s_{actor} \neq \hat{C}$, then S stores the entry $\left(s, s_{rand}, r_i, \mathsf{sk}_{s_{actor}}, \omega\right)$ in Q, else S stores the entry $(s, s_{rand}, l_s, *, \omega)$ in Q with $l_s = (str, l_{s'})$,
   - if $r = \mathcal{I}$, then S returns the Diffie–Hellman exponential g^κ to M, else S returns $\star$.

3. S stores entries of the form $(r, h, \kappa) \in \{0,1\}^* \times \mathbb{Z}_p \times \mathbb{Z}_p$ in table J. When M makes a query of the form (r, h) to the random oracle for H_1, answer it as follows:

[10] The value of $l_{s'}$ is the concatenation of the randomness of the current and the previous sessions of the same user.

- If $C = g^h$, then S aborts M and is successful by outputting $\mathrm{DLog}_g(C) = h$.
- Else if $(r, h, \kappa) \in J$ for some $\kappa \in \mathbb{Z}_p$, then S returns κ to M.
- Else if there exists an entry $\left(s, s_{rand}, l_s, \mathsf{sk}_{s_{actor}}, \kappa\right)$ in table Q with $l_s = r$ and $\mathsf{sk}_{s_{actor}} = h$, then S returns κ to M and stores the entry (r, h, κ) in table J.
- Else, S chooses $\kappa \in_R \mathbb{Z}_p$, returns it to M and stores the entry (r, h, κ) in table J.

4. $\mathsf{send}(\hat{P}, i, V)$ to send message V to session $s = (\hat{P}, i)$: If $s_{status} \neq$ `active`, then S returns $\perp$. Else if $s_{role} = \mathcal{I}$, then S does the following. If $V \notin G$, then the status of session s is set to `rejected`. Else, the status of session s is set to `accepted`, and
   - If there exists an entry $\left(s_{peer}, s_{actor}, \mathcal{R}, s_{recv}, s_{sent}, \lambda\right)$ in table T, then S stores the entry $\left(s_{actor}, s_{peer}, \mathcal{I}, s_{sent}, s_{recv}, \lambda\right)$ in table T.
   - Else if there exists an entry $\left(\sigma_1, \sigma_2, \sigma_3, s_{actor}, s_{peer}, \lambda\right)$ in table L, for some $\lambda \in \{0,1\}^k$, such that $\mathrm{DDH}(s_{recv}, s_{sent}, \sigma_3) = 1$, $\mathrm{DDH}(s_{sent}, \mathsf{pk}_{s_{peer}}, \sigma_2) = 1$ and $\mathrm{DDH}(s_{recv}, \mathsf{pk}_{s_{actor}}, \sigma_1) = 1$, then S stores $\left(s_{actor}, s_{peer}, \mathcal{I}, s_{sent}, s_{recv}, \lambda\right)$ in table T.
   - Else, S chooses $\mu \in_R \{0,1\}^k$ and stores the entry $(s_{actor}, s_{peer}, \mathcal{I}, s_{sent}, s_{recv}, \mu)$ in T.

   Else if $s_{role} = \mathcal{R}$, then S does the following. If $V \notin G$, then the status of session s is set to `rejected`. Else, S sets the status of session s to `accepted`, returns g^{κ} to M, where κ denotes the last element of the entry $\left(s, s_{rand}, l_s, \mathsf{sk}_{s_{actor}}, \kappa\right)$ in table Q, and proceeds in a similar way as in the previous case.
5. When M makes a query of the form $\left(\sigma_1, \sigma_2, \sigma_3, \hat{P}_i, \hat{P}_j\right)$ to the random oracle for H_2, answer it as follows:
   - If $\left(\sigma_1, \sigma_2, \sigma_3, \hat{P}_i, \hat{P}_j, \lambda\right) \in L$ for some $\lambda \in \{0,1\}^k$, then S returns λ to M.
   - Else if there exist entries $\left(\hat{P}_i, \hat{P}_j, \mathcal{I}, U, V, \lambda\right)$ or $\left(\hat{P}_j, \hat{P}_i, \mathcal{R}, V, U, \lambda\right)$ in table T, for some $\lambda \in \{0,1\}^k$ and $U, V \in G$, such that $\mathrm{DDH}(V, U, \sigma_3) = 1$, $\mathrm{DDH}(V, \mathsf{pk}_{\hat{P}_i}, \sigma_1) = 1$ and $\mathrm{DDH}(U, \mathsf{pk}_{\hat{P}_j}, \sigma_2) = 1$, then S returns λ to M and stores the entry $\left(\sigma_1, \sigma_2, \sigma_3, \hat{P}_i, \hat{P}_j, \lambda\right)$ in table L.
   - Else, S chooses $\mu \in_R \{0,1\}^k$, returns it to M and stores the entry $\left(\sigma_1, \sigma_2, \sigma_3, \hat{P}_i, \hat{P}_j, \mu\right)$ in L.
6. $\mathsf{randomness}(s)$: If $s_{status} = \perp$, then S returns $\perp$. Otherwise, S returns s_{rand}.
7. $\mathsf{session\text{-}key}(s)$: If $s_{status} \neq$ `accepted`, then S returns $\perp$. Otherwise, S answers this query by lookup in table T.
8. $\mathsf{test\text{-}session}(s)$: If $s \neq s^*$, then S aborts; otherwise S answers the query in the appropriate way.
9. $\mathsf{corrupt}(\hat{P})$: If $\hat{P} \notin \mathcal{P}$, then S returns $\perp$. Else if $\hat{P} = \hat{C}$, then S aborts. Else, S returns $\mathsf{sk}_{\hat{P}}$.
10. M outputs a guess: S aborts.

Analysis of event $DL \wedge K$
Similar to the analysis of the related event $DL \wedge K$ in the proof of [19, Proposition 7].
Event $T_O \wedge DL^c \wedge K$
Let s^* and s' denote the test session and the origin-session for the test session, respectively. We split event $Evt := T_O \wedge DL^c \wedge K$ into the following events $B_1, \ldots, B_3$ so that $Evt = B_1 \vee B_2 \vee B_3$:

1. B_1 : *Evt* occurs and $s^*_{peer} = s'_{actor}$.
2. B_2 : *Evt* occurs and $s^*_{peer} \neq s'_{actor}$ and M does not issue the queries randomness or cr-create to all sessions of s'_{actor} that were created prior to creation of the origin-session s' of s^*, including the origin-session itself, but may issue a corrupt(s^*_{peer}) query.
3. B_3 : *Evt* occurs and $s^*_{peer} \neq s'_{actor}$ and M does not issue a corrupt(s^*_{peer}) query, but may issue the queries randomness or cr-create to all session created prior to creation of the origin-session, including the origin-session s' itself.

Event B_1

Let the input to the GDH challenge be (X_0, Y_0). Suppose that event B_1 occurs with non-negligible probability. In this case S chooses long-term secret/public key pairs for all the honest parties and stores the associated long-term secret keys. Additionally S chooses two random values $m, n \in_R \{1, 2, \ldots, q_s\}$. The m'th activated session by adversary M will be called s^* and the n'th activated session will be called s'. Suppose further that $s^*_{actor} = \hat{A}$, $s^*_{peer} = \hat{B}$ and $s^*_{role} = \mathcal{I}$, w.l.o.g. We now define S's responses to M's queries. S maintains tables Q, J, T and L, all of which are initially empty, as well as a variable ω initialized with 1.

1. create($\hat{A}, \mathcal{I}, \hat{B}$) or cr-create($\hat{A}, \mathcal{I}, str, \hat{B}$) to create session s^*: If create is issued, S chooses $s^*_{rand} \in_R \{0, 1\}^k$. Else, S sets $s^*_{rand} \leftarrow str$. S (a) returns the message X_0, where (X_0, Y_0) is the GDH challenge, and (b) stores the entry $(s^*, s^*_{rand}, l_{s^*}, \mathsf{sk}_{\hat{A}}, *)$ in table Q, where $l_{s^*} = (s^*_{rand}, l_s)$ if there exists a previously created session s of user $s_{actor} = \hat{A}$ with an entry in table Q, and $l_{s^*} = s^*_{rand}$ if there no such session exists.
2. create($\hat{B}, r, \hat{Q}$) or cr-create($\hat{B}, r, str, \hat{Q}$) with $r \in \{\mathcal{I}, \mathcal{R}\}$ to create session s': If create is issued, S chooses $s'_{rand} \in_R \{0, 1\}^k$. Else, S sets $s'_{rand} \leftarrow str$. S stores the entry $(s', s'_{rand}, l_{s'}, \mathsf{sk}_{\hat{B}}, *)$ in table Q, where $l_{s'} = (s'_{rand}, l_s)$ if there exists a previously created session s of user $s_{actor} = \hat{A}$ with an entry in table Q, and $l_{s'} = s'_{rand}$ if there no such session exists. If $r = \mathcal{I}$, then S returns message Y_0 to M, where (X_0, Y_0) is the GDH challenge. Else, $\star$ is returned.
3. send($\hat{B}, i, Z$) with $(\hat{B}, i) = s'$: If $s'_{status} \neq \texttt{active}$, then S returns $\perp$. Else if $s'_{role} = \mathcal{R}$ and $Z \in G$, then S returns message Y_0 to M, where (X_0, Y_0) is the GDH challenge, sets the status of session s' to `accepted`, and proceeds as in the previous simulation for completing the session. Else, S proceeds as in the previous simulation.
4. send($\hat{A}, i, Y_0$) with $(\hat{A}, i) = s^*$: S proceeds as in the previous simulation for completing the session.
5. Other create, cr-create and send queries are answered as in the simulation relative to event $DL \wedge K$.
6. When M makes a query of the form $\left(\sigma_1, \sigma_2, \sigma_3, \hat{P}_i, \hat{P}_j\right)$ to the random oracle for H_2, answer it as follows:
   - If $\left\{\hat{P}_i, \hat{P}_j\right\} = \left\{\hat{A}, \hat{B}\right\}$, $\sigma_1 = Y_0^a$, $\sigma_2 = X_0^b$ and $\mathrm{DDH}(X_0, Y_0, \sigma_3) = 1$, then S aborts M and is successful by outputting $CDH(X_0, Y_0) = \sigma_3$.
   - Else if $\left(\sigma_1, \sigma_2, \sigma_3, \hat{P}_i, \hat{P}_j, \lambda\right) \in L$ for some $\lambda \in \{0, 1\}^k$, then S returns λ to M.
   - Else if there exist entries $\left(\hat{P}_i, \hat{P}_j, \mathcal{I}, U, V, \lambda\right)$ or $\left(\hat{P}_j, \hat{P}_i, \mathcal{R}, V, U, \lambda\right)$ in table T, for some $\lambda \in \{0, 1\}^k$ and $U, V \in G$, such that $\mathrm{DDH}(V, U, \sigma_3) = 1$, $\mathrm{DDH}(V, \mathsf{pk}_{\hat{P}_i}, \sigma_1) = 1$ and $\mathrm{DDH}(U, \mathsf{pk}_{\hat{P}_j}, \sigma_2) = 1$, then S returns λ to M and stores the entry $\left(\sigma_1, \sigma_2, \sigma_3, \hat{P}_i, \hat{P}_j, \lambda\right)$ in table L.

– Else, S chooses $\mu \in_R \{0,1\}^k$, returns it to M and stores the entry $\left(\sigma_1, \sigma_2, \sigma_3, \hat{P}_i, \hat{P}_j, \mu\right)$ in L.

7. $\mathsf{randomness}(s)$: If $s_{status} = \perp$, then S returns $\perp$. Otherwise, S returns s_{rand}.
8. $\mathsf{session\text{-}key}(s)$: If $s_{status} \neq \texttt{accepted}$, then S returns $\perp$. Otherwise, S answers this query by lookup in table T.
9. $\mathsf{test\text{-}session}(s)$: If $s \neq s^*$ or if s' is not the origin-session for session s^*, then S aborts; otherwise S answers the query in the appropriate way.
10. $H_1(r, h)$: If $r = l_{s^*}$ and $h = \mathsf{sk}_{\hat{A}}$ or if $r = l_{s'}$ and $h = \mathsf{sk}_{\hat{B}}$, then S aborts. Otherwise S simulates a random oracle as in the simulation relative to event $DL \wedge K$.
11. $\mathsf{corrupt}(\hat{P})$: If $\hat{P} \notin \mathcal{P}$, then S returns $\perp$. Else, S returns $\mathsf{sk}_{\hat{P}}$.
12. M outputs a guess: S aborts.

Analysis of event B_1
S's simulation of M's environment is perfect except with negligible probability. The probability that M selects s^* as the test-session and s' as the origin-session for the test-session is $\frac{1}{(q_s)^2}$. Assuming that this is indeed the case, S does not abort in Step 9. Under event DL^c, the adversary first issues a $\mathsf{corrupt}(\hat{P})$ query to party $\hat{P}$ before making an H_1 query that involves the long-term secret key of party $\hat{P}$. Freshness of the test session guarantees that the adversary can reveal/determine either l_{s^*} or $\mathsf{sk}_{\hat{A}}$, but not both. Similar for $l_{s'}$ and $\mathsf{sk}_{\hat{B}}$. Hence S does not abort in Step 10. Under event K, except with negligible probability of guessing $CDH(X_0, Y_0)$, S is successful as described in the first case of Step 6 and does not abort as in Step 12. Hence, if event B_1 occurs, then the success probability of S is given by $P(S) \geq \frac{1}{(q_s)^2} P(B_1)$.
Event B_2
Let the input to the GDH challenge be (X_0, Y_0). Suppose that event B_2 occurs with non-negligible probability. The simulation of S proceeds in the same way as for event B_1 with the following changes. S additionally keeps a history H of M's queries.

– $\mathsf{randomness}(s)$: If $s_{status} = \perp$, then S returns $\perp$. Else if $s = s'$ and there were queries ($\mathsf{randomness}$ or $\mathsf{cr\text{-}create}$) to all previous sessions of the same user s'_{actor}, then S aborts. Else, S returns s_{rand}.
– $H_1(r, h)$: If $r = l_{s^*}$ and $h = \mathsf{sk}_{\hat{A}}$, then S aborts. Otherwise S simulates a random oracle as in the previous simulation.

Analysis of event B_2
Similar to the analyses of the related event B_2 in the proof of [19, Proposition 7] and event B_1.
Event B_3
Let the input to the GDH challenge be (X_0, B). Suppose that event B_3 occurs with non-negligible probability. In this case, S chooses one user $\hat{B} \in \mathcal{P}$ at random from the set $\mathcal{P}$ and sets its long-term public key to B. S chooses long-term secret/public key pairs for the remaining parties in $\mathcal{P}$ and stores the associated long-term secret keys. Additionally S chooses two random values $m, n \in_R \{1, 2, \ldots, q_s\}$. We denote the m'th activated session by adversary M by s^* and the n'th activated session by s'. Suppose further that $s^*_{actor} = \hat{A}$, $s^*_{peer} = \hat{B}$ and $s^*_{role} = \mathcal{I}$, w.l.o.g. Algorithm S maintains tables Q, J, T and L, all of which are initially empty. S also maintains a variable ω initialized with 1.

1. $\mathsf{create}(\hat{A}, \mathcal{I}, \hat{B})$ or $\mathsf{cr\text{-}create}(\hat{A}, \mathcal{I}, str, \hat{B})$ to create session s^*: If create is issued, S chooses $s^*_{rand} \in_R \{0,1\}^k$. Else, S sets $s^*_{rand} \leftarrow str$. S (a) returns the message X_0, where (X_0, B) is the GDH challenge, and (b) stores the entry $(s^*, s^*_{rand}, l_{s^*}, \mathsf{sk}_{\hat{A}}, *)$ in table Q,

where $l_{s^*} = (s^*_{rand}, l_s)$ if there exists a previously created session s of user $s_{actor} = \hat{A}$ with an entry in table Q, and $l_{s^*} = s^*_{rand}$ if there no such session exists.

2. $\mathsf{create}\left(\hat{P}, r, \hat{Q}\right)$ to create session s: S checks whether $\hat{P} \in \mathcal{P}$, $\hat{Q} \in \mathcal{P}$, and $r \in \{\mathcal{I}, \mathcal{R}\}$. If one of the checks fails, then S returns $\perp$. Else, S initializes the session variables according to the protocol specification, and stores an entry of the form $\left(s, s_{rand}, l_s, \mathsf{sk}_{s_{actor}}, \kappa\right) \in (\mathcal{P} \times \mathbb{N}) \times \{0, 1\}^k \times \{0, 1\}^* \times (\mathbb{Z}_p \cup \{*\}) \times \mathbb{Z}_p$ in table Q as follows:

   - S chooses $s_{rand} \in_R \{0, 1\}^k$ (i.e. the randomness of session s),
   - S chooses $\kappa \in_R \mathbb{Z}_p$,
   - if there is no entry $\left(s, s_{rand}, l_s, \mathsf{sk}_{s_{actor}}, \kappa\right)$ in table Q such that $s_{actor} = \hat{P}$, then S sets the value of l_s to s_{rand}, else S sets the value of l_s to $(s_{rand}, l_{s'})$, where s' is the previous session with $s'_{actor} = s_{actor}$ for which an entry in table Q has been made.
   - if $s_{actor} \neq \hat{B}$, then S stores the entry $\left(s, s_{rand}, s_{data}, \mathsf{sk}_{s_{actor}}, \kappa\right)$ in Q, else S stores the entry $(s, s_{rand}, s_{data}, *, \kappa)$ in Q, and
   - if $r = \mathcal{I}$, then S returns the Diffie–Hellman exponential g^κ to M, else S returns $\star$.

3. $\mathsf{cr\text{-}create}\left(\hat{P}, r, str, \hat{Q}\right)$ to create session s: S checks whether $\hat{P} \in \mathcal{P}$, $\hat{Q} \in \mathcal{P}$, and $r \in \{\mathcal{I}, \mathcal{R}\}$. If one of the checks fails, then S returns $\perp$. Else, S initializes the session variables according to the protocol specification, and stores an entry of the form $\left(s, s_{rand}, l_s, \mathsf{sk}_{s_{actor}}, \kappa\right) \in (\mathcal{P} \times \mathbb{N}) \times \{0, 1\}^k \times \{0, 1\}^* \times (\mathbb{Z}_p \cup \{*\}) \times \mathbb{Z}_p$ in table Q as follows:

   - if there is an entry (r_i, h_i, κ_i) in table J such that $r_i = (str, l_{s'})$, and $h_i = \mathsf{sk}_{\hat{P}}$, where s' is the previous session with $s'_{actor} = s_{actor}$ for which an entry in table Q has been made, then S sets $\omega \leftarrow \kappa_i$, else S chooses $\kappa \in_R \mathbb{Z}_p$ and sets $\omega \leftarrow \kappa$.
   - if $s_{actor} \neq \hat{B}$, then S stores the entry $\left(s, s_{rand}, r_i, \mathsf{sk}_{s_{actor}}, \omega\right)$ in Q, else S stores the entry $(s, s_{rand}, l_s, *, \omega)$ in Q with $l_s = (str, l_{s'})$,
   - if $r = \mathcal{I}$, then S returns the Diffie–Hellman exponential g^κ to M, else S returns $\star$.

4. S stores entries of the form $(r, h, \kappa) \in \{0, 1\}^* \times \mathbb{Z}_p \times \mathbb{Z}_p$ in table J. When M makes a query of the form (r, h) to the random oracle for H_1, answer it as follows:

   - If $r = l_{s^*}$ and $h = \mathsf{sk}_{\hat{A}}$, then S aborts.
   - Else if $(r, h, \kappa) \in J$ for some $\kappa \in \mathbb{Z}_p$, then S returns κ to M.
   - Else if there exists an entry $\left(s, s_{rand}, l_s, \mathsf{sk}_{s_{actor}}, \kappa\right)$ in table Q with $l_s = r$ and $\mathsf{sk}_{s_{actor}} = h$, then S returns κ to M and stores the entry (r, h, κ) in table J.
   - Else, S chooses $\kappa \in_R \mathbb{Z}_p$, returns it to M and stores the entry (r, h, κ) in table J.

3. $\mathsf{send}(\hat{P}, i, V)$ to send message V to session $s = (\hat{P}, i)$: If $s_{status} \neq \mathtt{active}$, then S returns $\perp$. Else if $s_{role} = \mathcal{I}$, then S does the following. If $V \notin G$, then the status of session s is set to $\mathtt{rejected}$. Else, the status of session s is set to $\mathtt{accepted}$, and

   - If there exists an entry $\left(s_{peer}, s_{actor}, \mathcal{R}, s_{recv}, s_{sent}, \lambda\right)$ in table T, then S stores the entry $\left(s_{actor}, s_{peer}, \mathcal{I}, s_{sent}, s_{recv}, \lambda\right)$ in table T.
   - Else if there exists an entry $\left(\sigma_1, \sigma_2, \sigma_3, s_{actor}, s_{peer}, \lambda\right)$ in table L, for some $\lambda \in \{0, 1\}^k$, such that $\mathrm{DDH}(s_{recv}, s_{sent}, \sigma_3) = 1$, $\mathrm{DDH}(s_{sent}, \mathsf{pk}_{s_{peer}}, \sigma_2) = 1$ and $\mathrm{DDH}(s_{recv}, \mathsf{pk}_{s_{actor}}, \sigma_1) = 1$, then S stores $\left(s_{actor}, s_{peer}, \mathcal{I}, s_{sent}, s_{recv}, \lambda\right)$ in table T.
   - Else, S chooses $\mu \in_R \{0, 1\}^k$ and stores the entry $(s_{actor}, s_{peer}, \mathcal{I}, s_{sent}, s_{recv}, \mu)$ in T.

Else if $s_{role} = \mathcal{R}$, then S does the following. If $V \notin G$, then the status of session s is set to `rejected`. Else, S sets the status of session s to `accepted`, returns g^{κ} to M, where κ denotes the last element of the entry $\left(s, s_{rand}, l_s, \mathsf{sk}_{s_{actor}}, \kappa\right)$ in table Q, and proceeds in a similar way as in the previous case.

6. When M makes a query of the form $\left(\sigma_1, \sigma_2, \sigma_3, \hat{P}_i, \hat{P}_j\right)$ to the random oracle for H_2, answer it as follows:
   - If $\left\{\hat{P}_i, \hat{P}_j\right\} = \left\{\hat{A}, \hat{B}\right\}$, $\sigma_1 = A^{\kappa}$, $\mathrm{DDH}(X_0, B, \sigma_2) = 1$, and $\sigma_3 = X_0^{\kappa}$, where κ denotes the last element of the entry $(s', s'_{rand}, l_{s'}, \mathsf{sk}_{s'_{actor}}, \kappa)$ in table Q, then S aborts M and is successful by outputting $CDH(X_0, B) = \sigma_2$.
   - Else if $\left(\sigma_1, \sigma_2, \sigma_3, \hat{P}_i, \hat{P}_j, \lambda\right) \in L$ for some $\lambda \in \{0, 1\}^k$, then S returns λ to M.
   - If $\left(\sigma_1, \sigma_2, \sigma_3, \hat{P}_i, \hat{P}_j, \lambda\right) \in L$ for some $\lambda \in \{0, 1\}^k$, then S returns λ to M.
   - Else if there exist entries $\left(\hat{P}_i, \hat{P}_j, \mathcal{I}, U, V, \lambda\right)$ or $\left(\hat{P}_j, \hat{P}_i, \mathcal{R}, V, U, \lambda\right)$ in table T, for some $\lambda \in \{0, 1\}^k$ and $U, V \in G$, such that $\mathrm{DDH}(V, U, \sigma_3) = 1$, $\mathrm{DDH}(V, \mathsf{pk}_{\hat{P}_i}, \sigma_1) = 1$ and $\mathrm{DDH}(U, \mathsf{pk}_{\hat{P}_j}, \sigma_2) = 1$, then S returns λ to M and stores the entry $\left(\sigma_1, \sigma_2, \sigma_3, \hat{P}_i, \hat{P}_j, \lambda\right)$ in table L.
   - Else, S chooses $\mu \in_R \{0, 1\}^k$, returns it to M and stores the entry $\left(\sigma_1, \sigma_2, \sigma_3, \hat{P}_i, \hat{P}_j, \mu\right)$ in L.
7. $\mathsf{randomness}(s)$: If $s_{status} = \perp$, then S returns $\perp$. Otherwise, S returns s_{rand}.
8. $\mathsf{session\text{-}key}(s)$: If $s_{status} \neq$ `accepted`, then S returns $\perp$. Otherwise, S answers this query by lookup in table T.
9. $\mathsf{test\text{-}session}(s)$: If $s \neq s^*$ or if s' is not the origin-session for session s^*, then S aborts; otherwise S answers the query in the appropriate way.
10. $\mathsf{corrupt}(\hat{P})$: If $\hat{P} \notin \mathcal{P}$, then S returns $\perp$. Else if $\hat{P} = \hat{B}$, then S aborts. Else, S returns $\mathsf{sk}_{\hat{P}}$.
11. M outputs a guess: S aborts.

Analysis of event B_3
Similar to the analysis of the related event B_3 in the proof of [19, Proposition 7].
Event $(T_O)^c \wedge DL^c \wedge K$
The simulation and analysis are very similar to the simulation and analysis related to event B_3. □

References

1. Debian, Debian Security Advisory DSA-1571-1 openssl—predictable random number generator. http://www.debian.org/security/2008/dsa-1571. Accessed 05 Nov 2013.
2. Lenstra A., Hughes J., Augier M., Bos J., Kleinjung T., Wachter C.: Public keys. In: Advances in Cryptology (Crypto 2012). LNCS, vol. 7417, pp. 626–642. Springer, Heidelberg (2012).
3. Marvin R.: Google admits an Android crypto PRNG flaw led to Bitcoin heist (2013). http://sdt.bz/64008 Accessed 01 Oct 2013.
4. Perlroth N., Larson J., Shane S.: N.S.A. able to foil basic safeguards of privacy on web. The New York Times (2013).
5. Koblitz N., Menezes A.: The random oracle model: a twenty-year retrospective. Cryptology ePrint Archive, Report 2015/140 (2015). http://eprint.iacr.org/.
6. Bernstein D.J., Lange T., Niederhagen R.: Dual EC: a standardized back door. Cryptology ePrint Archive, Report 2015/767 (2015). http://eprint.iacr.org/. Accessed July 2015.

7. Pornin T.: Deterministic usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA), RFC 6979 (2013).
8. Bellare M., Brakerski Z., Naor M., Ristenpart T., Segev G., Shacham H., Yilek S.: Hedged public-key encryption: how to protect against bad randomness. In: Advances in Cryptology (ASIACRYPT 2009). LNCS, pp. 232–249. Springer, Heidelberg (2009).
9. Yilek S.: Resettable public-key encryption: how to encrypt on a virtual machine. In: Proceedings of the 2010 International Conference on Topics in Cryptology (CT-RSA'10), pp. 41–56. Springer, Berlin (2010).
10. LaMacchia B., Lauter K., Mityagin A.: Stronger security of authenticated key exchange. In: Susilo W., Liu J.K., Mu Y. (eds.) ProvSec'07. LNCS, vol. 4784, pp. 1–16. Springer, Berlin (2007).
11. Canetti R., Krawczyk H.: Analysis of key-exchange protocols and their use for building secure channels. In: Pfitzmann B. (ed.) EUROCRYPT'01. LNCS, vol. 2045, pp. 453–474. Springer, London (2001).
12. Yang G., Duan S., Wong D.S., Tan C.H., Wang H.: Authenticated key exchange under bad randomness. In: Proceedings of the 15th International Conference on Financial Cryptography and Data Security. FC'11, pp. 113–126. Springer, Berlin (2012). doi:10.1007/978-3-642-27576-0_10.
13. Ristenpart T., Yilek S.: When good randomness goes bad: virtual machine reset vulnerabilities and hedging deployed cryptography. In: Proceedings of the Network and Distributed System Security Symposium (NDSS'10) (2010).
14. Kamara S., Katz J.: How to encrypt with a malicious random number generator. In: Fast Software Encryption. LNCS, vol. 5086, pp. 303–315. Springer, Berlin (2008).
15. Bellare M., Tackmann B.: Nonce-based cryptography: retaining security when randomness fails. Cryptology ePrint Archive, Report 2016/290 (2016). http://eprint.iacr.org/.
16. Krawczyk H.: HMQV: a high-performance secure Diffie–Hellman protocol. In: Shoup, V. (ed.) Advances in Cryptology (CRYPTO 2005). LNCS, vol. 3621, pp. 546–566. Springer, Berlin (2005).
17. Ustaoglu B.: Obtaining a secure and efficient key agreement protocol from (H)MQV and NAXOS. Cryptology ePrint Archive, Report 2007/123, 2007, version June 22 (2009).
18. Blake-Wilson S., Johnson D., Menezes A.: Key agreement protocols and their security analysis. In: Darnell M. (ed.) Crytography and Coding. LNCS, vol. 1355, pp. 30–45. Springer, Berlin (1997). doi:10.1007/BFb0024447.
19. Cremers C., Feltz M.: Beyond eCK: perfect forward secrecy under actor compromise and ephemeral-key reveal. Des. Codes Cryptogr. **74**(1), 183–218 (2015).
20. Brzuska C., Fischlin M., Warinschi B., Williams S.: Composability of Bellare-Rogaway key exchange protocols. In: Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11). pp. 51–62. ACM, New York (2011). doi:10.1145/2046707.2046716.
21. Boyd C., Cremers C., Feltz M., Paterson K., Poettering B., Stebila D.: ASICS: authenticated key exchange security incorporating certification systems. In: Crampton J., Jajodia S., Mayes K. (eds.) Computer Security (ESORICS 2013). LNCS, vol. 8134, pp. 381–399. Springer, Berlin (2013).
22. Bellare M., Rogaway P.: Entity authentication and key distribution. In: 13th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO'93), pp. 232–249. Springer, New York (1994).
23. Bellare M., Rogaway P.: Provably secure session key distribution: the three party case. In: 27th Annual ACM Symposium on Theory of Computing (STOC'95), pp. 57–66. ACM, New York (1995).
24. Bellare M., Pointcheval D., Rogaway P.: Authenticated key exchange secure against dictionary attacks. In: 19th International Conference on Theory and Application of Cryptographic Techniques (EUROCRYPT'00), pp. 139–155. Springer, Berlin (2000).
25. Cremers C., Feltz M.: Beyond eCK: perfect forward secrecy under actor compromise and ephemeral-key reveal. In: Proceedings of the 17th European Conference on Research in Computer Security. ESORICS. Springer, Berlin (2012).
26. Okamoto T., Pointcheval D.: The gap-problems: a new class of problems for the security of cryptographic schemes. In: Kim K. (ed.) PKC'2001. LNCS, vol. 1992, pp. 104–118. Springer, Berlin (2001).
27. Feltz M., Cremers C.: On the limits of authenticated key exchange security with an application to bad randomness. Cryptology ePrint Archive, Report 2014/369 (2014). http://eprint.iacr.org/.
28. Choo K.-K.R., Boyd C., Hitchcock Y.: Examining indistinguishability-based proof models for key establishment protocols. In: Advances in Cryptology—ASIACRYPT 2005, 11th International Conference on the Theory and Application of Cryptology and Information Security, Chennai, India, 4–8 Dec 2005, Proceedings. Lecture Notes in Computer Science, vol. 3788, pp. 585–604. Springer, Berlin (2005).
29. Schneier B., Fredrikson M., Kohno T., Ristenpart T.: Surreptitiously weakening cryptographic systems. Cryptology ePrint Archive, Report 2015/097 (2015). http://eprint.iacr.org/. Accessed March 2015.