A Cryptographically Sound Security Proof of the Needham-Schroeder-Lowe Public-Key Protocol

Michael Backes, Birgit Pfitzmann
IBM Zurich Research Lab
{mbc,bpf}@zurich.ibm.com

Abstract

We present the first cryptographically sound security proof of the well-known Needham-Schroeder-Lowe public-key protocol. More precisely, we show that the protocol is secure against arbitrary active attacks if it is implemented using provably secure cryptographic primitives. Although we achieve security under cryptographic definitions, our proof does not have to deal with probabilistic aspects of cryptography and is hence in the scope of current proof tools. The reason is that we exploit a recently proposed ideal cryptographic library, which has a provably secure cryptographic implementation. Besides establishing the cryptographic security of the Needham-Schroeder-Lowe protocol, our result also exemplifies the potential of this cryptographic library and paves the way for cryptographically sound verification of security protocols by means of formal proof tools.

1 Introduction

In recent times, the analysis of cryptographic protocols has been getting more and more attention, and the demand for rigorous proofs of cryptographic protocols has been rising.

One way to conduct such proofs is the cryptographic approach, whose security definitions are based on complexity theory, e.g., [12, 11, 13, 6]. The security of a cryptographic protocol is proved by reduction, i.e., by showing that breaking the protocol implies breaking one of the underlying cryptographic primitives with respect to its cryptographic definition. This approach captures a very comprehensive adversary model and allows for mathematically rigorous and precise proofs. However, because of probabilism and complexity-theoretic restrictions, these proofs have to be done by hand so far, which yields proofs with faults and imperfections. Moreover, such proofs rapidly become too complex for larger protocols.

The alternative is the formal-methods approach, which is concerned with the automation of proofs using model checkers and theorem provers. As these tools currently cannot deal with cryptographic details like error probabilities and computational restrictions, abstractions of cryptography are used. They are almost always based on the so-called Dolev-Yao model [10]. This model simplifies proofs of larger protocols considerably and gave rise to a large body of literature on analyzing the security of protocols using various techniques for formal verification, e.g., [19, 17, 14, 7, 21, 1].

A prominent example demonstrating the usefulness of the formal-methods approach is the work of Lowe [15], where he found a man-in-the-middle attack on the well-known Needham-Schroeder public-key protocol [20]. Lowe later proposed a repaired version of the protocol [16] and used the model checker FDR to prove that this modified protocol (henceforth known as the Needham-Schroeder-Lowe protocol) is secure in the Dolev-Yao model. The original and the repaired Needham-Schroeder public-key protocols are two of the most often investigated security protocols, e.g., [25, 18, 24, 26]. Various new approaches and formal proof tools for the analysis of security protocols were validated by showing that they can discover the known flaw or prove the fixed protocol in the Dolev-Yao model.

It is well-known and easy to show that the security flaw of the original protocol in the formal-methods approach can as well be used to mount a successful attack against any cryptographic implementation of the protocol. However, all existing proofs of security of the fixed protocol are restricted to the Dolev-Yao model, i.e., no theorem exists which allows for carrying over the results of an existing proof to the cryptographic approach with its much more comprehensive adversary. Although recent research focused on moving towards such a theorem, i.e., a cryptographically sound foundation of the formal-methods approach, the results are either specific for passive adversaries [3, 2] or they do not capture the local evaluation of nested cryptographic terms [8, 22], which is needed to model many usual cryptographic protocols. A recently proposed cryptographic library [5] allows for such nesting, but has not been applied to any security protocols yet. Thus, despite of the tremendous amount of research dedicated to the Needham-Schroeder-Lowe protocol, it is still an open question whether an actual implementation based on provably secure cryptographic primitives is secure under cryptographic security definitions.

We close this gap by providing the first security proof of the Needham-Schroeder-Lowe protocol in the cryptographic approach. We show that the protocol is secure against arbitrary active attacks if the Dolev-Yao-based abstraction of public-key encryption is implemented using a chosen-ciphertext secure public-key encryption scheme with small additions like ciphertext tagging. Chosen-ciphertext security was introduced in [23] and formulated as "IND-CCA2" in [6]. Efficient encryption systems secure in this sense exist under reasonable assumptions [9].

Obviously, establishing a proof in the cryptographic approach presupposes dealing with the mentioned cryptographic details, hence one naturally assumes that our proof heavily relies on complexity theory and is far out of scope of current proof tools. However, our proof is not performed from scratch in the cryptographic setting, but based on the mentioned cryptographic library [5]. This library provides cryptographically faithful, deterministic abstractions of cryptographic primitives, i.e., the abstractions can be securely implemented using actual cryptography. Moreover, the library allows for nesting the abstractions in an arbitrary way, quite similar to the original Dolev-Yao model. In a nutshell, it is sufficient to prove the security of the Needham-Schroeder-Lowe protocol based on the deterministic abstractions; then the result automatically carries over to the cryptographic setting. As the proof is deterministic and rigorous, it should be easily expressible in formal proof tools, in particular theorem provers. Even done by hand, our proof is much less prone to error than a reduction proof conducted from scratch in the cryptographic approach. We also want to point out that our result not only provides the up-to-now missing cryptographic security proof of the Needham-Schroeder-Lowe protocol, but also exemplifies the usefulness of the cryptographic library of [5] for the cryptographically sound verification of cryptographic protocols.

2 Preliminaries

In this section, we give an overview of the ideal cryptographic library of [5] and briefly sketch its provably secure implementation. We start by introducing the notation used in this paper.

2.1 Notation

We write "$:=$" for deterministic and "$\leftarrow$" for probabilistic assignment, and "$\xleftarrow{\mathcal{R}}$" for uniform random choice from a set. By $x := y{+}{+}$ for integer variables x, y we mean $y := y + 1; x := y$. The length of a message m is denoted as $|m|$, and $\downarrow$ is an error element available as an addition to the domains and ranges of all functions and algorithms. The list operation is denoted as $l := (x_1, \ldots, x_j)$, and the arguments are unambiguously retrievable as $l[i]$, with $l[i] = \downarrow$ if $i > j$. A database D is a set of functions, called entries, each over a finite domain called attributes. For an entry $x \in D$, the value at an attribute att is written $x.att$. For a predicate $pred$ involving attributes, $D[pred]$ means the subset of entries whose attributes fulfill $pred$. If $D[pred]$ contains only one element, we use the same notation for this element. Adding an entry x to D is abbreviated $D :\Leftarrow x$.

2.2 Overview of the Ideal and Real Cryptographic Library

The ideal (abstract) cryptographic library of [5] offers its users abstract cryptographic operations, such as commands to encrypt or decrypt a message, to make or test a signature, and to generate a nonce. All these commands have a simple, deterministic semantics. To allow a reactive scenario, this semantics is based on state, e.g., of who already knows which terms; the state is represented as a database. Each entry has a type (e.g., "ciphertext"), and pointers to its arguments (e.g., a key and a message). Further, each entry contains handles for those participants who already know it. A send operation makes an entry known to other participants, i.e., it adds handles to the entry. The ideal cryptographic library does not allow cheating. For instance, if it receives a command to encrypt a message m with a certain key, it simply makes an abstract database entry for the ciphertext. Another user can only ask for decryption of this ciphertext if he has obtained handles to both the ciphertext and the secret key.

To allow for the proof of cryptographic faithfulness, the library is based on a detailed model of asynchronous reactive systems introduced in [22] and represented as a deterministic machine $\mathsf{TH}_{\mathcal{H}}$, called *trusted host*. The parameter $\mathcal{H} \subseteq \{1 \ldots, n\}$ denotes the honest participants, where n is a parameter of the library denoting the overall number of participants. Depending on the considered set $\mathcal{H}$, the trusted host offers slightly extended capabilities for the adversary. However, for current purposes, the trusted host can be seen as a slightly modified Dolev-Yao model together with a network and intruder model, similar to "the CSP Dolev-Yao model" or "the inductive-approach Dolev-Yao model".

The real cryptographic library offers its users the same commands as the ideal one, i.e., honest users operate on cryptographic objects via handles. The objects are now real cryptographic keys, ciphertexts, etc., handled by real distributed machines. Sending a term on an insecure channel releases the actual bitstring to the adversary, who can do with it what he likes. The adversary can also insert arbitrary bitstrings on non-authentic channels. The implementation of the commands is based on arbitrary secure encryption and signature systems according to standard cryptographic definitions, with certain additions like type tagging and additional randomizations.

The security proof of [5] states that the real library is *at least as secure* as the ideal library. This is captured using the notion of *simulatability*, which states that whatever an adversary can achieve in the real implementation, another adversary can achieve given the ideal library, or otherwise the underlying cryptography can be broken [22]. This is the strongest possible cryptographic relationship between a real and an ideal system. In particular it covers active attacks. Moreover, a composition theorem exists in the underlying model [22], which states that one can securely replace the ideal library in larger systems with the real library, i.e., without destroying the already established simulatability relation.

3 The Needham-Schroeder-Lowe Public-Key Protocol

The original Needham-Schroeder protocol and Lowe's variant consist of seven steps, where four steps deal with key generation and public-key distribution. These steps are usually omitted in a security analysis, and it is simply assumed that keys have already been generated and distributed. We do this as well to keep the proof short. However, the underlying cryptographic library offers commands for modeling the remaining steps as well. The main part of the Needham-Schroeder-Lowe public-key protocol consists of the following three steps, expressed in the typical protocol notation, as in, e.g., [15].

$$\begin{array}{lcll} 1. & u \rightarrow v & : & E_{pk_v}(N_u, u) \\ 2. & v \rightarrow u & : & E_{pk_u}(N_u, N_v, v) \\ 3. & u \rightarrow v & : & E_{pk_v}(N_v). \end{array}$$

Here, user u seeks to establish a session with user v. He generates a nonce N_u and sends it to v together with its identity, encrypted with v's public key (first message). Upon receiving this message, v decrypts it to obtain the nonce N_u. Then v generates a new nonce N_v and sends both nonces and its identity back to u, encrypted with u's public key (second message). Upon receiving this message, u decrypts it and tests whether the contained identity v equals the sender of the message and whether u earlier sent the first contained nonce to user v. If yes, u sends the second nonce back to v, encrypted with v's public key (third message). Finally, v decrypts this message; and if v had earlier sent the contained nonce to u, then v believes to speak with u.

3.1 The Needham-Schroeder-Lowe Protocol Using the Abstract Library

We now show how to model the Needham-Schroeder-Lowe protocol in the framework of [22] and using the ideal cryptographic library. For each user $u \in \{1, \dots, n\}$, we define a machine $\mathsf{M}_u^{\mathsf{NS}}$, called a *protocol machine*, which executes the protocol sketched above for participant identity u. It is connected to its user via ports $\mathsf{EA\_out}_u!$, $\mathsf{EA\_in}_u?$ ("EA" for "Entity Authentication", because the behavior at these ports is the same for all entity authentication protocols) and to the cryptographic library via ports $\mathsf{in}_u!$, $\mathsf{out}_u?$. The notation follows the CSP convention, e.g., the cryptographic library has a port $\mathsf{in}_u?$ where it obtains messages output at $\mathsf{in}_u!$. The combination of the protocol machines $\mathsf{M}_u^{\mathsf{NS}}$ and the trusted host $\mathsf{TH}_{\mathcal{H}}$ is the *ideal Needham-Schroeder-Lowe system* $Sys^{\mathsf{NS,id}}$. It is shown in Figure 1; H and A model the arbitrary joint honest users and the adversary, respectively.

Using the notation of [5], the system $Sys^{\mathsf{NS,id}}$ consists of several *structures* $(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}})$, one for each value of the parameter $\mathcal{H}$. Each structure consists of a set $\hat{M}_{\mathcal{H}} := \{\mathsf{TH}_{\mathcal{H}}\} \cup \{\mathsf{M}_u^{\mathsf{NS}} \mid u \in \mathcal{H}\}$ of machines, i.e., for a given set $\mathcal{H}$ of honest users, only the machines $\mathsf{M}_u^{\mathsf{NS}}$ with $u \in \mathcal{H}$ are actually present in a protocol run. The others are subsumed in the adversary. $S_{\mathcal{H}}$ denotes those ports of $\hat{M}_{\mathcal{H}}$ that the honest users connect to, i.e., $S_{\mathcal{H}} := \{\mathsf{EA\_in}_u?, \mathsf{EA\_out}_u! \mid u \in \mathcal{H}\}$. Formally, we obtain $Sys^{\mathsf{NS,id}} := \{(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}}) \mid \mathcal{H} \subseteq \{1, \dots, n\}\}$.

In order to capture that keys have been generated and distributed, we assume that suitable entries for the keys already exist in the database. We denote the handle of u_1 to the public key as $pke_{u,u_1}^{\mathsf{hnd}}$ and the handle of u to its secret key as ske_u^{hnd}. We show in Section 6.2 how to deal with this formally, after we have given a detailed description of the ideal cryptographic library.

The state of the machine $\mathsf{M}_u^{\mathsf{NS}}$ consists of the bitstring u and a family $(Nonce_{u,v})_{v \in \{1,\dots,n\}}$ of sets of handles. Each set $Nonce_{u,v}$ is initially empty. We now define how the machine $\mathsf{M}_u^{\mathsf{NS}}$ evaluates inputs. They either come from user u at port $\mathsf{EA\_in}_u?$ or from $\mathsf{TH}_{\mathcal{H}}$ at port $\mathsf{out}_u?$. The behavior of $\mathsf{M}_u^{\mathsf{NS}}$ in both cases is described in Algorithm 1 and 2 respectively, which we will describe below. We refer to Step i of Algorithm j as Step $j.i$. Both algorithms should immediately abort if a command to the cryptographic library does not yield the desired result, e.g., if a decryption requests fails. For readability we omit these abort checks in the algorithm descriptions; instead we impose the following convention on both algorithms.

Convention 1 *If* $\mathsf{M}_u^{\mathsf{NS}}$ *enters a command at port* $\mathsf{in}_u!$ *and receives* $\downarrow$ *at port* $\mathsf{out}_u?$ *as the immediate answer of the cryptographic library, then* $\mathsf{M}_u^{\mathsf{NS}}$ *aborts the execution of the current algorithm, except if the command was of the form* list_proj *or* send_i.

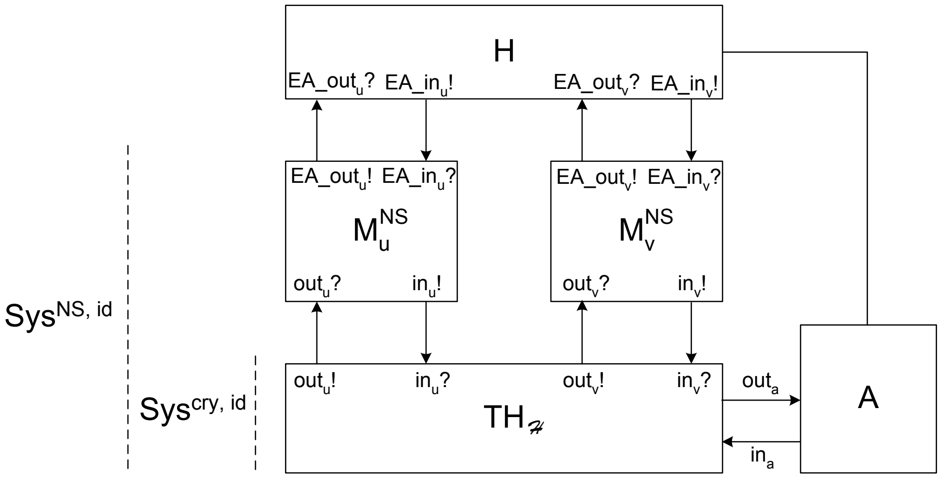

Figure 1: Overview of the Needham-Schroeder-Lowe Ideal System.

The user of the machine $\mathsf{M}_u^{\mathsf{NS}}$ can start a new protocol with user $v \in \{1, \ldots, n\} \setminus \{u\}$ by inputting $(\mathsf{new\_prot}, v)$ at port $\mathsf{EA\_in}_u?$. Our security proof holds for all adversaries and all honest users, i.e., especially those that start protocols with the adversary (respectively a malicious user) in parallel with protocols with honest users. Upon such an input, $\mathsf{M}_u^{\mathsf{NS}}$ builds up the term corresponding to the first protocol message using the ideal cryptographic library $\mathsf{TH}_\mathcal{H}$ according to Algorithm 1. The command gen_nonce generates the ideal nonce. $\mathsf{M}_u^{\mathsf{NS}}$ stores the resulting handle n_u^{hnd} in $Nonce_{u,v}$ for future comparison. The command store inputs arbitrary application data into the cryptographic library, here the user identity u. The command list forms a list and encrypt is encryption. Since only lists are allowed to be transferred in $\mathsf{TH}_\mathcal{H}$ (because the list-operation is a convenient place to concentrate all verifications that no secret items are put into messages), the encryption is packed as a list again. The final command send_i means that $\mathsf{M}_u^{\mathsf{NS}}$ sends the resulting term to v over an insecure channel. The effect is that the adversary obtains a handle to the term and can decide what to do with it (such as forwarding it to $\mathsf{M}_v^{\mathsf{NS}}$).

The behavior of $\mathsf{M}_u^{\mathsf{NS}}$ upon receiving an input from the cryptographic library at port $\mathsf{out}_u?$ (corresponding to a message that arrives over the network) is defined similarly in Algorithm 2. By construction of $\mathsf{TH}_\mathcal{H}$, such an input is always of the form $(v, u, \mathsf{i}, m^{\mathsf{hnd}})$ where m^{hnd} is a handle to a list. $\mathsf{M}_u^{\mathsf{NS}}$ first decrypts the list content using the secret key of user u, which yields a handle l^{hnd} to an inner list. This list is parsed into at most three components using the command list_proj. If the list has two elements, i.e., it could correspond to the first message of the protocol, $\mathsf{M}_u^{\mathsf{NS}}$ generates a new nonce and stores its handle in $Nonce_{u,v}$. After that, $\mathsf{M}_u^{\mathsf{NS}}$ builds up a new list according to the protocol description, encrypts the list and sends it to user v. If the list has three elements, i.e., it could correspond to the second message of the protocol, then $\mathsf{M}_u^{\mathsf{NS}}$ tests whether the third list element equals v and whether the first list element s already contained in the set $Nonce_{u,v}$. If one of these tests does not succeed, $\mathsf{M}_u^{\mathsf{NS}}$ aborts. Otherwise, it again builds up a term according to the protocol description and sends it to user v. Finally, if the list has only one element, i.e., it could correspond to the third message of the protocol, then $\mathsf{M}_u^{\mathsf{NS}}$ tests if the handle of this element is already contained in the set $Nonce_{u,v}$. If so, $\mathsf{M}_u^{\mathsf{NS}}$ outputs (ok, v) at $\mathsf{EA\_out}_u!$. This signals that the protocol with user v has terminated successfully, i.e., u believes to speak with v.

3.2 On Polynomial Runtime

In order to use existing composition results of the underlying model, the machines $\mathsf{M}_u^{\mathsf{NS}}$ have to be polynomial-time. Similar to the cryptographic library, we hence define that each machine $\mathsf{M}_u^{\mathsf{NS}}$ main-

Algorithm 1 Evaluation of Inputs from the User (Protocol Start)

Input: $(\mathsf{new\_prot}, v)$ at $\mathsf{EA\_in}_u?$ with $v \in \{1, \ldots, n\} \setminus \{u\}$.

$n_u^{\mathsf{hnd}} \leftarrow \mathsf{gen\_nonce}()$.

$Nonce_{u,v} := Nonce_{u,v} \cup \{n_u^{\mathsf{hnd}}\}$.

$u^{\mathsf{hnd}} \leftarrow \mathsf{store}(u)$.

$l_1^{\mathsf{hnd}} \leftarrow \mathsf{list}(n_u^{\mathsf{hnd}}, u^{\mathsf{hnd}})$.

$c_1^{\mathsf{hnd}} \leftarrow \mathsf{encrypt}(pke_{v,u}^{\mathsf{hnd}}, l_1^{\mathsf{hnd}})$.

$m_1^{\mathsf{hnd}} \leftarrow \mathsf{list}(c_1^{\mathsf{hnd}})$.

$\mathsf{send\_i}(v, m_1^{\mathsf{hnd}})$.

tains explicit polynomial bounds on the message lengths and the number of inputs accepted at each port.

4 The Security Property

Our security property states that an honest participant v only successfully terminates a protocol with an honest participant u if u has indeed started a protocol with v, i.e., an output (ok, u) at $\mathsf{EA\_out}_v!$ can only happen if there was a prior input $(\mathsf{new\_prot}, v)$ at $\mathsf{EA\_in}_u?$. This property and also the actual protocol does not consider replay attacks, i.e., a user v could successfully terminate a protocol with u multiple times but u only once started a protocol with v. However, this can easily be avoided as follows: If $\mathsf{M}_u^{\mathsf{NS}}$ receives a message from v containing a nonce and $\mathsf{M}_u^{\mathsf{NS}}$ created this nonce, then it additionally removes this nonce from the set $Nonce_{u,v}$.Formally, this means that after Steps 2.20 and 2.25, the handle x_1^{hnd} is removed from $Nonce_{u,v}$.

Integrity properties in the underlying model are formally sets of traces at the in- and output ports connecting the system to the honest users, i.e., here traces at the port set $S_{\mathcal{H}} = \{\mathsf{EA\_out}_u!, \mathsf{EA\_in}_u? \mid u \in \mathcal{H}\}$. Intuitively, such an integrity property Req states which are the "good" traces at these ports. A trace is a sequence of sets of events. We write an event $p?m$ or $p!m$, meaning that message m occurs at input or output port p. The t-th step of a trace r is written r_t; we also speak of the step at time t. Thus the integrity requirement Req^{EA} for the Needham-Schroeder-Lowe protocol is formally defined as follows:

Definition 4.1 *(Entity Authentication Requirement) A trace r is contained in Req^{EA} if for all $u, v \in \mathcal{H}$:*

$$
\begin{array}{ll}
\exists t_1 \in \mathbb{N}\colon \mathsf{EA\_out}_v!(\mathsf{ok}, u) \in r_{t_1} & \textit{\# If } v \textit{ believes to speak with } u \textit{ at time } t_1 \\
\;\Rightarrow \exists t_0 < t_1\colon & \textit{\# then there exists a past time } t_0 \\
\mathsf{EA\_in}_u?(\mathsf{new\_prot}, v) \in r_{t_0} & \textit{\# in which } u \textit{ started a protocol with } v
\end{array}
$$

$\diamond$

The notion of a system Sys fulfilling an integrity property Req essentially comes in two flavors [4]. *Perfect fulfillment*, $Sys \models^{\mathsf{perf}} Req$, means that the integrity property holds for all traces arising in runs of Sys (a well-defined notion from the underlying model [22]). *Computational fulfillment*, $Sys \models^{\mathsf{poly}} Req$, means that the property only holds for polynomially bounded users and adversaries, and only with negligible error probability. Perfect fulfillment implies computational fulfillment.

The following theorem captures the security of the ideal Needham-Schroeder-Lowe protocol.

Theorem 4.1 *(Security of the Needham-Schroeder-Lowe Protocol based on the Ideal Cryptographic Library) Let $Sys^{\mathsf{NS,id}}$ be the ideal Needham-Schroeder-Lowe system defined in Section 3, and Req^{EA} the integrity property of Definition 4.1. Then $Sys^{\mathsf{NS,id}} \models^{\mathsf{perf}} Req^{\mathsf{EA}}$.* $\square$

Algorithm 2 Evaluation of Inputs from $\mathsf{TH}_{\mathcal{H}}$ (Network Inputs)

Input: $(v, u, \mathsf{i}, m^{\mathsf{hnd}})$ at $\mathsf{out}_u?$ with $v \in \{1, \ldots, n\} \setminus \{u\}$.

$c^{\mathsf{hnd}} \leftarrow \mathsf{list\_proj}(m^{\mathsf{hnd}}, 1)$
$l^{\mathsf{hnd}} \leftarrow \mathsf{decrypt}(ske_u^{\mathsf{hnd}}, c^{\mathsf{hnd}})$
$x_i^{\mathsf{hnd}} \leftarrow \mathsf{list\_proj}(l^{\mathsf{hnd}}, i)$ for $i = 1, 2, 3$.
if $x_1^{\mathsf{hnd}} \neq \downarrow \wedge x_2^{\mathsf{hnd}} \neq \downarrow \ \wedge x_3^{\mathsf{hnd}} = \downarrow$ **then** {First Message is input}
  $x_2 \leftarrow \mathsf{retrieve}(x_2^{\mathsf{hnd}})$.
  if $x_2 \neq v$ **then**
    Abort
  end if
  $n_u^{\mathsf{hnd}} \leftarrow \mathsf{gen\_nonce}()$.
  $Nonce_{u,v} := Nonce_{u,v} \cup \{n_u^{\mathsf{hnd}}\}$.
  $u^{\mathsf{hnd}} \leftarrow \mathsf{store}(u)$.
  $l_2^{\mathsf{hnd}} \leftarrow \mathsf{list}(x_1^{\mathsf{hnd}}, n_u^{\mathsf{hnd}}, u^{\mathsf{hnd}})$.
  $c_2^{\mathsf{hnd}} \leftarrow \mathsf{encrypt}(pke_{v,u}^{\mathsf{hnd}}, l_2^{\mathsf{hnd}})$.
  $m_2^{\mathsf{hnd}} \leftarrow \mathsf{list}(c_2^{\mathsf{hnd}})$.
  $\mathsf{send\_i}(v, m_2^{\mathsf{hnd}})$.
else if $x_1^{\mathsf{hnd}} \neq \downarrow \ \wedge x_2^{\mathsf{hnd}} \neq \downarrow \wedge x_3^{\mathsf{hnd}} \neq \downarrow$ **then** {Second Message is input}
  $x_3 \leftarrow \mathsf{retrieve}(x_3^{\mathsf{hnd}})$.
  if $x_3 \neq v \vee x_1^{\mathsf{hnd}} \notin Nonce_{u,v}$ **then**
    Abort
  end if
  $l_3^{\mathsf{hnd}} \leftarrow \mathsf{list}(x_2^{\mathsf{hnd}})$.
  $c_3^{\mathsf{hnd}} \leftarrow \mathsf{encrypt}(pke_{v,u}^{\mathsf{hnd}}, l_3^{\mathsf{hnd}})$.
  $m_3^{\mathsf{hnd}} \leftarrow \mathsf{list}(c_3^{\mathsf{hnd}})$.
  $\mathsf{send\_i}(v, m_3^{\mathsf{hnd}})$.
else if $x_1^{\mathsf{hnd}} \in Nonce_{u,v} \wedge \ x_2^{\mathsf{hnd}} = x_3^{\mathsf{hnd}} = \downarrow$ **then** {Third Message is input}
  Output (ok, v) at $\mathsf{EA\_out}_u!$.
end if

5 Proof of the Cryptographic Realization

If Theorem 4.1 has been proven, it follows easily that the Needham-Schroeder-Lowe protocol based on the real cryptographic library computationally fulfills the integrity requirement Req^{EA}. The main tool is the following *preservation theorem* from [4].

Theorem 5.1 *(Preservation of Integrity Properties (Sketch)) Let two systems Sys_1, Sys_2 be given such that Sys_1 is at least as secure as Sys_2 (written $Sys_1 \geq_{\mathsf{sec}}^{\mathsf{poly}} Sys_2$). Let Req be an integrity requirement for both Sys_1 and Sys_2, and let $Sys_2 \models^{\mathsf{poly}} Req$. Then also $Sys_1 \models^{\mathsf{poly}} Req$.* □

Let $Sys^{\mathsf{cry,id}}$ and $Sys^{\mathsf{cry,real}}$ denote the ideal and the real cryptographic library from [5], and $Sys^{\mathsf{NS,real}}$ the Needham-Schroeder-Lowe protocol based on the real cryptographic library. This is well-defined given the formalization with the ideal library because the real library has the same ports and offers the same commands.

Theorem 5.2 *(Security of the Real Needham-Schroeder-Lowe Protocol) Let Req^{EA} denote the integrity property of Definition 4.1. Then $Sys^{\mathsf{NS,real}} \models^{\mathsf{poly}} Req^{\mathsf{EA}}$.* □

Proof. In [5] it has already been shown that $Sys^{\mathsf{cry,real}} \geq_{\mathsf{sec}}^{\mathsf{poly}} Sys^{\mathsf{cry,id}}$ holds for suitable parameters in the ideal system. Since $Sys^{\mathsf{NS,real}}$ is derived from $Sys^{\mathsf{NS,id}}$ by replacing the ideal with the real cryptographic library, $Sys^{\mathsf{NS,real}} \geq_{\mathsf{sec}}^{\mathsf{poly}} Sys^{\mathsf{NS,id}}$ follows from the composition theorem of [22]. We only have to show that the theorem's preconditions are in fact fulfilled. This is straightforward, since the machines $\mathsf{M}_u^{\mathsf{NS}}$ are polynomial-time (cf. Section 3.2). Now Theorem 4.1 implies $Sys^{\mathsf{NS,id}} \models^{\mathsf{poly}} Req^{\mathsf{EA}}$, hence Theorem 5.1 yields $Sys^{\mathsf{NS,real}} \models^{\mathsf{poly}} Req^{\mathsf{EA}}$. ∎

6 Proof in the Ideal Setting

This section contains the proof of Theorem 4.1, i.e., the proof of the Needham-Schroeder-Lowe protocol using the ideal, deterministic cryptographic library. The proof idea is to go backwards in the protocol step by step, and to show that a specific output always requires a specific prior input. For instance, when user v successfully terminates a protocol with user u, then u has sent the third protocol message to v; thus v has sent the second protocol message to u; and so on. The main challenge in this proof was to find suitable invariants on the state of the ideal Needham-Schroeder-Lowe system.

We start with the rigorous definition of the state and the commands of the ideal cryptographic library used for modeling the Needham-Schroeder-Lowe protocol. We also describe the *local adversary commands* that model the slightly extended capabilities of the adversary. After that, we state the invariants of the system $Sys^{\mathsf{NS,id}}$.

6.1 Detailed Description of the Cryptographic Library

6.1.1 States of the Library

The machine $\mathsf{TH}_{\mathcal{H}}$ has ports $\mathsf{in}_u?$ and $\mathsf{out}_u!$ for inputs from and outputs to each user $u \in \mathcal{H}$ and for $u = \mathsf{a}$, denoting the adversary. Besides the number n of users, the ideal cryptographic library is parameterized by a tuple L of length functions which are used to calculate the "length" of an abstract entry, corresponding to the length of the corresponding bitstring in the real implementation. Moreover, L contains bounds on the message lengths and the number of accepted inputs at each port. These bounds can be arbitrarily large, but have to be polynomially bounded in the security parameter. Using the notation of [5], the ideal cryptographic library is a *system* $Sys_{n,L}^{\mathsf{cry,id}} := \{(\{\mathsf{TH}_{\mathcal{H}}\}, S_{\mathcal{H}}) \mid \mathcal{H} \subseteq \{1, \ldots, n\}\}$, cf. the definition of the ideal Needham-Schroeder-Lowe system in Section 3.1. In the following, we omit the parameters n and L for simplicity.[1]

As the machines $\mathsf{M}_u^{\mathsf{NS}}$ of the Needham-Schroeder-Lowe protocol only make bounded-length inputs to $\mathsf{TH}_{\mathcal{H}}$ given n (this follows from the fixed term structure and coding conventions in [5]), the bounds in L can easily be chosen large enough so that all these inputs are legal. Further, as we only prove an integrity property, it is not a problem in the proof that the number of accepted inputs might be exceeded. Hence we omit the details of the length functions from [5]. We present the full definitions of the commands, but the reader need not worry about functions with names x_len.

The main data structure of $\mathsf{TH}_{\mathcal{H}}$ is a database D. The entries of D are abstract representations of the data produced during a system run, together with the information on who knows these data. Each entry in D is of the form (recall the notation in Section 2.1)

$$(ind, type, arg, hnd_{u_1}, \ldots, hnd_{u_m}, hnd_{\mathsf{a}}, len)$$

where $\mathcal{H} = \{u_1, \ldots, u_m\}$. For each entry $x \in D$:

[1]Formally, these parameters are thus also parameters of the ideal Needham-Schroeder-Lowe system $Sys^{\mathsf{NS,id}}$.

- $x.ind \in \mathcal{INDS}$, called index, consecutively numbers all entries in D. The set $\mathcal{INDS}$ is isomorphic to $\mathbb{N}$ and is used to distinguish index arguments from others. The index is used as a primary key attribute of the database, i.e., we write $D[i]$ for the selection $D[ind = i]$.
- $x.type \in typeset$ identifies the *type* of x.
- $x.arg = (a_1, a_2, \ldots, a_j)$ is a possibly empty list of arguments. Many values a_i are indices of other entries in D and thus in $\mathcal{INDS}$. We sometimes distinguish them by a superscript "ind".
- $x.hnd_u \in \mathcal{HNDS} \cup \{\downarrow\}$ for $u \in \mathcal{H} \cup \{\mathsf{a}\}$ are handles by which a user or adversary u knows this entry. $x.hnd_u = \downarrow$ means that u does not know this entry. The set $\mathcal{HNDS}$ is yet another set isomorphic to $\mathbb{N}$. We always use a superscript "hnd" for handles.
- $x.len \in \mathbb{N}_0$ denotes the "length" of the entry; it is computed by applying the functions from L.

Initially, D is empty. $\mathsf{TH}_\mathcal{H}$ has a counter $size \in \mathcal{INDS}$ for the current size of D. For the handle attributes, it has counters $curhnd_u$ (current handle) initialized with 0.

6.1.2 Evaluation of Commands

Each input c at a port $\mathsf{in}_u?$ with $u \in \mathcal{H} \cup \{\mathsf{a}\}$ should be a list $(cmd, x_1, \ldots, x_j)$ and cmd from a fixed list of commands. We usually write it $y \leftarrow cmd(x_1, \ldots, x_j)$ with a variable y designating the result that $\mathsf{TH}_\mathcal{H}$ returns at $\mathsf{out}_u!$. The algorithm $i^{\mathsf{hnd}} := \mathsf{ind2hnd}_u(i)$ (with side effect) denotes that $\mathsf{TH}_\mathcal{H}$ determines a handle i^{hnd} for user u to an entry $D[i]$: If $i^{\mathsf{hnd}} := D[i].hnd_u \neq \downarrow$, it returns that, else it sets and returns $i^{\mathsf{hnd}} := D[i].hnd_u := curhnd_u{++}$. On non-handles, it is the identity function. The function $\mathsf{ind2hnd}_u^*$ applies $\mathsf{ind2hnd}_u$ to each element of a list.

Basic Commands. In the following definitions, we assume that a basic commands is input at the port $\mathsf{in}_u?$ with $u \in \mathcal{H} \cup \{\mathsf{a}\}$. First, we describe the commands for storing and retrieving data via handles.

- *Storing:* $m^{\mathsf{hnd}} \leftarrow \mathsf{store}(m)$, for $m \in \{0,1\}^{\mathsf{max\_len}(k)}$.

  If $i := D[type = \mathsf{data} \wedge arg = (m)].ind \neq \downarrow$ then return $m^{\mathsf{hnd}} := \mathsf{ind2hnd}_u(i)$.[2] Otherwise if $\mathsf{data\_len}^*(|m|) > \mathsf{max\_len}(k)$ return $\downarrow$. Else set $m^{\mathsf{hnd}} := curhnd_u{++}$ and

  $$D :\Leftarrow (ind := size{++}, type := \mathsf{data}, arg := (m), hnd_u := m^{\mathsf{hnd}}, len := \mathsf{data\_len}^*(|m|)).$$

- *Retrieval:* $m \leftarrow \mathsf{retrieve}(m^{\mathsf{hnd}})$.

  $m := D[hnd_u = m^{\mathsf{hnd}} \wedge type = \mathsf{data}].arg[1]$.[3]

Next we describe list creation and list projection. Lists cannot include secret keys of the public-key systems (entries of type ske, sks) because no information about those must be given away.

- *Generate a list:* $l^{\mathsf{hnd}} \leftarrow \mathsf{list}(x_1{}^{\mathsf{hnd}}, \ldots, x_j{}^{\mathsf{hnd}})$, for $0 \leq j \leq \mathsf{max\_len}(k)$.

  Let $x_i := D[hnd_u = x_i{}^{\mathsf{hnd}}].ind$ for $i = 1, \ldots, j$. If any $D[x_i].type \in \{\mathsf{sks}, \mathsf{ske}\}$, set $l^{\mathsf{hnd}} := \downarrow$.

  If $l := D[type = \mathsf{list} \wedge arg = (x_1, \ldots, x_j)].ind \neq \downarrow$, then return $l^{\mathsf{hnd}} := \mathsf{ind2hnd}_u(l)$. Otherwise, set $length := \mathsf{list\_len}^*(D[x_1].len, \ldots, D[x_j].len)$ and return $\downarrow$ if $length > \mathsf{max\_len}(k)$. Else set $l^{\mathsf{hnd}} := curhnd_u{++}$ and

  $$D :\Leftarrow (ind := size{++}, type := \mathsf{list}, arg := (x_1, \ldots, x_j), hnd_u := l^{\mathsf{hnd}}, len := length).$$

[2] Hence if the same string m is stored twice, $\mathsf{TH}_\mathcal{H}$ reuses the first result.

[3] This implies that m^{hnd} was created by a store command, as no other command creates entries with $type = \mathsf{data}$. Thus only explicitly stored data can be retrieved and not, e.g., keys or ciphertexts.

- *i-th projection:* $x^{\mathsf{hnd}} \leftarrow \mathsf{list\_proj}(l^{\mathsf{hnd}}, i)$, for $1 \le i \le \mathsf{max\_len}(k)$.

  If $D[hnd_u = l^{\mathsf{hnd}} \wedge type = \mathsf{list}].arg = (x_1, \ldots, x_j)$ with $j \ge i$, then $x^{\mathsf{hnd}} := \mathsf{ind2hnd}_u(x_i)$, otherwise $x^{\mathsf{hnd}} := \downarrow$.

The abstract command to create a fresh nonce simply creates a new entry in $\mathsf{TH}_{\mathcal{H}}$.

- *Generate a nonce:* $n^{\mathsf{hnd}} \leftarrow \mathsf{gen\_nonce}()$.

  Set $n^{\mathsf{hnd}} := curhnd_u$++ and

$$D :\Leftarrow (ind := size\text{++}, type := \mathsf{nonce}, arg := (), hnd_u := n^{\mathsf{hnd}}, len := \mathsf{nonce\_len}^*(k)).$$

Finally, we used commands to encrypt and decrypt a list. Since we assume that keys have already been generated, we omit a detailed description of the key generation command gen_enc_keypair.

- *Encryption:* $c^{\mathsf{hnd}} \leftarrow \mathsf{encrypt}(pk^{\mathsf{hnd}}, l^{\mathsf{hnd}})$.

  Let $pk := D[hnd_u = pk^{\mathsf{hnd}} \wedge type = \mathsf{pke}].ind$ and $l := D[hnd_u = l^{\mathsf{hnd}} \wedge type = \mathsf{list}].ind$ and $length := \mathsf{enc\_len}^*(k, D[l].len)$. If $length > \mathsf{max\_len}(k)$ or $pk = \downarrow$ or $l = \downarrow$, then return $\downarrow$. Else set $c^{\mathsf{hnd}} := curhnd_u$++ and

$$D :\Leftarrow (ind := size\text{++}, type := \mathsf{enc}, arg := (pk, l), hnd_u := c^{\mathsf{hnd}}, len := length).$$

- *Decryption:* $l^{\mathsf{hnd}} \leftarrow \mathsf{decrypt}(sk^{\mathsf{hnd}}, c^{\mathsf{hnd}})$.

  Let $sk := D[hnd_u = sk^{\mathsf{hnd}} \wedge type = \mathsf{ske}].ind$ and $c := D[hnd_u = c^{\mathsf{hnd}} \wedge type = \mathsf{enc}].ind$. Return $\downarrow$ if $c = \downarrow$ or $sk = \downarrow$ or $pk := D[c].arg[1] \ne sk + 1$ or $l := D[c].arg[2] = \downarrow$. Else return $l^{\mathsf{hnd}} := \mathsf{ind2hnd}_u(l)$.

Local Adversary Commands. From the set of local adversary commands, which capture additional commands for the adversary at port $\mathsf{in_a}?$, we only describe the command adv_parse. It allows the adversary to retrieve all information that we do not explicitly require to be hidden. This command returns the type and usually all the abstract arguments of a value (with indices replaced by handles), except in the case of ciphertexts.

- *Parameter retrieval:* $(type, arg) \leftarrow \mathsf{adv\_parse}(m^{\mathsf{hnd}})$.

  Let $m := D[hnd_{\mathsf{a}} = m^{\mathsf{hnd}}].ind$ and $type := D[m].type$. In most cases, set $arg := \mathsf{ind2hnd}^*_{\mathsf{a}}(D[m].arg)$. (Recall that this only transforms arguments in $\mathcal{INDS}$.) The only exception is for $type = \mathsf{enc}$ and $D[m].arg$ of the form (pk, l) (a valid ciphertext) and $D[pk-1].hnd_{\mathsf{a}} = \downarrow$ (the adversary does not know the secret key); then $arg := (\mathsf{ind2hnd_a}(pk), D[l].len)$.

About the remaining local adversary commands we only need to know that they do not output handles to already existing entries of type list or nonce.

Send Commands. We finally describe the send commands for sending messages on insecure channels.

- $\mathsf{send\_i}(v, l^{\mathsf{hnd}})$, for $v \in \{1, \ldots, n\}$ at port $\mathsf{in}_u?$ for $u \in \mathcal{H}$.

  Let $l^{\mathsf{ind}} := D[hnd_u = l^{\mathsf{hnd}} \wedge type = \mathsf{list}].ind$. If $l^{\mathsf{ind}} \ne \downarrow$, then output $(u, v, \mathsf{i}, \mathsf{ind2hnd_a}(l^{\mathsf{ind}}))$ at $\mathsf{out_a}!$.

- $\mathsf{adv\_send\_i}(u, v, l^{\mathsf{hnd}})$, for $u \in \{1, \ldots, n\}$ and $v \in \mathcal{H}$ at port $\mathsf{in_a}?$.

  Intuitively, the adversary wants to send list l to v, pretending to be u. Let $l^{\mathsf{ind}} := D[\mathit{hnd}_{\mathsf{a}} = l^{\mathsf{hnd}} \wedge \mathit{type} = \mathsf{list}].\mathit{ind}$. If $l^{\mathsf{ind}} \neq \downarrow$, output $(u, v, \mathsf{i}, \mathsf{ind2hnd}_v(l^{\mathsf{ind}}))$ at $\mathsf{out}_v!$.

For the proof of Theorem 4.1, the following property of $\mathsf{TH}_{\mathcal{H}}$ proven in [5] will be useful.

Lemma 6.1 The ideal cryptographic library $\mathit{Sys}^{\mathsf{cry,id}}$ has the following property: The only modifications to existing entries x in D are assignments to previously undefined attributes $x.\mathit{hnd}_u$ (except for counter updates in entries for signature keys, which we do not have to consider here). □

6.2 Capturing Distributed Keys

For the ideal cryptographic library, the assumption that keys have already been generated and distributed (Section 3.1) means that we start with an initially empty database D, and for each user $u \in \mathcal{H}$ two entries of the following form are added:

$$(\mathit{ske}_u, \mathit{type} := \mathsf{ske}, \mathit{arg} := (), \mathit{hnd}_u := \mathit{ske}_u^{\mathsf{hnd}}, \mathit{len} := 0);^4$$

$$\begin{aligned}&(\mathit{pke}_u, \mathit{type} := \mathsf{pke}, \mathit{arg} := (), \mathit{hnd}_{u_1} := \mathit{pke}_{u,u_1}^{\mathsf{hnd}}, \ldots, \mathit{hnd}_{u_m} := \mathit{pke}_{u,u_m}^{\mathsf{hnd}},\\ &\mathit{hnd}_{\mathsf{a}} := \mathit{pke}_{u,\mathsf{a}}^{\mathsf{hnd}}, \mathit{len} := \mathsf{pke\_len}^*(k)).\end{aligned}$$

Here ske_u and pke_u are two consecutive natural numbers. We omit the details of how the entries for user u are added by a command $\mathsf{gen\_enc\_keypair}$, followed by send commands for the public keys over authenticated channels.

6.3 Invariants

This section contains invariants of the system $\mathit{Sys}^{\mathsf{NS,id}}$, which are needed for the proof of Theorem 4.1. The first invariants, *correct nonce owner* and *unique nonce use*, are easily proved and essentially state that handles contained in a set $\mathit{Nonce}_{u,v}$ indeed point to entries of type nonce, and that no nonce is in two such sets. The next two invariants, *nonce secrecy* and *nonce-list secrecy*, deal with the secrecy of certain terms. They are mainly needed to prove the last invariant, *correct list owner*, which establishes who created certain terms.

- *Correct Nonce Owner.* For all $u \in \mathcal{H}, v \in \{1, \ldots, n\}$ and for all $x^{\mathsf{hnd}} \in \mathit{Nonce}_{u,v}$, it holds $D[\mathit{hnd}_u = x^{\mathsf{hnd}}] \neq \downarrow$ and $D[\mathit{hnd}_u = x^{\mathsf{hnd}}].\mathit{type} = \mathsf{nonce}$.

- *Unique Nonce Use.* For all $u, v \in \mathcal{H}$, all $w, w' \in \{1, \ldots, n\}$, and all $j \leq \mathit{size}$: If $D[j].\mathit{hnd}_u \in \mathit{Nonce}_{u,w}$ and $D[j].\mathit{hnd}_v \in \mathit{Nonce}_{v,w'}$, then $(u, w) = (v, w')$.

Nonce secrecy states that the nonces exchanged between honest users u and v remain secret from all other users and from the adversary. For the formalization, note that the handles to these nonces form the sets $\mathit{Nonce}_{u,v}$. The claim is that the other users and the adversary have no handles to such a nonce in the database D of $\mathsf{TH}_{\mathcal{H}}$:

- *Nonce Secrecy.* For all $u, v \in \mathcal{H}$ and for all $j \leq \mathit{size}$: If $D[j].\mathit{hnd}_u \in \mathit{Nonce}_{u,v}$ then $D[j].\mathit{hnd}_w = \downarrow$ for all $w \in (\mathcal{H} \cup \{\mathsf{a}\}) \setminus \{u, v\}$.

[4] Treating secret keys as being of length 0 is a technicality in the proof of [5] and will not matter in the sequel.

Similarly, the invariant *nonce-list secrecy* states that a list containing such a handle can only be known to u and v. Further, it states that the identity fields in such lists are correct. Moreover, if such a list is an argument of another entry, then this entry is an encryption with the public key of u or v.

- *Nonce-List Secrecy.* For all $u, v \in \mathcal{H}$ and for all $j \leq \mathit{size}$ with $D[j].\mathit{type} = \mathsf{list}$: Let $x_i^{\mathsf{ind}} := D[j].\mathit{arg}[i]$ for $i = 1, 2, 3$. If $D[x_i^{\mathsf{ind}}].\mathit{hnd}_u \in \mathit{Nonce}_{u,v}$ then

  a) $D[j].\mathit{hnd}_w = \downarrow$ for all $w \in (\mathcal{H} \cup \{\mathsf{a}\}) \setminus \{u, v\}$.
  b) if $D[x_{i+1}^{\mathsf{ind}}].\mathit{type} = \mathsf{data}$, then $D[x_{i+1}^{\mathsf{ind}}].\mathit{arg} = (u)$.
  c) for all $k \leq \mathit{size}$ it holds $j \in D[k].\mathit{arg}$ only if $D[k].\mathit{type} = \mathsf{enc}$ and $D[k].\mathit{arg}[1] \in \{\mathit{pke}_u, \mathit{pke}_v\}$.

The invariant *correct list owner* states that certain protocol messages can only be constructed by the "intended" users. For example, if a database entry is structured like the cleartext of a first protocol message, i.e., it is of type list, its first argument belongs to the set $\mathit{Nonce}_{u,v}$, and its second argument is a non-cryptographic construct (formally of type data) then it must have been created by user u. Similar statements exist for the second and third protocol message.

- *Correct List Owner.* For all $u, v \in \mathcal{H}$ and for all $j \leq \mathit{size}$ with $D[j].\mathit{type} = \mathsf{list}$: Let $x_i^{\mathsf{ind}} := D[j].\mathit{arg}[i]$ and $x_{i,u}^{\mathsf{hnd}} := D[x_i^{\mathsf{ind}}].\mathit{hnd}_u$ for $i = 1, 2$.

  a) If $x_{1,u}^{\mathsf{hnd}} \in \mathit{Nonce}_{u,v}$ and $D[x_2^{\mathsf{ind}}].\mathit{type} = \mathsf{data}$, then $D[j]$ was created by $\mathsf{M}_u^{\mathsf{NS}}$ in Step 1.4.
  b) If $D[x_1^{\mathsf{ind}}].\mathit{type} = \mathsf{nonce}$ and $x_{2,u}^{\mathsf{hnd}} \in \mathit{Nonce}_{u,v}$, then $D[j]$ was created by $\mathsf{M}_u^{\mathsf{NS}}$ in Step 2.12.
  c) If $x_{1,u}^{\mathsf{hnd}} \in \mathit{Nonce}_{u,v}$ and $x_2^{\mathsf{ind}} = \downarrow$, then $D[j]$ was created by $\mathsf{M}_v^{\mathsf{NS}}$ in Step 2.21.

This invariant is key for proceeding backwards in the protocol. For instance, if v terminates a protocol with user u, then v must have received a third protocol message. *Correct list owner* implies that this message has been generated by u. Now u only constructs such a message if it received a second protocol message. Applying the invariant two more times shows that u indeed started a protocol with v. The proof described below will take care of the details. Formally, the invariance of the above statements is captured in the following lemma.

Lemma 6.2 *The statements* correct nonce owner, unique nonce use, nonce secrecy, nonce-list secrecy, *and* correct list owner *are invariants of* $\mathit{Sys}^{\mathsf{NS,id}}$*, i.e., they hold at all times in all runs of* $\{\mathsf{M}_u^{\mathsf{NS}} \mid u \in \mathcal{H}\} \cup \{\mathsf{TH}_{\mathcal{H}}\}$ *for all* $\mathcal{H} \subseteq \{1, \ldots, n\}$. □

The proof is postponed to Appendix A.

6.4 Authenticity Proof

To increase readability, we partition the proof into several steps with explanations in between. Assume that $u, v \in \mathcal{H}$ and that $\mathsf{M}_v^{\mathsf{NS}}$ outputs (ok, u) to its user, i.e., a protocol between u and v has terminated successfully. We first show that this implies that $\mathsf{M}_v^{\mathsf{NS}}$ has received a message corresponding to the third protocol step, i.e., of the form that allows us to apply *correct list owner* to show that it was created by $\mathsf{M}_v^{\mathsf{NS}}$.

Proof. (Theorem 4.1) Assume that $\mathsf{M}_v^{\mathsf{NS}}$ outputs (ok, u) at $\mathsf{EA\_out}_v!$ for $u, v \in \mathcal{H}$ at time t_4. By definition of Algorithms 1 and 2, this can only happen if there was an input $(u, v, \mathsf{i}, m_v^{3\ \mathsf{hnd}})$ at $\mathsf{out}_v?$ at a time $t_3 < t_4$. Here and in the sequel we use the notation of Algorithm 2, but we distinguish the variables from its different executions by a superscript indicating the number of the (claimed) received protocol message, here ^{3}, and give handles an additional subscript for their owner, here v.

The execution of Algorithm 2 for this input must have given $l_v^{3\,\mathsf{hnd}} \neq \downarrow$ in Step 2.2, since it would otherwise abort by Convention 1 without creating an output. Let $l^{3\,\mathsf{ind}} := D[hnd_v = l_v^{3\,\mathsf{hnd}}].ind$. The algorithm further implies $D[l^{3\,\mathsf{ind}}].type = \mathsf{list}$. Let $x_i^{3\,\mathsf{ind}} := D[l^{3\,\mathsf{ind}}].arg[i]$ for $i = 1, 2$ at the time of Step 2.3. By definition of $\mathsf{list\_proj}$ and since the condition of Step 2.25 is true immediately after Step 2.3, we have

$$x_{1,v}^{3\,\mathsf{hnd}} = D[x_1^{3\,\mathsf{ind}}].hnd_v \text{ at time } t_4 \tag{1}$$

and

$$x_{1,v}^{3\,\mathsf{hnd}} \in Nonce_{v,u} \wedge x_2^{3\,\mathsf{ind}} = \downarrow \text{ at time } t_4, \tag{2}$$

since $x_{2,v}^{3\,\mathsf{hnd}} = \downarrow$ after Step 2.3 implies $x_2^{3\,\mathsf{ind}} = \downarrow$. ■

This first part of the proof shows that $\mathsf{M}_v^{\mathsf{NS}}$ has received a list corresponding to a third protocol message. Now we apply *correct list owner* to the list entry $D[l^{3\,\mathsf{ind}}]$ to show that this entry was created by $\mathsf{M}_u^{\mathsf{NS}}$. Then we show that $\mathsf{M}_u^{\mathsf{NS}}$ only generates such an entry if it has received a second protocol message. To show that this message contains a nonce from v, as needed for the next application of *correct list owner*, we exploit the fact that v accepts the same value as its nonce in the third message, which we know from the first part of the proof.

Proof. (cont'd with 3rd message) Equations (1) and (2) are the preconditions for Part c) of *correct list owner*. Hence the entry $D[l^{3\,\mathsf{ind}}]$ was created by $\mathsf{M}_u^{\mathsf{NS}}$ in Step 2.21.

This algorithm execution must have started with an input $(w, u, \mathsf{i}, m_u^{2\,\mathsf{hnd}})$ at $\mathsf{out}_u?$ at a time $t_2 < t_3$ with $w \neq u$. As above, we conclude $l_u^{2\,\mathsf{hnd}} \neq \downarrow$ in Step 2.2, set $l^{2\,\mathsf{ind}} := D[hnd_u = l_u^{2\,\mathsf{hnd}}].ind$, and obtain $D[l^{2\,\mathsf{ind}}].type = \mathsf{list}$. Let $x_i^{2\,\mathsf{ind}} := D[l^{2\,\mathsf{ind}}].arg[i]$ for $i = 1, 2, 3$ at the time of Step 2.3. As the condition of Step 2.16 is true immediately afterwards, we obtain $x_{i,u}^{2\,\mathsf{hnd}} \neq \downarrow$ for $i \in \{1, 2, 3\}$. The definition of $\mathsf{list\_proj}$ and Lemma 6.1 imply

$$x_{i,u}^{2\,\mathsf{hnd}} = D[x_i^{2\,\mathsf{ind}}].hnd_u \text{ for } i \in \{1, 2, 3\} \text{ at time } t_4. \tag{3}$$

Step 2.18 ensures $x_3^2 = w$ and $x_{1,u}^{2\,\mathsf{hnd}} \in Nonce_{u,w}$. Thus *correct nonce owner* implies

$$D[x_1^{2\,\mathsf{ind}}].type = \mathsf{nonce}. \tag{4}$$

Now we exploit that $\mathsf{M}_u^{\mathsf{NS}}$ creates the entry $D[l^{3\,\mathsf{ind}}]$ in Step 2.21 with the input $\mathsf{list}(x_{2,u}^{2\,\mathsf{hnd}})$. With the definitions of list and $\mathsf{list\_proj}$ this implies $x_2^{2\,\mathsf{ind}} = x_1^{3\,\mathsf{ind}}$. Thus Equations (1) and (2) imply

$$D[x_2^{2\,\mathsf{ind}}].hnd_v \in Nonce_{v,u} \text{ at time } t_4. \tag{5}$$

■

We have now shown that $\mathsf{M}_u^{\mathsf{NS}}$ has received a list corresponding to the second protocol message. We apply *correct list owner* to show that $\mathsf{M}_v^{\mathsf{NS}}$ created this list, and again we can show that this can only happen if $\mathsf{M}_v^{\mathsf{NS}}$ received a suitable first protocol message. Further, the next part of the proof shows that $w = v$ and thus $\mathsf{M}_u^{\mathsf{NS}}$ got the second protocol message from $\mathsf{M}_v^{\mathsf{NS}}$, which remained open in the previous proof part.

Proof. (cont'd with 2nd message) Equations (3) to (5) are the preconditions for Part b) of *correct list owner*. Thus the entry $D[l^{2\,\mathsf{ind}}]$ was created by $\mathsf{M}_v^{\mathsf{NS}}$ in Step 2.12. The construction of this entry in Steps

2.11 and 2.12 implies $x_3^2 = v$ and hence $w = v$ (using the definitions of store and $\mathsf{retrieve}$, and list and $\mathsf{list\_proj}$). With the results from before Equation (4) and Lemma 6.1 we therefore obtain

$$x_3^2 = v \wedge x_{1,u}^{2\ \mathsf{hnd}} \in Nonce_{u,v} \text{ at time } t_4. \tag{6}$$

The algorithm execution where $\mathsf{M}_v^{\mathsf{NS}}$ creates the entry $D[l^{2^{\mathsf{ind}}}]$ must have started with an input $(w', v, \mathsf{i}, m_v^{1\ \mathsf{hnd}})$ at $\mathsf{out}_v?$ at a time $t_1 < t_2$ with $w' \neq v$. As above, we conclude $l_v^{1\ \mathsf{hnd}} \neq \downarrow$ in Step 2.2, set $l^{1^{\mathsf{ind}}} := D[hnd_v = l_v^{1\ \mathsf{hnd}}].ind$, and obtain $D[l^{1^{\mathsf{ind}}}].type = \mathsf{list}$. Let $x_i^{1^{\mathsf{ind}}} := D[l^{1^{\mathsf{ind}}}].arg[i]$ for $i = 1, 2, 3$ at the time of Step 2.3. As the condition of Step 2.4 is true, we obtain $x_{i,v}^{1\ \mathsf{hnd}} \neq \downarrow$ for $i \in \{1, 2\}$. Then the definition of $\mathsf{list\_proj}$ and Lemma 6.1 yield

$$x_{i,v}^{1\ \mathsf{hnd}} = D[x_i^{1^{\mathsf{ind}}}].hnd_v \text{ for } i \in \{1,2\} \text{ at time } t_4. \tag{7}$$

When $\mathsf{M}_v^{\mathsf{NS}}$ creates the entry $D[l^{2^{\mathsf{ind}}}]$ in Step 2.12, its input is $\mathsf{list}(x_{1,v}^{1\ \mathsf{hnd}}, n_v^{\mathsf{hnd}}, v^{\mathsf{hnd}})$. This implies $x_1^{1^{\mathsf{ind}}} = x_1^{2^{\mathsf{ind}}}$ (as above). Thus Equations (3) and (6) imply

$$D[x_1^{1^{\mathsf{ind}}}].hnd_u \in Nonce_{u,v} \text{ at time } t_4. \tag{8}$$

The test in Step 2.6 ensures that $x_2^1 = w' \neq \downarrow$. This implies $D[x_2^{1^{\mathsf{ind}}}].type = \mathsf{data}$ by the definition of $\mathsf{retrieve}$, and therefore with Lemma 6.1,

$$D[x_2^{1^{\mathsf{ind}}}].type = \mathsf{data} \text{ at time } t_4. \tag{9}$$

■

We finally apply *correct list owner* again to show that $\mathsf{M}_u^{\mathsf{NS}}$ has generated this list corresponding to a first protocol message. We then show that this message must have been intended for user v, and thus user u has indeed started a protocol with user v.

Proof. (cont'd with 1st message) Equations (7) to (9) are the preconditions for Part a) of *correct list owner*. Thus the entry $D[l^{1^{\mathsf{ind}}}]$ was created by $\mathsf{M}_u^{\mathsf{NS}}$ in Step 1.4. The construction of this entry in Steps 1.3 and 1.4 implies $x_2^1 = u$ and hence $w' = u$.

The execution of Algorithm 1 must have started with an input $(\mathsf{new\_prot}, w'')$ at $\mathsf{EA\_in}_u?$ at a time $t_0 < t_1$. We have to show $w'' = v$. When $\mathsf{M}_u^{\mathsf{NS}}$ creates the entry $D[l^{1^{\mathsf{ind}}}]$ in Step 1.4, its input is $\mathsf{list}(n_u^{\mathsf{hnd}}, u^{\mathsf{hnd}})$ with $n_u^{\mathsf{hnd}} \neq \downarrow$. Hence the definition of $\mathsf{list\_proj}$ implies $D[x_1^{1^{\mathsf{ind}}}].hnd_u = n_u^{\mathsf{hnd}} \in Nonce_{u,w''}$. With Equation (8) and *unique nonce use* we conclude $w'' = v$.

In a nutshell, we have shown that for all times t_4 where $\mathsf{M}_v^{\mathsf{NS}}$ outputs (ok, u) at $\mathsf{EA\_out}_v!$, there exists a time $t_0 < t_4$ such that $\mathsf{M}_u^{\mathsf{NS}}$ receives an input $(\mathsf{new\_prot}, v)$ at $\mathsf{EA\_in}_u?$ at time t_0. This proves Theorem 4.1. ■

7 Conclusion

We have proven the Needham-Schroeder-Lowe public-key protocol in the real cryptographic setting via a deterministic, provably secure abstraction of a real cryptographic library. Together with composition and integrity preservation theorems from the underlying model, this library allowed us to perform the actual proof effort in a deterministic setting corresponding to a slightly extended Dolev-Yao model. This was the first example of such a proof. We hope that it paves the way for the actual use of automatic proof tools for this and many similar cryptographically faithful proofs of security protocols.

References

[1] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[2] M. Abadi and J. Jürjens. Formal eavesdropping and its computational interpretation. In *Proc. 4th International Symposium on Theoretical Aspects of Computer Software (TACS)*, pages 82–94, 2001.

[3] M. Abadi and P. Rogaway. Reconciling two views of cryptography: The computational soundness of formal encryption. In *Proc. 1st IFIP International Conference on Theoretical Computer Science*, volume 1872 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2000.

[4] M. Backes and C. Jacobi. Cryptographically sound and machine-assisted verification of security protocols. In *Proc. 20th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 2607 of *Lecture Notes in Computer Science*, pages 675–686. Springer, 2003.

[5] M. Backes, B. Pfitzmann, and M. Waidner. A universally composable cryptographic library. IACR Cryptology ePrint Archive 2003/015, Jan. 2003. `http://eprint.iacr.org/`.

[6] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption schemes. In *Advances in Cryptology: CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 26–45. Springer, 1998.

[7] M. Burrows, M. Abadi, and R. Needham. A logic for authentication. Technical Report 39, SRC DIGITAL, 1990.

[8] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proc. 42nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 136–145, 2001.

[9] R. Cramer and V. Shoup. Practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In *Advances in Cryptology: CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 13–25. Springer, 1998.

[10] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[11] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game – or – a completeness theorem for protocols with honest majority. In *Proc. 19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 218–229, 1987.

[12] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28:270–299, 1984.

[13] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–207, 1989.

[14] R. Kemmerer. Analyzing encryption protocols using formal verification techniques. *IEEE Journal on Selected Areas in Communications*, 7(4):448–457, 1989.

[15] G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–135, 1995.

[16] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proc. 2nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 1996.

[17] C. Meadows. Using narrowing in the analysis of key management protocols. In *Proc. 10th IEEE Symposium on Security & Privacy*, pages 138–147, 1989.

[18] C. Meadows. Analyzing the Needham-Schroeder public key protocol: A comparison of two approaches. In *Proc. 4th European Symposium on Research in Computer Security (ESORICS)*, volume 1146 of *Lecture Notes in Computer Science*, pages 351–364. Springer, 1996.

[19] J. K. Millen. The interrogator: A tool for cryptographic protocol security. In *Proc. 5th IEEE Symposium on Security & Privacy*, pages 134–141, 1984.

[20] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 12(21):993–999, 1978.

[21] L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Cryptology*, 6(1):85–128, 1998.

[22] B. Pfitzmann and M. Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *Proc. 22nd IEEE Symposium on Security & Privacy*, pages 184–200, 2001.

[23] C. Rackoff and D. R. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *Advances in Cryptology: CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 433–444. Springer, 1992.

[24] S. Schneider. Verifying authentication protocols with CSP. In *Proc. 10th IEEE Computer Security Foundations Workshop (CSFW)*, pages 3–17, 1997.

[25] P. Syverson. A new look at an old protocol. *Operation Systems Review*, 30(3):1–4, 1996.

[26] F. J. Thayer Fabrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Why is a security protocol correct? In *Proc. 19th IEEE Symposium on Security & Privacy*, pages 160–171, 1998.

A Proof of the Invariants

A.1 Correct Nonce Owner and Unique Nonce Use

We start with the proof of *correct nonce owner*.

Proof. (*Correct nonce owner*) Let $x^{\mathsf{hnd}} \in Nonce_{u,v}$ for $u \in \mathcal{H}$ and $v \in \{1, \ldots, n\}$. By construction, x^{hnd} has been added to $Nonce_{u,v}$ by $\mathsf{M}_u^{\mathsf{NS}}$ in Step 1.2 or Step 2.10. In both cases, x^{hnd} has been generated by the command $\mathsf{gen\_nonce}()$ at some time t, input at port $\mathsf{in}_u?$ of $\mathsf{TH}_{\mathcal{H}}$. Convention 1 implies $x^{\mathsf{hnd}} \neq \downarrow$, as $\mathsf{M}_u^{\mathsf{NS}}$ would abort otherwise and not add x^{hnd} to the set $Nonce_{u,v}$. The definition of $\mathsf{gen\_nonce}$ then implies $D[hnd_u = x^{\mathsf{hnd}}] \neq \downarrow$ and $D[hnd_u = x^{\mathsf{hnd}}].type = \mathsf{nonce}$ at time t. Because of Lemma 6.1 this also holds at all later times $t' > t$, which finishes the proof. ∎

The following proof of *unique nonce use* is quite similar.

Proof. (*Unique Nonce Use*) Assume for contradiction that both $D[j].hnd_u \in Nonce_{u,w}$ and $D[j].hnd_v \in Nonce_{v,w'}$ at some time t. Without loss of generality, let t be the first such time and let $D[j].hnd_v \notin Nonce_{v,w'}$ at time $t-1$. By construction, $D[j].hnd_v$ is thus added to $Nonce_{v,w'}$ at time t by Step 1.2 or Step 2.10. In both cases, $D[j].hnd_v$ has been generated by the command $\mathsf{gen\_nonce}()$ at time $t-1$. The definition of $\mathsf{gen\_nonce}$ implies that $D[j]$ is a new entry and $D[j].hnd_v$ its only handle at time $t-1$, and thus also at time t. With *correct nonce owner* this implies $u = v$. Further, $Nonce_{v,w'}$ is the only set into which the new handle $D[j].hnd_v$ is put at times $t-1$ and t. Thus also $w = w'$. This is a contradiction. ■

A.2 Correct List Owner

In the following subsections, we prove *correct list owner*, *nonce secrecy*, and *nonce-list secrecy* by induction. Hence assume that all three invariants hold at a particular time t in a run of the system, and we have to show that they still hold at time $t+1$.

Proof. (*Correct list owner*) Let $u, v \in \mathcal{H}$, $j \le size$ with $D[j].type = \mathsf{list}$. Let $x_i^{\mathsf{ind}} := D[j].arg[i]$ and $x_{i,u}^{\mathsf{hnd}} := D[x_i^{\mathsf{ind}}].hnd_u$ for $i = 1, 2$ and assume that $x_{i,u}^{\mathsf{hnd}} \in Nonce_{u,v}$ for $i = 1$ or $i = 2$ at time $t+1$.

The only possibilities to violate the invariant *correct list owner* are that (1) the entry $D[j]$ is created at time $t+1$ or that (2) the handle $D[j].hnd_u$ is created at time $t+1$ for an entry $D[j]$ that already exists at time t or that (3) the handle $x_{i,u}^{\mathsf{hnd}}$ is added to $Nonce_{u,v}$ at time $t+1$. In all other cases the invariant holds by the induction hypothesis and Lemma 6.1.

We start with the third case. Assume that $x_{i,u}^{\mathsf{hnd}}$ is added to $Nonce_{u,v}$ at time $t+1$. By construction, this only happens in a transition of $\mathsf{M}_u^{\mathsf{NS}}$ in Step 1.2 and Step 2.10. However, here the entry $D[x_i^{\mathsf{ind}}]$ has been generated by the command $\mathsf{gen\_nonce}$ input at $\mathsf{in}_u?$ at time t, hence x_i^{ind} cannot be contained as an argument of an entry $D[j]$ at time t. Formally, this corresponds to the fact that D is *well-formed*, i.e., index arguments of an entry are always smaller than the index of the entry itself; this has been shown in [5]. Since a transition of $\mathsf{M}_u^{\mathsf{NS}}$ does not modify entries in $\mathsf{TH}_{\mathcal{H}}$, this also holds at time $t+1$.

For proving the remaining two cases, assume that $D[j].hnd_u$ is created at time $t+1$ for an already existing entry $D[j]$ or that $D[j]$ is generated at time $t+1$. Because both can only happen in a transition of $\mathsf{TH}_{\mathcal{H}}$, this implies $x_{i,u}^{\mathsf{hnd}} \in Nonce_{u,v}$ already at time t, since transitions of $\mathsf{TH}_{\mathcal{H}}$ cannot modify the set $Nonce_{u,v}$. Because of $u, v \in \mathcal{H}$, *nonce secrecy* implies $D[x_i^{\mathsf{ind}}].hnd_w \neq \downarrow$ only if $w \in \{u, v\}$. Lists can only be constructed by the basic command list, which requires handles to all its elements. More precisely, if $w \in \mathcal{H} \cup \{\mathsf{a}\}$ creates an entry $D[j']$ with $D[j'].type = \mathsf{list}$ and $(x'_1, \ldots, x'_k) := D[j].arg$ at time $t+1$ then $D[x'_i].hnd_w \neq \downarrow$ for $i = 1, \ldots, k$ already at time t. Applied to the entry $D[j]$, this implies that either u or v have created the entry $D[j]$.

We now only have to show that the entry $D[j]$ has been created by u in the claimed steps. This can easily be seen by inspection of Algorithms 1 and 2. We only show it in detail for the first part of the invariant; it can be proven similarly for the remaining two parts.

Let $x_{1,u}^{\mathsf{hnd}} \in Nonce_{u,v}$ and $D[x_2^{\mathsf{ind}}].type = \mathsf{data}$. By inspection of Algorithms 1 and 2 and because $D[j].type = \mathsf{list}$, we see that the entry $D[j]$ must have been created by either $\mathsf{M}_u^{\mathsf{NS}}$ or $\mathsf{M}_v^{\mathsf{NS}}$ in Step 1.4. (The remaining list generation commands either only have one element, which implies $x_2^{\mathsf{ind}} = \downarrow$ and hence $D[x_2^{\mathsf{ind}}].type \neq \mathsf{data}$, or we have $D[x_2^{\mathsf{ind}}].type = \mathsf{nonce}$ by construction.) Now assume for contradiction that the entry $D[j]$ has been generated by $\mathsf{M}_v^{\mathsf{NS}}$. This implies that also the entry $D[x_1^{\mathsf{ind}}]$ has been newly generated by the command $\mathsf{gen\_nonce}$ input at $\mathsf{in}_v?$. However, only $\mathsf{M}_u^{\mathsf{NS}}$ can add a handle to the set $Nonce_{u,v}$ (it is the local state of $\mathsf{M}_u^{\mathsf{NS}}$), but every nonce that $\mathsf{M}_u^{\mathsf{NS}}$ adds to the set $Nonce_{u,v}$ is newly generated by the command $\mathsf{gen\_nonce}$ input by $\mathsf{M}_u^{\mathsf{NS}}$ by construction. This implies

$x_{1,u}^{\mathsf{hnd}} \not\in Nonce_{u,v}$ at all times, which yields a contradiction to $x_{1,u}^{\mathsf{hnd}} \in Nonce_{u,v}$ at time $t+1$. Hence $D[j]$ has been created by user u. ∎

A.3 Nonce Secrecy

Proof. (*Nonce secrecy*) Let $u, v \in \mathcal{H}$, $j \leq size$ with $D[j].hnd_u \in Nonce_{u,v}$, and $w \in (\mathcal{H} \cup \{\mathsf{a}\}) \setminus \{u, v\}$ be given. Because of *correct nonce owner*, we know that $D[j].type = \mathsf{nonce}$. The invariant could only be affected if (1) the handle $D[j].hnd_u$ is put into the set $Nonce_{u,v}$ at time $t+1$ or (2) if a handle for w is added to the entry $D[j]$ at time $t+1$.

For proving the first case, note that the set $Nonce_{u,v}$ is only extended by a handle n_u^{hnd} by $\mathsf{M}_u^{\mathsf{NS}}$ in Steps 1.2 and 2.10. In both cases, n_u^{hnd} has been generated by $\mathsf{TH}_{\mathcal{H}}$ at time t since the command gen_nonce was input at $\mathsf{in}_u?$ at time t. The definition of gen_nonce immediately implies that $D[j].hnd_w = \downarrow$ at time t if $w \neq u$. Moreover, this also holds at time $t+1$ since a transition of $\mathsf{M}_u^{\mathsf{NS}}$ does not modify handles in $\mathsf{TH}_{\mathcal{H}}$, which finishes the claim for this case.

For proving the second case, we only have to consider those commands that add handles for w to entries of type nonce. These are only the commands list_proj or adv_parse input at $\mathsf{in}_w?$, where adv_parse has to be applied to an entry of type list, since only entries of type list can have arguments which are indices to nonce entries. More precisely, if one of the commands violated the invariant there would exist an entry $D[i]$ at time t such that $D[i].type = \mathsf{list}$, $D[i].hnd_w \neq \downarrow$ and $j \in (x_1^{\mathsf{ind}}, \ldots, x_m^{\mathsf{ind}}) := D[i].arg$. However, both commands do not modify the set $Nonce_{u,v}$, hence we have $D[j].hnd_u \in Nonce_{u,v}$ already at time t. Now *nonce secrecy* yields $D[j].hnd_w = \downarrow$ at time t and hence also at all times $< t$ because of Lemma 6.1. This implies that the entry $D[i]$ must have been created by either u or v, since generating a list presupposes handles for all elements (cf. the previous proof). Assume without loss of generality that $D[i]$ has been generated by u. By inspection of Algorithms 1 and 2, this immediately implies $j \in (x_1^{\mathsf{ind}}, x_2^{\mathsf{ind}})$, since handles to nonces only occur as first or second element in a list generation by u. Because of $j \in D[i].arg[1, 2]$ and $D[j].hnd_u \in Nonce_{u,v}$ at time t, *nonce-list secrecy* for the entry $D[i]$ implies that $D[i].hnd_w = \downarrow$ at time t. This yields a contradiction. ∎

A.4 Nonce-List Secrecy

Proof. (*Nonce-list secrecy*) Let $u, v \in \mathcal{H}$, $j \leq size$ with $D[j].type = \mathsf{list}$. Let $x_i^{\mathsf{ind}} := D[j].arg[i]$ and $x_{i,u}^{\mathsf{hnd}} := D[x_i^{\mathsf{ind}}].hnd_u$ for $i = 1, 2$, and $w \in (\mathcal{H} \cup \{\mathsf{a}\}) \setminus \{u, v\}$. Let $x_{i,u}^{\mathsf{hnd}} \in Nonce_{u,v}$ for $i = 1$ or $i = 2$.

We first show that the invariant cannot be violated by adding the handle $x_{i,u}^{\mathsf{hnd}}$ to $Nonce_{u,v}$ at time $t+1$. This can only happen in a transition of $\mathsf{M}_u^{\mathsf{NS}}$ in Step 1.2 or 2.10. As shown in the proof of *correct list owner*, the entry $D[x_i^{\mathsf{ind}}]$ has been generated by $\mathsf{TH}_{\mathcal{H}}$ at time t. Since D is well-formed, this implies that $x_i^{\mathsf{ind}} \not\in D[j].arg$ for all entries $D[j]$ that already exist at time t. This also holds for all entries at time $t+1$, since the transition of $\mathsf{M}_u^{\mathsf{NS}}$ does not modify entries of $\mathsf{TH}_{\mathcal{H}}$. This yields a contradiction to $x_i^{\mathsf{ind}} = D[j].arg[i]$. Hence we now know that $x_{i,u}^{\mathsf{hnd}} \in Nonce_{u,v}$ already holds at time t.

Part a) of the invariant can only be affected if a handle for w is added to an entry $D[j]$ that already exists at time t. (Creation of $D[j]$ at time t with a handle for w is impossible as above because that presupposes handles to all arguments, in contradiction to *nonce secrecy*.) The only commands that add new handles for w to existing entries of type list are list_proj, decrypt, adv_parse, send_i, and adv_send_i applied to an entry $D[k]$ with $j \in D[k].arg$. *Nonce-list secrecy* for the entry $D[j]$ at time t then yields $D[k].type = \mathsf{enc}$. Thus the commands list_proj, send_i, and adv_send_i do not have to be considered any further. Moreover, *nonce-list secrecy* also yields $D[k].arg[1] \in \{pke_u, pke_v\}$. The secret keys of u and v are not known to $w \not\in \{u, v\}$, formally $D[hnd_w = ske_u^{\mathsf{hnd}}] = D[hnd_w = ske_v^{\mathsf{hnd}}] = \downarrow$;

this corresponds to the invariant *key secrecy* of [5]. Hence the command decrypt does not violate the invariant. Finally, the command adv_parse applied to an entry of type enc with unknown secret key also does not give a handle to the cleartext list, i.e., to $D[k].arg[2]$, but only outputs its length.

Part b) of the invariant can only be affected if the list entry $D[j]$ is created at time $t+1$. (By well-formedness, the argument entry $D[x_{i+1}^{\mathsf{ind}}]$ cannot be created after $D[j]$.) As in Part a), it can only be created by a party $w \in \{u, v\}$ because other parties have no handle to the nonce argument. Inspection of Algorithms 1 and 2 shows that this can only happen in Steps 1.4 and 2.12, because all other commands list have only one argument, while our preconditions imply $x_2^{\mathsf{ind}} \neq \downarrow$.

- If the creation is in Step 1.4, the preceding Step 1.2 implies $D[x_1^{\mathsf{ind}}].hnd_w \in Nonce_{w,w'}$ for some w' and Step 1.3 implies $D[x_2^{\mathsf{ind}}].type = \mathsf{data}$. Thus the preconditions of Part b) of the invariant can only hold for $i = 1$, and thus $D[x_1^{\mathsf{ind}}].hnd_u \in Nonce_{u,v}$. Now *unique nonce use* implies $u = w$. Thus Steps 1.3 and 1.4 yield $D[x_2^{\mathsf{ind}}].arg = (u)$.

- If the creation is in Step 2.12, the proof is analogous: The preceding steps 2.10 and 2.11 imply that the preconditions of Part b) of the invariant can only hold for $i = 2$. Then the precondition, Step 2.10, and *unique nonce use* imply $u = w$. Finally, Steps 2.11 and 2.12 yield $D[x_3^{\mathsf{ind}}].arg = (u)$.

Part c) of the invariant can only be violated if a new entry $D[k]$ is created at time $t+1$ with $j \in D[k].arg$ (by Lemma 6.1 and well-formedness). As $D[j]$ already exists at time t, *nonce-list secrecy* for $D[j]$ implies $D[j].hnd_w = \downarrow$ for $w \notin \{u, v\}$ at time t. We can easily see by inspection of the commands that the new entry $D[k]$ must have been created by one of the commands list and encrypt (or by sign, which creates a signature), since entries newly created by other commands cannot have arguments that are indices of entries of type list. Since all these commands entered at a port $\mathsf{in}_z?$ presuppose $D[j].hnd_z \neq \downarrow$, the entry $D[k]$ is created by $w \in \{u, v\}$ at time $t+1$. However, the only steps that can create an entry $D[k]$ with $j \in D[k].arg$ (with the properties demanded for the entry $D[j]$) are Steps 1.5, 2.13, and 2.22. In all these cases, we have $D[k].type = \mathsf{enc}$. Further, we have $D[k].arg[1] = pke_{w'}$ where w' denotes w's current believed partner. We have to show that $w' \in \{u, v\}$.

- Case 1: $D[k]$ is created in Step 1.5. By inspection of Algorithm 1, we see that the precondition of this proof can only be fulfilled for $i = 1$. Then $D[x_1^{\mathsf{ind}}].hnd_u \in Nonce_{u,v}$ and $D[x_1^{\mathsf{ind}}].hnd_w \in Nonce_{w,w'}$ and *unique nonce use* imply $w' = v$.

- Case 2: $D[k]$ is created in Step 2.13, and $i = 2$. Then $D[x_2^{\mathsf{ind}}].hnd_u \in Nonce_{u,v}$ and $D[x_2^{\mathsf{ind}}].hnd_w \in Nonce_{w,w'}$ and *unique nonce use* imply $w' = v$.

- Case 3: $D[k]$ is created in Step 2.13, and $i = 1$. This execution of Algorithm 2 must give $l^{\mathsf{hnd}} \neq \downarrow$ in Step 2.2, since it would otherwise abort by Convention 1. Let $l^{\mathsf{ind}} := D[hnd_w = l^{\mathsf{hnd}}].ind$. The algorithm further implies $D[l^{\mathsf{ind}}].type = \mathsf{list}$. Let $x_i^{0\,\mathsf{ind}} := D[l^{\mathsf{ind}}].arg[i]$ for $i = 1, 2, 3$ at the time of Step 2.3, and let $x_{i,w}^{0\;\mathsf{hnd}}$ be the handles obtained in Step 2.3. As the algorithm does not abort in Steps 2.5 and 2.7, we have $D[x_2^{0\,\mathsf{ind}}].type = \mathsf{data}$ and $D[x_2^{0\,\mathsf{ind}}].arg = (w')$.

  Further, the reuse of $x_{1,w}^{0\;\mathsf{hnd}}$ in Step 2.12 implies $x_1^{0\,\mathsf{ind}} = x_1^{\mathsf{ind}}$. Together with the precondition $D[x_1^{\mathsf{ind}}].hnd_u \in Nonce_{u,v}$, the entry $D[l^{\mathsf{ind}}]$ therefore fulfills the conditions of Part b) of *nonce-list secrecy* with $i = 1$. This implies $D[x_2^{0\,\mathsf{ind}}].arg = (u)$, and thus $w' = u$.

- Case 4: $D[k]$ is created in Step 2.22. With Step 2.21, this implies $x_2^{\mathsf{ind}} = \downarrow$ and thus $i = 1$. As in Case 3, this execution of Algorithm 2 must give $l^{\mathsf{hnd}} \neq \downarrow$ in Step 2.2, we set $l^{\mathsf{ind}} := D[hnd_w = l^{\mathsf{hnd}}].ind$, and we have $D[l^{\mathsf{ind}}].type = \mathsf{list}$. Let $x_i^{0\,\mathsf{ind}} := D[l^{\mathsf{ind}}].arg[i]$ for $i = 1, 2, 3$ at the time of Step 2.3, and let $x^{0\ \mathsf{hnd}}_{i,w}$ be the handles obtained in Step 2.3. As the algorithm does not abort in Steps 2.17 and 2.19, we have $D[x^{0\,\mathsf{ind}}_3].type = \mathsf{data}$ and $D[x^{0\,\mathsf{ind}}_3].arg = (w')$.

Further, the reuse of $x^{0\ \mathsf{hnd}}_{2,w}$ in Step 2.21 implies $x^{0\,\mathsf{ind}}_2 = x^{\mathsf{ind}}_1$. Together with the precondition $D[x^{\mathsf{ind}}_1].hnd_u \in Nonce_{u,v}$, the entry $D[l^{\mathsf{ind}}]$ therefore fulfills the condition of Part b) of *nonce-list secrecy* with $i = 2$. This implies $D[x^{0\,\mathsf{ind}}_3].arg = (u)$, and thus $w' = u$.

Hence in all cases we obtained $w' = u$, i.e., the list containing the nonce was indeed encrypted with the key of an honest participant. ■