To appear in *Proceedings of the 22nd IEEE Computer Security Foundations Symposium* (CSF 2009).

Authentication without Elision

Partially Specified Protocols, Associated Data, and Cryptographic Models Described by Code

Phillip Rogaway Till Stegers
Department of Computer Science
University of California at Davis
{*rogaway, stegers*}*@cs.ucdavis.edu*

Abstract

Specification documents for real-world authentication protocols typically mandate some aspects of a protocol's behavior but leave other features optional or undefined. In addition, real-world schemes often include parameter negotiations, authenticate associated data, and support a multiplicity of options. The cryptographic community has routinely elided such matters from our definitions, schemes, and proofs. We propose encompassing them by explicitly modeling the presence of unspecified protocol functionality. To demonstrate, we provide a new treatment for mutual authentication in the public-key setting, doing this in the computational cryptographic tradition. In our model, compactly described in pseudocode, a protocol core (PC) will call out to protocol details (PD), but, for defining security, such calls will be serviced by the adversary. Parties accepting an authentication exchange will output a string of associated data, the value of which may be determined by the PD calls. We illustrate the approach by re-proving security for the Needham-Schroeder-Lowe public-key protocol, but extended in a manner that would be typical were the mechanism embedded in a real-world standard.

Keywords: *authentication, associated data, Needham-Schroeder-Lowe protocol, provable security, security models.*

1. Introduction

In academic papers containing definitions and proofs, authentication protocols are usually quite simple schemes: a few messages exchanged, each computed by simple rules. Yet anyone who has examined a real-world authentication spec has seen a different kind of beast—schemes that don't seem simple after all. The Kerberos 5 spec is 138 pages, the TLS spec is 104 pages, and just one piece of the authentication underlying 802.11 security is 175 pages [20, 21, 38]. And it's not just the page counts that reflect the mechanisms' complexity; the real-world schemes are complex from many points of view. The specs often leave some protocol behaviors optional or intentionally unspecified. As an example, the TLS 1.2 spec has 61 SHOULDs, SHOULD NOTs, and MAYs [12, 20]. Because of all this complexity, we would claim that nobody has ever proven the cryptographic security of something like the "full" TLS 1.2; instead, one extracts some piece and goes from there [22, 36].

Certainly many people have observed this gulf between academic and real-world authentication. Consider the following comment from Krawczyk concerning a standardized key-exchange protocol, IKE [28], and how it is treated in the author's corresponding academic work:

> *A word of caution.* It is important to remark that all the protocols discussed in this paper are presented in their most basic form, showing only their cryptographic core. When used in practice it is essential to preserve this cryptographic core but also to take care of additional elements arising in actual settings. For example, if the protocol negotiates some security parameters or uses the protocol messages to send some additional information then the designers of such [a] full-fledge[d] protocol need to carefully expand the coverage of authentication also to these additional elements. [29, p. 408]

The cautionary note may sound routine; one expects to see some differences between an academic abstraction and a fully worked-out scheme. Abstraction is what we do. But, at another level, the proviso is actually quite alarming: if you prove something about the (self-identified) cryptographic core of an authentication protocol, does this actually prove *anything* about the full-fledged scheme? An approach in which security-relevant features of real-world protocols are routinely elided in corresponding analyses ought to raise foundational concerns.

Narrowing the gap. We suggest an approach aimed at bridging the gap between simple academic protocols and complex real-world schemes. The real-world scheme is to be regarded as comprising a *partially specified protocol* (PSP) plus additional *protocol details* (PD). The cryptographic core of a protocol will not be heuristically identified, but formally defined: it *is* the PSP. To prove a complex scheme secure, dissect it into a PSP and PD, then prove security for the PSP alone. Definitions for PSP security will be constructed so that this will prove security for the (PSP, PD) pair, and thus the real-world scheme.

The PSP will consist of a *key generator* (*KG*) and a *protocol core* (PC). Both will be specified in pseudocode. The PC is allowed to call out to the PD. But, for defining security, the *adversary itself* will service PD calls. This may sound a bit extreme: the PD isn't adversarial chosen, and regarding it as such might seem to give the adversary near-infinite power to disrupt a protocol's runs. But it does not, as the adversary will be allowed to see and influence only what's actually sent to the PD. The benefit to this pessimistic modeling of the PD is that a proof for a PSP becomes a proof for any full-fledged protocol that extends it by realizing the PD.

One might think that placing real-world protocol functionality within the PD boundary and then *ignoring* this functionality by replacing it with adversarially-controlled code would, in the end, be no different from ignoring the PD functionality in the first place. It is not so. First, it is a stronger requirement for something to be secure when it is adversarially controlled than when it is ignored. Second, formalizing what *is* the cryptographic core of a real-world protocol removes this step from the realm of informal, outside-of-the-model considerations. Third, the explicit presence of the PD will motivate an important definitional element: the inclusion of *associated data* (AD). Finally, allowing PD-determined AD opens the door to making security claims that reference the AD's contents, and therefore depend on what the PD does, without having to open it up.

What the PD might do. What, anyway, is all this alleged complexity in real-world protocols actually doing? A first purpose was hinted at in the earlier Krawczyk quote: when party A believes she has just had a conversation with B, she will typically hold some information, *ad*, that encodes *what* she believes that the two of them have agreed to. Formalizing this idea gives rise to a notion of authentication with associated data. While AD is known in the context of authenticated encryption [18, 40], it has never been formalized in the context of entity authentication, where it is equally applicable. So one important thing that the PD can do is to choose the AD. The contents of the AD may depend on the principal's security policy. Its value might encode things like the cryptographic primitives that were agreed to, perhaps through a multiple-flow negotiation.

The PD can do more. It can select *ancillary data* to augment each message flow. Such ancillary data may be used to provide additional functionality, like distributing a session key. The PD can also guide the PSP in selecting what long-lived key to use. It can help the PSP select which instance of a party to dispatch a message to. Or the PD might formulate an error message, request retransmission of a flow, or force a protocol to abort.

Defining MA. Our definitions are in the computational cryptographic tradition. We develop our ideas for the case of mutual authentication (MA), with associated data, in the public-key setting. Other goals and trust models can be handled in a similar vein. Our model refines earlier ones for entity authentication and key distribution (EA/KD) that go back to Bellare and Rogaway [8] and continue with works like [7, 10, 15, 23, 31, 44].

Our execution model will be described in pseudocode: four model-code procedures, about 30 lines of code, will define what the adversary can do and when it wins. Two more procedures, *KG* (key generator) and *PC* (protocol core), comprise the PSP. One last procedure, $\mathcal{A}$, is the adversary itself. The model-code procedures formalize the adversary's presumptive ability to send messages, look up public keys, and corrupt players. The protocol core *PC* may call a procedure *PD* for realizing protocol details, with such calls getting routed to the adversary $\mathcal{A}$.

Using a code-based descriptive language for cryptographic definitions not only enables the PSP idea, but may also prove helpful to overcome the length and ambiguity endemic to complex cryptographic notions described in English prose (we are thinking, for example, of UC [13]). We hope that writing cryptographic notions in code may make them more accessible, precise, and readily compared.

In revisiting notions for EA/KD we also take the opportunity to address a modeling concern. In previous work derived from [8], *instances*, not principals, are the conceptual entities to which one sends a message M. For example, the adversary might direct a query to a conceptualized entity Π_i^s representing instance s of party i. We find both philosophical and pragmatic difficulties with this choice. It implies that a party *must* tag each message with a session identifier, and that a principal *can't*, for example, refuse a create a session because too many are already in use. We prefer a model in which principals can create sessions at their discretion and route messages to sessions of their choice, potentially based on private state.

Security of NSL. After defining mutual authentication with associated data for a PSP, we exercise the definition by using it to prove security for the Needham-Schroeder-Lowe public-key protocol (NSL) [32, 33, 37]. Actually, our proof is not for NSL as it was first described, but for a variant of it that fits our syntax and adds in "hooks," in the form of PD calls and PD-interpreted strings in message flows, to incorporate additional functionality that a real-world NSL-based protocol might choose to add. Let NSL2 name our version of NSL. Using our notions, we will give a reduction to establish that NSL2 is secure as long as it is based on an encryption scheme that is CCA (chosen-ciphertext attack) secure. Our adversaries are dynamic (they may choose whom to corrupt based on information learned during the attack) and our security bound is concrete (as opposed to asymptotic). Our bound is worse than we expected: proven security drops in about $q^4/2^n$, where q is the number of adversary calls and n is the nonce length. We suspect this is

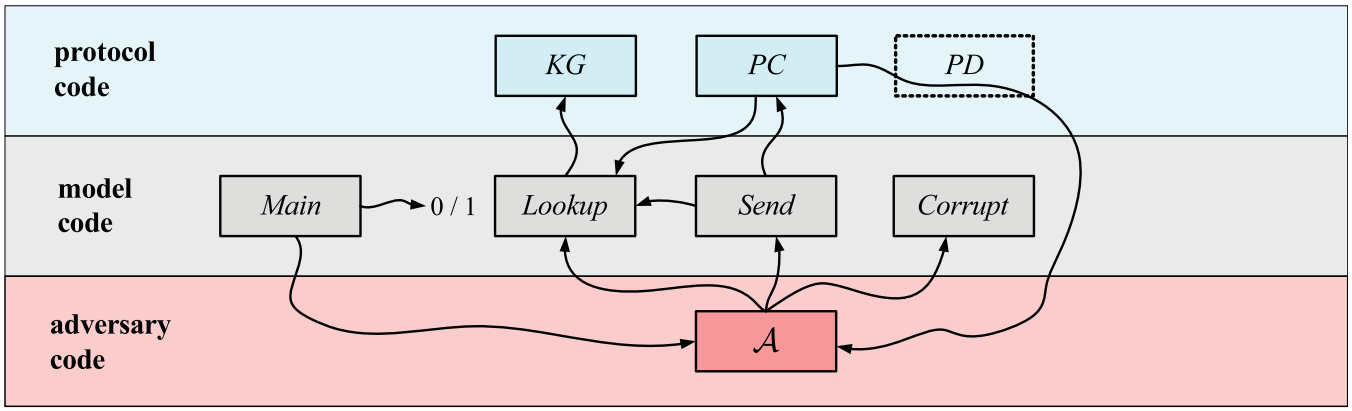

Figure 1. **Who may call whom.** A partially specified protocol $\Pi = (KG, PC)$ for MA is run with an adversary $\mathcal{A}$. Procedures are in boxes and an arc from one box to another means the first procedure can call the second.

just an artifact of the current proof but, in any case, it raises the question of how to design and analyze EA/KD schemes so as to not only get a strong notion of cryptographic security, but a tight bound as well.

Related work. We considered NSL not because it needs yet another proof of security, but because it has become somewhat of a "litmus test" for illustrating authentication methodologies. Within the computational-complexity tradition, the security of NSL was first established in independent papers by Backes and Pfitzmann [3] and by Warinschi [44]. The two works use radically different definitions and approaches. Backes and Pfitzmann employ a simulatability framework based on the cryptographic library of Backes, Pfitzmann, and Waidner [4]. While the initial result was a paper-and-pencil proof, subsequent work has come to encompass the use of automated theorem-provers and frameworks backed up by cryptographic soundness guarantees; see [14, 16, 34, 42] for work in this direction. Warinschi's paper [44] employs a definition that, like ours, is based on the simpler notions of Bellare and Rogaway [8]. Their prescriptive-style definition does not attempt to compare a real MA protocol to an idealized one; instead, one directly defines when the adversary does and doesn't win. Warinschi's result was stated asymptotically, assumed static adversaries, and defined partnering by matching conversations instead of through session identifiers (SIDs), but these differences are minor compared to the addition of PD-calls, associated data, and the absence of reified instances within the communications model.

Our execution model resembles a code-based game [9] and, in fact, we maintain that code-based games are a promising approach for giving clean definitions for many cryptographic goals. There is at least one difference between what we do and what code-based games have been asked to do: when an adversary interacts with a game, the adversary calls the game's procedures but the game doesn't turn around and call the adversary. For our definition, considering model code and protocol code as a game, the calls go both ways.

We have not provided a formal semantics for our code, a task we leave to future work. In recent work, Blanchet [11], Backes, Berg and Unruh [1] as well as Barthe, Grégoire and Zanella [5] each provide such semantics for some formulation of games. Namely, Blanchet represents games in a process calculus with probabilistic semantics with support for polynomial-time computable functions. His tool CryptoVerif heuristically finds transformations to simplify games for goals based on computational indistinguishability. Backes, Berg and Unruh define a typed lambda calculus with support for probabilistic functions (including oracles), user-defined types, and events (such as the canonical bad). They provide a semantics for their language in which programs evaluate to subprobability measures over program states and lists of events. Furthermore they define polynomial run time and formulate the Fundamental Lemma of Game-Playing Proofs [9] in their framework. They embed their type system in the Isabelle/HOL proof assistant to enable automated type inference. Barthe, Grégoire, and Zanella define a probabilistic imperative language pWHILE and show how a variety of bridging steps between games can be expressed as the transformation of pWHILE programs. They implement a tool, CertiCrypt, that verifies game-based proofs written in pWHILE and provides support for concrete-security resource accounting. Using CertiCrypt, they are able to find a tighter reduction for OAEP.

Hui and Lowe [27] share our motivation of bridging the gap between real-world protocols and their abstractions in a rigorous way, though they focus on simplifying the initially modeled protocol, whereas PSPs are useful for describing under-specified protocols as well. Working in a framework based on CSP [26], Hui and Lowe define transformations on protocols and their traces, giving sufficient conditions for these mappings to be *fault-preserving*, that is, to preserve the insecurity of protocols for certain definitions of mutual authentication and session-key privacy. They show that several mappings, such as omitting certain fields or extraneous

cryptographic operations, are fault-preserving. We are not aware of an analog to fault-preserving transformations in the computational cryptography tradition.

2. Defining Security of a PSP

Categories of code. Our execution model is described in code: the protocol and the adversary are regarded as procedures, and procedures are used to formalize how it all runs. Thus procedures are of three kinds: protocol code, adversary code, and model code. Figure 1 names the procedures of each kind and indicates which ones may call which others, while Figure 2 defines the model-code routines. In brief, the three types of code are as follows:

- **Protocol code** consists of two procedures: *KG*, the key generator, and *PC*, the protocol core. The latter routine defines how a principal responds when he receives a message. Procedure *PC* may call a procedure named *PD*, protocol details, but code for such a routine is not included in the execution model, as *PD* calls actually invoke the adversary $\mathcal{A}$. Taken together, *KG* and *PC* comprise the PSP $\Pi = (KG, PC)$. Protocol code may not employ static variables or otherwise maintain state; our model code does that on the protocol's behalf.
- **Adversary code** embodies functionality that we do not control yet hope to defeat. It consists of a single procedure, $\mathcal{A}$, which may call model-code procedures *Send*, *Lookup*, and *Corrupt*. We say that $\mathcal{A}$ has *oracles* for these functionalities. Adversary code is invoked by model code (to get things going) and by the protocol code (when it makes a *PD* call). When servicing a *PD* call, the adversary may not make *Send* or *Corrupt* calls. In order that a single conceptual entity services all adversary calls, the adversary may maintain its own global variables.
- **Model code** realizes the security definition. Our model code for mutual authentication with associated data in the public-key setting has procedures *Send*, *Lookup*, *Corrupt*, and *Main*. These procedures share global variables that are not visible to protocol or adversary code. Procedure *Main* calls the adversary $\mathcal{A}$ and then decides, based on the values of variables that have been set, if the adversary has won or lost. Our model code, specified in Figure 2, uses conventions that we will momentarily describe.

While the model does not dictate the internals of protocol or adversary code (routines *KG*, *PC*, and $\mathcal{A}$), we do insist that these procedures always terminate, regardless of coin tosses or responses to procedure calls, and that they do so within some bounded number of computational steps (this number of steps is associated to the procedure, of course).

Programming-language conventions. We do not formally define the programming language in which our code is written; it may be regarded as pseudocode. But the intended semantics is conventional enough that, with a bit of explanation, the execution of the model code (and, later, our protocol code for NSL2) should be clear.

To make our model code and protocol code compact, we adopt some specialized conventions. Any variable can take on the value $\diamond$, read as *invalid*, and all variables are silently initialized to this. In Boolean expressions, the symbol $\diamond$ is treated as FALSE, as is the integer 0, while nonzero integers are regarded as TRUE. Booleans are interchangeably regarded as bits. Our informal programming language supports variables recursively defined from types **array**, **boolean**, **integer**, **record**, **set**, or **string**. Strings are finite-length sequences over the alphabet $\Sigma = \{0,1\}$. The empty string is denoted ε. When $\langle X_1, \ldots, X_m \rangle$ appears as an R-value, as in Figure 2/line 11 (henceforth to be abbreviated as in 2:11), it means a string that encodes $(X_1, \ldots, X_m)$, including the type of each X_i. A *matching assignment* $\langle X_1, \ldots, X_m \rangle \leftarrow X$ (eg, 2:10) works like this. Parse the string X according to the reverse conventions used for encoding to get a tuple $(X'_1, \ldots, X'_{m'})$. If $m' = m$ and the type of each X_i is compatible with the type of X'_i then simultaneously set $X_i \leftarrow X'_i$ for all i; otherwise don't do anything. Arrays are associative arrays that map arbitrary values to values of some specified type. To declare an array, put brackets after the variable name, as in the declaration **string** *pk*[ ] at 2:1. Subscripts and brackets are interchangeably used for element-extraction from an array (eg, $pk_A = pk[A]$). Types are static and can often be inferred; when they cannot, they are strings. Variables are local unless they are declared globally at the beginning of the module in which they appear or are in a procedure's argument list. Procedure return types are inferred (usually they are strings). Variables are passed by value except where annotated by **var**, at the caller and called routine both, whence they are copy-in/copy-out. Macros can be defined using parentheses, which is the intent at 2:21–24. Further conventions will be specified as needed.

Defining adversarial advantage. Execution of $\Pi = (KG, PC)$ with $\mathcal{A}$ begins by calling *Main* and ends when *Main* terminates, returning a boolean indicating if the adversary has won. Runs will differ from one another because protocol and adversary code may be probabilistic. To realize this probabilism, the programming language includes a probabilistic-assignment statement $x \xleftarrow{\$} S$ that chooses an element x uniformly at random from a finite set S. The statement is not seen in Figure 2 simply because model-code routines for defining MA do not require probabilism.

We define $\mathbf{Adv}^{\mathrm{ma}}_{\Pi}(\mathcal{A})$ as the probability that $\mathcal{A}$ wins (*adversarywins* = TRUE) its execution with $\Pi = (KG, PC)$ using the model code of Figure 2. This quantity, the ad-

```
1   string pk[ ], sk[ ], party[ ], π[ ], sid[ ], pid[ ], ad[ ]
2   boolean corrupt[ ], role [ ], conf [ ]
3   integer n
4   procedure Send (string A, string M)
5   if corrupt_A then return ◇
6   n ← n + 1
7   party_n ← A
8   Lookup(A)
9   ⟨M′, decns⟩ ← PC (A, M, sk_A, var π_A)
10  ⟨sid_n, role_n, pid_n, ad_n, conf_n⟩ ← decns
11  return ⟨M′, decns⟩
12  procedure Lookup (string A)
13  if pk_A = ◇ then ⟨pk_A, sk_A⟩ ← KG(A)
14  return pk_A
15  procedure Corrupt (string A)
16  corrupt_A ← TRUE
17  return ⟨sk_A, π_A⟩
18  procedure Main
19  n ← 0
20  A(ε)
21  unfresh(sid) ← (∃1≤i≤n) (sid_i = sid) ∧ (corrupt_party_i ∨ corrupt_pid_i)
22  fresh(sid) ← ¬unfresh(sid)
23  partnered(i, j) ← sid_i = sid_j ≠ ◇ ∧ role_i ≠ role_j
24  detailsmatch(i, j) ← sid_i = sid_j ≠ ◇ ∧ role_i ≠ role_j ∧ pid_i = party_j ∧ pid_j = party_i ∧ ad_i = ad_j
25  proper ← (∀1≤i<j≤n) (sid_i = sid_j ≠ ◇ ∧ role_i = role_j) ⟹ (party_i = party_j ∧ pid_i = pid_j ∧ ad_i = ad_j)
26  validpartnering ← (∀1≤i,j≤n) partnered(i, j) ⟹ detailsmatch(i, j)
27  partnerconfirmation ← (∀1≤i≤n) (conf_i ∧ fresh(sid_i)) ⟹ (∃1≤j≤n) partnered(i, j)
28  adversarywins ← ¬proper ∨ ¬validpartnering ∨ ¬partnerconfirmation
29  return adversarywins
```

Figure 2. **Defining MA.** To run a protocol $\Pi = (KG, PC)$ with an adversary $\mathcal{A}$, procedure *Main* is called. The adversary $\mathcal{A}$ may call out to any of the procedures shown at lines 4–18. The protocol core PC, invoked at line 4, may call out to a procedure PD, which in fact invokes $\mathcal{A}$. It may also call *Lookup*.

versary's MA *advantage*, is a real number in $[0, 1]$, with 0 meaning that the adversary has done terribly and 1 meaning that it has done great. In the usual way, a protocol Π is regarded as "good" if $\mathbf{Adv}^{\mathrm{ma}}_{\Pi}(\mathcal{A})$ is "small" for any "reasonable" adversary $\mathcal{A}$. Concrete-security theorem statements, like ours, explicitly quantify how small is this advantage as a function of the computational resources spent by $\mathcal{A}$. The resources of interest include the number of procedure calls $\mathcal{A}$ makes to each model-code subroutine, as well as $\mathcal{A}$'s running time (which, by convention, includes the length of its code). For an asymptotic notion of security, a global integer constant k can be declared in the model code and exported to all routines. A protocol Π would be regarded as *secure* if $\varepsilon(k) = \mathbf{Adv}^{\mathrm{ma}}_{\Pi}(\mathcal{A})$ is negligible (meaning it is $k^{-\omega(1)}$) for every $\mathcal{A}$ that runs in time polynomial in k.

3. Explanation

At this point we have defined our notion of MA. Still, some English-language explanation may be useful to clarify what is going on in the model code of Figure 2, and why.

Explaining the code. At 2:1–2 we declare string-valued and boolean-valued associative arrays. Player A's public key and secret key are stored in pk_A ($= pk[A]$) and sk_A, and party A's state is kept in π_A. Recall that all values are initialized to $\diamond$, so, for example, initially $\pi_A = \diamond$ for all A.

At 2:19 procedure *Main* initializes a counter n for the number of *Send* calls to uncorrupted players. At 2:20 it calls the adversary $\mathcal{A}$, passing it the empty string as an indication to $\mathcal{A}$ that it has been called from *Main* (*PC* 's calls to $PD = \mathcal{A}$ should pass in something else). Any return value from $\mathcal{A}$ is discarded by 2:20.

As $\mathcal{A}$ executes, it may make calls to *Send*, *Lookup*, and *Corrupt*. The first is called to send a principal A some message M, the second to look up the public key of a principal, and the last to obtain a principal's internal state and secret key.

Starting with *Send*, we begin by disallowing queries to corrupted principals at 2:5. After recording the identity of the party that the adversary wants to send a message to at 2:7, we look up the public key for that party, which is done to make sure that the secret key for A is valid when we call *PC* at 2:9. There we pass *PC* the principal's name A (all identities are regarded as strings), the message M the adversary wants to send her, the party's secret key sk_A, and the party's state π_A. As protocol code may not maintain state across calls, the model code provides this service. The contents of π_A are uninterpreted by the model code, but the string π_A may be re-cast by *PC* to whatever type that procedure wants. The *PC* call must return and, when it does, its string-valued output is parsed and recorded in model-code globals at 2:10. The five components of *decns* encode: the session ID (SID), a string that names a session; the role, a symmetry-breaking device so that two entities can have the same SIDs but still be regarded as different instances; the partner ID (PID), a string that names a principal; the associated data (AD), a string that records a value that a party believes he shares with his communication partner;

and, finally, a bit indicating whether the instance believes to have received confirmation that his partner is there.

With a *Lookup* query the adversary can learn the public key for any player. The *PC* too has access to this functionality, which models an out-of-band mechanism to reliably map identities to public keys. With a *Corrupt* call the adversary can learn a player's secret key and state.

The most tricky part of Figure 2 are the macros and predicates at 2:21–29. Recall that 2:21–24 are macro definitions, to be used in the predicates that follow. Lines 2:21–22 say that an SID is *fresh* if any party that has that SID remains uncorrupted at the game's end, as does his partner. Line 2:23 captures the following notion. An *instance* is named by an SID and a role. Two SID/role pairs are regarded as *partnered* if they have the same SID and different roles (for example, one role may correspond to "initiator" and one to "responder"). Line 2:24 captures a stronger requirement than partnering: not only are the instances partnered, but they also agree in their AD values, and the PID of each party is the identity of the other. Line 2:25 captures the following, nearly "syntactic" requirement. Suppose that one and then a subsequent *Send* query are answered by the same instance. Then the identity of the party is not allowed to have changed, nor the PIDs, nor the ADs. In short, decisions stick: once an instance $(\mathit{sid}, \mathit{role})$ chooses a $(\mathit{pid}, \mathit{ad}, \mathit{conf})$, he is not allowed to change his mind. Moving on, line 2:26 captures the following idea. Whenever two instances are partnered—they have the same SID and opposite roles—then each party's PID should be the identity of the other, and they should have the same AD values, too. At 2:27 we formalize the following expectation. Suppose that an instance has confirmed his partner and has a fresh SID. Then there must exist an instance out there that it is partnered to. Finally, line 2:28 says that an adversary defeats the aim of MA if it managed to violate any of the expectations we have described.

Instances and dispatch. Under our model, messages are sent to principals, not to instances, and the model does not maintain separate state for each instance (of course a principal may choose to do so). Fundamentally, instances exist in our model only insofar as decisions coming out of principals (2:9–10) include an SID and role, and the predicates of *Main* interpret such pairs as though they named a communication endpoint. This set of choices are quite different from prior formalizations of EA/KD [7, 8, 10, 15, 35], where one addressed messages to instances and instances maintained their own, separate state. In prior definitions, peers were either *pre-specified* (one sends a message to a $\Pi^s_{i,j}$ that models instance s of party i wanting to authenticate party j) or *post-specified* (one sends a message to a Π^s_i that models instance s of party i willing to authenticate someone, their identity to be determined) [15, 35]. Either way, the SID s was specified with every message sent.

Our view is that *principals* receive messages, not instances. In that case a protocol must decide which instance to dispatch a message to. How should a principal decide? The selection *could* depend on an SID that is included in the message, but it could also be done based on information otherwise present in the flow. A natural choice is to let the PD choose the instance, based on information rich enough to inform the choice but not so rich as to obviate security. The issue there is that if message dispatch depends on *private* information known to the principal who receives a message, then one certainly can't quantify over all possible message-dispatch means, as a bad message-dispatch routine could divulge secret keys through the choice of which session a message is dispatched to. On the other hand, if message dispatch may not depend on any private information, then natural dispatch methods are impossible, including those which, as in NSL, need to decrypt an incoming flow to inspect a nonce contained within.

When messages are sent to instances rather than principals, protocols are unable, for example, to place limits on the number of open sessions, which implementations routinely do, or share other state. And session-identifiers effectively *must* be present in all flows, even if one could do without.

Extensions. It is possible to modify our definition of MA to handle a variety of trust models and goals. For example, to define the secrecy of session keys shared out in an authenticated key exchange (AKE) protocol, augment decision-vectors to include a session key *sn*, have *Main* flip a bit b, add in a *Reveal* query to obtain an already-distributed session key, add in a *Test* query to produce either a targeted session key *sn* or a sample from the distribution it is supposed to be drawn from, have the adversary output a bit b', and adjust *Main*'s predicates to say that the adversary wins when $b = b'$ and the adversary could not know b by trivial means. Switching to shared-key trust models is an easier change than this, as is the addition of a security parameter k, or the inclusion of schemes that employ globally-known parameters (eg, a randomly selected prime or elliptic-curve group).

Our model is not intended to be the strongest one feasible; we have opted for a relatively minimalist model instead. We do not require the concomitant distribution of a shared secret and so, correspondingly, there are no queries to reveal a session key or expire one. You can't reveal session state (no such notion is in the model). Our *Corrupt* queries provide a principal's state, but not his prior coins. No query allows the adversary to replace a principal's public key with a new one of his choice. We have in fact considered adding in this last capability, which models key-infrastructure attacks (where the adversary may, for example, obtain a valid certificate for an invalid public key). But we find that such queries would complicate our proof and necessitate a technical assumption on the encryption scheme's security beyond its being CCA.

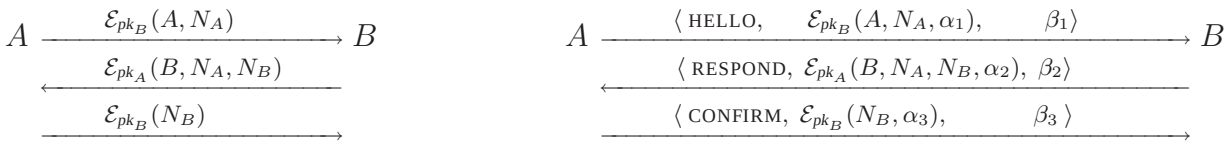

Figure 3. **Flow diagrams for NSL and NSL2.** For the latter, when B sends its RESPOND message it outputs decisions $\langle sid, role, pid, ad, conf\rangle$ of $sid = \langle A, B, C_1, C_2\rangle$, $role = 1$, $pid = A$, $ad = PD(\langle 0, A, B, \alpha_1, \alpha_2\rangle)$, and $conf =$ FALSE, where C_1 is the ciphertext within the HELLO message and C_2 is the ciphertext within the RESPOND. When A sends its CONFIRM message, it outputs decisions of $sid = \langle A, B, C_1, C_2\rangle$, $role = 0$, $pid = B$, $ad = PD(\langle 0, A, B, \alpha_1, \alpha_2\rangle)$, and $conf =$ TRUE. On receiving this message, B revises its *conf* flag to TRUE. Values $\alpha_1, \beta_1, \alpha_2, \beta_2, \alpha_3, \beta_3$ are all determined by *PD*-calls.

4. The NSL2 Protocol

Embellishing NSL. The Needham-Schroeder-Lowe public-key protocol [32, 37] (henceforth NSL) is usually summarized by the flow diagram given on the left-hand side of Figure 3. Such a diagram is useful and succinct, but it does of course leave much to the reader's imagination. Beyond this, we claim that the protocol suggested by the diagram elides cryptographically significant elements that one would expect to see in a real-world authentication scheme based on NSL. There are no parameter negotiations, public-key certificates, or error messages; there is no possibility of authenticating additional information by adding it within the scope of the (presumably non-malleable) encryption function $\mathcal{E}$; the scheme does not distribute a session key; and, as soon as one adds elements into the flows so that it will, cryptographic claims one has already made of the protocol will probably be lost, as there is no *a priori* reason to think that the MA property one shows for NSL will continue to hold on the embellished scheme.

To address these issues we define a version of NSL we'll call NSL2. The latter is, to a first approximation, NSL with generic "hooks" that are provided so that the scheme can be extended to include elements like those just mentioned. NSL2 is specified in Figure 4, doing this in a way that conforms to the syntax of Section 2. A corresponding flow diagram is given on the right-hand side of Figure 4, but it should be emphasized that such a figure remains no more than suggestive.

We call the α_i and β_i values that have been added to the flows *ancillary data*. We expect ancillary data to be chosen by the PD. For example, ancillary data might indicate the choice of an encryption scheme or key, say β_1 indicating A's choice for $\mathcal{E}_{pk_A}$ and β_2 indicating B's choice for $\mathcal{E}_{pk_B}$. It can be used to accomplish a key exchange intertwined with the MA, for instance, letting $\alpha_1 = g^a$ and $\alpha_2 = g^b$. Note that even though the β_i are not included within the scope of any cryptographic operator, this does not mean that they are security-irrelevant: the string β_1 can impact the PD's choice for α_2, which is part of what determines the AD.

Explaining the scheme. The partially specified protocol NSL2 depends on a public-key encryption scheme $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ and a nonce-length n; to be explicit, we will write NSL2$[\Pi, n]$. See Section 6 for a formalization of public-key encryption schemes and their security. Under that formulation, our encryption scheme is assumed to have the message space $\{0,1\}^*$ and, insofar as we omitted a security parameter in our formalization of MA, the key-generation algorithm $\mathcal{K}$ likewise does without.

To help with concision and readability, Figure 4 uses a programming-language construct that generalizes our earlier *matching assignment*. Suppose we write $\langle X_1, \ldots, X_m\rangle \leftarrow X$ where X is a string and each X_i is now a variable, an underlined variable, or an underlined constant. Then we parse X (using the reverse of whatever convention is used to encode tuples of variables into string) to get an m'-tuple of values $(X'_1, \ldots, X'_{m'})$. If $m = m'$ and each X_i has a type compatible with X'_i and $X_j = X'_j$ for each underlined X_j, then we simultaneously set $X_k \leftarrow X'_k$ for all non-underlined X_k and the matching-assignment statement itself returns TRUE. Otherwise, set $X_k \leftarrow \diamond$ for all non-underlined X_k and the matching-assignment statement returns FALSE. Either way, do not modify any underlined X_j. We allow a wildcard, $*$, to function as an anonymous non-underlined variable of the correct type. As an example of all this, at 4:15 we attempt to parse M into a triple consisting of the string constant RESPOND, a string C_2, and a third component we don't care about. If we succeed, the string C_2 will now be defined and we will continue evaluating the second half of the conjunct at 4:15 (conjunctions are assumed to evaluate from left-to-right), filling in *rcvdN* with the second component of the decryption of C_2. As an additional new convention, procedures are understood to return $\diamond$ in the absence of an explicit return statement.

After the type declaration at 4:1–3 we define the two routines exported from this module: *KG* and *PC*. The first is trivial, simply returning a string (which presumably encodes a pair of strings) returned by the encryption scheme's key-generation routine. In the declaration at 4:6, the final argument was a string when *PC* was called at 2:9, but it is now mapped into a value of type **PartyState**, with a $\diamond$-valued string taken to correspond to an array of $\diamond$-entries. We give *PD* every opportunity to interfere with protocol mechanics,

```
type InstanceState = record {boolean myRole, conf; string myN, yourN, sid, you, ad, lastrcvd, M′,
                                                    decns, α1, β1, C1, α2, β2, C2, α3, β3, C3 }
     PartyState = record {integer cnt; InstanceState instance[ ] }

procedure KG (string A)
return K ()

procedure PC (string A, string M, string sk, var PartyState π)
if π.cnt = ◇ then π.cnt ← 0;  M ← PD (⟨1, A, M, Sanitize (π)⟩)
⟨M′, decns′⟩ ← Dispatch (M, var π);  M′ ← PD (⟨2, A, M, M′, Sanitize (π)⟩)
return ⟨M′, decns′⟩

procedure Dispatch (string A, string M, string sk, PartyState π)
if ⟨START, ∗⟩ ← M then
     cur←++π.cnt;  return Start (A, M, sk, π, var π.instance[cur])
if ⟨HELLO, ∗⟩ ← M then
     cur←++π.cnt;  return Hello (A, M, π, var π.instance[cur])
if ⟨RESPOND, C2, ∗⟩ ← M and ⟨∗, rcvdN, ∗, ∗⟩ ← D(sk, C2) then
     cur ← min{cur ∈ {1, . . . , π.cnt} : π.instance[i].myN = rcvdN};  PD (⟨3, cur⟩)
     if cur = ◇ then return ◇ else return Respond (A, M, sk, π, var π.instance[cur])
if ⟨CONFIRM, C3, ∗⟩ ← M and ⟨rcvdN, ∗⟩ ← D(sk, C3) then
     cur ← min{cur ∈ {1, . . . , π.cnt} : π.instance[i].myN = rcvdN};  PD (⟨4, cur⟩)
     if cur = ◇ then return ◇ else return Confirm (A, M, sk, π, var π.instance[cur])

procedure Start (string A, string M, string sk, PartyState π, var InstanceState σ)
if ⟨START, σ.you⟩ ← M then
     σ.myN ←$ {0, 1}^n;  σ.myRole ← 0; σ.lastrcvd ← START;  ⟨σ.α1, σ.β1⟩ ← PD (⟨5, A, M, Sanitize (π)⟩)
     σ.C1 ← E(Lookup(σ.you), ⟨π.me, σ.myN, σ.α1⟩);  return ⟨⟨HELLO, σ.C1, σ.β1⟩, ◇⟩

procedure Hello (string A, string M, string sk, PartyState π, var InstanceState σ)
if ⟨HELLO, σ.C1, σ.β1⟩ ← M and ⟨σ.you, σ.yourN, σ.α1⟩ ← D(sk, σ.C1) then
     ⟨σ.α2, σ.β2⟩ ← PD (⟨6, A, M, Sanitize (π)⟩);  σ.myN ←$ {0, 1}^n
     σ.C2 ← E(Lookup(σ.you), ⟨π.me, σ.yourN, σ.myN, σ.α2⟩);  σ.M′ ← ⟨RESPOND, σ.C2, σ.β2⟩; σ.myRole ← 1
     σ.sid ← ⟨σ.you, π.me, σ.C1, σ.C2⟩;  σ.ad ← PD (⟨0, σ.you, A, σ.α1, σ.α2⟩);  σ.conf ← FALSE
     σ.lastrcvd ← HELLO;  σ.decns ← ⟨σ.sid, σ.myRole, σ.you, σ.ad, σ.conf⟩;  return ⟨σ.M′, σ.decns⟩

procedure Respond (string A, string M, string sk, PartyState π, var InstanceState σ)
if ⟨RESPOND, σ.C2, σ.β2⟩ ← M and σ.lastrcvd = START and ¬σ.conf
     and ⟨σ.you, σ.myN, σ.yourN, σ.α2⟩ ← D(sk, σ.C2) then
          ⟨σ.α3, σ.β3⟩ ← PD (⟨7, A, M, Sanitize (π)⟩)
          σ.C3 ← E(Lookup(σ.you), ⟨σ.yourN, σ.α3⟩)
          σ.M′ ← ⟨CONFIRM, σ.C3, σ.β3⟩;  σ.sid ← ⟨π.me, σ.you, σ.C1, σ.C2⟩;  σ.conf ← TRUE
          σ.ad ← PD (⟨0, A, σ.you, σ.α1, σ.α2⟩);  σ.decns ← ⟨σ.sid, σ.myRole, σ.you, σ.ad, σ.conf⟩
          return ⟨σ.M′, σ.decns⟩

procedure Confirm (string A, string M, string sk, PartyState π, var InstanceState σ)
if ⟨CONFIRM, σ.C3, σ.β3⟩ ← M and ¬σ.conf and σ.lastrcvd = HELLO
     and ⟨σ.myN, σ.α3⟩ ← D(sk, σ.C3) then
          σ.conf ← TRUE;  σ.decns ← ⟨σ.sid, σ.myRole, σ.you, σ.ad, σ.conf⟩;  return ⟨◇, σ.decns⟩

procedure Sanitize (PartyState π)
π′ ← π;   for i ∈ {1, . . . , cnt} do π′.instance[i].myN ← π′.instance[i].yourN ← ◇;  return π′
```

Figure 4. **Definition of NSL2.** The protocol depends on an encryption scheme $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ and a nonce length $n \geq 1$. For security of NSL2, one must assume that $PD(\langle 0, \cdots \rangle)$ calls are answered deterministically.

calling it at 4:7 to let it overwrite the message sent in, at 4:9 to let it overwrite the message going out. Such calls can be used for creating error messages, for example, or deleting communications not in conformance with a security policy. We continue to call *PD* at discretionary points throughout the code.

The *Dispatch* routine is responsible for sending the incoming message to the appropriate handler, sending that handler the protocol-maintained (as opposed to model-maintained) instance state for the appropriate instance. Instance identification is based on the nonces within messages or, when there is none, a new instance is created. Message are not annotated with anything "extra" to determine the session, as this can be gleaned from what was already in NSL flows.

Procedures *Start*, *Hello*, *Respond* and *Confirm* are called by *Dispatch* when a flow of that type is received. The first of these is used simply to request that the protocol begin. Procedure *Sanitize* is used to strip the state of the one thing that the adversary may not be given when *PD* is called: access to all nonces. Associated data is determined by the *PD* on the basis of the principal names, α_1, and α_2. For our security result on NSL2 we will need to assume that this particular *PD* call is deterministic, as we shall now explain.

5. Security of NSL2

We would like to claim that *no* reasonable adversary $\mathcal{A}$ can achieve good MA-advantage when attacking NSL2, but there is a technical issue that would make such a claim untrue. Since the AD is computed by a call to *PD*, the adversary could, for example, return a random bit at 4:29 and 4:37, in which case the pairs of AD values that "should" be equal will often disagree. To get around this, the *PD* calls that compute the AD ought to be deterministic. Formally, let's say that an adversary $\mathcal{A}$ is *valid* if $\mathcal{A}(\langle 0, \cdots \rangle)$ uses no probabilistic assignment and reads no global variable. This way $PD(\langle 0, \cdots \rangle)$ acts as a function. Of course calls of the form $PD(\varepsilon)$ or $PD(\langle i, \cdots \rangle)$ for $i \geq 1$ are not impacted; they can use probabilism or $\mathcal{A}$'s retained state.

We now quantify the MA-security of $\mathrm{NSL2}[\Pi, n]$ in terms of the CCA-security of the asymmetric encryption scheme Π.

Theorem 1. *Let $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be a public-key encryption scheme and let $n \in \mathbb{N}$ be a number. Then for any valid adversary $\mathcal{A}$ making $q < 2^{n/2}$ queries there exists an adversary $\mathcal{B}$ such that*

$$\mathbf{Adv}^{\mathrm{ma}}_{\mathrm{NSL2}[\Pi,n]}(\mathcal{A}) \leq \frac{8q^2}{1 - q^2/2^n} \cdot \mathbf{Adv}^{\mathrm{cca}}_{\Pi}(\mathcal{B}) + \frac{4q^4}{(2^n - q)(1 - q^2/2^n)} + \frac{q^2}{2^n} \quad (1)$$

where $\mathcal{B}$ makes at most $3q$ queries and runs in time comparable to $\mathcal{A}$'s.

More precisely, the phrase "time comparable to $\mathcal{A}$'s" means this. If $\mathcal{A}$ runs in time t and λ is the length of a longest string asked or answered by $\mathcal{A}$, then $\mathcal{B}$'s running time will be linear in $t \log q + q^2 \cdot \mathsf{Time}_{\Pi}(\lambda)$. Here $\mathsf{Time}_{\Pi}(\ell)$ denotes the maximum number of steps to compute $\mathcal{E}(pk, X)$, $\mathcal{D}(sk, X)$, and $\mathcal{K}(\,)$, where $|X| \leq \ell$.

The proof of Theorem 1 is given in Section 6. Let us here sketch some of its ideas.

Proof intuition. First we bound the probability that two nonces sampled in two distinct rounds collide. This is trivially seen to be rare, so, in the rest of the proof, we can then assume that there are no such nonce collisions.

The main part of the proof reduces the inability of $\mathcal{A}$ to create a "forgery" to the CCA-security of Π. Now it's not at all obvious what a forgery in NSL2 should *mean*; after all, we start with nothing but an encryption scheme, a kind of object that doesn't normally have an unforgeability guarantee, and an asymmetric one at that, so the adversary *can* come up with, say, a well-formed HELLO message. Fortunately, our code-based definitions provide a helpful language for what turns out to be a somewhat technical definition for what it means for the adversary to forge a ciphertext. Roughly said, we consider a HELLO message M a forgery if it is accepted by a principal B who believes it came from A, and in fact it does contain a nonce chosen by A for a session with B, but party A never sent M (so the adversary must have modified or created it). We define forgery notions for RESPOND and CONFIRM messages in a similar manner, this time requiring that the recipient's alleged nonce in the message match a nonce it previously chose for communication with the purported sender. Attacks on corrupted parties will not count as forgeries, as they can be trivially mounted.

Given an adversary $\mathcal{A}$ that forges NSL2 messages, we can build a CCA-distinguisher $\mathcal{B}$ against a nonstandard-but-equivalent formulation of CCA security. In this formulation, adversary $\mathcal{B}$ is given *two* encryption and decryption oracles, based on *two* independent key pairs. When started, our adversary $\mathcal{B}$ will pick two random principals A, B (this choice must be specified with care; our set of principals is infinite) and $\mathcal{B}$ will encrypt all messages between them using its left-or-right encryption oracles ("left-or-right" refers to the particular formalization of CCA security we will use). In the left-hand argument to the oracles, the adversary will submit the correct plaintext, as with NSL2. In the right-hand argument, it will pass in a plaintext with all nonces replaced by 0^n. (For parties other than A and B, distinguisher $\mathcal{B}$ will faithfully simulate all that goes on.) If all left plaintexts were encrypted, the specified behavior will perfectly simulate running NSL2 in our MA-model. If all right plaintexts were encrypted, then, since the nonces are excised from plaintexts and not surfaced through any other means, the messages exchanged between A and B are nearly independent of the

nonces. Thus, intuitively, even though the adversary may be likely to forge in the first setting, it cannot possibly do well in the second, which lets us distinguish among the encryption oracles we were given. If the adversary happens to corrupt A or B (which might in fact happen most of the time), then $\mathcal{B}$'s initial guess of A, B fails, and it gives up. The likelihood of this contributes a multiplicative loss of q^2 to the first two terms of bound (1).

The next portion of the proof shows that the absence of both forgeries and nonce collisions is enough to imply partner confirmation (the predicate at 2:25–26). Essentially, the proof for this backtracks the flows exchanged during a protocol, repeatedly using the condition that there are no forgeries to conclude the authenticity of received messages. In particular, we use the fact that $PD(\langle 0, \ldots \rangle)$ is deterministic and that both session participants have the same transcript of the session to conclude they computed the same AD. (One would expect that following this rather tedious portion of the proof would admit automation.)

Finally, the structure of our SID being $\langle A, B, C_1, C_2 \rangle$ together with the assumption that no nonces collide makes it fairly straightforward to show that variables *proper* and *validpartnering* will be set. Each of these is a claim about two executions of *PC* that return the same SID, therefore already agreeing about most of the first two flows, the session participants, and their roles.

The proof intuition we have just described omits many hidden complexities and details, yet we do not see the proof as containing very novel ideas or techniques. Perhaps the nicest thing about it is that writing the MA definition and the protocol in code lends itself to easy referencing and inference chains.

6. Proof of Theorem 1

Encryption schemes. Adapting the syntax of encryption schemes [6, 24, 39] so that their component algorithms can be called by code, we say that an (asymmetric) *encryption scheme* is a triple of procedures $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ such that $\mathcal{K}$ takes no input and returns a string $\langle pk, sk \rangle$; $\mathcal{E}$ takes a pair of strings pk, M and returns a string C; and $\mathcal{D}$ takes a pair of strings sk, C and returns a string M. We require that for every $\langle pk, sk \rangle$ returned by $\mathcal{K}$ and every string M we have $\mathcal{D}(sk, \mathcal{E}(pk, M)) = M$. This holds for all outcomes of probabilistic assignments that may appear in those procedures.

An adversary $\mathcal{A}$ attacking the CCA security of an encryption scheme $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ may call procedures *Encrypt* and *Decrypt* that work like this. Before the adversary is run, we set $b \xleftarrow{\$} \{0,1\}$, $\langle pk, sk \rangle \leftarrow \mathcal{K}$, and then call $\mathcal{A}(pk)$. Procedure *Encrypt* takes strings X_0 and X_1 and returns $\mathcal{E}(pk, X_b)$ if $|X_0| = |X_1|$, and $\diamond$ otherwise. Procedure *Decrypt* takes a string Y and returns $\mathcal{D}(sk, Y)$, unless Y was returned by an earlier call to *Encrypt*, in which case it returns $\diamond$. When $\mathcal{A}$ terminates, returning a bit b', it *wins* if $b = b'$. The *advantage* of $\mathcal{A}$ in attacking the CCA property of Π is defined as $\mathbf{Adv}^{\mathrm{cca}}_{\Pi}(\mathcal{A}) = \Pr[\mathcal{A} \textit{ wins}]$.

Proof strategy. Let G^{ma} be the game defined by running the model code for MA given in Figure 2 together with NSL2 as defined by Figure 4. Let $\mathcal{A}$ be a valid adversary as in Theorem 1, and denote $\mathcal{A}$ running in G^{ma} by $\mathcal{A}^{\mathrm{ma}}$. Let nc be the event that the nonces chosen in 4:23 or 4:27 during two different executions of *PC* coincide. For the variables *proper*, *validpartnering* and *partnerconfirmation* in lines 2:25–27, let proper, vp, and pc be events occurring if and only if the corresponding variable is TRUE when the model code terminates. Similarly, for integers i, j, let $\mathsf{partnered}(i, j)$ denote the value of the macro partnered(i, j) at the end of execution.

Suppose for the moment that $\mathsf{forgery}$ is some event. By the definition of $\mathbf{Adv}^{\mathrm{ma}}_{P}(\mathcal{A})$ and repeated application of the inequality $\Pr[\mathsf{E}_1 \vee \mathsf{E}_2] \leq \Pr[\mathsf{E}_1] + \Pr[\mathsf{E}_2 \mid \neg \mathsf{E}_1]$ we have:

$$
\begin{aligned}
&\mathbf{Adv}^{\mathrm{ma}}_{\mathrm{NSL2}[\Pi,n]}(\mathcal{A}) \\
&\quad \leq \quad \Pr[\,\mathcal{A}^{\mathrm{ma}} \colon \neg\mathsf{proper} \vee \neg\mathsf{vp} \vee \neg\mathsf{pc}\,] \\
&\quad \leq \quad \Pr[\,\mathcal{A}^{\mathrm{ma}} \colon \mathsf{nc}\,] \\
&\qquad + \Pr[\,\mathcal{A}^{\mathrm{ma}} \colon \neg\mathsf{proper} \vee \mathsf{forgery} \vee \neg\mathsf{vp} \vee \neg\mathsf{pc} \mid \neg\mathsf{nc}\,] \\
&\quad \leq \quad \Pr[\,\mathcal{A}^{\mathrm{ma}} \colon \mathsf{nc}\,] + \Pr[\,\mathcal{A}^{\mathrm{ma}} \colon \neg\mathsf{proper} \mid \neg\mathsf{nc}\,] \\
&\qquad + \Pr[\,\mathcal{A}^{\mathrm{ma}} \colon \mathsf{forgery} \mid \neg\mathsf{nc}\,] \\
&\qquad + \Pr[\,\mathcal{A}^{\mathrm{ma}} \colon \neg\mathsf{vp} \mid \mathsf{proper} \wedge \neg\mathsf{forgery} \wedge \neg\mathsf{nc}\,] \\
&\qquad + \Pr[\,\mathcal{A}^{\mathrm{ma}} \colon \neg\mathsf{pc} \mid \mathsf{proper} \wedge \neg\mathsf{forgery} \wedge \neg\mathsf{nc}\,]
\end{aligned}
$$

We continue by bounding the last probabilities above in a sequence of lemmas (and defining $\mathsf{forgery}$). The claim of Theorem 1 then follows by combining the results.

Lemma 2. *Let $\mathcal{A}$ be an adversary that makes q calls. Then* $\Pr[\,\mathcal{A}^{\mathrm{ma}} \colon \mathsf{nc}\,] \leq q^2/2^{n+1}$.

Proof: We notice that $\leq q$ nonces are sampled independently from $\Sigma^n = \{0,1\}^n$ and apply the birthday bound. □

Consider an execution of $\mathcal{A}^{\mathrm{ma}}$. If v is a variable in the pseudocode of NSL2, then denote by v^i the last value v holds in the i-th execution of *PC*. Further, let $corrupt_i$ be TRUE iff $party_i$ ever gets corrupted (a party A is corrupted at run j iff $corrupt_A$ is TRUE).

Lemma 3. *For any valid adversary $\mathcal{A}$ we have that* $\Pr[\,\mathcal{A}^{\mathrm{ma}} \colon \neg\mathsf{proper} \mid \neg\mathsf{nc}\,] = 0$.

Proof: To show proper holds, suppose that $i \leq j$, $sid_i = sid_j \neq \diamond$, and $role_i = role_j$. Since the identities of both initiator and responder are included in the session ID in 4:29, 4:36, it is clear that $party_i = party_j$ and $pid_i = pid_j$. Moreover, $\sigma.C_2^i = \sigma.C_2^j$. Without loss of generality we can assume that the field $\sigma.C_2$ was assigned in both call i and j. If $role_i = role_j = 0$, then we conclude from 4:33 $\sigma.myN^i = \sigma.myN^j$. If $role_i = role_j = 1$, then by $\sigma.C_2^i = \sigma.C_2^j$ and the injectivity of encryption $\sigma.myN^i = \sigma.myN^j$ as well. By $\neg\mathsf{nc}$

both of these nonces were chosen during the same execution of *PC*, and so $cur^i = cur^j$. Every field $\pi.instance[l].ad$ is written at most once; since $sid_i \neq \diamond$ we conclude by 4:29 and 4:36, respectively, that $\pi.instance[cur].ad$ must have been set by the i-th run, so $ad_i = ad_j$. □

Message authenticity. We define existential forgeries of messages in the context of NSL2 . Let forgery be the event that there exists a run i of *PC* such that $\mathsf{forgery}(i) = \mathsf{forgehello}(i) \vee \mathsf{forgerespond}(i) \vee \mathsf{forgeconfirm}(i)$ occurs, where these events are defined as follows:

1) $\mathsf{forgehello}(i)$: $role_i = 1$, and in the i-th *Send* query, $\mathcal{A}$ sends a message $M = \langle \text{HELLO}, C, \beta_1 \rangle$, where $\mathcal{D}_{sk^i}(C) = \langle \sigma.you^i, \sigma.myN^j, \alpha_1 \rangle$ for some $j < i$ such that $party_j = \sigma.you^i$, $\sigma.you^j = party_i$, $\alpha_1, \beta_1 \in \{0,1\}^*$, and $corrupt_i = corrupt_j = \text{FALSE}$, but M was never the response to a query $Send(\sigma.you^i, M')$;
2) $\mathsf{forgerespond}(i)$: $role_i = 0$, and the i-th *Send* is $Send(party_i, M)$, where $M = \langle \text{RESPOND}, C, \beta_2 \rangle$, and $\mathcal{D}_{sk^i}(C) = \langle \sigma.you^i, \sigma.myN^j, N, \alpha_2 \rangle$ for some $j < i$ such that $party_j = party_i$, $\sigma.you^j = \sigma.you^i$, $N \in \{0,1\}^n$, $\alpha_2, \beta_2 \in \{0,1\}^*$, and $corrupt_i = corrupt_j = \text{FALSE}$, but M was not replied in any *Send* query $k < i$ such that $party_k = \sigma.you^i$ and $\sigma.you^k = party_i$;
3) $\mathsf{forgeconfirm}(i)$: $role_i = 1$, and the i-th *Send* is $Send(party_i, M)$, where $M = \langle \text{CONFIRM}, C, \beta_3 \rangle$ and $\mathcal{D}_{sk^i}(C) = \langle \sigma.myN^j, \alpha_3 \rangle$ for some $j < i$ such that $party_j = party_i$, $\sigma.you^j = \sigma.you^i$, $\alpha_3, \beta_3 \in \{0,1\}^*$, and $corrupt_i = corrupt_j = \text{FALSE}$, but M was not replied in any run $k < i$ such that $party_k = \sigma.you^i$ and $\sigma.you^k = party_i$.

Lemma 4. *Let $\mathcal{A}$ be a valid adversary that makes q calls, where $q \notin \{2^{(n+1)/2}, 2^n\}$, and runs in time t. There exists a CCA adversary $\mathcal{B}$ such that*

$$\Pr[\mathcal{A}^{\mathrm{ma}}\colon \mathsf{forgery} \mid \neg\mathsf{nc}] \leq \frac{8q^2}{1 - q^2/2^{n+1}} \cdot \mathbf{Adv}^{\mathrm{cca}}_{\Pi}(\mathcal{B}) + \frac{4q^4}{2^n - q} \cdot \frac{1}{1 - q^2/2^{n+1}} .$$

Adversary $\mathcal{B}$ asks at most $3q$ queries and runs in time linear in $t \log q + q^2 \cdot \mathsf{Time}_{\Pi}(\lambda)$, where $\mathsf{Time}_{\Pi}(c)$ denotes the maximum number of steps to compute $\mathcal{E}(pk, X)$, $\mathcal{D}(sk, X)$, and $\mathcal{K}(\,)$ over all strings X with $|X| \leq c$.

Proof: We define an adversary $\mathcal{B}$ that runs $\mathcal{A}$ in a modified version of the game formed by the model code in Figure 2 and $\mathrm{NSL2}[\Pi, n]$ described in Figure 4. We assume that, for a fixed bit b, $\mathcal{B}$ has in fact access to *two* encryption oracles $Encrypt^1_b$, $Encrypt^2_b$ and corresponding decryption oracles $Decrypt^1$ and $Decrypt^2$ that use key pairs $(pk_1, sk_1), (pk_2, sk_1) \xleftarrow{\$} \mathcal{K}(\,)$, respectively. Denoting the game of running $\mathcal{B}$ with these oracles by $\mathcal{B}^b$, the advantage of $\mathcal{B}$ against Π in this setting is $\mathbf{Adv}^{\text{2-cca}}_{\Pi}(\mathcal{B}) = \Pr[\mathcal{B}^1 \Rightarrow 1] - \Pr[\mathcal{B}^0 \Rightarrow 1]$. A standard reduction shows that that for every adversary $\mathcal{B}$ there is a $\mathcal{B}'$ such that $\mathbf{Adv}^{\text{2-cca}}_{\Pi}(\mathcal{B}) \leq 2 \cdot \mathbf{Adv}^{\mathrm{cca}}_{\Pi}(\mathcal{B}')$ and $\mathcal{B}'$ runs in time linear in the running time of $\mathcal{B}$. Call this notion 2-CCA.

Let $p \leq 2q$ be the number of distinct parties mentioned in *Lookup* queries. First, the simulator $\mathcal{B}$ chooses $r, s \xleftarrow{\$} \{1, \ldots, p\}$ and sets $\mathcal{C} \leftarrow \emptyset$. Let A_1 and A_2 be the r-th and s-th (distinct) principal mentioned in a *Lookup* query, respectively. The simulator answers each query $Lookup(A_1)$ with pk_1 and each query $Lookup(A_2)$ with pk_2. (If $A_1 = A_2$ it also returns pk_1. For simplicity, we assume $A_1 \neq A_2$ in this description.) Adversary $\mathcal{B}$ hopes that $\mathcal{A}$ attempts to forge a message from A_1 to A_2 or from A_2 to A_1. To encrypt messages sent between A_1 and A_2, the simulator queries its corresponding oracle $Encrypt^i_b(\cdot,\cdot)$. For each plaintext X_1 that, say, A_1 would want to encrypt for A_2, $\mathcal{B}$ forms a plaintext X_0 by taking X_1 and replacing each nonce in it by 0^n. It then includes the ciphertext $Y \leftarrow Encrypt^2_b(X_0, X_1)$ in the outgoing message for A_2 and adds (A_2, X, Y) to $\mathcal{C}$. If $b = 1$ then Y has the same distribution as in G^{ma}, whereas if $b = 0$ then, intuitively, Y is (almost) independent of the choice of nonces. (We will make the latter precise.) For messages from A_2 to A_1 the simulator proceeds analogously, making use of its oracle $Encrypt^1_b(\cdot,\cdot)$. For all messages sent from or to a party $B \neq A_1, A_2$, the simulator uses Π's encryption and decryption algorithms as in Figure 4. Whenever A_i needs to decrypt, $\mathcal{B}$ checks if the ciphertext was the result of a query $Encrypt^i_b(X_0, X_1)$ by searching for a tuple $(A_i, X_1, Y) \in \mathcal{C}$. If so, it assumes X_1 is the plaintext. If not, it makes a query $Decrypt^i(Y)$ to obtain the plaintext. If A_1 or A_2 gets corrupted, $\mathcal{B}$ aborts and outputs 0, effectively giving up since it doesn't know their secret keys. All other *Corrupt* queries are answered as in Figure 4.

Let nc be defined as in G^{ma}, and let $\mathsf{forgery}^*(i)$ be the event that $\mathsf{forgery}(i)$ happens and $\{A^i_1, \sigma.you^i\} = \{A_1, A_2\}$, where A_1 and A_2 are the principals selected by $\mathcal{B}$ based on its random choices r, s. Define $\mathsf{forgery}^*$ as the union of all events $\mathsf{forgery}^*(i)$ where $1 \leq i \leq q$. The simulator $\mathcal{B}$ outputs 1 if $\neg\mathsf{forgery}^* \wedge \neg\mathsf{nc}$, and 0 otherwise. We have

$$\begin{aligned} \mathbf{Adv}^{\text{2-cca}}_{\Pi}(\mathcal{B}) &= \Pr[\mathcal{B}^1 \Rightarrow 1] - \Pr[\mathcal{B}^0 \Rightarrow 1] \\ &= \Pr[\mathcal{B}^1\colon \mathsf{forgery}^* \mid \neg\mathsf{nc}] \cdot \Pr[\mathcal{B}^0\colon \neg\mathsf{nc}] \\ &\quad - \Pr[\mathcal{B}^0\colon \mathsf{forgery}^* \mid \neg\mathsf{nc}] \cdot \Pr[\mathcal{B}^0\colon \neg\mathsf{nc}] . \end{aligned} \quad (2)$$

Note that the answers given to $\mathcal{A}$ in the games $\mathcal{B}^1$ and $\mathcal{A}^{\mathrm{ma}}$ are identically distributed, and the behavior of $\mathcal{A}$ in $\mathcal{B}^1$ is independent of the choice of i, j. In particular, the probability of the event $\{party_i, \sigma.you^i\} = \{A_1, A_2\}$ occurring conditioned on $\mathsf{forgery}(i) \wedge \neg\mathsf{nc}$ is $1/p^2$, thus $\Pr[\mathcal{B}^1\colon \mathsf{forgery}^*(i) \mid \neg\mathsf{nc}] = \Pr[\mathcal{B}^1\colon \mathsf{forgery}(i) \mid \neg\mathsf{nc}]/p^2$. Applying the union bound yields $\Pr[\mathcal{B}^1\colon \mathsf{forgery}^* \mid \neg\mathsf{nc}] \geq 1/p^2 \cdot \Pr[\mathcal{B}^1\colon \mathsf{forgery} \mid \neg\mathsf{nc}]$. The birthday bound gives the inequality $\Pr[\mathcal{B}^1\colon \neg\mathsf{nc}] \geq 1 - q^2/2^{n+1}$. To bound the remaining term, we will soon show that

$$\Pr[\mathcal{B}^0\colon \mathsf{forgery}^* \mid \neg\mathsf{nc}] \leq q^2/(2^n - q).$$

Rearranging equation (2) and applying all bounds we just derived yields:

$$\Pr[\,\mathcal{A}^{\mathrm{ma}}\colon \mathsf{forgery} \mid \neg\mathsf{nc}\,] \le \frac{p^2}{1-q^2/2^{n+1}} \cdot \mathbf{Adv}^{\text{2-cca}}_{\Pi}(\mathcal{B}) + \frac{p^2q^2}{2^n-q} \cdot \frac{1}{1-q^2/2^{n+1}}\,.$$

By the reduction from 2-CCA to CCA, there exists an adversary $\mathcal{B}'$ such that $\mathbf{Adv}^{\mathrm{cca}}_{\Pi}(\mathcal{B}')$ is at most $2\cdot\mathbf{Adv}^{\text{2-cca}}_{\Pi}(\mathcal{B})$. Since $\mathcal{B}$ asks at most $3q$ queries, $\mathcal{B}'$ makes at most $3q$ queries as well. The validity of of $\mathsf{forgery}^* \wedge \neg\mathsf{nc}$ can be checked in time proportional to $t \log q$, and $\mathcal{B}$ spends at most $q^2 \cdot \mathsf{Time}_{\Pi}(\lambda)$ steps in public-key operations. Thus both $\mathcal{B}$ and $\mathcal{B}'$ run in time $t' \le cq^2\mathsf{Time}_{\Pi}(\lambda) + ct\log q$, where c is a small constant depending only on the model of computation. Replacing $\mathcal{B}$ by $\mathcal{B}'$ and applying $p \le 2q$ yields the claim.

It remains to bound $\Pr[\,\mathcal{B}^0\colon \mathsf{forgery}^* \mid \neg\mathsf{nc}\,]$. We distinguish three cases. First suppose $\mathsf{forgehello}(i)$. Without loss of generality $\mathit{party}_i = A_1$. Then $\mathit{role}_i = 1$, and the i-th *Send* query was of the form $\langle \textsc{hello}, C, \beta_1\rangle$, where $X = \mathcal{D}_{\mathit{sk}^i}(C) = \langle A_2, N, \alpha_1\rangle$ for strings $N \in \{0,1\}^n$, $\alpha_1, \beta_1 \in \{0,1\}^*$, but M was never the response to a query $\mathit{Send}(A_2, M')$. Moreover, for a successful forgery it is necessary that $N = \sigma.\mathit{myN}^j$ for some $j < i$ such that $\mathit{party}_j = A_2$ and $\sigma.\mathit{you}^j = A_1$. Without loss of generality A_2 sampled $\sigma.\mathit{myN}$ in the j-th call. Since nonces chosen by A_1 and A_2 never actually get encrypted, the only place where they are used is to influence the control flow in the equality tests in 4:16, 4:19, 4:33, and 4:41. In particular, the nonce fields are redacted from the principal's state using $\mathit{Sanitize}(\pi)$ before being passed to the *PD*. By $\neg\mathsf{nc}$ the value $\sigma.\mathit{myN}^j$ does not coincide with any nonce chosen by any party in any other round. Thus, in a thought experiment one could swap $\sigma.\mathit{myN}^j$ for any one of the $\ge 2^n - q$ strings in $\{0,1\}^n$ that are never sampled during the game without changing $\mathcal{A}$'s view. Hence the probability that $N = \sigma.\mathit{myN}^j$ is at most $1/(2^n - q)$. Since there are most q choices for i and j, the probability that $\mathcal{A}$ succeeds in forging a HELLO message against A_1 or A_2 is at most $q^2/(2^n - q)$. Interchanging the roles of A_1 and A_2 yields the same bound for the case $\mathit{party}_i = A_2$.

Now let's consider the case where $\mathcal{A}$ attempts to forge a RESPOND message between A_1 and A_2. Suppose there exists some i such that $\mathit{party}_i = A_1$ and the i-th *Send* query is $\mathit{Send}(\mathit{party}_i, M)$, where $M = \langle \textsc{respond}, C, \beta_2\rangle$, and $\mathcal{D}_{\mathit{sk}^i}(C) = \langle A_2, N, N', \alpha_2\rangle$, $N, N' \in \{0,1\}^n$, and $\alpha_2, \beta_2 \in \{0,1\}^*$. We want to bound the probability that $N = \sigma.\mathit{myN}^j$ for some $j < i$ such that $\mathit{party}_j = A_1$, $\sigma.\mathit{you}^j = A_2$, but M was not replied in any run $k < i$ such that $\mathit{party}_k = A_2$ and $\sigma.\mathit{you}^k = A_1$. Since $\sigma.\mathit{myN}^j$ was chosen by A_1, we can apply a similar argument as for forgehello: there are at least $2^n - q$ possible values for $\sigma.\mathit{myN}^j$, and at most q choices for j. Hence the probability that $\mathcal{A}$ forges a RESPOND message between A_1 or A_2 is at most $q^2/(2^n - q)$. Similarly, the same bound holds for the case of CONFIRM messages. All cases are mutually exclusive because the number of components of the plaintext in the three cases above is distinct. Therefore $\Pr[\,\mathcal{B}^0\colon \mathsf{forgery}^* \mid \neg\mathsf{nc}\,] \le q^2/(2^n - q)$. □

Lemma 5. *For a valid adversary $\mathcal{A}$,*

$$\Pr[\,\mathcal{A}^{\mathrm{ma}}\colon \neg\mathsf{vp} \mid \mathsf{proper} \wedge \neg\mathsf{forgery} \wedge \neg\mathsf{nc}\,] = 0\,.$$

Proof: Suppose proper, $\neg\mathsf{forgery}$, and $\mathsf{partnered}(i,j)$. As $\mathit{sid}_i = \mathit{sid}_j \neq \diamond$ and $\mathit{role}_i \neq \mathit{role}_j$, it follows from the construction of $\sigma.\mathit{sid}$ in 4:29, 4:36 that $\mathit{pid}_i = \mathit{party}_j$, $\mathit{pid}_j = \mathit{party}_i$, and $\sigma.C_1^i = \sigma.C_1^j$, $\sigma.C_2^i = \sigma.C_2^j$. Since the partner IDs match, and thus matching keys were used, lines 4:24, 4:26 imply that $\sigma.\alpha_1^i = \sigma.\alpha_1^j$. Similarly, by lines 4:28, 4:33, we get $\sigma.\alpha_2^i = \sigma.\alpha_2^j$. Therefore $\sigma.\mathit{ad}$ was computed using the same deterministic call $\mathit{PD}\,(\langle 0, \sigma.\alpha_1, \sigma.\alpha_2\rangle)$ in 4:29 and 4:37, and thus $\mathit{ad}_i = \mathit{ad}_j$. □

Lemma 6. *For a valid adversary $\mathcal{A}$,*

$$\Pr[\,\mathcal{A}^{\mathrm{ma}}\colon \neg\mathsf{pc} \mid \mathsf{proper} \wedge \neg\mathsf{forgery} \wedge \neg\mathsf{nc}\,] = 0\,.$$

Proof: The proof somewhat tediously traces back the execution of both ends of a session to conclude that they are partnered. Crucially, we use $\neg\mathsf{forgery}$ to infer the authenticity of messages each participant received.

Suppose proper, $\neg\mathsf{forgery}$, and $\neg\mathsf{nc}$, but for some i with conf_i and $\mathsf{fresh}(\mathit{sid}_i)$ there is no j such that $\mathsf{partnered}(i,j)$, i.e., no j such that $\mathit{sid}_i = \mathit{sid}_j \neq \diamond$ and $\mathit{role}_i \neq \mathit{role}_j$. Note that $\mathit{sid}_i \neq \diamond$ since $\mathit{conf}_i \neq \diamond$. We distinguish two cases based on the role of party_i. If $\mathit{role}_i = 0$ then $\sigma.\mathit{conf}^i$ was set in 4:36, so by 4:15 and 4:32–33, party_i must have received a message M_1 of the form $\langle \textsc{respond}, \sigma.C_2^i, \sigma.\beta_2^i\rangle$ in run i, where $\mathcal{D}_{\mathit{sk}^i}(\sigma.C_2^i) = \langle \mathit{pid}_i, \sigma.\mathit{myN}^i, \sigma.\mathit{yourN}^i, \sigma.\alpha_2^i\rangle$. The nonce $\sigma.\mathit{myN}^i$ was generated in an earlier call in which $\sigma.\mathit{you}$ was set, necessarily to $\sigma.\mathit{you}^i$ since that field is written at most once. Since $\mathsf{fresh}(\mathit{sid}_i)$ implies that $\mathit{corrupt}_i = \textsc{false}$ and that $\mathit{corrupt}_B = \textsc{false}$, where $B = \mathit{pid}_i$, we conclude by $\neg\mathsf{forgerespond}(i)$ that M_1 was the response to some j-th *Send* query with $\mathit{party}_j = \mathit{pid}_i$ and $\mathit{pid}_j = \mathit{party}_i$. We aim to show $\mathsf{partnered}(i,j)$. As M_1 was sent in the j-th run, we have $\sigma.C_2^i = \sigma.C_2^j$. Encryptions are only performed in lines 4:24, 4:28, and 4:35. The plaintexts in those lines are encodings of tuples with three, four, and two components, respectively, so ciphertexts and the sender's $\sigma.\mathit{you}$ uniquely determine in which line of the pseudocode they were constructed. In particular, M_1 must have been constructed when executing line 4:28, and since $\sigma.\mathit{you}^j = \mathit{party}_i$'s public key was used to encrypt and its secret key to decrypt, the corresponding plaintexts are equal, that is, $\langle \mathit{pid}_i, \sigma.\mathit{myN}^i, \sigma.\mathit{yourN}^i, \alpha_2^i\rangle = \langle \mathit{party}_j, \sigma.\mathit{yourN}^j, \sigma.\mathit{myN}^j, \alpha_2^j\rangle$. (We shall use an analogous argument several times below, but keep it more succinct.)

By construction of M_1 in 4:28 we have that $\sigma.\mathit{myN}^i = \sigma.\mathit{yourN}^j$, and from line 4:29 see that $\mathit{role}_j = 1$. As

$sid_i = \langle party_i, pid_i, \sigma.C_1^i, \sigma.C_2^i \rangle$, it remains to show that $\sigma.C_1^i = \sigma.C_1^j$. By 4:26, $party_j$ received a message $M_2 = \langle \text{HELLO}, \sigma.C_1^j, \sigma.\beta_1^j \rangle$ in run j such that $\mathcal{D}_{sk^j}(\sigma.C_1^j) = \langle pid_j, \sigma.yourN^j, \sigma.\alpha_1^j \rangle = \langle pid_j, \sigma.myN^i, \sigma.\alpha_1^j \rangle$. Since sid_i is fresh, $pid_j = party_i$, and $party_i = pid_j$, $corrupt_j = corrupt_j = \text{FALSE}$ so $\neg\mathsf{forgehello}(j)$ implies that M_2 was indeed sent in k-th run of *PC*. By the construction of M_2 in 4:24 we have $\sigma.C_1^k = \sigma.C_1^j$, and $cur^i = cur^k$. $\sigma.myN^k = \sigma.yourN^j = \sigma.myN^i$. By $\neg\mathsf{nc}$, we get $cur^i = cur^k$, and thus $\sigma.C_1^i = \sigma.C_1^j$. By the construction of the session ID in 4:29 and 4:36, we conclude that $sid_i = sid_j$. Hence i and j are partnered.

Let's consider the case where $role_i = 1$. Because $conf_i = \text{TRUE}$, line 4:42 must have been executed during the i-th run of *PC*. Hence $party_i$ received a message $M_1 = \langle \text{CONFIRM}, \sigma.C_3^i, \sigma.\beta_3^i \rangle$ such that $\mathcal{D}_{sk^i}(\sigma.C_3^i) = \langle \sigma.myN^i, \sigma.\alpha_3^i \rangle$. By $\neg\mathsf{forgeconfirm}(i)$, there is a $j < i$ such that M_1 was sent by *PC* in the j-th run, and $party_j = pid_i$, $party_i = pid_j$. By the way M_1 was constructed in 4:36 it follows that $role_j = 0 \neq role_i$ and $\sigma.myN^i = \sigma.yourN^j$.

To prove $\mathsf{partnered}(i, j)$ it remains to show that we have $\sigma.C_1^i = \sigma.C_1^j$ and $\sigma.C_2^i = \sigma.C_2^j$. Since $party_j$ sent message M_1 in the j-th run, by 4:32 it must have received a message $M_2 = \langle \text{RESPOND}, \sigma.C_2^j, \sigma.\beta_2^j \rangle$ such that $\mathcal{D}_{sk^j}(\sigma.C_2^j) = \langle party_i, \sigma.myN^j, \sigma.yourN^j, \sigma.\alpha_2^j \rangle$. Since $\mathsf{fresh}(sid_i)$, $party_j = pid_i$, and $pid_j = party_i$, $\neg\mathsf{forgerespond}(j)$ shows that there exists $k < j$ such that $party_k$ sent M_2 in the k-th run, $party_k = pid_j = party_i$, and $pid_k = party_j$. Since $party_k = party_i$ and $\mathsf{fresh}(sid_i)$, line 4:24 implies in particular that $\sigma.myN^i = \sigma.yourN^j = \sigma.myN^k$. By the latter equation and $\neg\mathsf{nc}$, we find that $cur^i = cur^k$ and thus $\sigma.C_2^i. = \sigma.C_2^k = \sigma.C_2^j$.

Since *PC* executed 4:28 in run k, it must have executed 4:26 as well. Therefore it must have received a message $M_3 = \langle \text{HELLO}, \sigma.C_1^k, \sigma.\beta_1^k \rangle$, where $\mathcal{D}_{sk^k}(\sigma.C_1^k) = \langle pid_k, \sigma.yourN^k, \sigma.\alpha_1^k \rangle = \langle party_j, \sigma.myN^j, \sigma.\alpha_1^k \rangle$. Because $\mathsf{fresh}(sid_i)$, $pid_k = party_j = pid_i$, and $\neg\mathsf{forgehello}(k)$, there is some $l < k$ such that $party_l = pid_k$, $party_k = pid_l$, and M_3 was sent in run l. Thus $\sigma.C_1^k = \sigma.C_1^i$. The way M_2 was constructed in 4:24 and its contents verified in 4:40 together with $\sigma.you^l = A^k$ and $A^l = \sigma.you^k$ imply that $\sigma.myN^l = \sigma.yourN^k = \sigma.myN^j$. By $\neg\mathsf{nc}$, we get $\sigma.C_1^i = \sigma.C_1^k = \sigma.C_1^l = \sigma.C_1^j$, thus $sid_i = sid_j$ and $\mathsf{partnered}(i, j)$. This gives $\Pr[\neg\mathsf{pc} \mid \mathsf{proper} \wedge \neg\mathsf{forgery} \wedge \neg\mathsf{nc}] = 0$. □

7. Conclusions

The notions of this paper were not invented in a vacuum, but in an attempt to prove the security for a specific real-word protocol: the Abbreviated Handshake protocol of the draft IEEE 802.11s standard for mesh networking [19, 45]. The protocol there is for authenticated key exchange (AKE) in the symmetric setting. A conventional treatment of the scheme, which was our starting point, would begin by extracting out of the spec a simple four-flow sequence that one would claim to be the relevant object of study. After defining this protocol, one could prove it correct under one of the existing cryptographic definitions for AKE and claim success. But we decided that there would be something deeply unsatisfying with this whole approach. Most of the 802.11s spec would have been elided, including pieces that were clearly of cryptographic relevance. We wanted to make sure we had a proof that would say something about the actual protocol in the spec and in implementations, not some fiction of our design.

In the end, the complexity of the protocol core of the Abbreviated Handshake and the unsettledness of its spec proved to be a major impediment in communicating our ideas, so we switched to NSL, which is not a real-world scheme. It remains a *thesis* of our work that the PSP idea can be used to faithfully represent a real-world scheme without taking on an unmanageable amount of complexity.

Acknowledgment

The authors thank Mihir Bellare, Mark Gondree, Payman Mohassel, Jesse Walker, Meiyuan Zhao, and the anonymous referees for their comments and suggestions. We were supported in part by NSF 0208842 and a gift from Intel Corporation.

References

[1] M. Backes, M. Berg, and D. Unruh. A formal language for cryptographic pseudocode. *Logic for Programming, Artificial Intelligence, and Reasoning*, LNCS vol. 5330, Springer, pp. 353–373, 2008.

[2] M. Backes and P. Laud. Computationally sound secrecy proofs by mechanized flow analysis. *ACM Conference on Computer and Communications Security* (CCS 2006). ACM Press, pp. 370–379, 2006.

[3] M. Backes and B. Pfitzmann. A cryptographically sound security proof of the Needham-Schroeder-Lowe public-key protocol. *IEEE Journal on Selected Areas in Communications* 22(10), pp. 2075–2086, 2004.

[4] M. Backes, B. Pfitzmann, and M. Waidner. Symmetric authentication in a simulatable Dolev-Yao-style cryptographic library. *International Journal of Information Security*, 4(3), pp. 135–154, 2005.

[5] G. Barthe, B. Grégoire, and S. Zanella. Formal certification of code-based cryptographic proofs. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL 2009), pp. 90–101, ACM Press, 2009.

[6] M. Bellare, A. Desai, D. Pointcheval, P. Rogaway. Relations among notions of security for public-key encryption schemes. *Advances in Cryptology – CRYPTO '98*, LNCS vol. 1462, Springer, pp. 26–45, 1998.

[7] M. Bellare, D. Pointcheval, and P. Rogaway. Authenticated key exchange secure against dictionary attacks. *Advances in Cryptology – EUROCRYPT 2000*, LNCS vol. 1807, Springer, pp. 139–155, 2000.

[8] M. Bellare and P. Rogaway. Entity authentication and key distribution. *Advances in Cryptology – CRYPTO '93*, LNCS vol. 773, Springer, pp. 232–249, 1994.

[9] M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. *Advances in Cryptology – EUROCRYPT 2006.* LNCS vol. 4004, Springer, pp. 409–426, 2006.

[10] S. Blake-Wilson and A. Menezes. Entity authentication and authenticated key transport protocols employing asymmetric techniques. *Security Protocols Workshop 1997*, LNCS vol. 1361, Springer, pp. 137–158, 1997.

[11] B. Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4), pp. 193–207, 2008.

[12] S. Bradner. Key words for use in RFCs to indicate requirement levels. RFC 2119. March 1997.

[13] R. Canetti. Universally composable security: a new paradigm for cryptographic protocols. Cryptology ePrint Report 2000/067. Last revised 13 Dec 2005. Extended abstract in *FOCS 2001*.

[14] R. Canetti and J. Herzog. Universally composable symbolic analysis of mutual authentication and key-exchange protocols. *TCC 2006*, LNCS vol. 3876, pp. 380–403, 2006.

[15] R. Canetti and H. Krawczyk. Security analysis of IKE's signature-based key-exchange protocol. *Advances in Cryptology – CRYPTO 2002.* LNCS vol. 2442, Springer, pp. 143–161, 2002.

[16] V. Cortier and B. Warinschi. Computationally sound, automated proofs for security protocols. *European Symposium on Programming* (ESOP 2005). LNCS vol. 3444, Springer, pp. 157–171, 2005.

[17] A. Datta, A. Derek, J. C. Mitchell, and A. Roy. Protocol Composition Logic (PCL). *Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin*, ENTCS vol. 172, Elsevier, pp. 311–358, 2007.

[18] M. Dworkin. Recommendation for block cipher modes of operation: the CCM mode for authentication and confidentiality. NIST Special Publication 800-38C. May 2004.

[19] S. Connor. IEEE P802.11s draft 1.08. January 2008.

[20] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol: version 1.2. RFC 5246. August 2008.

[21] IEEE Standard 802.11i, Part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications: Amendment 6: Medium Access Control (MAC) security enhancements. IEEE Computer Society, 2004.

[22] S. Gajek, M. Manulis, O. Pereira, A Sadeghi, and J. Schwenk. Universally composable security analysis of TLS—secure sessions with handshake and record layer protocols. *ProvSec 2008*, LNCS 5324, Springer, pp. 313–327, 2008.

[23] R. Gennaro and Y. Lindell. A framework for password-based authenticated key exchange. *ACM TISSEC*, 9(2), pp. 181–234, 2006.

[24] S. Goldwasser, S. Micali. Probabilistic Encryption. *Journal of Computer and System Sciences*, 28(2), pp. 270–299, 1984.

[25] C. He, M. Sundararajan, A. Datta, A. Derek, J. C. Mitchell. A modular correctness proof of IEEE 802.11i and TLS. *ACM Conference on Computer and Communications Security* (CCS '05), pp. 2–15, ACM, 2005.

[26] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985. www.usingcsp.com

[27] M. L. Hui and G. Lowe. Fault-preserving simplifying transformations for security protocols or Not just the Needham Schroeder Public Key Protocol. *Journal of Computer Security*, vol. 9, pp. 3–46, 2001.

[28] C. Kaufman. Internet key exchange (IKEv2) protocol. RFC 4306. December 2005. See also RFC 2409, The Internet key exchange, by D. Harkins and D. Carrel, November 1998.

[29] H. Krawczyk. SIGMA: The 'SIGn-and-MAc' approach to authenticated Diffie-Hellman and its use in the IKE-protocols. *Advances in Cryptology – CRYPTO 2003.* LNCS vol. 2729, Springer, pp. 400–425, 2003.

[30] D. Kuhlman, R. Moriarty, T. Braskich, S. Emeott, and M. Tripunitara. A correctness proof of a mesh security architecture. *IEEE Computer Security Foundations Symposium 2008* (CSF 2008), IEEE Press, pp. 315–330, 2008.

[31] B. LaMacchia, K. Lauter, and A. Mityagin. Stronger security of authenticated key exchange. *ProvSec 2007.* LNCS vol. 4784, Springer, pp. 1–16, 2007.

[32] G. Lowe. An attack on the Needham-Schroeder public key authentication protocol. *Information Processing Letters*, 56(3), pp. 131–136, 1995.

[33] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. *Tools and Algorithms for Construction and Analysis of Systems* (TACAS 1996), LNCS vol. 1055, Springer, pp. 147–166, 1996.

[34] D. Micciancio and B. Warinschi. Soundness of formal encryption in the presence of active adversaries. *Theory of Cryptography Conference* (TCC 2004), LNCS vol. 2951, Springer, pp. 133–151, 2004.

[35] A. Menezes and B. Ustaoglu. Comparing the pre- and post-specified peer models for key agreement. *ACISP 2008.* LNCS vol. 5107, Springer, pp. 53–68, 2008.

[36] P. Morrissey, N. Smart, and B. Warinschi. A modular security analysis of the TLS handshake protocol. *Advances in Cryptology – ASIACRYPT 2008.* LNCS vol. 5350, Springer, pp. 55–73, 2008.

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

[38] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. The Kerberos network authentication service (V5). RFC 4120. July 2005.

[39] C. Rackoff and D. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. *Advances in Cryptology – CRYPTO 1991*, LNCS vol. 576, Springer, pp. 433–444, 1991.

[40] P. Rogaway. Authenticated-encryption with associated-data. *ACM Conference on Computer and Communications Security* (CCS 2002), ACM Press, pp. 98–107, 2002.

[41] V. Shoup. On formal models for secure key exchange. Manuscript, 1999.

[42] C. Sprenger, M. Backes, D. Basin, B. Pfitzmann, and M. Waidner. Cryptographically sound theorem proving. *Computer Security Foundations Workshop* (CSFW 19), IEEE Press, pp. 153–166, 2006.

[43] V. Shoup and A. Rubin. Session key distribution using smart cards. *Advances in Cryptology – EUROCRYPT 1996*, LNCS vol. 1070, Springer, pp. 321–331, 1996.

[44] B. Warinschi. A computational analysis of the Needham-Schroeder-(Lowe) protocol. *Journal of Computer Security*, 13(3), pp. 565–591, 2005. Earlier version in *Computer Security Foundations Workshop* (CSFW 16), IEEE Press, pp. 248–262, 2003.

[45] M. Zhao, J. Walker, S. Conner, H. Suzuki, and J. Kruys. Abbreviated handshake for authenticated peer link establishment. Document IEEE 802.11-07/1999r4. Proposal to the to IEEE 802.11s Task Group, September 2007.