Cryptographically-sound Protocol-model Abstractions*

Christoph Sprenger
ETH Zurich, Switzerland
sprenger@inf.ethz.ch

David Basin
ETH Zurich, Switzerland
basin@inf.ethz.ch

Abstract

We present a formal theory for cryptographically-sound theorem proving. Our starting point is the Backes-Pfitzmann-Waidner (BPW) model, which is a symbolic protocol model that is cryptographically sound in the sense of blackbox reactive simulatability. To achieve cryptographic soundness, this model is substantially more complex than standard symbolic models and the main challenge in formalizing and using this model is overcoming this complexity. We present a series of cryptographically-sound abstractions of the original BPW model that bring it much closer to standard Dolev-Yao style models. We present a case study showing that our abstractions enable proofs of complexity comparable to those based on more standard models. Our entire development has been formalized in Isabelle/HOL.

1. Introduction

Cryptographically-sound symbolic models. It is well-known that security protocols are difficult to get right. Cryptographic security proofs offer a potential remedy, but they are long, complicated, and error-prone. Machine-checked proofs are an alternative, but they are typically based on the Dolev-Yao model [9], where messages are algebraic terms and cryptography is assumed to be perfect. However, in general, this model lacks a cryptographic justification. We would ideally like the best of both worlds: cryptographically-sound, machine-checked proofs.

The first Dolev-Yao model with a cryptographic justification under arbitrary attacks was introduced by Backes, Pfitzmann, and Waidner [3]. This model, called the *BPW model*, can be implemented in the sense of blackbox reactive simulatability (BRSIM) [15] by real cryptographic systems that are secure under standard cryptographic definitions. As a result, we can replace an abstract system (here the BPW model) by a concrete one (here its cryptographic realization) in arbitrary protocol contexts while preserving its security properties [15, 5, 2]; BRSIM is also called UC for its universal composition properties. Hence, after a one-time BRSIM reduction proof, we can reason about protocols in the substantially simpler symbolic setup, knowing that properties we prove transfer to the cryptographic world.

Roughly speaking, the BPW model can be viewed as a centralized cryptographic library component. It has an encapsulated state where it tracks which party (honest users or the adversary) knows which messages and it provides interface functions for constructing, decomposing, and sending messages. For cryptographic soundness in the sense of BRSIM, the BPW model has certain *non-standard aspects* in comparison with other Dolev-Yao models. For example, ciphertexts include randomness tags (modeling probabilistic encryption) and leak the length of the underlying cleartext and the adversary may generate invalid ciphertexts. Moreover, protocol messages are DAG-structured and are not manipulated directly, but rather are referred to indirectly using handles (*i.e.*, pointers).

Towards more abstract models. In this paper, we describe our formalization and use of increasingly abstract, cryptographically-sound symbolic models. Our starting point is the original BPW model, which provides cryptographic soundness, but whose complexity raised the following question: *Is it possible to reason efficiently about protocols based on this model, using a theorem prover, without sacrificing cryptographic soundness?* The BPW model introduces a number of complexities, including those listed above, which are obstacles to efficient formal reasoning. As a consequence, standard techniques for reasoning about state-based systems, such as Hoare logics and weakest precondition calculi, scale poorly to the complex state spaces and pointer structures that arise. Indeed, after our initial formalization of this model in Isabelle/HOL [13], the complexity was such that we were not even able to prove the relatively simple Needham-Schroeder-Lowe (NSL) protocol [11]. Given this, the focus of our work has shifted from formalizing and using the BPW model to developing methods to simplify proofs by lifting the level of modeling ab-

*This work was partially supported by the Zurich Information Security Center. It represents the views of the authors. David Basin acknowledges the support of IBM Zurich Research Laboratory during his sabbatical.

$$\begin{aligned} \text{Crypt. Impl.} &\leq \text{Original BPW Model} \\ &\subseteq \text{DAG-based BPW Model} \\ &\leq \text{Term-based BPW Model} \end{aligned}$$

Figure 1. Protocol model abstractions

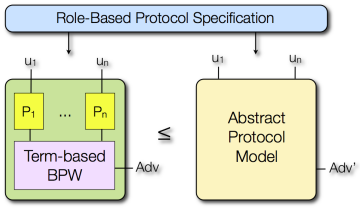

Figure 2. Overview of the last abstraction

stractions. Given these methods, we can now answer the above question affirmatively.

Figures 1 and 2 summarize the different models involved and their relationships. In [3], the original BPW model was shown to be a BRSIM (denoted by $\leq$) abstraction of the cryptographic implementation. Our first formalization of the BPW model is the *DAG-based BPW model*. This model is a compositional, property-preserving over-approximation (denoted by $\subseteq$) of the original BPW model. Our second formalization, the *term-based BPW model*, replaces the DAG-structured messages in the machine's state by inductively-defined messages. We formally prove that the term-based and the DAG-based models are bisimilar, which is a special case of BRSIM. Finally, as depicted in Figure 2, we define our *abstract protocol model*, which interprets a class of role-based protocol specifications, using high-level operations for pattern matching and message construction. For each protocol in our class, this model reactively simulates the term-based model. As both $\leq$ and $\subseteq$ are compositional and preserve security properties (including integrity and secrecy), it follows that such properties proved for role-based protocols in the abstract protocol model also hold for the cryptographic protocol implementation. Moreover, these properties are preserved in all (protocol) contexts.

We describe in this paper our models and their relationships, formalized in Isabelle/HOL. We also investigate the effects of abstraction. Namely, each abstraction step brings us closer to standard Dolev-Yao models and substantially simplifies theorem proving. We expand on this below.

The abstraction step from the DAG-based to the term-based model replaces operations that manipulate pointer-structured messages by equivalent operations that manipulate terms over an inductively-defined data type. The use of inductively-defined messages is more in-line with standard models and it substantially simplifies reasoning by allowing structural induction. Moreover, it enables concise property specifications using functional closure operators, such as Paulson's *analyze* [14] which closes a set of messages under cryptographically-accessible submessages. The equational theories associated with these operators enable the efficient use of Isabelle's term rewriter for simplification. These enhancements allowed us to give a (cryptographically-sound) proof of the NSL protocol.

Although the abstraction introduced by the term-based model is substantial, there is still a significant "semantic gap" between this model and standard Dolev-Yao models. This can largely be attributed to the different granularities and side-effect behavior of the message-handling operations. In standard Dolev-Yao models, messages are manipulated directly and without side effects (*e.g.*, matching a message m against a pattern). In contrast, in the BPW model, messages are manipulated constructor-wise by passing their handles to interface functions, which may produce side effects (*e.g.*, generating handles for m's subterms). As a result, protocols must be specified at a low level using the BPW interface functions and the term-based model requires fine-grained, lengthy security proofs. In particular, due to interface functions' side effects, the proof of each protocol invariant must be decomposed into preservation results for each of the 17 imperative BPW-model operations.

Moving to the abstract protocol model largely eliminates the remaining semantic gap and brings the complexity of theorem proving in-line with standard Dolev-Yao models. Namely, we specify the abstract model (right-hand side of Figure 2) as an instance of a generic security-protocol interpreter. The interpreter executes security protocols, formalized as role-based protocol specifications built from sequences of patterns. We give a second operational semantics to role-based descriptions, instantiating our generic interpreter to interpret protocols using the operations of the term-based model (left-hand side of Figure 2).

This last abstraction overcomes the problems of the term-based model. First, role-based specifications allow us to concisely and naturally specify protocols. Second, and most importantly, when reasoning about protocols, we reason in the abstract model about the interpretation of entire protocol steps, where messages are manipulated directly. We prove general logical characterizations of these high-level protocol operations, which substantially simplify protocol proofs. Moreover, since these operations are side-effect-free, invariant preservation comes for free. This allows us to shift reasoning from Hoare logic to assertional reasoning in HOL. Hence, reasoning in this model is at a much higher abstraction level than in the term-based model.

Contributions. Our primary contribution is to close the gap between the BPW model and standard Dolev-Yao models. While our abstract model still has some non-standard aspects, the level of abstraction and complexity is comparable to standard Dolev-Yao models, such as Paul-

son's [14]. In particular, the protocol interpreter operates directly on messages and patterns from the protocol specification and the adversary capabilities are defined in the standard Dolev-Yao-style using message-derivation operators. A secondary contribution is an infrastructure for effective reasoning about security protocols, namely a much simpler model along with general protocol-independent theorems. These theorems, formulated in Hoare logic, include invariants (*e.g.*, well-formedness of protocol threads) and logical characterizations of the operational protocol interpretation. We use the NSL protocol as a case study, documenting that this infrastructure leads to substantially simpler proofs, whose complexity is comparable to Paulson's proofs.

Our entire development, including the three models and all proofs, has been formalized in Isabelle/HOL. We provide some statistics in Sections 6 and 7. Due to the technical detail and intricacy of the proofs involved, we strongly believe that formal theorem proving has a central role to play in relating *and* reasoning about security protocol models. Our work is unique in this area and shows that this is possible.

2. Background on Isabelle/HOL

Isabelle is a generic, tactic-based theorem prover. We use Isabelle's implementation of higher-order logic (HOL). HOL can roughly be seen as logic on top of functional programming. We will assume that the reader has basic familiarity with both logic and typed functional programming.

In Isabelle/HOL, $t :: T$ denotes a term t of type T. The expression $c\ x \equiv t$ defines the function constant c with parameter x as the term t. Type variables are denoted by lowercase Greek letters. Given types α and β, $\alpha \Rightarrow \beta$ is the type of (total) functions from α to β, $\alpha \times \beta$ is the product type, and α *set* is the type of sets with elements of type α. Moreover, $[\alpha, \beta] \Rightarrow \gamma$ abbreviates $\alpha \Rightarrow \beta \Rightarrow \gamma$. There are several mechanisms for defining new types, such as the datatype declaration, which defines an inductive data type. For example, the polymorphic option type is defined as datatype α *option* = *None* | *Some* α. Functions of type $\alpha \Rightarrow \beta$ *option* model partial functions from α to β. The declaration types $T_1 = T_2$ merely introduces a new name for the type T_2, possibly with parameters, as in types $\alpha \rightharpoonup \beta = \alpha \Rightarrow \beta$ *option*. Isabelle/HOL includes a package supporting record types. For example, a type of points defined by record *point* = $x :: nat$ $y :: nat$ contains records like $r = (\!|x{=}1, y{=}2|\!)$ as elements. Each field determines a projection, like in $x\ r = 1$. Records are extensible: record *cpoint* = *point* + $c :: color$ extends points with a color field.

3. Monadic modeling and verification tools

We now briefly describe our monad-based component model and associated logics. For further details see [18].

3.1. Monads and components

Monads are a category-theoretic notion used to abstractly model different computational phenomena in a functional setting. From a programming perspective, a monad is a type constructor with a unit and a binding operation, enjoying unit and associativity properties. In our case, we work with a deterministic state-exception monad.

```
datatype α result = Exception | Value α
types (α, σ) S = σ ⇒ α result × σ

return :: α ⇒ (α, σ) S
return a ≡ λs. (Value a, s)

bind :: (α, σ) S ⇒ (α ⇒ (β, σ) S) ⇒ (β, σ) S
bind m k ≡ λs. let (a, t) = m s in
  case a of Exception ⇒ (Exception, t)
          | Value x ⇒ k x t
```

A term of type $(\alpha, \sigma)\ S$ is called a *computation*. Computations act on a state of type σ and return either a value of type α or result in an exception. The monad unit, called return, embeds a value into a computation. Binding acts as sequential composition with value passing in the style of a let-binding. We write do $x \leftarrow m$; $k\ x$ for *bind* $m\ k$ and omit the '$x \leftarrow$' when the return value is ignored. Our monad also provides functions for raising and catching exceptions. To build more complex computations, we directly use control structures from Isabelle/HOL, such as pattern matching (case) and recursive functions.

Our components consist of a state manipulated by a set of *interface functions*. We model a state as an extensible record whose fields are the state variables. We give components an operational semantics in terms of a transition system. A computation *m* gives rise to a transition relation $tr\ m \equiv \{(s, t) \mid t = snd\ (m\ s)\}$. The transition relation of a component is the union of the transition relations of its interface functions. We obtain the transition system associated with a component by adding a set of initial states. We define the runs of a transition system as infinite sequences of states, starting in an initial state and where successive states are related by transitions.

3.2. Program and temporal logics

We have implemented a weakest pre-condition (WP) calculus, a Hoare logic, and a linear-time temporal logic (LTL). All these logics use HOL sets (of states) as their basic assertions and set operations as boolean connectives. Hence, when we say that a state s satisfies a basic assertion P, we mean $s \in P$. The WP calculus and the Hoare logic are both tailored to our state-exception monad, whereas the temporal logic is interpreted over standard transition systems and linked to the other logics via "gluing lemmas".

Our weakest pre-condition calculus is inspired by Pitts' evaluation logic [16], which generalizes dynamic logic by interpreting it over monads. We formalize a shallow semantic embedding of a variant of evaluation logic, instantiated to the state-exception monad.

$$[x \leftarrow m]Q\,x \equiv \{s \mid \forall\, x\ t.\ (m\,s) = (\mathit{Value}\ x, t) \longrightarrow t \in Q\,x\}$$
$$[@\ m]Q \equiv \{s \mid \forall\, t.\ (m\,s) = (\mathit{Exception}, t) \longrightarrow t \in Q\}$$

We have two box modalities: $[x \leftarrow m]Q\,x$ for normal termination and $[@\ m]Q$ for exceptional termination. These correspond to the weakest pre-condition of the computation m with respect to post-condition Q. Note the dependence of the assertion Q on the result value x in $[x \leftarrow m]Q\,x$.

We construct our Hoare logic on top of the WP calculus.

$$\{P\}\ m\ \{> Q\} \equiv P \subseteq [x \leftarrow m](Q\,x)$$
$$\{P\}\ m\ \{@\ Q\} \equiv P \subseteq [@\ m]Q$$

We have one type of Hoare triple for each type of termination. We formalize and derive the rules for Hoare logic, expressed in terms of this embedding. For example:

$$\frac{\{P\}\ p\ \{> Q\} \qquad \bigwedge x.\ \{Q\ x\}\ q\ x\ \{> R\}}{\{P\}\ \mathsf{do}\ x \leftarrow p;\ q\ x\ \{> R\}}\ \mathit{bindN}$$

Here, $\bigwedge x$ denotes a universal (meta-logic) quantification over all inputs x to q. A companion rule handles the case of termination with an exception.

We formulate component properties as invariants in our standard embedding of LTL in Isabelle/HOL. The proof of each invariant I is reduced to a set of preservation results of the form $\{I \cap J\}\ f\ x\ \{> \lambda\, z.\ I\}$ in monadic Hoare logic, one triple for each interface function f of the component under study. Here, J denotes a previously proved auxiliary invariant. We decompose the monadic program constructs using our Hoare logic, which may require user-supplied intermediate assertions (*e.g.* Q in the rule bindN). We deal with the program control structures (*e.g.*, pattern matching and recursion) directly in HOL (*e.g.*, using case analysis and induction). Once this decomposition yields sufficiently simple program parts, we may switch to the WP calculus by unfolding the definition of the respective Hoare triple. We then finish the proof using assertional reasoning in HOL.

3.3. Relational Hoare logic and bisimulation

To establish BRSIM relations between our models, we must reason about bisimulation between components. We define bisimilarity, *bisim* $R\ m_1\ m_2$, in terms of the following general notion of a *relational Hoare tuple*.

$$\{R\}\ m_1\ m_2\ \{S, T\} \equiv \forall\, s\ t\ s'\ t'\ x\ y.$$
$$(s,\ t) \in R \wedge m_1\ s = (x, s') \wedge m_2\ t = (y, t') \longrightarrow$$
$$(\exists\, a\ b.\ x = \mathit{Value}\ a \wedge y = \mathit{Value}\ b \wedge (s',\ t') \in S\ a\ b)$$
$$\mid (x = \mathit{Exception} \wedge y = \mathit{Exception} \wedge (s',\ t') \in T)$$

$$\mathit{bisim}\ R\ m_1\ m_2 \equiv \{R\}\ m_1\ m_2\ \{(\lambda\, a\ b.\ \{(s, t) \mid a = b\} \cap R), R\}$$

Relational Hoare tuples have two postconditions. The first postcondition is a parametrized relation $S\ a\ b$, required to hold whenever the computation m_1 returns the value a and m_2 returns the value b. The second postcondition T covers the case where both computations terminate with an exception. Bisimulation is the special case where the bisimulation relation appears in both the pre and post-relations and, moreover, the result values are required to be identical.

To prove the bisimilarity of pairs of computations we have derived a set of proof rules for relational Hoare logic (see also [4]). An example is the following rule for sequential compositions, where S_n is the intermediate relation for normal termination of m_1 and m_2.

$$\frac{\{R\}\ m_1\ m_2\ \{S_n, T_e\} \quad \bigwedge a\ b.\ \{S_n\ a\ b\}\ (k_1\ a)\ (k_2\ b)\ \{T_n, T_e\}}{\{R\}\ (\mathsf{do}\ a \leftarrow m_1;\ k_1\ a)\ (\mathsf{do}\ b \leftarrow m_2;\ k_2\ b)\ \{T_n, T_e\}}$$

To prove that two components with identical interfaces are bisimilar, we show that all pairs of identically named interface functions are bisimilar, under a given bisimulation relation R. Since the interface functions are deterministic, this suffices to establish the two components' bisimilarity.

Blackbox reactive simulatability. We define a version of BRSIM for our deterministic setting. For the probabilistic polynomial-time setting we refer the reader to [15, 5, 2]. The models we consider have separate interfaces for honest users and for the adversary. We define properties that are preserved from abstract models to concrete models in terms of traces of *user input/output events* at the user interface.

Let C and D be two models (components) with identical user interfaces, but possibly different adversary interfaces. We say that D *reactively simulates* C *in the sense of BRSIM*, written $C \leq D$, if there is a simulator S such that C is bisimilar to the component obtained by connecting S to D's adversary interface; the simulator maps C's concrete adversary interface to D's abstract adversary interface. Hence, the I/O traces at C's user interface are also possible user I/O traces of D. Moreover, security properties expressed in terms of user I/O traces are preserved from D to C. Equivalently, any attack on D translates to an attack on C.

It follows from the definition of BRSIM that BRSIM is preserved in all contexts. Let $\mathit{Con}[\cdot]$ be a *protocol context* that can be connected to the user interfaces of both C and D and itself provides a user interface. Then we have $\mathit{Con}[C] \leq \mathit{Con}[D]$. Hence, the above statements about user I/O traces and their properties also hold for $\mathit{Con}[C]$ and $\mathit{Con}[D]$.

4. Abstractions of the BPW model

We describe, in this and the next section, the models and the relationships depicted in Figures 1 and 2. We omit, however, the BPW model's cryptographic realization as this is not necessary for understanding this paper's contributions.

4.1. Original BPW model

The BPW model constitutes a library of cryptographic operations, which tracks the messages known by each party. The model provides *local functions* for manipulating messages and *send functions* for exchanging messages between an arbitrary, but fixed, number N of users and the adversary. Some of these functions implement extended capabilities available only to the adversary. At the interface, messages are referred to by *handles*. This indirection is necessary for cryptographic soundness (BRSIM), since the model and its cryptographic realization work with vastly different objects: abstract terms and bitstrings, respectively. Handles uniformly present these objects to the users and hence avoid that the two models can be trivially distinguished.

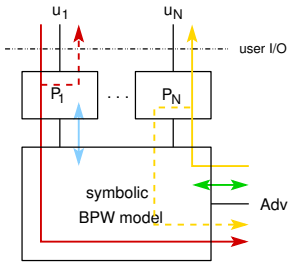

Figure 3. Components and control flow

To model a security protocol within the BPW model, we compose the model with a component P_i for each user u_i, which implements the protocol by invoking the BPW-model functions. Each P_i maintains its own local state (*e.g.*, to store nonces it has generated) and provides interfaces for communicating with u_i and with the BPW model. Figure 3 depicts some typical control flows through the system, which can be classified by whether they are initiated by a user or the adversary. First, a user may give input to its protocol component, which then constructs the first protocol message by invoking local BPW functions. This message may then be sent to the network (*i.e.*, the adversary). Second, the adversary may decompose messages and construct new ones using the local BPW-model functions. He may also send a message to some user u_i. The BPW model delivers such messages to the protocol component P_i, where they are processed according to the protocol.

4.2. Modeling decisions and formalization

We now summarize the principal abstraction steps and design choices that we have employed in our first formalization of the BPW model. For more details, see [17]. The combination of these steps results in our over-approximation relation $\subseteq$, which is compositional and preserves security properties, including integrity and secrecy.

Polynomial bounds. In the original BPW model, users and the adversary constitute probabilistic, polynomially-bounded machines. We model these by universal quantification over all possible inputs, corresponding to a single unbounded machine that non-deterministically produces arbitrary input to the system. This safely over-approximates the original setting, since the unbounded machine can (weakly) simulate any combination of probabilistic, polynomially-bounded users and the adversary. Moreover, we have formalized the enforcement of polynomial bounds on message lengths using an uninterpreted function of the security parameter as the bound. This subsumes all instances with polynomially-bounded functions and thus constitutes a safe over-approximation.

Components and communication. Each machine transition in the original BPW model can be seen as a function call (with parameters passed at an input port) producing either a return value (at an output port) or an exception. We replace the BPW machine model by monadic components (Section 3.1) and asynchronous communication by function calls, thereby eliminating the message buffers in the original model. As a consequence, we pass from a *small-step semantics*, with its internal transitions such as communication via buffers, to a (weakly) bisimilar *big-step semantics*, where such transitions are suppressed.

4.3. The DAG-based BPW model

Our first formalization follows the above modeling decisions. This model (and subsequent ones) formalize different *parties* and their knowledge, given by *knowledge maps*.

```
datatype party = User user | Adv
types μ kmap = party ⇒ hnd ⇀ μ
```

Here, *user* denotes the type of honest users, which is isomorphic to the set $\{1, \ldots, N\}$, and *hnd* is the type of handles, which is isomorphic to $\mathbb{N}$. Knowledge maps track who knows which messages (of generic type μ).

As in the original BPW model, the DAG-based model uses a pointer-like structure to represent messages. The state consists of a database storing messages, which are referred to by indices (of type *ind*, isomorphic to $\mathbb{N}$), together with a knowledge map instantiated to indices.

```
record δ iLibState =
  db :: ind ⇒ δ entry        −− the database
  knowsI :: ind kmap         −− knowledge map
```

The database models a heap where entries are allocated. Database entries have a content and a length field. We have also formalized signatures, which we omit here for brevity.

```
datatype δ content =
    iNonce                   −− nonce
  | iGarbage                 −− garbage
```

```
  | iPke ind                 -- public encryption key
  | iSke                     -- private encryption key
  | iData δ                  -- payload data
  | iPair ind ind            -- pair
  | iEncv ind ind            -- valid ciphertext
  | iEnci ind                -- invalid ciphertext

record δ entry =
  cont :: δ content          -- content
  len :: nat                 -- length of entry
```

Elements of the type δ *content* are messages (polymorphic in the type δ of payload data), built from message constructors. In a well-formed database, each index known by some party determines a directed acyclic graph, the DAG-based BPW-model representation of a message. Hence, constructor arguments of type *ind* point to other database entries. For example, in the term *iEncv pki mi*, representing a valid encryption, the first argument points to the public key used and the second to the plaintext message. In contrast to commonly used Dolev-Yao models, our adversary may create garbage entries (*iGarbage*) or invalid ciphertexts (*iEnci*). The length field in each entry enables the length to leak to the adversary and the model to enforce length bounds.

The BPW-model interface functions manipulate DAG-based messages. As examples of local interface functions, the operations *encryptI* and *decryptI* for public key encryption and decryption have the types [*party*, *hnd*, *hnd*] $\Rightarrow$ (*hnd*, δ *iLibState*) *S*. The function *encryptI* takes a public key and a cleartext and returns the ciphertext and *decryptI* takes a secret key and a ciphertext and returns a cleartext. Message arguments and results are referred to by handles. If an argument is invalid, an exception is raised.

An important difference between this model and other Dolev-Yao models is that encrypting a message with the same key always results in a new ciphertext, *i.e.* a fresh database entry. This reflects that secure encryption is probabilistic and shows the role of indices in modeling idealized randomness. In fact, all message constructors, except for payload data and pairs, produce new database entries with each invocation. Payload data and pairs are allocated only once and shared between users. Another difference is that the adversary may learn the length of the plaintext underlying a ciphertext, modeling a length-revealing crypto system.

Although our formalization considerably simplifies the original BPW model, the DAG-based model is still too complex for a practically useful verification framework. We address some of its problems, including the lack of an inductive message structure, in the term-based BPW model.

4.4. The term-based BPW model

Our second model, the *term-based BPW model*, abstracts messages to inductively defined terms. It is based on the observation that message sharing between users in the DAG-based model is inessential and can be eliminated. We thereby obtain a more abstract representation of messages using an inductive data type. Moreover, Isabelle automatically generates an induction scheme for each such type.

```
datatype δ msg =
    mNonce tag                -- nonce
  | mGarbage tag len          -- adversary garbage
  | mPke key                  -- public key
  | mSke key                  -- private key
  | mData δ                   -- data item
  | mPair (δ msg) (δ msg)     -- pair of messages
  | mEncv tag key (δ msg)     -- valid ciphertext
  | mEnci tag key len         -- invalid ciphertext
```

This definition replaces indices previously in the content fields of database entries with messages. Additionally, the role of indices in allocating fresh database entries for cryptographic messages is taken by the elements of a new, but isomorphic, type *tag*, which can be thought of as an (abstraction of) random coins. The type *key* is an alias for *tag*. Moreover, we now determine the length of messages by a partially interpreted function *len_ofM* of type δ *msg* $\Rightarrow$ *len*, which allows us to remove redundant length information from the state. Length fields are still required for garbage and invalid ciphertexts, as the adversary can choose arbitrary lengths for these two message types.

This abstraction step substantially simplifies the structure of states by eliminating the database and (largely) the length fields: a state of the term-based BPW model simply consists of a knowledge map storing messages.

```
record δ mLibState = knowsM :: δ msg kmap
```

The second substantial improvement, leading to more concise specifications and improved proof automation, stems from adapting to our setting the closure operators *parts* and *analyze* and their equational theories developed by Paulson [14]. The term *parts H* denotes the closure of the set of messages *H* under all submessages, whereas *analyze H* closes *H* under all cryptographically-accessible submessages. Hence, *analyze* (*ran* (*knowsM s u*)) denotes the set of messages that the party *u* can derive from his knowledge in state *s* (*ran f* denotes the range of the partial function *f*). Using *analyze* and *parts*, we define secrecy of an atomic message *m* as follows.

$$\mathit{secret}\ s\ m\ U \equiv \forall u.\ m \in \mathit{analyze}\ (\mathit{ran}\ (\mathit{knowsM}\ s\ u)) \longrightarrow u \in U$$

This states that in state *s* the message *m* is a secret shared by (at most) the parties in the set *U*.

4.5. Bisimulation result

We have used our relational Hoare logic to establish:

Theorem 4.1. *The DAG-based and term-based BPW models are bisimilar.*

Thus these models are also BRSIM, since bisimulation is a special case of BRSIM where the simulator is the identity.

Central to our bisimulation relation is the message abstraction relation *message s i2t* :: (*ind* × δ *msg*) *set*. This associates database indices to messages and is parametrized by a state *s* of the DAG-based model and a function *i2t* mapping indices to tags, which witnesses that tags play the role of indices for message freshness. Note that this relation is defined independently of the states of the term-based model. The inductive definition of *message* contains a rule for each constructor of the type δ *content*. For example, the rule for valid ciphertexts is:

$$\frac{s \in \mathit{contains}\ i\ (\mathit{iEncv}\ \mathit{pki}\ \mathit{mi}) \qquad \mathit{tg} = \mathit{i2t}\ i \qquad (\mathit{pki}, \mathit{mPke}\ k) \in \mathit{message}\ s\ \mathit{i2t} \qquad (\mathit{mi}, m) \in \mathit{message}\ s\ \mathit{i2t}}{(i, \mathit{mEncv}\ \mathit{tg}\ k\ m) \in \mathit{message}\ s\ \mathit{i2t}}$$

This rule states that, at some fixed state *s*, an index *i* abstracts to the ciphertext message (*mEncv tg k m*) if the index *i* contains (*iEncv pki mi*), the index *pki* abstracts to the public key message (*mPke k*), the index *mi* abstracts to message *m*, and the tag *tg* is the image of *i* under *i2t*. The main property proved for *message s i2t* is its functionality.

We define a family of bisimulation relations, consisting of pairs of states for which the domains of the knowledge maps are identical and the message at *knowsM s u h* (if defined) is an abstraction of the index at *knowsI s u h*. Moreover, the parameter *i2t* is required to be bijective to map different database entries to different messages.

$$\begin{aligned}
&\mathit{I2M}\ \mathit{i2t} \equiv \{(s, t) \mid \mathit{bij}\ \mathit{i2t} \wedge \\
&\quad (\forall u.\ \mathit{dom}\ (\mathit{knowsI}\ s\ u) = \mathit{dom}\ (\mathit{knowsM}\ t\ u)) \wedge \\
&\quad (\forall u\ h\ i\ m. \\
&\qquad \mathit{knowsI}\ s\ u\ h = \mathit{Some}\ i \wedge \mathit{knowsM}\ t\ u\ h = \mathit{Some}\ m \\
&\qquad\quad \longrightarrow (i, m) \in \mathit{message}\ s\ \mathit{i2t}) \}
\end{aligned}$$

The bisimulation relation itself is the union over all family members, *i.e.*, a second-order property. Since indices and tags are allocated dynamically, but not every index is associated with a tag (*e.g.*, payload data is untagged), the parameter *i2t* cannot be determined statically.

4.6. Modeling protocols

We now construct a protocol context for our formalizations of the BPW model, *cf.* Figure 3, in the form of a generic protocol component for each user. Afterwards we instantiate these components to a concrete protocol. From Theorem 4.1 and the compositionality of BRSIM, any protocol interpreted in the term-based model reactively simulates its interpretation in the DAG-based model.

Protocol components. The global state extends the BPW model state with the local state of each protocol component and the trace observed at the user interface. The trace is a list of pairs of a user name and an input or output event.

```
datatype (ι, o) uio = uIn ι | uOut o    −− user i/o

record (ι, o, δ, σ) gState = δ mLibState +
  loc :: user ⇒ σ                       −− local state
  trace :: (user × (ι, o) uio) list     −− user i/o trace
```

Our setup is polymorphic in the type δ of payload data (from the BPW model), the type σ of local states, as well as ι and *o*, the types of user input and output. Concrete protocols later instantiate these type parameters to concrete types.

The protocol component interface consists of a user and a network input handler, specifying how the component reacts to user and network input. Each handler may manipulate the component's local state to record session-specific protocol data and may produce output for either the user or the network (cf. Figure 3). A protocol is then defined as a function from users to protocol components.

```
datatype o proto_out = pToUser o | pToNet party hnd

record (ι, o, δ, σ) proto_comp =
  proto_user_handler :: ι ⇒
    (o proto_out, (ι, o, δ, σ) gState) S
  proto_net_handler :: [party, hnd] ⇒
    (o proto_out, (ι, o, δ, σ) gState) S

types (ι, o, δ, σ) protocol =
  user ⇒ (ι, o, δ, σ) proto_comp
```

Note that we do not explicitly represent protocol sessions. We leave session handling to the protocol implementation. Typically, each protocol session is initiated and terminated by user interaction, possibly with additional user interaction during the session. These user I/O events are recorded in the history variable *trace*.

Complete system. We compose the BPW model with the protocol components yielding the complete system. The system has two kinds of interface functions: (1) the system user and network handlers and (2) the 14 local adversary functions of the BPW model. Here we only present the system user and network handlers, whose types are:

```
sys_user_handler :: (ι, o, δ, σ) protocol ⇒
  [user, ι] ⇒ (o sys_out, (ι, o, δ, σ) gState) S
sys_net_handler :: (ι, o, δ, σ) protocol ⇒
  [party, user, hnd] ⇒ (o sys_out, (ι, o, δ, σ) gState) S
```

Both handlers produce a system output of type *o sys_out*, the system-level version of type *o proto_out*. The user handler takes an input from the user (of type ι), while the network handler takes the presumed sender, the recipient, and a handle to the message as an argument.

To illustrate the message flow through the system (*cf.* Figure 3), let us consider the system network handler.

```
sys_net_handler proto v u h ≡
  do hu ← adv_send_i v u h;        −− u receives msg
```

```
do pout ← proto_net_handler (proto u) v hu;
case pout of
  pToUser uom ⇒
    do log (u, uOut uom);          −− log output
    return (sToUser u uom)         −− user output
| pToNet vd hd ⇒                   −− send reply
    do ha ← send_i u vd hd;
    return (sToNet u vd ha)
```

This handler uses the BPW-model send functions for the user (*send_i*) and the adversary (*adv_send_i*), which transfer a message from a user's to the adversary's knowledge map or vice versa. The handler's input is a message *h* from the adversary, sent to the user *u* (pretending to be party *v*) using the adversary send function. The resulting handle *hu* for *u*, is fed into *u*'s protocol network handler. The handler's output is either intended for user *u*, in which case the output is logged by the observer and returned to *u*, or it is a reply message handle *hd* intended for party *vd* that is sent back to the network (adversary) using the user send function.

We specify a concrete protocol by providing its user and network handlers. This determines concrete types for user I/O, payload data, and the local state of protocol components, instantiating the type variables ι, o, δ, and σ. Afterwards, we can specify and verify protocol properties.

Example 4.2 (NSL protocol)**.** We model the well-known three message version of the Needham-Schroeder-Lowe authentication protocol [11].

$$\begin{array}{lll} \text{NSL1.} & u \to v: & \{N_u, u\}_{PK(v)} \\ \text{NSL2.} & v \to u: & \{N_u, N_v, v\}_{PK(u)} \\ \text{NSL3.} & u \to v: & \{N_v\}_{PK(v)} \end{array}$$

We specify this protocol by defining a protocol component for each user. Each such component P_u records the set of nonces it generates in protocol sessions with the party *v* in the local state variable *nonces*, under *v*'s name.

```
record ustate = nonces :: party ⇒ hnd set
```

We can initiate a protocol run by indicating the name of the responder. The protocol is terminated by returning the name of the initiator to the responder. Thus, both user input and output are of type *user*. Moreover, the only payload data used in the NSL protocol are user names. Therefore, we use an abbreviation for the states of the protocol.

```
types NSLstate = (user, user, user, ustate) gState
```

We then specify the NSL protocol by instantiating the user and network handlers.

```
NSL :: (user, user, user, ustate) protocol
NSL u ≡ (|
  proto_user_handler = λv.          −− initiate with v
    do nh ← gen_add_nonce u v;      −− fresh nonce
    do uih ← store (User u) u;
    do mh ← pair (User u) (nh, uih)
    do emh ← encrypt (User u) (pke (User u) v) mh;
    return (pToNet (u, v, emh))     −− send 1st msg

  proto_net_handler = λv emh.       −− reply to messages
    do pm ← parse_msg u v emh;
    case pm of
      msg1 vnh vid ⇒ mk_msg2 u v vnh      −− 2nd msg
    | msg2 unh vnh vid ⇒ mk_msg3 u v vnh  −− 3rd msg
    | msg3 vnh ⇒ return (pToUser v) |)    −− terminate
```

The user handler for user *u* initiates a protocol session with party *v* by constructing the first protocol message. This can be done any time and thus arbitrarily many sessions are possible. Here, *pke* (*User u*) *v* denotes the handle by which *u* refers to *v*'s public key. The statement *gen_add_nonce u v* generates a fresh nonce and adds it to *nonces* (*loc s u*) *v*, i.e. the nonces used by *u* in sessions with *v*. The subsequent calls incrementally construct the message.

The network handler takes the sender's name *v* and a message handle *emh*, and parses the message. Depending on the result, it either produces a reply message or terminates the protocol, returning the name of the (presumably) authenticated user. Besides parsing messages, the function *parse_msg* ensures correct message sequencing by verifying that all prerequisites for replying to messages are satisfied (*e.g.*, message NSL2 contains the nonce sent in NSL1).

5. Role-based protocol models

We will now focus on two models and their relationship: the term-based protocol model from Section 4.6 and a more high-level model that we call the abstract protocol model. For the purpose of comparison we will refer to the former as the *concrete* model and the latter as the *abstract* model.

We compare them on a particular class of protocols defined using *role-based protocol specifications*. This not only provides a basis for comparing the models, it also facilitates the concise, high-level, specification of protocols themselves. A protocol is specified by a set of roles (*e.g.*, initiator, responder, or key server), where each role is a sequence of pairs of input and output *patterns*, describing which input messages are accepted by a user in the role and the response. In our formalization, we simplify matters by considering the class of protocols using only public-key encryption, a single set of pre-distributed asymmetric key pairs, and no key generation. However, our framework can accommodate signatures, key generation, and symmetric cryptography.

5.1. Role-based protocol specifications

We declare a type *role* of roles and a type *mvar* of message variables, each isomorphic to the natural numbers. Patterns are described by an inductive data type.

```
datatype pat = pVar mvar | pRole role |
               pPair pat pat | pEnc role pat
```

The constructors represent patterns for message variables, protocol roles, pairs, and ciphertexts (where *role* refers to the role's public or private key). Patterns are used for pattern matching and message construction.

A *role script* consists of a list of *protocol steps*, each of which is described by a pair of input and output role events. A *role event* is a pattern associated with either the user or adversary interface, indicating whether a message is exchanged with an honest user or the adversary, respectively (*cf.* Figure 2). In *rNet r p*, the role *r* indicates the intended source or destination role. A *protocol specification* maps a protocol role to a role specification, which consist of a variable typing context and a role script.

```
datatype roleEvent = rUsr pat | rNet role pat
types roleScript = (roleEvent × roleEvent) list

record roleSpec =
  contxt :: mvar ⇒ msgType
  script :: roleScript

types protoSpec = role ⇀ roleSpec
```

A well-formedness condition ensures role executability and restricts the messages exchanged at the user interface to non-cryptographic messages, namely pairs and roles.

Example 5.1 (NSL protocol)**.** Below is the role-based specification of the NSL protocol. We use the notation $\{\!|p|\!\}_R$ for ciphertext patterns, where p is a non-empty list of patterns constructed from pair patterns. We omit variable and role constructors for the sake of readability.

```
[ I ↦ (| contxt = ctxtAny(nI := mt_nonce, nR := mt_nonce),
         script = [ (rUsr R,                    rNet R {|nI, I|}_R),
                    (rNet R {|nI, nR, R|}_I, rNet R {|nR|}_R) ] |),
  R ↦ (| contxt = ctxtAny(nI := mt_nonce, nR := mt_nonce),
         script = [ (rNet I {|nI, I|}_R, rNet I {|nI, nR, R|}_I),
                    (rNet I {|nR|}_R,    rUsr I)  ] |) ]
```

Both initiator and responder roles declare two nonce variables, *nI* and *nR*, and are composed of two steps. Consider the initiator role. First, the initiator requires the name of the desired responder and sends an encrypted message containing a fresh nonce *nI* and his own name to the responder. Second, the initiator expects a ciphertext from the network, which it decrypts and matches against *nI*, a responder nonce *nR*, and the responder's name. Finally, the initiator re-encrypts the nonce *nR* for the responder and sends it back to the network. The responder role *R* is analogous, but its final output is the name of the presumed initiator.

5.2. Generic protocol interpretation

Our plan (recall Figure 2) is to interpret role-based protocol specifications within two different models and afterwards to relate these interpretations. Therefore, we have factored much of their operational semantics into a generic protocol interpreter, which we introduce here. In Sections 5.3 and 5.4, we instantiate this interpreter twice to obtain these two different models. This factorization makes it much easier to relate the states and actions of the two models and thereby greatly facilitates the BRSIM proof in Section 5.5. We first define threads and then how the generic interpreter executes protocol steps.

Environments and threads. An *environment* is an interpretation of the message variables and roles appearing in role scripts by messages and party names, respectively. Let *aMsg* = *party msg* be the type of messages carrying party names as payload data. The polymorphic type μ of network messages will be instantiated to handles, which refer to messages of type *aMsg*, on the concrete side and directly to messages of type *aMsg* on the abstract side.

```
record μ env =
  imsg :: mvar ⇀ μ        −− message interpretation
  iusr :: role ⇀ party    −− role interpretation

record μ thread =
  rol :: role             −− role executed by thread
  scr :: roleScript       −− remaining role script
  env :: μ env            −− interpretation
```

A *thread* is an executing role instance, represented as a record with three fields: the executed role, a role script corresponding to the protocol script still to be executed, and an environment assigning parties to roles and messages to the variables instantiated so far.

Generic protocol interpreter. The protocol interpreter is an event handler that, given an input event, executes a protocol step and produces an output event. Input/output events associate messages with a (user/network) interface.

```
datatype μ ioEvent = tUsr aMsg | tNet party μ
```

Note that the type of messages exchanged with the honest users is *aMsg* in both interpreter instantiations. Only the type of network messages μ is variable. The party indicated in a network event specifies the presumed source (intended destination) of an input (or output) message. Next, let us consider the definition of the protocol interpreter.

```
proto_interpreter par proto u tid iev ≡
  do thd ← lookup_thread (get_thds par) u tid;
  do (thd', oev) ← proto_step proto (match par)
                                (compos par) u iev thd;
  do update_threads proto (upd_thds par) u tid thd';
  return oev
```

The interpreter is parametrized by a *static parameter record* (*par*) consisting of functions for pattern matching (*match par*) and message composition (*compos par*) as well

as accessing (*get_thds par*) and updating (*upd_thds par*) the threads of a given user. A protocol interpretation step consists of three phases. First, the interpreter checks the existence of *u*'s thread, identified by *tid*. Second, it executes the protocol step itself, *i.e.*, it processes the input event *iev* and generates an output event *oev* according to the role script of the scheduled thread *thd*. Finally, it updates the state by replacing the executed thread identified by *tid* with its updated version *thd'* and returns the output event *oev*.

Let us now focus on the protocol step.

```
proto_step proto match compos u iev thd ≡
  case scr thd of
    [] ⇒ throw ()
  | (ire, ore) # rs ⇒
      do theta ← ev_match match (env thd) u iev ire;
      do (oev, rho) ← ev_compos compos theta u ore;
      return (thd(| scr := rs, env := rho |), oev)
```

A terminated thread (one with an empty role script) throws an exception. Otherwise the event pattern pair (*ire*, *ore*) at the role script's head is extracted and the input event *iev* is matched against the input role event *ire*. If successful, this results in an environment *theta*, which extends the thread's original environment (*env thd*) with bindings for messages matching previously undefined variables in the input pattern. The event composition algorithm takes *theta* and the output role event *ore* and returns an output event and an updated environment *rho*, where unassigned nonce variables are bound to fresh nonces. Finally, the protocol step returns a pair consisting of the updated thread and the generated output event. In the updated thread, denoted by *thd*(| ... |), the protocol step just executed is removed from the remaining role script and *rho* becomes the new environment.

5.3. Term-based BPW as instance

We now present the term-based model as a concrete instance. Here, as well as the forthcoming abstract instance, we define the model's state and the behavior of users and the adversary. The users' behavior is determined by instantiating the generic protocol interpreter with an appropriate static parameter record. The concrete instantiation interprets role-based protocol specifications in the term-based BPW model, while the abstract one interprets them directly without BPW model. Both models embed the protocol interpreter into two system-level interface functions, namely the *user* and *network event handlers*. These add protocol-independent processing steps such as logging events in a trace or, in the case of the term-based model, sending messages from a user to the adversary and vice versa.

The behavior of the concrete adversary is determined by the BPW model, while we define the abstract adversary in terms of Dolev-Yao-style message-derivation operators.

State. In Section 4.6, we described the concrete model. Here we instantiate the four type variables parametrizing its state: the types of user input and output both become *aMsg*, payload data becomes *party* and the local state of each protocol component becomes a partial map from thread identifiers to threads. To simplify the presentation, we repeat here the fully instantiated state, which we call *cState*.

```
datatype uio = uIn aMsg | uOut aMsg   −− user I/O

record cState = party mLibState +
  loc :: user ⇒ TID ⇀ hnd thread      −− user's threads
  trace :: (user × uio) list          −− user I/O trace
```

As protocol components refer to messages using handles, the thread environments map variables to handles.

Message handling. The pattern matching and message composition operations are built from BPW-model interface functions. Here, we focus on the pattern-matching algorithm and show the fragment dealing with decryption.

```
cl_match kt rho u eh (pEnc r p) =
  do h ← decrypt (User u) (ske kt (iusr rho r)) eh;
  cl_match kt rho u h p
```

We call the BPW-model decryption function with the presumed ciphertext at handle *eh* using the secret key for role *r* at the handle (*ske kt* (*iusr rho r*)). When successful, we get a cleartext handle *h*, which we recursively match against the cleartext pattern *p*. The cases for roles (matched against party names) and pair patterns use the BPW-model functions in similar ways. When matching a message variable *v* with a handle *h*, either the environment already maps *v* to *h* or it is undefined at *v* and updated with such a mapping.

System interface and invariants. As described in Section 4.6, the system-level user and network handlers call the send functions of the term-based BPW model to transmit messages from the protocol interpreter (*i.e.*, some user *u*'s knowledge map) to the adversary (*i.e.*, the adversary's knowledge map) and vice versa. Moreover, these two handlers log I/O with the users in the *trace* variable. The rest of the interface is composed of the 14 BPW-model adversary functions for message construction and access. The main invariant of the concrete model, *wf_cThreadEnv*, states that handles occurring in the thread environments correspond to messages defined in the respective user's knowledge map.

5.4. Abstract protocol model as instance

While the role-based specifications relieve the user from specifying protocols as low-level imperative programs, the fine granularity of the BPW-model interface functions is still apparent (due to their side effects) when reasoning about protocols in the concrete model. Hence, we define

an abstract protocol model, independent of the BPW model, based on high-level message manipulation operations and a concisely specified adversary.

State. In the abstract protocol model, the type variable μ is instantiated to the type *aMsg* of messages carrying party names as payload data. The abstract state contains for each user a partial map from thread identifiers to threads and a global event trace recording the system history.

```
datatype event = eLocal party aMsg
  | eProtoInp user (aMsg ioEvent)
  | eProtoOut user (aMsg ioEvent)

record aState =
  threads :: user ⇒ TID ⇀ aMsg thread      −− threads
  trace :: event list                      −− execution trace
```

An abstract trace can contain input, output, and local events. In contrast to the concrete model, it records I/O events at both the user and the adversary interface. Local events are currently used only by the adversary to record a message in the trace. The adversary can observe the set of all messages occurring in *network* I/O events and in his own local events, denoted by (*sees Adv* (*trace t*)) for an abstract state *t*. As we shall see, this set corresponds exactly to the set of messages known to the concrete adversary.

Message handling. As in the concrete case, we focus on pattern matching, which is done using Isabelle/HOL's matching primitives. Here is the case for decryption.

```
amatch kt rho (mEncv tg k m) (pEnc r p) =
  do enforceb (kt (iusr rho r) = k); −− check key
  amatch kt rho m p                  −− match plaintext
```

We simply check that the key *k* in the ciphertext corresponds to the key assigned to the role *r*, *i.e.*, (*kt* (*iusr rho r*)). If so, we recursively match the plaintext *m* against the pattern *p*. The cases for the other constructors are defined analogously. Note that, unlike in the concrete model, abstract message manipulation is direct (*i.e.*, without using handles), atomic (*i.e.*, not composed of lower-level message operations), and free of side effects. This decisively simplifies reasoning about protocols in the abstract model, *e.g.*, because these operations trivially preserve all invariants.

System interface and invariants. The second important simplification in our abstract protocol model concerns the adversary. We collapse the 14 concrete adversary interface functions for message manipulation into a single one.

```
adv_deriveA m ≡
  do enforce (λt. m ∈ synth (analyze (sees Adv (trace t))));
  logA (eLocal Adv m)
```

This function uses the standard operations *analyze* and *synth* on sets of messages (see [14]). It ensures that the adversary can derive the message *m* and logs it as a local event in the event trace. Failure to derive *m* leads to an exception. The system-level user and network handlers invoke the protocol interpreter. Before and after this call, the input and output events are logged in the trace, respectively. Moreover, messages from the network are subject to the above derivability check. The abstract adversary (unlike the concrete one) derives messages without producing side effects.

The main invariant of the abstract protocol model describes the well-formedness of the user threads. For example, the thread environment defines exactly the set of variables occurring in the already executed portion of the role.

5.5. Blackbox reactive simulatability proof

Let $\mathsf{A}(p)$ (respectively $\mathsf{C}(p)$) denote the abstract (respectively concrete) model instantiated with a protocol p, *i.e.*, where the protocol interpreter *proto_interpreter* executes p. Our main result is that the concrete model is a cryptographically-sound refinement of the abstract one.

Theorem 5.2. *Let p be a well-formed, role-based protocol specification of type protoSpec. Then $\mathsf{C}(p) \leq \mathsf{A}(p)$.*

We now sketch some of the ideas behind our proof of the theorem including the simulator and bisimulation relation.

The simulator. The simulator emulates the concrete model's adversary interface using the abstract adversary's interface functions. It is largely a replication of the adversary part of the term-based BPW model. Thus, its state is a knowledge map, associating handles with messages.

```
record simState = aState +
  iknows :: hnd ⇀ aMsg     −− adversary knowledge map
```

The simulator provides all local functions of the concrete model adversary, which are almost identical to their original versions. The main difference is that when the simulator creates a new message *m*, it calls *adv_derive m* to ensure its derivability and to log it as a local event in the trace.

The simulator also provides a user and an adversary send function. These functions translate between handles and messages using the adversary's knowledge map *iknows*, rather than between the handles of the honest users and those of the adversary, as in the concrete model. We compose abstract protocol system and the simulator by connecting these send functions to the user and network handlers of the abstract protocol system. We call the resulting system $\mathsf{AS}(p)$. The main invariant of $\mathsf{AS}(p)$ states that the set of messages observed by the adversary in the trace equals the range of the knowledge map *iknows* and is defined by *iknowsTraced* ≡ {*t* | *ran* (*iknows t*) = *sees Adv* (*trace t*)}. The logging of local events for the messages derived by the simulator is essential for this invariant.

The bisimulation relation. The bisimulation relation *Rsim* between the states of the concrete model and system $\mathsf{AS}(p)$ consists of four conjuncts, of which we present three.

$$\{(s,\ t)\ \mid\ \forall\, u.\ abs \circ (loc\ s\ u) = threads\ \ t\ \ u \quad \textsf{where}\ abs = c2aThread\ (knowsM\ s\ (User\ u))\}$$
$$\{(s,\ t)\ \mid\ knowsM\ s\ Adv = iknows\ t\}$$
$$\{(s,\ t)\ \mid\ cState\ .\ trace\ \ s\ =\ trace\_proj\ \ (aState\ .\ trace\ \ t)\}$$

The first conjunct relates threads. For each user u, composing the concrete thread map (of type $TID \rightharpoonup hnd\ thread$) with the thread abstraction function $c2aThread\ \ (knowsM\ s\ (User\ u))$ (of type $hnd\ thread \Rightarrow aMsg\ thread$) yields the abstract thread map (of type $TID \rightharpoonup aMsg\ thread$). The thread abstraction function composes the concrete thread environment with user u's knowledge map to yield an abstract environment. The invariant *wf_cThreadEnv* (Section 5.3) ensures that the handles in the concrete thread environments are defined in the user's knowledge map. The second conjunct relates intruder knowledge. The adversary knowledge map in the simulator equals the one in the concrete model. From invariant *iknowsTraced*, it then follows that the set of messages the concrete adversary knows are the adversary-observable messages in the abstract trace. The third conjunct relates traces. The projection of the abstract trace to the user interface events (I/O events labeled by *tUsr*), yields the observer trace in the concrete model.

With this relation at hand, we establish the following lemma, from which our theorem immediately follows.

Lemma 5.3. *For all well-formed p of type protoSpec, Rsim is a bisimulation between $\mathsf{C}(p)$ and $\mathsf{AS}(p)$.*

Our relational Hoare logic proof relies on two types of auxiliary results for the two systems: (1) the invariants described in Sections 5.3 and 5.4, and (2) logical properties of pattern matching and message composition expressed in monadic Hoare logic. These results about the abstract protocol system are not only needed for meta-level reasoning *about* the models (*i.e.*, the BRSIM abstraction proof), they are also essential ingredients for reasoning *within* the abstract protocol model, *i.e.*, carrying out protocol security proofs (*cf.* Section 6).

6. Case study: NSL protocol

In this section, we describe the verification of the NSL protocol in our three formalized models. The term-based and role-based specifications of this protocol have already been presented in Examples 4.2 and 5.1. The DAG-based specification is almost identical to the term-based one. Rather than presenting proof details, we explain how the increased abstraction of our models dramatically improves our property specification and proof techniques.

6.1. DAG-based BPW model

Our attempt to verify the NSL protocol in the DAG-based model failed. We abandoned the proof of nonce secrecy when its length exceeded 80 pages of PDF documentation (ca. 4k lines of Isabelle/HOL source). There were two main problems. First, the imperative representation of messages as DAGs complicated the proofs. Second, our property specifications (*e.g.*, of nonce secrecy) were ad hoc and could not employ general notions and results, *e.g.*, for the closure operators *parts* and *analyze*.

These two problems thwarted proof automation. The WP calculus usually produced prohibitively large expressions. Hence, proving even simple properties of the BPW-model interface functions necessitated interactive Hoare logic proofs and required complicated intermediate assertions. As a result, the proof quickly became infeasible.

6.2. Term-based BPW model

Invariants. The main property we proved is that the responder authenticates the initiator. This is formulated as an invariant of the observed user I/O trace and therefore transfers to the cryptographic implementation.

$$authRI \equiv \{s\ \mid\ \forall\, u\ v.\ \ (v,\ uOut\ u) \in set\ \ (trace\ \ s) \longrightarrow (u,\ uIn\ \ v) \in set\ \ (trace\ \ s)\}$$

Here it is sufficient just to consider $set\ \ (trace\ \ s)$, the unordered *set* of I/O events at state s.

The proof of this invariant depends on the auxiliary invariants listed in Figure 4, along with their dependencies. We focus our discussion on some higher-level invariants. The invariants *uniqueInitNonce* and *uniqueResponseNonce* express that the initiator nonce in message NSL1 and the responder nonce in NSL2 uniquely determine all the other fields of the respective message. Based on these invariants, we prove that the protocol nonces remain secret (invariant *nonceSecrecy*) between the protocol participants.

$$nonceSecrecy \equiv \{s\ \mid\ \ \forall\, u\ v\ n.\ \ n \in Nonces\ s\ u\ v \longrightarrow secret\ \ s\ \ (mNonce\ n)\ \{User\ u,\ User\ v\}\}$$

Here, *Nonces* returns the set of nonces denoted by handles stored in the variable *nonces* (see Section 4.2) and secrecy was already defined in Section 4.4. Nonce secrecy transfers to the cryptographic implementation [2]. Finally, the authentication property *authRI* is derived from the conjunction of four auxiliary invariants, *beforeCommit*, *beforeM3*, *beforeM2*, and *beforeM1*, each of these going one message back in the protocol (dashed line in Figure 4).

Verification. Our verification of the invariant *authRI* took 130 pages of PDF documentation (6.5k lines of source). Much of this stems from the large number (17) of interface functions in the model and the many auxiliary preservation

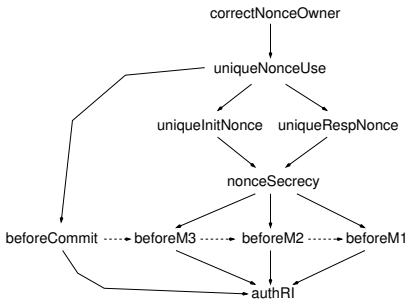

Figure 4. NSL invariants (BPW version)

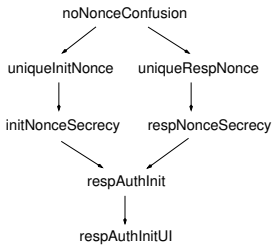

Figure 5. NSL invariants (APM version)

results that must be proved for each invariant for reuse in later invariant proofs.

The term-based model dramatically improved proof automation. We could use the WP calculus in most proofs at the level of the BPW-model interface functions. For the protocol handlers, we could often automate Hoare logic proofs, by pulling the precondition over all calls to the BPW-model interface functions. Only when reasoning about the complete system with the send functions did we need to resort to interactive Hoare logic proofs, since reasoning about the user send function required inventing an additional precondition. This improved automation has two sources: (1) the simplified BPW-model state using message terms, and (2) the use of systematic property specifications based on the message closure operators *parts* and *analyze*.

While the proof was manageable, it was still quite detailed and 1-2 orders of magnitude longer than Paulson's proof (which fits on 3.5 PDF pages). This is mainly due to the fine granularity and side-effect behavior of the BPW-model interface functions and the lack of general protocol-independent results supporting the proofs. Indeed, due to the absence of high-level message manipulation operations, it is hardly possible to obtain such results. We designed the abstract protocol model to overcome these problems.

6.3. Abstract Protocol Model

Invariants. In this model, we express the user I/O-interface property of responder authentication as follows.

$$\begin{aligned}&\mathit{respAuthInitUI} \equiv \{s \mid \forall\, u\ v.\\ &\quad \mathit{eProtoOut}\ v\ (\mathit{tUsr}\ (\mathit{mParty}\ (\mathit{User}\ u))) \in \mathit{set}\ (\mathit{trace}\ s)\\ &\quad \longrightarrow \mathit{eProtoInp}\ u\ (\mathit{tUsr}\ (\mathit{mParty}\ (\mathit{User}\ v))) \in \mathit{set}\,(\mathit{trace}\ s)\}\end{aligned}$$

The main auxiliary invariants needed to prove that *respAuthInitUI* is an invariant are depicted in Figure 5.

The principal invariant used in this proof is the following authentication property, formulated in terms of the observations of the adversary and the thread state.

$$\begin{aligned}&\mathit{respAuthInit} \equiv \{s \mid \forall\, u\ v\ \mathit{ni}\ \mathit{nr}\ \mathit{tg2}\ \mathit{tg3}.\\ &\quad (\mathit{mEncv}\ \mathit{tg3}\ (\mathit{aKt}\ (\mathit{User}\ v))\ (\mathit{mNonce}\ \mathit{nr})\\ &\quad \in \mathit{parts}\ (\mathit{seesAdv}\ s) \wedge\\ &\quad (\mathit{eProtoOut}\ v\ (\mathit{tNet}\ (\mathit{User}\ u)\\ &\qquad (\mathit{mEncv}\ \mathit{tg2}\ (\mathit{aKt}\ (\mathit{User}\ u))\\ &\qquad\quad \{\!|\ \mathit{mNonce}\ \mathit{ni}, \mathit{mNonce}\ \mathit{nr}, \mathit{mParty}\ (\mathit{User}\ v)\ |\!\}\,)))\\ &\quad \in \mathit{set}\ (\mathit{trace}\ s)) \longrightarrow\\ &\qquad \exists\ \mathit{tid}\ \mathit{thd}.\ \mathit{threads}\ s\ u\ \mathit{tid} = \mathit{Some}\ \mathit{thd}\\ &\qquad \wedge\ \mathit{imsg}\ (\mathit{env}\ \mathit{thd})\ \mathit{nI} = \mathit{Some}\ (\mathit{mNonce}\ \mathit{ni})\\ &\qquad \wedge\ \mathit{iusr}\ (\mathit{env}\ \mathit{thd})\ R = \mathit{Some}\ (\mathit{User}\ v)\ \wedge\ \mathit{rol}\ \mathit{thd} = I\ \}\end{aligned}$$

Here $\{\!|\ ...\ |\!\}$ denotes message tupling. This invariant states that whenever a message of shape NSL3 appears on the network and the responder v has sent a corresponding message of shape NSL2 containing the nonce ni, then there is an initiator thread in a session with v that generated ni. Note that, unlike *respAuthInitUI*, this invariant also specifies agreement on the initiator nonce. However, its proof requires three additional precedence invariants linking thread states to related trace events. These are not shown in Figure 5, as they are independent from all other invariants.

The invariant *respAuthInit* depends on the secrecy of the nonces. In this model, a separate invariant specifies the secrecy of each nonce. The one for the initiator nonce is:

$$\begin{aligned}&\mathit{initNonceSecrecy} \equiv \{s \mid \forall\, u\ \mathit{tg}\ v\ \mathit{ni}.\\ &\quad \mathit{eProtoOut}\ u\ (\mathit{tNet}\ (\mathit{User}\ v)\\ &\qquad\qquad (\mathit{mEncv}\ \mathit{tg}\ (\mathit{aKt}\ (\mathit{User}\ v))\\ &\qquad\qquad\quad \{\!|\ \mathit{mNonce}\ \mathit{ni}, \mathit{mParty}\ (\mathit{User}\ u)\ |\!\}))\\ &\quad \in \mathit{set}\ (\mathit{trace}\ s)\\ &\quad \longrightarrow \mathit{mNonce}\ \mathit{ni} \notin \mathit{analyze}\ (\mathit{sees}\ \mathit{Adv}\ (\mathit{trace}\ s))\ \}\end{aligned}$$

The invariants *uniqueInitNonce* and *uniqueRespNonce* are similar to the respective invariants of the concrete model.

Verification. Our verification of invariant *respAuthInitUI* within the abstract protocol model still takes about 80 pages of PDF documentation (4k lines of source). However, the proof of each invariant is reduced to two *core lemmas*, one for the protocol steps on the user side and one for the adversary, which take up only about 10 pages or 13% of this space. The remaining proofs are completely straightforward and could be automated. In the following, we elaborate on the reduction of the proofs to these core lemmas. We consider the user and adversary sides in turn.

On the user side, we reduce preservation results (formulated in Hoare logic) for the user and network handlers to a HOL assertion about the protocol step, which is the user-

side core lemma. Here is a typical instance of such a lemma, for the case of initiator nonce secrecy.

lemma *initNonceSecrecy__protocol_step*:
$s \in proto\_step\_spec\ NSLproto\ u\ iev\ thda\ oev\ thdb$
$\wedge\ s \in initNonceSecrecy\ \wedge\ wf\_aThread\ NSLproto\ u\ thda$
$\wedge\ eProtoInp\ u\ iev \in set\ (trace\ s)$
$\longrightarrow s(\!|\ trace := eProtoOut\ u\ oev\ \#\ (trace\ s)\ |\!)$
$\in initNonceSecrecy$

The premises of this lemma are proved as post-conditions of the protocol step and its conclusion is needed as a pre-condition for logging the output event of the protocol step.

The first premise, *proto_step_spec*, specifies the protocol step where *iev* is the input event, *thda* is user *u*'s thread to be executed, *oev* is the output event resulting from the step, and *thdb* is the updated thread. This is a general, previously proved post-condition that characterizes the protocol step. For instance, it states that the output event *oev* is an instance of the current output role event of the thread's script and names the freshly generated nonces. The other premises are: initiator nonce secrecy, which is trivially preserved by the side-effect-free protocol step; a thread well-formedness condition; and the statement that *iev* occurs in an input trace event. The premises include three more auxiliary invariants, which we have omitted above.

The proof of user-side core lemmas is based on another lemma for the case analysis on the role and position in the role script. We prove such a lemma once for each protocol. Here is the one for NSL.

lemma *NSLproto_step_casesD*:
$(s \in proto\_step\_spec\ NSLproto\ u\ iev\ thda\ oev\ thdb$
$\wedge\ wf\_aThread\ NSLproto\ u\ thda)$
$\longrightarrow s \in NSLproto\_event\_cases\ u\ iev\ thda\ oev\ thdb$

Its conclusion is defined as a large disjunction with one disjunct for each role and script position, which specializes the general information in the protocol step specification to the concrete protocol step being taken. For example, the following case describes the first step in the NSL initiator role.

$(\exists v\ tg\ ni.\ rol\ thdb = I$
$\wedge\ iusr\ (env\ thdb)\ R = Some\ v$
$\wedge\ imsg\ (env\ thdb)\ nI = Some\ (mNonce\ ni) \quad (*\ after\ *)$
$\wedge\ imsg\ (env\ thdb)\ nR = None$
$\wedge\ imsg\ (env\ thda) = empty \quad (*\ before\ *)$
$\wedge\ iev = tUsr\ (mParty\ v)$
$\wedge\ oev = tNet\ v\ (mEncv\ tg\ (aKt\ v)$
$\{\!|\ mNonce\ ni, mParty\ (User\ u)\ |\!\})$
$\wedge\ mNonce\ ni \notin parts\ (seesAdv\ s))$

This fixes all the relevant bindings in the environment, describes the shapes of the input and output events, and states that the nonce *ni* is freshly generated in this step.

The adversary-side core lemma for initiator nonce secrecy is as follows.

lemma *initNonceSecrecy__adversary_derivable*:
$(s \in initNonceSecrecy\ \wedge\ s \in adv\_derivable\ m)$
$\longrightarrow s(\!|\ trace := eProtoInp\ u\ (tNet\ v\ m)\ \#\ (trace\ s)\ |\!)$
$\in initNonceSecrecy$

This lemma states that adding a network input event to the trace containing a message derived by the adversary preserves the invariant. We prove this using standard results from the theories of *synth* and *analyze* [14].

Discussion. By proving a number of strong, general results about protocol steps and the adversary, we reduce invariant proofs to two core lemmas, whose complexity is directly comparable to the inductive cases arising in Paulson's invariant proofs. For the adversary, it would be possible to adapt Paulson's specialized tactics to our setting. Moreover, the other, completely straightforward parts of the invariant proof, which we currently handle using an invariant proof template, could be automatically proved. While the invariants shown in Figure 5 closely reflect Paulson's, one difference with his proof of this protocol is that we need three additional precedence invariants (not shown in Figure 5) that glue together the thread state and the trace. However, these are local invariants about a single thread and should be derivable from general results about the protocol model.

In conclusion, our results provide strong evidence that cryptographically-sound reasoning about security protocols is indeed possible with a complexity comparable to more standard, not necessarily cryptographically-sound models. This is one of the main results of our work.

7. Related work and conclusions

Related work. Understanding relationships between cryptographically-sound models is an active research area. There are two established techniques to show the cryptographic soundness of a Dolev-Yao model: mapping-based [12, 8] and simulatability-based [15, 5]. These notions preserve trace properties and certain forms of secrecy from the symbolic to the cryptographic world. Simulatability-based soundness additionally provides composition guarantees.

Micciancio *et al.* [12] present a mapping-sound Dolev-Yao model with public-key encryption, which is extended in [8] with signatures and ciphertext forwarding. In this context, Cortier *et al.* [7] show that randomness tags in ciphertexts can be safely omitted for a restricted class of property specifications that cannot express the equality between ciphertexts and includes secrecy and authentication properties. Although the setup is different, this work is closest to ours. They also further abstract non-standard aspects of a cryptographically-sound Dolev-Yao model to enable effective tool-supported protocol analysis.

Canetti and Herzog [6] consider mutual authentication and key exchange protocols based on public-key encryption in the UC framework. They reduce the proof that such

Module	Theories	Pages	Lines	%
Modeling & verification tools	9	54	2.7k	7
DAG-based model + BRSIM	10	119	6k	16
Term-based model	17	96	4.8k	13
Term-based NSL	14	136	6.8k	18
APM + BRSIM	31	266	13.3k	35
APM-based NSL	9	83	4.2k	11
Total	90	754	37.8k	100

Table 1. Isabelle/HOL Statistics

a protocol is UC-secure to showing that a Dolev-Yao-style symbolic abstraction of the protocol satisfies a certain symbolic property. They then use the ProVerif tool to symbolically analyze secrecy aspects of the NSL protocol. We share with their work the simulatability-based setup, but our starting point is not a computational model, but a very general Dolev-Yao-style model (the BPW model), which we further abstract and specialize to role-based protocols. The BPW model also includes signatures, symmetric encryption, and MACs, which can easily be added to our formal model.

Laud [10] has designed a type system for secrecy aspects of security protocols specified in a process calculus tailored to the BPW model. He shows that typeable protocols keep their payload inputs cryptographically secret. Backes and Laud [1] automate such secrecy proofs using message flow analysis. While they only consider the payload secrecy property, their language covers a large class of protocols definable in the BPW model.

Conclusions and future work. Overall, we have shown that it is possible to close the gap between a very general, cryptographically-sound model and standard Dolev-Yao models. Abstraction is the key and with the right modeling tools one can formally justify the abstractions as well. Of course this required some work as indicated in Table 1.

The gap has both semantic and deductive aspects. First, through our work, we gain a better understanding of sufficient semantic conditions for cryptographic soundness. Although our abstractions are substantial, our abstract protocol model still contains some non-standard aspects, *e.g.*, randomness tags, invalid ciphertexts, and message length restrictions. Further investigation is needed to determine which of these are essential, *cf.* Cortier *et al.* [7].

The second aspect concerns the complexity of deduction. By reducing the models' complexity we have gained the ability to formally verify protocols with proofs of complexity directly comparable to those based on more standard symbolic models, *e.g.*, essentially the same creative steps and reasoning are required as in Paulson's proofs. Moreover, we believe we can overcome the remaining difference in the proof length by extending our proof infrastructure to automate the proofs of adversary core lemmas and of standard preservation lemmas needed in invariant proofs.

Acknowledgements. We thank Michael Backes and Birgit Pfitzmann for their past collaboration and support.

References

[1] M. Backes and P. Laud. Computationally sound secrecy proofs by mechanized flow analysis. In *Proc. 13th CCS*, pages 370–379. ACM, 2006.

[2] M. Backes and B. Pfitzmann. Relating symbolic and cryptographic secrecy. *IEEE Transactions on Dependable and Secure Computing*, 2(2):109–123, 2005.

[3] M. Backes, B. Pfitzmann, and M. Waidner. A universally composable cryptographic library. IACR Cryptology ePrint Archive 2003/015, Jan. 2003.

[4] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proc. Principles of Programming Languages (POPL)*, pages 14–25, 2004.

[5] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proc. 42nd IEEE FOCS*, pages 136–145, 2001.

[6] R. Canetti and J. Herzog. Universally composable symbolic analysis of mutual authentication and key exchange protocols. In *Proc. 3rd Theory of Cryptography Conference (TCC)*, volume 3876 of *LNCS*, pages 380–403, 2006.

[7] V. Cortier, H. Hördegen, and B. Warinschi. Explicit randomness is not necessary when modeling probabilistic encryption. *ENTCS*, 186:49–65, 2007.

[8] V. Cortier and B. Warinschi. Computationally sound, automated proofs for security protocols. In *Proc. 14th European Symposium on Programming (ESOP)*, pages 157–171, 2005.

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

[10] P. Laud. Secrecy types for a simulatable cryptographic library. In *Proc. 12th ACM Conference on Computer and Communications Security*, pages 26–35, 2005.

[11] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. *Software — Concepts and Tools*, 17:93–102, 1996.

[12] D. Micciancio and B. Warinschi. Soundness of formal encryption in the presence of active adversaries. In *Proc. 1st Theory of Cryptography Conference (TCC)*, volume 2951 of *LNCS*, pages 133–151, 2004.

[13] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[14] L. Paulson. The inductive approach to verifying cryptographic protocols. *J. Computer Security*, 6:85–128, 1998.

[15] B. Pfitzmann and M. Waidner. Composition and integrity preservation of secure reactive systems. In *Proc. 7th CCS*, pages 245–254, 2000.

[16] A. M. Pitts. Evaluation logic. In G. Birtwistle, editor, *IVth Higher Order Workshop, Banff 1990*, Workshops in Computing, pages 162–189. Springer, Berlin, 1991.

[17] C. Sprenger, M. Backes, D. Basin, B. Pfitzmann, and M. Waidner. Cryptographically sound theorem proving. In *19th IEEE CSFW*, pages 153–166, July 2006.

[18] C. Sprenger and D. Basin. A monad-based modeling and verification toolbox with application to security protocols. In *20th TPHOLs*, LNCS Volume 4732, pages 302–318, 2007.