Composing Proofs of Security Protocols Using Isabelle/IOA

Oleg Sheyner Jeannette Wing *

February 1, 2000

Abstract

Complex security protocols require a formal approach to ensure their correctness. The protocols are frequently composed of several smaller, simpler components. We would like to take advantage of the compositional nature of such protocols to split the large verification task into separate and more manageable pieces.

Various formalisms have been used successfully for reasoning about large protocol compositions by hand. However, hand proofs are prone to error. Automated proof systems can help make the proofs more rigorous. The goal of our work is to develop an automated proof environment for compositional reasoning about systems. This environment would combine the power of compositional reasoning with the rigor of mechanically-checked proofs. The hope is that the resulting system would be useful in verification of security protocols of real-life size and complexity.

Toward this goal, we present results of a case study in compositional verification of a private communication protocol with the aid of automated proof tool Isabelle/IOA.

1 Introduction

Today's security protocols require a formal approach to ensure that they satisfy important correctness properties. Traditional ways of verifying correctness by hand are prone to error and require a large investment of human effort and patience. Furthermore, these problems tend to grow worse as the size and complexity of the system being verified both increase. Automated proof tools can help make the proofs more rigorous. Such tools also need a lot of human guidance, and the automation they do provide typically does not scale well with the size of the problem.

The protocols we are interested in are frequently composed of several smaller, simpler protocols. We would like to take advantage of the compositional nature of such protocols to split the large verification task into separate, more manageable pieces. Existing proof systems do not provide a structured environment for compositional reasoning about systems. The goal of our work is to develop such an environment. This environment would combine the power of compositional reasoning with the rigor of mechanically-checked proofs. We would like the resulting system to be useful in verification of security protocols of real-life size and complexity. Toward that goal we have conducted a case study in compositional verification of a private communication protocol with the aid of the automated proof tool Isabelle. This paper describes our experiences with the case study.

I/O automata [Lyn96, LT89] have been successfully employed in hand verification of large reactive systems. I/O automata express reactive distributed systems concisely as compositions of

*Authors' contact information: Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, {sheyner, wing}@cs.cmu.edu

several smaller subsystems. Meta-theorems about compositional properties of I/O automata help prove correctness theorems about the systems they describe.

Lynch has applied the I/O automata formalism to verifying a private communication protocol [Lyn99]. In this paper we take a version of the same protocol and verify its properties using the theorem prover Isabelle and I/O automata meta-theory developed by Müller [Mül98]. The protocol is decomposed into two components whose properties are proven separately. The top-level proofs then combine correctness theorems about the components and obtain a correctness proof about the composite protocol. The resulting formal description could be further combined with other security protocol components, and such compositions can be verified in Isabelle using the same compositional reasoning techniques.

The rest of this paper is organized as follows. Section 2 gives an introduction to I/O automata and the meta-theorems used in the Isabelle proofs. Section 3 describes two recent efforts to incorporate I/O automata into mechanical theorem provers, PVS [ORSvH95] and Isabelle [Pau94]. Sections 4 and 5 discuss our experiences with verifying a private communication protocol of I/O using the Isabelle theorem prover. In Section 6 we discuss our results.

2 An Introduction to I/O Automata

The Input/Output Automaton (I/O Automaton) model [Lyn96, LT89] is a general model used for formal descriptions of distributed reactive systems. An I/O Automaton A is a state machine in which the state transitions are associated with named actions. The actions are classified as either *input*, *output*, or *internal*. The input and output actions are called *external* actions. We let $ext(A)$ designate the set of external actions of automaton A. External actions are used for communication with the automaton's environment, while the internal actions are visible only to the automaton itself.

I/O Automata state machines are typically given in a *precondition-effect* style. For each action, the code specifies the *preconditions* under which the action is permitted to occur, as a predicate on the automaton state, and the *effects* of the action on the automaton state. The *effects* may be given as a series of imperative statements, in which case it is understood that all of the effects occur in one atomic step.

The *composition* operator $||$ allows an automaton representing a complex system to be constructed by composing automata representing individual system components. The composition identifies actions with the same name in different component automata. When any component automaton performs a step involving an action π, so do all component automata that include π. The state of the composition is the product of the states of its components.

When we compose a collection of automata, output actions of the components become output actions of the composition, internal actions of the components become internal actions of the composition, and actions that are inputs to some components but outputs of none become input actions of the composition.

A triple (s, π, s') is a *step* of an I/O automaton A if A has a transition from state s to state s' via action π. An *execution fragment* of A is a finite or infinite sequence $s_0\pi_0 s_1\pi_1 \ldots$ of alternating states and actions of A, where each subseqence $s_i\pi_i s_{i+1}$ is a step of A. An *execution* of A is an execution fragment whose first state is a start state of A. The *trace* of an execution α is the subsequence γ of α consisting of external actions of A. The set of all traces of A is designated $traces(A)$.

Let γ be a finite (possibly empty) sequence of external actions of automaton A, and let s and t be states of A. The triple (s, γ, t) is a *move* of A (written $s \stackrel{\gamma}{\Rightarrow}_A t$) if there exists a finite execution fragment α of A starting in s and ending in t such that $trace(\alpha) = \gamma$. Thus, a move $s \stackrel{\gamma}{\Rightarrow}_A t$ is a series of state transitions with the externally-visible behavior γ.

For reasoning about correctness properties of I/O automata, we use the notion of *implementation relation*, also called *trace inclusion*.

Definition 2.1. *Given two I/O automata A and C with sets of identical external actions, we say that C implements A (denoted $C \preceq A$) iff traces(C) $\subseteq$ traces(A).* □

Implementation relations are used to show that a concrete system C safely implements an abstract system A. Typically, A is a specification of *safety* properties we would like the concrete system to exhibit. Proving the relation $C \preceq A$ guarantees that C exhibits only the external behaviors allowed by the specification A.

Implementation relations can be established by exhibiting *simulation relations* between the concrete and abstract automata.

Definition 2.2. *Let C and A be I/O automata with identical external actions. A* forward simulation *from C to A is a relation R over $states(C) \times states(A)$ that satisfies the following conditions:*

- *If s is a start state of C, then there is a start state s' of A such that $(s, s') \in R$.*
- *If state s is reachable in C, state $s' \in R[s]$ is reachable in A, $a \in ext(C)$, and (s, a, t) is a step of C, then there is a move $s' \stackrel{a}{\Rightarrow}_A t'$ in A, where $t' \in R[t]$.*

Intuitively, every externally visible step (s, a, t) of automaton C is simulated by a move $s' \stackrel{a}{\Rightarrow}_A t'$ of automaton A. The move must include exactly one external action a, but may include any finite number of internal actions.

We write $C \leq_F A$ when there is a forward simulation from C to A. The utility of forward simulations is established by the following theorem.

Theorem 2.1. *Let C and A be I/O automata with identical external actions. If $C \leq_F A$, then $C \preceq A$.*

In this paper we make use of two weaker forms of forward simulations: *refinement mappings* and *weak refinement mappings*. A refinement mapping is a restricted form of forward simulations that allows each state of the concrete automaton C to be related to exactly one state of the abstract automaton A. Weak refinement mappings are further restricted. They allow the abstract automaton A to simulate a step of the concrete automaton C by at most one step.

Definition 2.3. *Let C and A be I/O automata with identical external actions. A* refinement mapping *from C to A is a function M from $states(C)$ to $states(A)$ that satisfies the following conditions:*

- *If $s \in start(C)$ then $M(s) \in start(A)$.*
- *If state s is reachable in C, $a \in ext(C)$, and (s, a, t) is a step of C, then state $M(s)$ is reachable in A and there is a move $M(s) \stackrel{a}{\Rightarrow}_A M(t)$ in A.*

Definition 2.4. *Let C and A be I/O automata with identical external actions. A* weak refinement mapping *from C to A is a function M from $states(C)$ to $states(A)$ that satisfies the following conditions:*

- *If $s \in start(C)$ then $M(s) \in start(A)$.*
- *If state s is reachable in C, $a \in ext(C)$, and (s, a, t) is a step of C, then state $M(s)$ is reachable in A and $(M(s), a, M(t))$ is a step of A.*

It is trivial to show that weak refinement mappings are refinement mappings, and that refinement mappings are forward simulations.

To prove trace inclusion $C \preceq A$ by hand, one usually performs the following steps:

- Find a simulation relation R (or a refinement mapping M) over the states of C and A.
- Roughly speaking, to prove that the relation R is a simulation, for each transition (s, a, t) of C that begins with a reachable state s, and for each state $s' \in R[s]$ reachable in A, we must exhibit a transition (s', a, t') of A, where $t' \in R[t]$.

  The proof usually proceeds by induction on the length of the execution leading up to the state s. For the base case, we verify that each start state of C has an R-related start state of A. For the inductive step, we consider each transition (s, a, t) of C that starts in a reachable state s. For each $s' \in R[s]$ we exhibit a transition (s', a, t') of A and prove that $(t, t') \in R$.
- During the proof of the inductive step, showing $(t, t') \in R$ sometimes requires us to place constraints on the possible values of t and t'. Here it is often helpful to prove invariant properties about the reachable states of C and A. These invariant properties provide us with the necessary constraints on t and t'.

The following theorem defines compositional properties of I/O automata and enables us to reason about individual components of complex systems.

Theorem 2.2. *Let $C = C_1||\ldots||C_n$ and $A = A_1||\ldots||A_n$ be parallel compositions of I/O automata, where $ext(A_i) = ext(C_i)$ and $C_i \preceq A_i$ for every i. Then $ext(A) = ext(C)$ and $C \preceq A$.*

Hence, if we can decompose complex systems C and A into simpler components, we can prove trace inclusion between C and A by proving trace inclusion between individual components and then applying Theorem 2.2.

3 I/O Automata and Mechanical Theorem Proving

When a trace inclusion proof is attempted using a typical generic theorem prover, many issues crop up. The first question is how I/O automata should be represented in the specification language of the theorem prover. The language may lack expressive power or convenient features because the language is tailored for the theorem prover, rather than the user's needs.

Once the representation has been designed, it is necessary to verify that the representation satisfies the meta-theorems about I/O automata, in particular Theorems 2.1 and 2.2. These essential theorems may be difficult to prove for the chosen representation of I/O automata. One possible solution is to supply these and other theorems to the theorem prover in the form of axioms. This approach defeats some of the value of mechanical verification, since we could not be sure that our representation of I/O automata is sound.

If we are verifying a complex composition of multiple smaller automata, each individual automaton has to be hand-translated to the input language of the theorem prover—a laborious process that is prone to error. In our experience, subsequent attempts to prove properties of the system reveal many more errors resulting from faulty translations than errors inherent in the original I/O automata specification.

The process of proving theorems about automata in a theorem prover can be tedious. Prover commands are typically very different from the reasoning steps that humans usually make. Even if the user knows how the high-level proof should go, translating this knowledge into a complete proof in a mechanical prover can be a frustrating experience.

Recent work has addressed these complications and attempted to make automated verification of I/O automata systems more closely resemble hand verification. Archer *et al.* recently developed TAME [AHS98], a high-level interface to the higher-order logic theorem prover PVS for specifying and proving invariant properties of I/O automata models. The TAME interface provides a template for translating I/O automata specifications into the PVS input language. A set of high-level commands lets the user prove invariant properties with the same type of steps that are commonly taken in hand proofs. However, TAME has significant shortcomings as well. There is no natural way to define an I/O automaton type and formalize I/O meta-theory, including the composition operator and theorems about simulations and compositional reasoning. This is due to restrictions in the polymorphic features of the PVS specification language. Hence, TAME is suitable primarily for verifying invariant properties of relatively small systems.

Müller formalized a large part of the basic I/O automata meta-theory using the theorem prover Isabelle [Mül98]. Isabelle's specification language has rich polymorphic mechanisms, making it suitable for concise specifications of I/O automata and associated operators. Müller's meta-theory includes a definition of the composition operator and proofs of Theorems 2.1 and 2.2. We used Müller's Isabelle/IOA system for our case study.

4 Trace Inclusion Proofs in Isabelle/IOA

The first step in the verification process is converting I/O automata specification and implementation (written in the traditional precondition-effect style) into the Isabelle input language. This task is reasonably easy because Isabelle/IOA contains the composition operator, an operator for hiding external actions (which helps make automata compatible for composition), and other standard operators from I/O automata theory. It is therefore not necessary to compose automata by hand, or otherwise modify them before doing the translation.

A template for formalizing automata in Isabelle's language is shown in Figure 1. The template assumes that the actions of the automaton have been defined as a datatype `action` in a separate theory named `Action`. To create a specific automaton out of the template, the user must fill in items 1 through 11 (marked in bold numbers in the figure), as follows.

The definition `auto_trans_def` specifies the transition relation on the state of the automaton using set comprehension notation. The relation is a set of triples $tr = (s, \alpha, t)$ satisfying the boolean `case` expression on the action name α. The user fills out items 6 through 8 to set up the transition relation for a specific instantiation of the template. Items 6 and 7 pair an action name with a

```
auto = IOA + Action + ..1.. +
types
    auto_state = ..2..
consts
    auto_asig :: action signature
    auto_trans :: (action, auto_state) transition set
    auto_ioa :: (action, auto_state) ioa
defs
    auto_asig_def "auto_asig == ({..3..}, {..4..}, {..5..})"
    auto_trans_def "auto_trans ==
            { tr. let s = fst(tr);
                          t = snd(snd(tr));
                          α = fst(snd(tr))
                      in
                  case α of
                      ..6.. ⇒ ..7.. |
                      .
                      8
                      .
            }"
    auto_ioa_def "auto_ioa == (auto_asig, {..9..}, auto_trans, {..10..}, {..11..})"
end
```

Figure 1: Template for specifying I/O automata in Isabelle

boolean expression constraining the set of transitions labeled by the action name. All other actions of the automaton follow in item 8, using the same syntax.

Finally, the definition `auto_ioa_def` defines the entire I/O automaton as a 5-tuple consisting of the action signature, the set of initial states (item 9), the transition relation, and two types of fairness conditions (items 10 and 11). In this paper we will consider only safety properties, so the fairness conditions will always be empty sets. Section 5.1.1 contains an example translation of an I/O automaton into Isabelle using the template.

Once the I/O automata have been encoded in Isabelle, the next step is stating and proving invariant properties that will be used later in the implementation proof. A typical hand invariant proof proceeds by induction. For the base case, we show that the invariant holds in all initial states. For the inductive step, we check that each action preserves the invariant property. A similar strategy works in our Isabelle proofs of invariant properties. We have developed an Isabelle tactic (called `simplify_inv_goal_tac`) that takes the invariant goal, applies induction (thereby breaking the goal into subgoals for the base case and for each action) and automatically proves the "trivial" cases. In particular, the cases that do not modify the parts of the state involved in the invariant are proven automatically. After this tactic is applied, the user is left with the task of proving the remaining cases. In each case, the necessary reasoning is localized to the effects of one action, eliminating the need to reason about the entire automaton. Appendix A.7 shows an Isabelle proof of one of the invariants as an example.

The final step in the implementation proof is exhibiting a simulation relation or a refinement mapping from the states of the implementation to the states of the specification. The proof that a function is a refinement mapping is structurally similar to the proofs of invariant properties. Once again, we apply induction on the length of the execution to the goal and automatically discharge the "trivial" cases among the resulting subgoals. The rest of the subgoals are proven in the manner similar to the hand proof. Each subgoal corresponds to one step of the implementation automaton; the user must exhibit a corresponding move of the specification automaton and prove that the end states of the implementation step and the specification move are related by the refinement mapping.

When we want to generate a trace inclusion proof between two compositions of automata $C = C_1 || \ldots || C_n$ and $A = A_1 || \ldots || A_n$, we can take advantage of Theorem 2.2. Once we have obtained separate trace inclusion proofs for each pair of component automata C_i and A_i separately, we can apply the compositionality theorem to get trace inclusion between C and A. This step is easy, and requires only a side proof that the automata being composed are compatible with each other. We are developing Isabelle tactics that discharge most of this proof automatically.

5 Case Study: Verification of a Private Communication Protocol

We have taken a modified version of a private communication service protocol specified as I/O automata in [Lyn99] and used Müller's Isabelle/IOA to verify secrecy properties of the service. The main point of this exercise is to investigate the feasibility of using the theoretical machinery provided by I/O automata to perform compositional analysis of complex systems in an automated proof environment. A full description of the system appears in the Appendices. In the rest of the paper we give a high-level description of the system and discuss our experiences with Isabelle/IOA.

The private communication service is specified as an I/O automaton PC. The service lets clients exchange messages with each other using an insecure transmission channel. The specification guarantees that messages are delivered at most once, and their content remains secret from the adversary.

The service is implemented using a shared-key cryptosystem and contains a number of automata. Before going out on the *insecure communication channel*, each message passes through an *encoder* automaton and gets encrypted with a key that the encoder shares with a corresponding *decoder* automaton on the receiving side. The decoder decrypts the messages and passes them on to the client. The implementation model also includes a *passive eavesdropper* automaton. The eavesdropper can intercept messages appearing in the insecure channel and also compute new messages (via encryption and decryption functions) from the available information. Using a technique similar to assume-guarantee proofs, the *environment* automaton records our assumptions about the environment in which the service can operate correctly. In particular, the environment must not give away secrets to the eavesdropper.

The shared keys are generated by a *key distribution service*. The full implementation employs a version of the Diffie-Hellman protocol to generate and distribute shared keys. Since the analysis of key distribution is fairly involved, we decompose the implementation into two parts that can be verified independently. Figure 2 shows the structure of an I/O automata composition $PCImpl_1 = IC||Eve||KD||Enc||Dec||Env$ implementing specification PC.

In $PCImpl_1$, KD is a high-level specification, leaving out the details of key distribution and thus simplifying the structure of $PCImpl_1$. The Diffie-Hellman key distribution protocol can now be verified independently of the rest of the private communication protocol. The protocol consists of Diffie-Hellman nodes (one per client) and an insecure channel. Diffie-Hellman nodes exchange several messages over the channel in order to establish a shared key for a pair of clients. Just as in the private communication implementation, there is a passive eavesdropper and and an environment. Figure 3 shows the structure of an I/O automata composition $KDImpl = DH_1||DH_2||IC||Eve||Env$ implementing specification KD.

For simplicity, we assume (unrealistically) that the key distribution protocol and the private

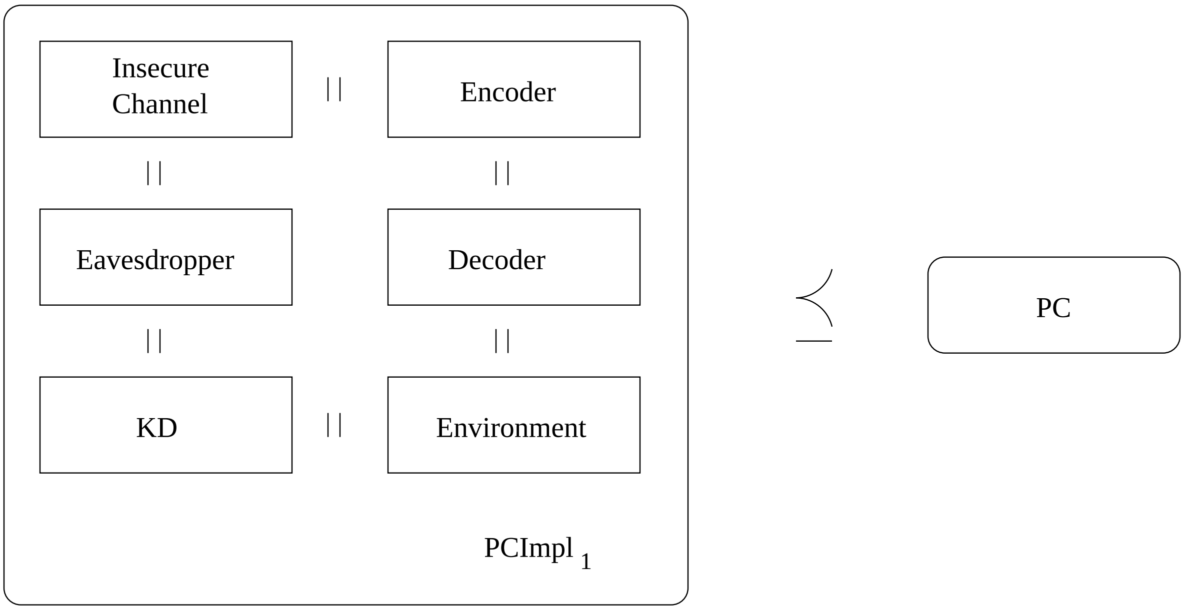

Figure 2: Composition *PCImpl₁* implements specification *PC*

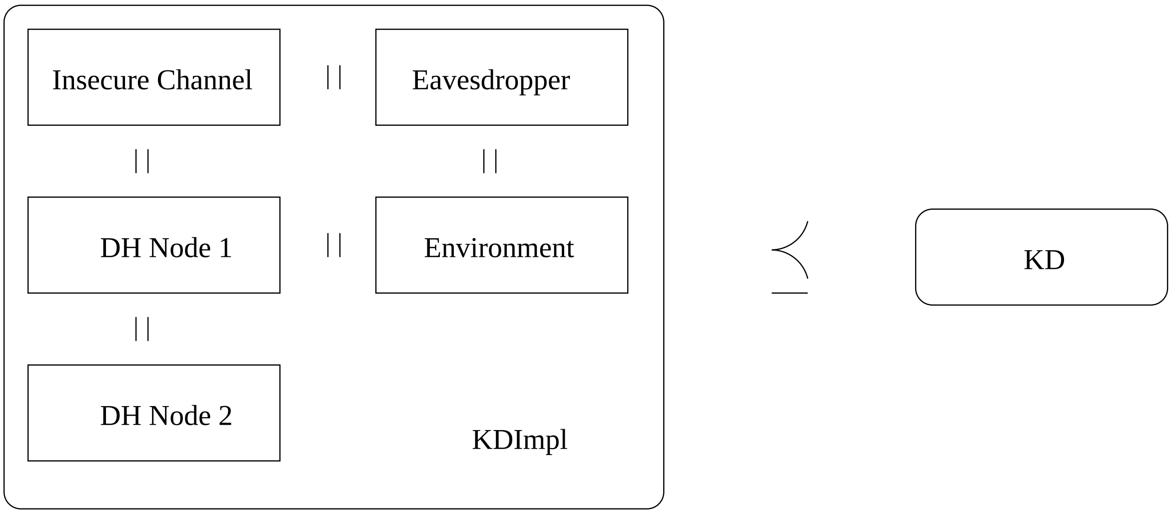

Figure 3: Composition *KDImpl* implements specification *KD*

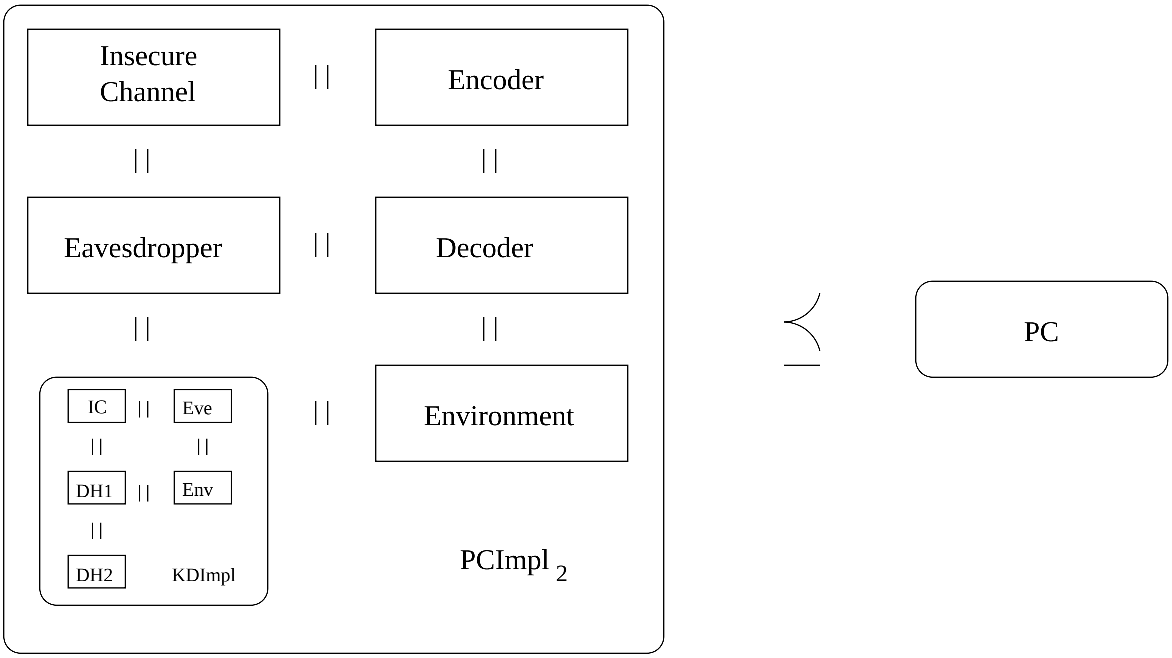

Figure 4: Composition *PCImpl₂* implements specification *KD*

communication protocol have separate insecure channels and eavesdroppers, and the eavesdroppers do not communicate with each other. See [Lyn99] for a more realistic treatment combining the insecure channels and the eavesdroppers.

Breaking up the implementation in this manner lets us take advantage of the compositionality theorem about I/O automata (Theorem 2.2). We prove trace inclusion for compositions shown in Figures 2 and 3 in Isabelle. Theorem 2.2 then lets us substitute the Diffie-Hellman implementation *KDImpl* in place of the specification *KD* while preserving trace inclusion between $PCImpl_1$ and *PC*. The resulting implementation $PCImpl_2$ is shown in Figure 4.

Below we give a more detailed description of the *PC* and *KD* service specifications. Appendices A and B describe the compositions $PCImpl_1$ and *KDImpl* and state the invariants, lemmas, and theorems needed to perform Isabelle trace inclusion proofs for Figures 2 and 3 (all their proofs are in [SW00], and many follow straightforwardly by induction). Appendix C shows how the compositionality theorem is applied to obtain the implementation relation for Figure 4.

5.1 The Services

In this section, we describe the two services that are implemented by the protocols and verified in this paper. The use of input and output actions provides convenient ways of composing these automata with others, and of describing what is preserved by implementation relationships. These specifications describe only safety properties, although the same methods can be used to handle liveness properties, formulated as *live I/O Automata* [GSSL93].

5.1.1 Private Communication

This section contains a specification of the problem of achieving private communication among the members of a finite collection P of clients. The specification expresses three properties: (1) only messages that are sent are delivered, (2) messages are delivered at most once, and (3) none of the messages are revealed by an "adversary." We describe the problem using a high-level I/O automaton specification $PC(U, P, M, A)$, where U is a universal set of data values, P is an arbitrary finite set of client ports, $M \subseteq U$ is a set of messages, and A is an arbitrary finite set of adversary ports. This specification makes no mention of distribution or keys; these aspects will appear in implementations of this specification, but not in the specification itself. The specification simply describes the desired properties, as an abstract machine. As usual for automaton specifications, the properties, listed separately above, are intermingled in one description.

$PC(U, P, M, A)$:

Signature:

Input:
  $PC\text{-}send(m)_{p,q}$, $m \in M$, $p, q \in P$, $p \neq q$

Output:
  $PC\text{-}receive(u)_{p,q}$, $u \in U$, $p, q \in P$, $p \neq q$
  $reveal(u)_a$, $u \in U$, $a \in A$

States:

for every pair $p, q \in P$, $p \neq q$:
  $buffer(p, q)$, a multiset of M

Transitions:

$PC\text{-}send(m)_{p,q}$
  Effect:
    add m to $buffer(p, q)$

$PC\text{-}receive(u)_{p,q}$
  Precondition:
    $u \in buffer(p, q)$
  Effect:
    remove one copy of u from $buffer(p, q)$

$reveal(u)_a$
  Precondition:
    $u \notin M$
  Effect:
    none

The first two properties listed above, which amount to at-most-once delivery of messages that were actually sent, are ensured by the transition definitions for *PC-send* and *PC-receive*. The third property, privacy, is expressed by the constraint for *reveal*.

The following figure demonstrates the private communication specification translated into Isabelle/IOA. The translation fills in specific information about the specification into the template shown in Section 4.

```
PC = IOA + Action + InfMultiset +
types
   PC_state = "P × P ⇒ U tmultiset"
consts
   PC_asig :: action signature
   PC_trans :: (action, PC_state) transition set
   PC_ioa :: (action, PC_state) ioa
defs
   PC_asig_def "PC_asig ==
         ((UN m p q. {PC_send m p q}),
```

```
        (UN u msg p q. {PC_receive u msg p q}) ∪ (UN u a. {reveal u a}),
        {})"
    PC_trans_def "PC_trans ==
        { tr. let s = fst(tr);
                  t = snd(snd(tr));
                  α = fst(snd(tr))
                in
              case α of
                reveal u a ⇒ u ∉ M_set |
                PC_send m p q ⇒
                  (m ∈ M_set) &
                  (t = (λ (p', q').
                         if (p = p') & (q = q') then
                           s (p', q') + {m}
                         else
                           s (p', q'))) |
                PC_receive u msg p q ⇒
                  (u ∈ s (p, q)) &
                  (t = (λ (p', q').
                         if (p = p') & (q = q') then
                           s (p', q') - {u}
                         else
                           s (p', q')))
        }"
    PC_ioa_def "PC_ioa == (PC_asig, {λ (p, q). ∅}, PC_trans, {}, {})"
end
```

The state of PC is represented as a function from a pair of clients of type P to a multiset of messages of type U. The definition of the transition relation gives a boolean expression for every triple (s, α, t), where s and t are states and α is an action of PC. The boolean expression includes the precondition of α and relates t to s via the effects of α. Thus, the expression is true if and only if (s, α, t) is a step of PC.

5.1.2 Key Distribution

This is a drastically simplified key distribution service, which distributes a single key to several participants. We do not model requests for the keys, but assume that the service generates the key spontaneously. The simplified key distribution problem is specified by the automaton $KD(U, P, K, A)$, where U is a universal set of data values, P is an arbitrary finite set of client ports, $K \subseteq U$ is a set of keys, and A is a finite set of adversary ports.

$KD(U, P, K, A)$:

Signature:

Input:	Internal:
none	*choose-key*
Output:	
$grant(u)_p$, $u \in U$, $p \in P$	
$reveal(u)_a$, $u \in U$, $a \in A$	

States:

chosen-key, an element of $K \cup \{\perp\}$, initially $\perp$
notified $\subseteq P$, initially $\emptyset$

Transitions:

choose-key
  Precondition:
    $chosen\text{-}key = \bot$
  Effect:
    $chosen\text{-}key :=$ choose k where $k \in K$

reveal$(u)_a$
  Precondition:
    $u \notin K$
  Effect:
    none

grant$(u)_p$
  Precondition:
    $chosen\text{-}key \neq \bot$
    $u = chosen\text{-}key$
    $p \notin notified$
  Effect:
    $notified := notified \cup \{p\}$

6 Discussion

The benefits of decomposing large systems into smaller parts for verification are twofold. From the software engineering perspective, formalizing and reasoning about large monolithic systems quickly becomes unmanageable. The number of potential interactions between state components typically increases exponentially with the size of the state and the size of the transition relation. When the system has more than a few state components, just formulating the necessary invariants can prove to be a daunting task. Compositional reasoning lets us take a modular approach to verification. We can focus on proving properties of self-contained systems of reasonable size and build up a component library for constructing larger systems. Compositionality results let us combine proven properties of components and obtain new results about the larger system without going through the verification process from scratch. One can imagine that somewhat more realistic versions of the *PC* and *KD* services and their implementations could be a part of a library of formalized security and cryptography components.

Decomposition also helps avoid the state explosion problems common to all automated verification tools. Isabelle's simplifier was valuable in reducing the human effort in our verification exercise, but in our experience its running time greatly depends on the size of automata being verified. The table below shows the running time on a set of theorems proven automatically by an identical invocation of the simplifier. Each theorem describes how a transition of an n-automata composition is projected onto the individual components. The table gives the timings for $n \in \{3, 4, 5, 6\}$.

n	3	4	5	6
time	5.5 *sec*	27.9 *sec*	3.8 *min*	40.1 *min*

We did not prove the theorems for higher values of n because for $n \geq 7$ the simplifier requires more than the 256MB of RAM available on the test machine. But the data in the table suggest that even without the space restriction, the automatic proof tools in Isabelle would not be able to handle larger systems in a reasonable amount of time, and without them the verification effort is prohibitively expensive. In the small example verified in this paper, we split the task of verifying trace inclusion for a nine-component system $PCImpl_2$ into two separate tasks, one of which deals with a six-component system $PCImpl_1$ and the other with a five-component system *KDImpl*. Notably, we could not prove the projection theorems for the nine-component case, but could do so for the smaller component cases. This modest division resulted in substantial savings primarily because complexity, running time, and space requirements appear to be exponentially related to problem size. In the context of real-world systems that can have dozens of such components, abstraction and decomposition become essential.

6.1 Observations on Benefits of Formal Verification

Refinement proofs turned out to be a more effective way of fleshing out specification problems than invariant proofs. Invariant proofs may touch only specific parts of the protocol state and leave untouched more abstract questions about what the protocol is doing. The refinement proof makes explicit all the assumptions about why the implementation does what the specification intended.

In particular, during the refinement proof for the implementation PCImpl_1 we were forced to go back and prove several auxiliary invariants whose utility were not obvious *a priori*. This in turn led us to typos and errors in our formalization of the cryptosystems and component automata. Although the bugs caught during the process of proving invariant lemmas and trace inclusion were mostly errors in our formalization, we caught one error in the original description of PCImpl_1 protocol (some uninitialized variables led to failed proofs of the base case) and a typo in an invariant statement in [Lyn99] (see remark about invariant B.1 in Appendix B.3).

6.2 Efficiency Issues

The human effort spent on the project included (1) twelve weeks for formalizing and verifying $\mathit{PCImpl}_1 \preceq \mathit{PC}$, (2) three weeks for verifying $\mathit{KDImpl} \preceq \mathit{KD}$, and (3) three days for verifying $\mathit{PCImpl}_2 \preceq \mathit{PC}$. A substantial fraction of the time in stage 1 was spent learning Isabelle/IOA and setting up the procedure for formalizing I/O automata, stating and proving invariants, and proving trace inclusion. This accounts for most of the difference in effort between stages 1 and 2. Stage 3 was much shorter due to our use of the compositionality theorem.

We believe that additional automation can reduce the human effort substantially in all phases of the verification process. At the level of the prover, additional tactics can automate tasks that commonly show up in reasoning about I/O automata. These tactics fall into two categories. One set of tactics would simulate the high-level proof steps used in human-style I/O automata proofs. These would be similar to the proof strategies offered by Archer's TAME environment for PVS. Another set of tactics would help the user deal with proof obligations specific to Isabelle and the Isabelle formalization of I/O automata meta-theory. For example, applying the compositionality theorem requires proofs for side conditions that the Isabelle type checker does not guarantee. It must be shown that the I/O automata definitions are well formed—the sets of input, output, and internal actions are disjoint, and the transition relation is defined only for the actions in the automaton signature definition. Furthermore, the user must show that the automata being composed are compatible with each other. These proofs have common structure and can therefore be effectively encapsulated in a higher-level Isabelle tactic. The tactic would be used with every application of the compositionality theorem.

There are also ways to improve efficiency at the user interface level. A compiler can take care of translating I/O automata (expressed in a suitable way) into an Isabelle formalization. It is also possible to generate a general framework for invariant definitions and trace inclusion proofs automatically, letting the user fill in definitions and proof script details specific to the problem.

One of the biggest obstacles to formal reasoning with theorem provers remains their cumbersome nature and the level of attention to low-level details required of the user. Isabelle is not an exception. Interacting with the bare-bones prover throughout the verification cycle can be a frustrating experience, which is why we emphasize the need to automate as much of the process as possible. With the enchancements discussed above, the task of formalizing the specification and setting up proof goals and induction can be substantially automated. Most user interaction with the prover would take place when reasoning about individual automata actions. The actions typically have a small and localized effect on the automaton state, which makes the proofs more manageable.

6.3 Technical Issues

In Müller's formalization of I/O automata meta-theory the binary automata composition operator has the following type, given in Isabelle's ML-like notation:

$$\| \quad :: \quad (\alpha, \sigma)\ ioa \rightarrow (\alpha, \tau)\ ioa \rightarrow (\alpha, \sigma \times \tau)\ ioa$$

where α is the action type, σ and τ are state types of the automata being composed, and $\sigma \times \tau$ is the state type of the composition. The composition operator requires that both automata be defined over the same action space α. If we apply the operator multiple times to compose several automata, every action of every component must be a member of the same action space. Mechanized induction on the action datatype generates a subcase for each action in the action space, including those that do not belong to the component being verified. This means that inductive proofs do not scale well for large compositions of automata. This is a serious problem, as it undermines the primary benefit of compositional reasoning: scalability. It takes over an hour for Isabelle (ver. 99) to execute in interactive mode the invariant and refinement proof scripts developed in this project. The simplifier spends the majority of that time reducing inductive subcases for actions, considering many more cases than necessary.

Fixing the problem without completely revising the meta-theory requires a richer type system than supported by Isabelle/HOL. For example, in a polymorphic language with subtyping and union types [Pie91], the composition operator could be given the following type:

$$\| \quad :: \quad (\alpha, \sigma)\ ioa \rightarrow (\beta, \tau)\ ioa \rightarrow (\alpha \vee \beta, \sigma \times \tau)\ ioa$$

The action type of the composition $\alpha \vee \beta$ is the union type derived from the action types α and β of the components. Assuming that the usual binary operators on sets (union, intersection, difference) have the type $\alpha\ set \rightarrow \beta\ set \rightarrow (\alpha \vee \beta)\ set$, the existing definition of the composition operator would still make sense in this setting.

7 Conclusions

Existing compositional proof methods, including implementation relations between I/O Automata, are adequate for handling large classes of verification problems. Numerous case studies have used these techniques by hand to prove global properties of non-trivial systems. Until recently, automated verification tools have not included compositional techniques in their repertoire. Yet, the strengths of compositional reasoning and automated reasoning have the potential to complement each other.

Automation demands that compositional proofs be made strictly rigorous. It does not tolerate typos or imprecise wording, which can lead to subtle errors in hand proofs. Forced to develop proofs according to these exacting standards, the user gains deeper understanding of the subtleties of the system and more confidence in the final product. Although time consuming to use, automated proof tools make proof re-checking much easier, which can result in substantial time savings in the iterative development/verification cycle. Conversely, compositional techniques offer the best hope of dealing with state explosion and complexity problems associated with automated verification of non-trivial systems.

Our experience with the Isabelle/IOA verification environment leads us to conclude that there is a lot of work yet to be done before the potential benefits of automated compositional reasoning are fully realized. Using Isabelle/IOA is a labor-intensive undertaking, and the environment does not appear to be sufficiently scalable. These issues can be resolved with additional effort, and we believe that the benefits of the compositional approach make the effort worthwhile.

References

[AHS98] M. Archer, C. Heitmeyer, and S. Sims. TAME: A PVS interface to simplify proofs for automata models. *UITP '98*, July 1998.

[GSSL93] R. Gawlick, R. Segala, J.F. Sogaard-Andersen, and N. Lynch. Liveness in timed and untimed systems. Technical Report MIT/LCS/TR-587, MIT, Laboratory for Computer Science, Cambridge, MA., December 1993.

[LT89] N. Lynch and M. Tuttle. An Introduction to Input/Output Automata. *CWI-Quarterly*, 2(3):219–246, September 1989.

[Lyn96] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, March 1996.

[Lyn99] N. Lynch. I/O automaton models and proofs for shared-key communication systems. *12th IEEE Computer Security Foundations Workshop (CSFW12)*, pages 14–29, June 1999.

[Mül98] O. Müller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory*. PhD thesis, Technische Universitäat München, 1998.

[ORSvH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[Pau94] L.C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[Pie91] B.C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, February 1991.

[SW00] O. Sheyner and J. Wing. Toward compositional analysis of security protocols using theorem proving. Technical Report CMU-CS-00-106, CMU Computer Science Department, Pittsburgh, PA., January 2000.

A Private Communication Implementation

In this appendix, we describe the protocol that implements the private communication specification from Section 5.1.1. The protocol uses a shared-key cryptosystem $\mathcal{C}$ to encrypt messages before sending them over an insecure communication channel. The protocol keeps the messages secure against passive eavesdroppers.

We give automaton models for some system components that appear in many security-related settings: environments for security services, insecure channels, and eavesdroppers. They are presented in a parameterized fashion so that they can be used in different contexts. We then put these components together in the private communication protocol.

A.1 Cryptosystems

A *cryptosystem signature* $\mathcal{S}$ consists of:

- $TN_{\mathcal{S}}$, a set of *type names*.
- $FN_{\mathcal{S}}$, a set of *function names*.
- $domain_{\mathcal{S}}$, a mapping from $FN_{\mathcal{S}}$ to $(TN_{\mathcal{S}})^*$.
- $range_{\mathcal{S}}$, a mapping from $FN_{\mathcal{S}}$ to $TN_{\mathcal{S}}$.
- $EN_{\mathcal{S}} \subseteq FN_{\mathcal{S}}$, a set of *easy* function names.

A *constant name* is a function name f such that $domain_{\mathcal{S}}(f) = \lambda$. Let $CN_{\mathcal{S}} \subseteq FN_{\mathcal{S}}$ denote the set of constant names of $\mathcal{C}$. We omit the subscript $\mathcal{S}$ where no confusion seems likely. A *cryptosystem* $\mathcal{C}$ consists of:

- A cryptosystem signature $sig_{\mathcal{C}}$. We write $TN_{\mathcal{C}}$ as shorthand for $TN_{sig_{\mathcal{C}}}$, etc.
- $set_{\mathcal{C}}$, a mapping from $TN_{\mathcal{C}}$ to disjoint sets.
- $fun_{\mathcal{C}}$, a mapping from $FN_{\mathcal{C}}$ to functions; We require that if $domain_{\mathcal{C}}(f) = (t_1, \ldots, t_k)$ and $range_{\mathcal{C}}(f) = t$ then $fun_{\mathcal{C}}(f) : set_{\mathcal{C}}(t_1) \times \cdots \times set_{\mathcal{C}}(t_k) \rightarrow set_{\mathcal{C}}(t)$.

We write $set_{\mathcal{C}}$ for $\bigcup_{t \in TN_{\mathcal{C}}} set_{\mathcal{C}}(t)$. We omit the subscript $\mathcal{C}$ where no confusion seems likely. If $X \cup \{y\} \subseteq set_{\mathcal{C}}$, we say that y is *easily reachable* from X in $\mathcal{C}$ provided that y is obtainable starting from elements of X, by applying only functions denoted by function names in $EN_{\mathcal{C}}$.

In [SW00] we define formally a general *term cryptosytem* and two specific instances: a shared-key cryptosystem, used in shared key communication; and a base-exponenet cryptosystem used in the Diffie-Hellman key distribution protocols.

A.2 Environment Automata

Here we assume that U is a universal set of data values, A is an arbitrary finite set of adversary ports, that is, locations where information can be communicated to the adversary, and $N \subseteq U$. The environment automaton $Env(U, A, N)$ models any entities other than the channels from which an eavesdropper may learn information. It says that the environment is capable of communicating elements of U at any adversary port $a \in A$, but in fact does not communicate any elements of N.

$Env(U, A, N)$:

Signature:

Input:
None

Output:
$learn(u)_a$, $u \in U$, $a \in A$

States:

No variables

Transitions:

$learn(u)_a$
Precondition:
$u \notin N$
Effect:
none

A.3 Insecure Channel Automata

Here we assume that U is a universal set of data values, P is an arbitrary finite set of client ports, and A is an arbitrary finite set of adversary ports. The insecure channel admits *send* and *receive* actions for all elements of U and also has *eavesdrop* output actions, by which information in transit passes to an outsider. The insecure channel allows any message in transit to be communicated to an outsider via the *eavesdrop* actions.

$IC(U, P, A)$:

Signature:

Input:
$IC\text{-}send(u)_{p,q}$, $u \in U$, $p, q \in P$, $p \neq q$

Output:
$IC\text{-}receive(u)_{p,q}$, $u \in U$, $p, q \in P$, $p \neq q$
$eavesdrop(u)_{p,q,a}$, $u \in U$, $p, q \in P$, $p \neq q$, $a \in A$

States:

for every $p, q \in P$, $p \neq q$:
$buffer(p, q)$, a multiset of U, initially empty

Transitions:

$IC\text{-}send(u)_{p,q}$
Effect:
add u to $buffer(p, q)$

$IC\text{-}receive(u)_{p,q}$
Precondition:
$u \in buffer(p, q)$
Effect:
remove one copy of u from $buffer(p, q)$

$eavesdrop(u)_{p,q,a}$
Precondition:
$u \in buffer(p, q)$
Effect:
none

A.4 Eavesdropper Automata

Here we assume that $\mathcal{C}$ is a cryptosystem, P is an arbitrary finite set of client ports, and A is an arbitrary finite set of adversary ports. We define a model for an eavesdropper, as a nondeterministic automaton $Eve(\mathcal{C}, P, A)$. *Eve* simply remembers everything it learns and hears, and can reveal anything it has, at any time. It does this by maintaining a variable *has*, initially $\emptyset$. The value of *has* may change only in restricted ways: Namely, when $eavesdrop(u)_{p,q,a}$ or $learn(u)_a$ occurs, u gets added to *has*. When an internal *compute* action occurs, the value resulting from applying an easy function (one in $EN_{\mathcal{C}}$) to values in *has* may be added to *has*. We restrict the $reveal(u)$ output so that $u \in has$, that is, *Eve* can only report a value that it has. Similar treatments of known information appear elsewhere in the literature.

$Eve(\mathcal{C}, P, A)$:

Signature:

Input:
- $eavesdrop(u)_{p,q,a}$, $u \in set_{\mathcal{C}}$, $p, q \in P$, $p \neq q$, $a \in A$
- $learn(u)_a$, $u \in set_{\mathcal{C}}$, $a \in A$

Output:
- $reveal(u)_a$, $u \in set_{\mathcal{C}}$, $a \in A$

Internal:
- $compute(u, f)_a$, $f \in EN_{\mathcal{C}}$, $a \in A$

States:

$has \subseteq set_{\mathcal{C}}$, initially $\emptyset$

Transitions:

$eavesdrop(u)_{p,q,a}$
Effect:
$has := has \cup \{u\}$

$learn(u)_a$
Effect:
$has := has \cup \{u\}$

$reveal(u)_a$
Precondition:
$u \in has$
Effect:
none

$compute(u, f)_a$
Precondition:
$\{u_1, \ldots, u_k\} \subseteq s.has$
$u = f(u_1, \ldots, u_k)$
Effect:
$has := has \cup \{u\}$

The rest of this appendix describes a straightforward shared-key communication protocol. The protocol simply uses a shared key, obtained from a key distribution service, to encode and decode messages. Throughout the section, we assume that $\mathcal{C}$ is a shared-key cryptosystem, P is a set (of clients) with at least 2 elements, and A is a nonempty finite set (of adversaries).

A.5 The Encoder and Decoder

We define parameterized encoder and decoder automata, parameterized by the shared-key cryptosystem $\mathcal{C}$, the set P of clients, and elements $p, q \in P$, $p \neq q$. Note that, in the code for $IC\text{-}send(u)$, we are using the abbreviation *enc* for $fun_{\mathcal{C}}(enc)$ – that is, we are suppressing mention of the particular cryptosystem $\mathcal{C}$.

$Enc(\mathcal{C}, P)_{p,q}$, **where** $p, q \in P$, $p \neq q$:

Signature:

Input:
    $PC\text{-}send(m)_{p,q}$, $m \in [MConst_{\mathcal{C}}]$
    $grant(u)_p$, $u \in set_{\mathcal{C}}$

Output:
    $IC\text{-}send(u)_{p,q}$, $u \in set_{\mathcal{C}}$

States:

$buffer$, a multiset of elements of $[MConst_{\mathcal{C}}]$, initially empty
$shared\text{-}key \in [KConst_{\mathcal{C}}] \cup \{\bot\}$, initially $\bot$

Transitions:

$PC\text{-}send(m)_{p,q}$
  Effect:
    add m to $buffer$

$IC\text{-}send(u)_{p,q}$
  Precondition:
    m is in $buffer$
    $shared\text{-}key \neq \bot$
    $u = enc(m, shared\text{-}key)$
  Effect:
    remove one copy of m from $buffer$

$grant(u)_p$
  Effect:
    if $u \in [KConst_{\mathcal{C}}]$ then
      $shared\text{-}key := u$

More-or-less symmetrically, we have:

$Dec(\mathcal{C}, P)_{p,q}$**, where** $p, q \in P$**,** $p \neq q$ **:**
Signature:

Input:
    $IC\text{-}receive(u)_{p,q}$, $u \in set_{\mathcal{C}}$
    $grant(u)_q$, $u \in set_{\mathcal{C}}$

Output:
    $PC\text{-}receive(u)_{p,q}$, $u \in set_{\mathcal{C}}$

States:

$buffer$, a multiset of elements of $set_{\mathcal{C}}(\text{“}M\text{”})$, initially empty
$shared\text{-}key \in [KConst_{\mathcal{C}}] \cup \{\bot\}$, initially $\bot$

Transitions:

$IC\text{-}receive(u)_{p,q}$
  Effect:
    if $u \in set_{\mathcal{C}}(\text{“}M\text{”})$ then
      add u to $buffer$

$PC\text{-}receive(u)_{p,q}$
  Precondition:
    m is in $buffer$
    $shared\text{-}key \neq \bot$
    $u = dec(m, shared\text{-}key)$
  Effect:
    remove one copy of m from $buffer$

$grant(u)_q$
  Effect:
    if $u \in [KConst_{\mathcal{C}}]$ then
      $shared\text{-}key := u$

A.6 The Complete Implementation

In the rest of this section, we assume: $U = set_{\mathcal{C}}$; $M = [MConst_{\mathcal{C}}]$; $K = [KConst_{\mathcal{C}}]$; $N = M \cup K$; U' is an arbitrary set with $K \subseteq U'$; A' is an arbitrary set, disjoint from A.

The implementation consists of encoder and decoder components, an insecure channel, eavesdropper and environment, plus a key distribution service. More precisely, the implementation, *PCImpl₁* $(\mathcal{C}, P, A, U', A')$, is obtained by composing the following automata and then hiding certain actions.

- $Enc(\mathcal{C}, P)_{p,q}$, $Dec(\mathcal{C}, P)_{p,q}$, $p, q \in P$, $p \neq q$.
- $IC(U, P, A)$, $Eve(\mathcal{C}, P, A)$, $Env(U, A, N)$.
- $KD(U', P, K, A')$, a key distribution service.

In this system, the eavesdropper *Eve* does not acquire any information directly from the *KD* component.

To get $PCImpl_1(\mathcal{C}, P, A, U', A')$, we hide the following actions in the composition just defined: $eavesdrop_{p,q,a}$, $p, q \in P$, $a \in A$; $IC\text{-}send_{p,q}$, $IC\text{-}receive_{p,q}$, $p, q \in P$; $grant_p$, $p \in P$; $learn_a$, $a \in A$; $reveal_a$, $a \in A'$.

A.7 Correctness of the Private Communication Implementation

To prove correctness of $PCImpl_1$, we demonstrate an implementation relationship between $PCImpl_1$ and PC. The invariant and implementation proofs are similar to [Lyn99], though ours were modified to mirror the Isabelle proofs (seec̃itesheyner-wing-tr). In particular, instead of a simulation relation we use a weak refinement mapping between the states of $PCImpl_1$ and PC.

A.7.1 Invariants

Invariant A.1. *In all reachable states of $PCImpl_1$, the following are true:*

1. *If $Enc_{p,q}.shared\text{-}key \neq \bot$ then $Enc_{p,q}.shared\text{-}key = KD.chosen\text{-}key$.*
2. *If $Dec_{p,q}.shared\text{-}key \neq \bot$ then $Dec_{p,q}.shared\text{-}key = KD.chosen\text{-}key$.*

Invariant A.2. *In all reachable states of $PCImpl_1$, the following are true:*

1. *If $Enc_{p,q}.shared\text{-}key = \bot$ then IC.buffer(p, q) is empty.*
2. *If $Enc_{p,q}.shared\text{-}key = \bot$ then $Dec_{p,q}.buffer$ is empty.*

Invariant A.3. *In all reachable states of $PCImpl_1$ the following holds: for all p, $q \in P$, and all $u \in N$, $u \notin IC.buffer(p, q)$.*

Invariant A.4. *In all reachable states of $PCImpl_1$, if $u \in N$ then $u \notin Eve.has$.*

We present the Isabelle proof of invariant A.4.

```
Goal "invariant PCImpl_ioa lemmaA_6";

(* Apply simplification tactic to reduce the goal to (non-trivial) subgoals
   corresponding to automata actions
```

```
*)
by (simplify_inv_goal_tac lemmaA_6_def 1);

(* There are three cases still left to show: eavesdrop, learn, and compute. *)

(* eavesdrop *)

(* Strip outer quantifiers and implications, flatten conjunctions
   in hypotheses *)
by (REPEAT (rtac allI 1 ORELSE rtac impI 1 ORELSE etac conjE 1));
by (rename_tac "s t u p q" 1);

(* Apply invariant lemma A.5 *)
by (apply_inv_tac lemmaA_5 lemmaA_5_def 1);

(* The rest is definition expansion, quantifier instantiation,
   and simplification *)
sf [Ball_def] 1;
by (strip_tac 1);

by (thin_tac "ALL x. x : N_set --> x ~: eve s" 1);
by (eres_inst_tac [("x", "p")] allE 1);
by (eres_inst_tac [("x", "q")] allE 1);
by (eres_inst_tac [("x", "x")] allE 1);

by (case_tac "x = u" 1);
sf [] 1;
ba 1;

(* learn *)
(* Solved automatically by Isabelle *)
by (Blast_tac 1);

(* compute *)
(* Solved automatically by Isabelle after expanding some definitions *)
by (asm_full_simp_tac (simpset() addsimps [N_set_def]) 1);
by (Blast_tac 1);

(* done *)
qed "lemmaA_6";
```

High-level tactic `simplify_inv_goal_tac` takes care of breaking down the invariant definition, applying the Isabelle induction tactic, and simplifying the resulting cases. In the example invariant proof shown here, the user is left with the same three non-trivial cases that were considered in the hand proof. The tactic `apply_inv_tac` is another instance where a high-level step in the hand proof can be simulated effectively by a high-level Isabelle tactic. In the example proof for the case $\alpha = \mathit{eavesdrop}(u)_{p,q,a}$, `apply_inv_tac` applies the invariant A.3 to state s and adds the result to the list of assumptions in the current goal, to be used later in the proof.

A.7.2 Implementation Proof

We show that PCImpl_1 implements PC by exhibiting a *weak refinement mapping* F from the states of PCImpl_1 to the states of PC. $F(s)$ is the multiset union of three multisets, A_1, A_2, and A_3, where:

1. $A_1 = s.Enc_{p,q}.buffer$.

2. $A_2 = dec(s.IC.buffer(p,\ q),\ s.KD.chosen\text{-}key)$ if $s.KD.chosen\text{-}key \neq \perp$ else $\emptyset$.

3. $A_3 = dec(s.Dec_{p,q}.buffer,\ s.KD.chosen\text{-}key)$ if $s.KD.chosen\text{-}key \neq \perp$ else $\emptyset$.

Thus, the multiset of messages in transit at the specification level is obtained by combining the multisets of messages at the encoder and the decoder and the multiset of messages in the insecure channel. The messages in the insecure channel and decoder buffers must be decoded with the shared key to get the correspondence.

Theorem A.5. *F is a weak refinement mapping.*

Theorem A.6. *$PCImpl_1(\mathcal{C}, P, A, U', A') \preceq PC(U, P, M, A)$.*

B Diffie-Hellman Key Distribution Implementation

This section describes the Diffie-Hellman key distribution protocol. Throughout the section, we assume $\mathcal{C}$ is an augmented base-exponent cryptosystem, $P = \{p1, p2\}$, and A is a nonempty set.

B.1 The Endpoint Automata

We define two symmetric automata, for the two elements of P.

$DH(\mathcal{C}, P)_{p1}$:

Signature:

Input:
$IC\text{-}receive(b)_{p2,p1}$, $b \in set_{\mathcal{C}}(\text{“}B\text{”})$

Output:
$IC\text{-}send(b)_{p1,p2}$, $b \in set_{\mathcal{C}}(\text{“}B\text{”})$
$grant(b)_{p1}$, $b \in set_{\mathcal{C}}(\text{“}B\text{”})$

Internal:
$choose\text{-}exp_{p1}$

States:

$chosen\text{-}exp \in [XConst1_{\mathcal{C}}] \cup \{\perp\}$, initially $\perp$
$base\text{-}sent$, a Boolean, initially *false*
$rcvd\text{-}base \in set_{\mathcal{C}}(\text{“}B\text{”}) \cup \{\perp\}$, initially $\perp$
$granted$, a Boolean, initially *false*

Derived variables:
$chosen\text{-}base \in set_{\mathcal{C}}(\text{“}B\text{”}) \cup \{\perp\}$, given by:
if $chosen\text{-}exp \neq \perp$ then $exp([b0_{\mathcal{C}}], chosen\text{-}exp)$ else $\perp$

Transitions:

$choose\text{-}exp_{p1}$
Precondition:
$chosen\text{-}exp = \perp$
Effect:
$chosen\text{-}exp :=$ choose x
where $x \in [XConst1_{\mathcal{C}}]$

$IC\text{-}send(b)_{p1,p2}$
Precondition:
$chosen\text{-}exp \neq \perp$
$b = chosen\text{-}base$
$base\text{-}sent = false$
Effect:
$base\text{-}sent := true$

$IC\text{-}receive(b)_{p2,p1}$
Effect:
$rcvd\text{-}base := b$

$grant(b)_{p1}$
Precondition:
$chosen\text{-}exp \neq \perp$
$rcvd\text{-}base \neq \perp$
$b = exp(rcvd\text{-}base, chosen\text{-}exp)$
$granted = false$
Effect:
$granted := true$

The automaton for $p2$ is the same, but interchanges uses of $p1$ and $p2$, and likewise of *XConst1* and *XConst2*.

$DH(\mathcal{C}, P)_{p2}$:

Signature:

Input:
$IC\text{-}receive(b)_{p1,p2}$, $b \in set_{\mathcal{C}}(\text{“}B\text{”})$
Output:
$IC\text{-}send(b)_{p2,p1}$, $b \in set_{\mathcal{C}}(\text{“}B\text{”})$
$grant(b)_{p2}$, $b \in set_{\mathcal{C}}(\text{“}B\text{”})$

Internal:
$choose\text{-}exp_{p2}$

States:

$chosen\text{-}exp \in [XConst2_{\mathcal{C}}] \cup \{\perp\}$, initially $\perp$
base-sent, a Boolean, initially *false*
$rcvd\text{-}base \in set_{\mathcal{C}}(\text{“}B\text{”}) \cup \{\perp\}$, initially $\perp$
granted, a Boolean, initially *false*

Derived variables:
$chosen\text{-}base \in set_{\mathcal{C}}(\text{“}B\text{”}) \cup \{\perp\}$, given by:
if $chosen\text{-}exp \neq \perp$ then $exp([b0_{\mathcal{C}}], chosen\text{-}exp)$ else $\perp$

Transitions:

$choose\text{-}exp_{p2}$
Precondition:
$chosen\text{-}exp = \perp$
Effect:
$chosen\text{-}exp :=$ choose x
where $x \in [XConst2_{\mathcal{C}}]$

$IC\text{-}send(b)_{p2,p1}$
Precondition:
$chosen\text{-}exp \neq \perp$
$b = chosen\text{-}base$
$base\text{-}sent = false$
Effect:
$base\text{-}sent := true$

$IC\text{-}receive(b)_{p1,p2}$
Effect:
$rcvd\text{-}base := b$

$grant(b)_{p2}$
Precondition:
$chosen\text{-}exp \neq \perp$
$rcvd\text{-}base \neq \perp$
$b = exp(rcvd\text{-}base, chosen\text{-}exp)$
$granted = false$
Effect:
$granted := true$

B.2 The Complete Implementation

In the rest of this section, we assume: $U' = set_{\mathcal{C}}$; $K' = [B2_{\mathcal{C}}]$; $X' = [XConst1_{\mathcal{C}}] \cup [XConst2_{\mathcal{C}}]$; $N' = K' \cup X'$.

The implementation consists of two endpoint automata, an insecure channel, an eavesdropper and an environment. Specifically, implementation $KDImpl(\mathcal{C}, P, A)$ is the composition of the following automata, with certain actions hidden:

- $DH(\mathcal{C}, P)_p$, $p \in P$, endpoint automata.
- $IC(U', P, A)$, $Eve(\mathcal{C}, P, A)$, $Env(U', A, N')$.

To get $KDImpl(\mathcal{C}, P, A)$, we hide: $eavesdrop_{p,q,a}$, $p, q \in P$, $p \neq q$, $a \in A$; $IC\text{-}send_{p,q}$, $IC\text{-}receive_{p,q}$, $p, q \in P$, $p \neq q$; $learn_a$, $a \in A$.

B.3 Invariants

In the system *KDImpl*, we use $DH(p)$ for $p \in P$, *IC*, and *Eve* as handles to help in naming state variables in the composed state. The first invariant says that messages that have been received or are in transit are correct:

Invariant B.1. *In all reachable states of KDImpl, the following are true:*

1. *If* $DH(p).rcvd\text{-}base \neq \perp$ *and* $q \neq p$ *then* $DH(q).chosen\text{-}exp \neq \perp$, *and* $DH(p).rcvd\text{-}base = DH(q).chosen\text{-}base$.
2. *If* $u \in IC.buffer(p, q)$, *then* $DH(p).chosen\text{-}exp \neq \perp$, *and* $u = DH(p).chosen\text{-}base$.

Remark: There was a typo in part 1 of invariant B.1 as stated in [Lyn99]. Client names p and q were reversed in the conclusion, which read $DH(q).rcvd\text{-}base = DH(p).chosen\text{-}base$ instead of $DH(p).rcvd\text{-}base = DH(q).chosen\text{-}base$.

The next two invariants say that no N' elements ever appear in *Eve.has* or in the insecure channel.

Invariant B.2. *In all reachable states of KDImpl, for all* $p, q \in P$, $p \neq q$, *and all* $u \in N'$, $u \notin IC.buffer(p, q)$.

Invariant B.3. *In all reachable states of KDImpl, if* $u \in N'$ *then* $u \notin Eve.has$.

B.4 Implementation Proof

We show that $KDImpl(\mathcal{C}, P, A)$ implements $KD(U, P, K, A)$ using a refinement mapping. The mapping F is defined as follows:

1. If $s.DH(p).chosen\text{-}exp \neq \perp$ for all $p \in P$, then $F(s).chosen\text{-}key = exp(s.DH(p1).chosen\text{-}base, s.DH(p2).chosen\text{-}exp)$, and otherwise $F(s).chosen\text{-}key = \perp$.
2. $F(s).notified = \{p \in P : s.DH(p).granted\}$.

Theorem B.4. *F is a refinement mapping.*

Theorem B.5. $KDImpl(\mathcal{C}, P, A) \preceq KD(U, P, K, A)$.

C Combining Diffie-Hellman Key Distribution with Private Communication

Now we are ready to combine the Diffie-Hellman key distribution implementation with the rest of the private communication protocol. The new private communication protocol is identical to *PCImpl₁*, with one change: the key distribution specification *KD* is replaced by Diffie-Hellman key distribution. As we said earlier, we assume that the key distribution protocol and the private communication protocol have separate insecure channels and eavesdroppers, and the eavesdroppers do not communicate with each other.

Let $\mathcal{C}$ be a shared-key cryptosystem and $\mathcal{E}$ a base-exponent cryptosystem. In this section, we assume: $U = set_{\mathcal{C}}$; $M = [MConst]_{\mathcal{C}}$; $K = [[B2]_{\mathcal{E}}]_{\mathcal{C}}$; $N = M \cup K$; $U' = set_{\mathcal{E}}$; $K' = [B2_{\mathcal{E}}]$; $X' = [XConst1]_{\mathcal{E}} \cup [XConst2]_{\mathcal{E}}$; $N' = K' \cup X'$; $P = \{p1, p2\}$; A is a nonempty set; A' is an arbitrary set, disjoint from A. Note that the key set K of cryptosystem $\mathcal{C}$ consists of doubly-exponentiated bases of cryptosystem $\mathcal{E}$.

As before, the implementation consists of encoder and decoder components, an insecure channel, eavesdropper and environment, plus a key distribution module. More precisely, the implementation, $PCImpl_2(\mathcal{C}, \mathcal{E}, P, A, A')$, is obtained by composing the following automata and then hiding certain actions:

- $Enc(\mathcal{C}, P)_{p,q}$, $Dec(\mathcal{C}, P)_{p,q}$, $p, q \in P$, $p \neq q$.
- $IC(U, P, A)$, $Eve(\mathcal{C}, P, A)$, $Env(U, A, N)$.
- $KDImpl(\mathcal{E}, P, A')$, the key distribution module.

In this system, the eavesdropper *Eve* does not acquire any information directly from the *KDImpl* component, and conversely, the eavesdropper inside *KDImpl* cannot receive information from outside *KDImpl*.

To get $PCImpl_2(\mathcal{C}, \mathcal{E}, P, A, A')$, we hide the following actions in the composition just defined: $eavesdrop_{p,q,a}$, $p, q \in P$, $a \in A$; $IC\text{-}send_{p,q}$, $IC\text{-}receive_{p,q}$, $p, q \in P$; $grant_p$, $p \in P$; $learn_a$, $a \in A$; $reveal_a$, $a \in A'$.

Theorem C.1. $PCImpl_2(\mathcal{C}, \mathcal{E}, P, A, A') \preceq PC(U, P, M, A)$.

Proof. From theorem B.5 we have $KDImpl(\mathcal{E}, P, A') \preceq KD(U', P, K, A')$. The only difference between $PCImpl_2$ and $PCImpl_1$ is that we have substituted $KDImpl(\mathcal{E}, P, A')$ for $KD(U', P, K, A')$. So by the compositionality theorem 2.2, $PCImpl_2(\mathcal{C}, \mathcal{E}, P, A, A') \preceq PCImpl_1(\mathcal{C}, P, A, U', A')$. The result then follows by theorem A.6 and transitivity of $\preceq$. □

Acknowledgments

The authors wish to thank Nancy Lynch for her support and encouragement for our project and her feedback to an earlier draft of this paper.

This research is sponsored in part by the the National Science Foundation under Grant No. CCR-9523972 and the Department of Defense under Award Number MDA904-99-C-5020. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied of the U.S. Government.