# First-Order Verification of Cryptographic Protocols

Ernie Cohen
Microsoft Research

**Abstract**

We describe a verification method for cryptographic protocols, based on first-order invariants. For typical protocols, a suitable invariant can be generated mechanically from the program text, allowing safety properties to be proved by ordinary first-order reasoning.

The method has been implemented in an automatic verifier, TAPS, that proves safety properties comparable to those in published Isabelle verifications, but does so much faster with little or no guidance from the user. TAPS has verified properties of about 80 protocols, including all but three protocols from the Clark & Jacob survey; on average, these verifications require less than a second of CPU time and less than 4 bytes of hints from the user.

## 1 Introduction

A *cryptographic protocol* is a program that uses cryptographic primitives (e.g., ciphers and hash functions), typically to implement distributed computing primitives (e.g., reliable messaging, shared memory) in a hostile communication environment. These protocols are notoriously tricky; for example, bugs have been found in a number of published authentication protocols. The brevity and subtlety of these protocols make them an attractive target for formal methods.

Most modern methods for verifying these protocols model them as concurrent systems, consisting of a collection of principals following the rules of the protocol along with an active adversary who controls all message delivery and can generate new messages from old ones using a limited set of operations (typically tupling, untupling, encryption, decryption, and nonce/key generation). Protocol properties (e.g. authentication) are then checked using conventional program verification techniques. This approach seems to strike a good balance between the complexity-based proofs favored by cryptographers [3, 4] (which cannot be produced for most existing protocols) and proofs in special-purpose authentication logics [7, 11] (which are insensitive to many flaws of interest).

A variety of verification techniques have been applied to the resulting concurrent systems; we give only some examples here. Model checkers have been used to find bugs [15, 9], but combinatorial explosion in the number of sessions/roles forces the models to be severely constrained, which can lead to missed attacks[1]. Bounded-session techniques using symbolic traces and constraints (e.g. [6]) suffer from the same problem[2]. Symbolic search tools like Athena [23] and Rankanalyser [13] are much faster, and can often prove protocols correct without bounding the number of sessions, but depend on bounded message size[3] and structural limitations on the protocol[4]. Finally, approaches that use classical program invariants (e.g. [21, 22, 17, 10]) handle unbounded message depth and protocol instances, but require judiciously chosen recursively defined sets of messages [18], and require substantial effort and proof-checking expertise.

We present a very different approach to verifying protocols in the unbounded model, based on first-order invariants. Unlike other approaches, our invariants are constructed without regard for the properties being checked. Nevertheless, suitable invariants can be constructed automatically for most protocols, allowing safety properties to be proved with first-order reasoning.

Our method is implemented in an automatic verifier, TAPS, that proves safety properties roughly equivalent to those proved in published Isabelle verifications. However, TAPS generates these proofs quickly (typically around a second or less), with almost no user guidance (none for 90% of our examples, an average of about 40 bytes each for the remaining ones). TAPS has verified about 80 protocols, including almost all of the Clark & Jacob Survey (see section 5). TAPS can perform sophisticated mathematical reasoning on functions and predicates as part of a verification process, a feature important for our long term goal of analyzing systems in which cryptography is used as part of a larger security infrastructure. It can also handle recursive protocols to a limited extent, and has verified protocols that use chains of certificates and hashes. The main downside compared to search-based systems is that TAPS does not generate counterexamples (although the failure of a proof obligation can provide information useful in the construction of such counterexamples).

Although TAPS can verify many protocols quickly and fully automatically, we emphasize that *TAPS searches for proofs, not attacks.* The utility of TAPS lies not in its ability to find attacks that search tools miss, but in its ability to quickly find proofs that make such searches unnecessary.

[1] For example, overconstraint caused Murphi to miss a serious bug in a rationalized reconstruction of SSL [20], a bug we found by trying to prove the protocol correct.

[2] They do, however, handle unbounded message size.

[3] These bounds can often be justified by strong typing, which can be added to most to most protocols using the tagging scheme of [12]. However, this doesn't work for protocols that require unbounded encryption depth (e.g. SKEY and paywords), and the additional explicitness of type tagging can open up new avenues of attack (e.g., offline guessing attacks). Moreover, our experience TAPS shows that strong typing is rarely needed for protocol correctness.

[4] For example, protocols must be loop-free (so they can't handle recursive protocols, repeated authentication, or certificate chains) and can't test messages for inequality (so they can't handle protocols that use such tests to prevent replay or reflection attacks). Similar limitations apply to verifiers based on monadic Horn theories, e.g. [25, 5]

## 1.1 Roadmap

We model protocols as transition systems, where the state of the system is given by the set of transitions that have been executed and the set of messages that have been *published* (i.e., sent in the clear). A typical transition generates some fresh values (to be used as nonces or keys), checks that some precondition holds, records that the transition has taken place, and publishes a new message. Several implicit transitions model the actions of the spy, and the states of the system can be further restricted by user-supplied *axioms*.

From the protocol description, we generate a number of invariants. All but one of these is invariant by construction. The one exception is the *secrecy invariant*, which says that if a message is published, then either we know something interesting about the states of some principals, or the message is a tuple or encryption whose components are also published[5]. Safety properties like authentication are proved from the invariants using ordinary first-order reasoning.

To show the invariance of the secrecy invariant, we have to show that it is closed under the destructors; proving this usually requires substantial reasoning about message authentication. Fortunately, we have all of the invariants to work with (including the secrecy invariant), so these proofs are rather easy. If one of these proofs fails, it suggests the last step of an attack through which illicit information can leak out to the spy. (It does not, however, guarantee the existence of such an attack.)

In TAPS, all of this logical reasoning is performed by a resolution theorem prover[6]; For most protocols, TAPS generates a suitable secrecy invariant without any help, but for some protocols, the user has to provide hints (typically a formula proposing conditions necessary for the publication of a particular nonce, key, or nested encryption)[7].

We illustrate the method with what has become the standard example, the Needham-Schroeder-Lowe (NSL) protocol [14], for which the complete TAPS input appears in figure 1. Because we want to show both how the method can be applied by hand and how it is used within TAPS, we present a complete hand proof, but describe TAPS only down to the level of first-order proof obligations. (To make the proof easier to follow, and to illustrate some additional features, figure 1 includes some axioms and hints that TAPS doesn't actually need.) Additional examples of TAPS input appear in the appendix.

[5]Backward chaining through this invariant leads one to consider only smaller published messages, so this method does not suffer from the same infinite regression problem as backward symbolic search through destructors.

[6]The current version of TAPS uses SPASS [26]; previous versions used Gandalf [24] and Otter [16].

[7]These hints only effect the choice of secrecy invariant; they do not affect the soundness of the verification.

```
Protocol NeedhamSchroederLowe
/* k(X) represents X's public key, but the protocol below works
   for any type of key                                          */

Definitions {
    m0 = {A,Na}_k(B)
    m1 = {B,Na,Nb}_k(A)
    m2 = {Nb}_k(B)
}
Transitions {
    /* A->B */  Na: pub(A) /\ pub(B)    -p0-> m0
    /* B->A */  Nb: pub(B) /\ pub(m0)   -p1-> m1
    /* A->B */      p0 /\ pub(m1)       -p2-> m2
    /* B    */      p1 /\ pub(m2)       -p3-> {}
    /* A    */      p0 /\ dk(k(A))      -oopsNa-> Na
    /* B    */      p1 /\ dk(k(B))      -oopsNb-> Nb
}
Axioms {
   /* Not necessary - honest principals can safely share keys  */
   k injective
}
Labels {
   /* Hints used to generate the secrecy invariant (illustration
      only - TAPS generates better ones automatically)
      the spy sees Na/Nb only if A or B is compromised          */

    Na, Nb: dk(k(A)) \/ dk(k(B))
}
Goals {
    /* if A has executed p2, then B has executed p1 with the
       same values for A,B,Na,Nb, or one of their private keys
       has been compromised */

    p2 => p1 \/ dk(k(A)) \/ dk(k(B))

    /* if B has executed p3, then A has executed p2 with the
       same values for A,B,Na,Nb, or one of their private keys
       has been compromised */

    p3 => p2 \/ dk(k(A)) \/ dk(k(B))
}
```

Figure 1: TAPS input for the Needham-Schroeder-Lowe (NSL) protocol

# 2 The protocol model

## 2.1 Messages

Each protocol makes use of an underlying set of *messages* whose structure is given by a first-order theory[8]. Identifiers starting with uppercase letters ($X$,$Y$,$Na$,…) are first-order variables, ranging over messages; in displayed formulas, all such variables are implicitly universally quantified. Identifiers starting with lowercase letters ($nil$, $pub$, $p0$,…) are first-order functions, first-order predicates, and state predicates (section 2.2). The message theory includes the following predicate and function symbols (individual protocols may use additional predicate and function symbols):

| | |
|---|---|
| $nil$ | the empty message |
| $cons(X, Y)$ | the ordered pair $\langle X, Y\rangle$ |
| $enc(X, Y)$ | encryption of $Y$ under the key $X$ |
| $atom(X)$ | iff $X$ is an atomic message (e.g., a nonce or a key) |
| $d(X, Y)$ | iff messages encrypted under $X$ can be decrypted using $Y$ |

The first-order theory says that *nil*, *cons*, and *enc* are injective, with disjoint ranges, and do not yield atoms . The user can provide additional axioms in the `Axioms` section.

**example:** The NSL protocol makes use of the additional function $k$, mapping names of principals to their public keys. The injectivity of $k$ means that principals do not share public keys[9].
**end of example**

TAPS allows the standard comma-separated list notation for right-associated tuples; for example, `{X,Y,z}` abbreviates $cons(X, cons(Y, cons(z, nil)))$. TAPS uses the binary infix operator `_` as a synonym for *enc* (with arguments reversed), so `{A,B}_X` abbreviates $enc(X, cons(A, cons(B, nil)))$. To make protocols easier to read and modify, TAPS allows the user to define abbreviations for message terms (in the `Definitions` section).

**example (continued):** The `Definitions` section of the NSL protocol introduces the definition

$$m0 \quad = \quad enc(k(B), cons(A, cons(Na, nil)))$$

**end of example**

[8]We do not assume an induction principle for message destructors, so for example there is nothing to prevent the message theory from having a fixed point for encryption.

[9]Using the built-in declarations for injective functions and hash functions lets TAPS do a better job of simplifying lemmas and proof obligations, leading to more efficient verification.

## 2.2 States

We model protocols as transition systems, where the state of the system is given by the interpretations of a (protocol-specific) collection of state predicates. (Recall that in standard model theory, the interpretation of a predicate is a set of tuples, so a state predicate can also be viewed as an ordinary program variable ranging over sets of message tuples.) Each protocol makes use of the special state predicate *pub*; $pub(X)$ means that the message $X$ has been *published* (sent in the clear). The remaining state predicates are *history predicates* that track which transitions have been executed. For each history predicate $\mathsf{p}$, TAPS generates a *signature* $\Sigma_{\mathsf{p}}$ (a list of distinct variables), as described in section 2.3.1; in formulas, $\mathsf{p}$ abbreviates the formula $\mathsf{p}(\Sigma_{\mathsf{p}})$. TAPS does not allow history predicates to appear with explicit arguments (although argument substitution can be achieved using the import notation of section 3); this prevents users from writing formulas whose meaning depends on the ordering of variables within a signature. This discipline makes it much easier to read and modify large protocols (whose state predicates might contain a dozen or more arguments).

**example (continued):** In the NSL protocol, $p0$ is a history predicate, with signature $\Sigma_{p0} = A, B, Na$. Thus, the guard of the third transition, $p0 \wedge pub(m1)$, is syntactic sugar for the formula $p0(A, B, Na) \wedge pub(m1)$.
**end of example**

TAPS also defines[10]

$$dk(X) \Leftrightarrow (\exists\, Y : d(X, Y) \wedge pub(Y))$$

*dk* is, in effect, a state predicate (except that it can't be assigned to); $dk(X)$ means that messages encrypted under $X$ are decryptable by the spy.

Various flavors of keys can be defined using the predicate $d$, but weaker specifications in terms of *dk* are strong enough for all practical purposes. For example, the predicate $sk(X)$ ("$X$ is a symmetric key") could be defined with the axiom

$$sk(X) \Leftrightarrow (\forall Y : d(X, Y) \Leftrightarrow X = Y)$$

but we prefer the weaker axiom

$$sk(X) \Rightarrow (dk(X) \Leftrightarrow pub(X))$$

Explicit axioms for keys are usually needed only for protocols that dynamically generate such keys (for example, the protocols in the appendix); simple protocols like NSL can be written in a form that is independent of the kind of encryption used (e.g., by writing goals using *dk* instead of *pub*).

A formula without free (message) variables is said to be *stable* if, once true, it is guaranteed to remain true thereafter. (A formula with free variables is stable

[10]Simply adding this formula as an axiom would be unsound, because TAPS would treat *dk* as an ordinary first-order predicate (i.e., assume that its interpretation remains fixed during execution).

iff each of its instances is stable.) All state predicates are stable (messages, once published, remain published; transitions, once executed, remain executed). A formula is *positive* if no state predicate occurs with negative polarity (i.e., guarded by an odd number of negations); a simple inductive argument shows that positive formulas are also stable.

## 2.3 Protocols

The behavior of a protocol is given by a set of *actions*, each of the form

$$\mathit{guard} \longrightarrow \mathsf{p}(\mathit{args})$$

where *guard* is a formula, $\mathsf{p}$ is a state predicate, and *args* is a list of terms. This action is executed as follows:

1. choose arbitrary message values for the free variables of *guard* and *args*;
2. check that *guard* holds in the current state;
3. modify the state by adding the tuple *args* to the interpretation of $\mathsf{p}$;
4. check that all user-supplied axioms hold.

A protocol is executed by starting from a state where all state predicates are *false* and all user-supplied axioms hold, and executing an arbitrary sequence of actions.

### 2.3.1 Transitions

Protocol actions are not given directly; instead, the user provides a set of *transitions*, labeled with distinct history predicates. The transition labeled $\mathsf{p}$ has the form

$$\mathsf{nv_p} : \quad \mathsf{g_p} \quad \xrightarrow{\mathsf{p}} \quad \mathsf{M_p}$$

where $\mathsf{nv_p}$ is a list of distinct variables (the *nonce variables* of $\mathsf{p}$), $\mathsf{g_p}$ is a positive formula, and $\mathsf{M_p}$ is a message term. For convenience, we will assume that the $\mathsf{nv_p}$'s are disjoint, although this restriction is not essential. TAPS treats any predicate symbol that appears as the label of a transition as a history predicate, and assumes the remaining predicate symbols are ordinary first-order predicates of the underlying message theory. Then, for each history predicate $\mathsf{p}$, TAPS chooses a minimal signature $\Sigma_\mathsf{p}$ that includes all free variables of $\mathsf{nv_p}$, $\mathsf{g_p}$ and $\mathsf{M_p}$; this restriction determines the signatures uniquely, up to variable reordering[11]. The transition above is syntactic sugar for the two actions

$$\begin{aligned} \mathit{fresh}(\mathsf{nv_p}) \wedge \mathsf{g_p} \quad &\longrightarrow \quad \mathsf{p}(\Sigma_\mathsf{p}) \\ \mathsf{p}(\Sigma_\mathsf{p}) \quad &\longrightarrow \quad \mathit{pub}(\mathsf{M_p}) \end{aligned}$$

[11]Because TAPS allows recursive protocols (e.g., where $\mathsf{p}$ occurs in its own guard), this definition of $\Sigma_\mathsf{p}$ is actually recursive, but it still has a minimal solution.

where $\mathit{fresh}(\mathsf{nv_p})$ means that the values assigned to the variables of $\mathsf{nv_p}$ are distinct atoms that have not been previously chosen as instantiations for nonce variables of state predicates[12]; formally,

$$\begin{array}{rcl} \mathit{fresh}(\mathsf{v}_0, \ldots) & \Leftrightarrow & (\forall i : \mathit{atom}(\mathsf{v}_i)) \wedge (\forall i, j : i \neq j \Rightarrow \mathsf{v}_i \neq \mathsf{v}_j) \\ & \wedge & (\forall i, \mathsf{p}, \mathsf{w} : \mathsf{w} \in \mathsf{nv_p} \wedge \mathsf{p}(\Sigma_{\mathsf{p}}{}') \Rightarrow \mathsf{v}_i \neq \mathsf{w}') \end{array}$$

(This effect is approximated in real protocols by choosing fresh atoms randomly from a large pool of candidates.)

**example (continued):** Here are the transitions of the NSL protocol:

$$\begin{array}{rlcl} \mathit{Na}: & \mathit{pub}(A) \wedge \mathit{pub}(B) & \xrightarrow{p0} & m0 \\ \mathit{Nb}: & \mathit{pub}(B) \wedge \mathit{pub}(m0) & \xrightarrow{p1} & m1 \\ & p0 \wedge \mathit{pub}(m1) & \xrightarrow{p2} & m2 \\ & p1 \wedge \mathit{pub}(m2) & \xrightarrow{p3} & \mathit{nil} \\ & p0 \wedge \mathit{dk}(k(A)) & \xrightarrow{oopsNa} & \mathit{Na} \\ & p1 \wedge \mathit{dk}(k(B)) & \xrightarrow{oopsNb} & \mathit{Nb} \end{array}$$

From these transitions, TAPS generates the signatures

$$\begin{array}{rcl} \Sigma_{p0} = \Sigma_{oopsNa} & = & A, B, \mathit{Na} \\ \Sigma_{p1} = \Sigma_{p2} = \Sigma_{p3} = \Sigma_{oopsNb} & = & A, B, \mathit{Na}, \mathit{Nb} \end{array}$$

The first transition translates to the (desugared) actions

$$\begin{array}{rcl} \mathit{fresh}(\mathit{Na}) \wedge \mathit{pub}(A) \wedge \mathit{pub}(B) & \longrightarrow & p0(A, B, \mathit{Na}) \\ p0(A, B, \mathit{Na}) & \longrightarrow & \mathit{pub}(m0) \end{array}$$

The first of these actions is executed by choosing arbitrary values for $A$, $B$, and $\mathit{Na}$, checking that $A$ and $B$ are published (a common, weak way to test values as being suitable for use as names of principals) and that $\mathit{Na}$ is a fresh atom, adding the tuple $\langle A, B, \mathit{Na} \rangle$ to the interpretation of $p0$, and finally checking that all of the user-defined axioms hold. (The NSL axioms don't mention any state predicates, so the last check is unnecessary.)
**end of example**

In addition to the transitions explicitly given, each protocol implicitly includes actions modeling the capabilities of the spy. These include the transition

$$\mathit{Trash}: \quad \mathit{true} \quad \xrightarrow{trash} \quad \mathit{Trash}$$

[12] Note that nothing prevents a transition from making use of an atom that has not yet been issued as a nonce value; typically, this will prevent verification of the secrecy invariant.

(the spy can generate new atoms) and the actions

$$\begin{array}{lcl}
\mathit{true} & \longrightarrow & \mathit{pub}(\mathit{nil}) \\
\mathit{pub}(X) \wedge \mathit{pub}(Y) & \longrightarrow & \mathit{pub}(\mathit{cons}(X,Y)) \\
\mathit{pub}(X) \wedge \mathit{pub}(Y) & \longrightarrow & \mathit{pub}(\mathit{enc}(X,Y)) \\
\mathit{pub}(\mathit{cons}(X,Y)) & \longrightarrow & \mathit{pub}(X) \\
\mathit{pub}(\mathit{cons}(X,Y)) & \longrightarrow & \mathit{pub}(Y) \\
\mathit{pub}(\mathit{enc}(X,Y)) \wedge \mathit{dk}(X) & \longrightarrow & \mathit{pub}(Y)
\end{array}$$

(the spy can tuple, encrypt, and untuple previously published messages, and decrypt messages encrypted with decryptable keys). Note that in each case, the spy publishes the result, allowing these actions to be chained together.

## 3 Imports

To motivate our last bit of notational machinery, consider a transition system with the single action

$$g \longrightarrow \mathit{pub}(t)$$

where $g$ is positive (hence stable) and $t$ is a term. Suppose we arrive in a state where $\mathit{pub}(m)$ holds, for some value $m$. If $m$ has no free variables, we can conclude

$$(\exists\, V : g \wedge t = m)$$

where $V$ is a superset of the variables occurring free in $g$ or $t$. Intuitively, this is because $\mathit{pub}(m)$ must have been truthified by the action above (it's the only one); the execution that truthified it must have chosen variable bindings for which $t = m$; $g$ must have held just before the action was executed (for the chosen bindings), and must hold now (because $g$ is stable). In the general case where $m$ might contain free variables, we have to rename variables of $g$ and $t$ to avoid capturing free variables of $m$ in the scope of the existential quantification, so we write instead

$$(\mathbf{I}\, V' : g' \wedge t' = m)$$

where $g'$ is $g$ with each variable primed (similarly for $t'$). It is helpful to think of this formula as specifying a new context (the one given by the values of the primed variables), related to the surrounding context by the constraint $t' = m$. (In general, we may have several such constraints.) Since the list of variables $V$ and the primed symbols are really not carrying any information other than separating the constraints from the new context, we abbreviate this formula with the new notation

$$(\mathbf{I}\, [m\colon t] :\ g)$$

("import $m$ as $t$ in $g$").

In general, let $V$ be the set of all free variables in the formula $\mathsf{f}$ or in any of the terms $\mathsf{Y}_i$, let $V'$ be $V$ with all variables primed, and let $\mathsf{f}'$ be $\mathsf{f}$ with all variables primed. We define

$$\begin{array}{rcl}(\mathbf{I}\ [\mathsf{X}_0 : \mathsf{Y}_0, \mathsf{X}_1 : \mathsf{Y}_1, \ldots] :\ \mathsf{f}) & \Leftrightarrow & (\exists\ V' : \mathsf{X}_0 = \mathsf{Y}'_0 \wedge \mathsf{X}_1 = \mathsf{Y}'_1 \wedge \ldots \wedge \mathsf{f}') \\ (\mathbf{I}\ [\mathsf{X}_0, \mathsf{X}_1, \ldots] :\ \mathsf{f}) & \Leftrightarrow & (\mathbf{I}\ [\mathsf{X}_0 : \mathsf{X}_0, \mathsf{X}_1 : \mathsf{X}_1, \ldots] :\ \mathsf{f})\end{array}$$

(The first form is pronounced "import $\mathsf{Y}_0$ as $\mathsf{X}_0$, $\mathsf{Y}_1$ as $\mathsf{X}_1$,... from $\mathsf{f}$", the second as "import $\mathsf{X}_0$,$\mathsf{X}_1$,... from $\mathsf{f}$".) For example,

$$\begin{array}{lcl}(\mathbf{I}\ [X \cdot Y] :\ p(X, Z)) & \Leftrightarrow & (\exists\ X', Y', Z' : X \cdot Y = X' \cdot Y' \wedge p(X', Z')) \\ (\mathbf{I}\ [f(X) : X] :\ p(X)) & \Leftrightarrow & p(f(X))\end{array}$$

In hand proofs, we work directly with imports using the following rules (here, $h$ is injective and $\mathsf{fv.f}$ is the set of free variables in $\mathsf{f}$):

$$\begin{array}{rcl}\mathsf{f} & \Rightarrow & (\mathbf{I}\ [\ldots] :\ \mathsf{f}) \\ (\mathbf{I}\ [\ldots] :\ \mathsf{f} \vee \mathsf{g}) & \Leftrightarrow & (\mathbf{I}\ [\ldots] :\ \mathsf{f}) \vee (\mathbf{I}\ [\ldots] :\ \mathsf{g}) \\ (\mathbf{I}\ [h(\mathsf{X}), \ldots] :\ \mathsf{f}) & \Leftrightarrow & (\mathbf{I}\ [\mathsf{X}, \ldots] :\ \mathsf{f}) \\ \mathsf{f} \wedge (\mathbf{I}\ [\mathsf{fv.f}, \ldots] :\ \mathsf{g}) & \Leftrightarrow & (\mathbf{I}\ [\mathsf{fv.f}, \ldots] :\ \mathsf{f} \wedge \mathsf{g})\end{array}$$

TAPS uses imports internally only to facilitate communication with the user; it replaces imports with existential quantification before doing any deduction.

## 4 Invariants

For each protocol, TAPS generates several invariants:

- *guard lemmas* say that history predicates imply their guards;
- *unicity lemmas* say that nonce values are not reused[13];
- the *secrecy invariant* classifies the messages that the protocol can publish, and what conditions are implied by each.

The guard and unicity lemmas are generated in such a way that their invariance is guaranteed without further justification. However, the secrecy invariant is conjectural, and its invariance depends on discharging a number of proof obligations.

### 4.1 Guard Lemmas

For each history variable $\mathsf{p}$ (other than *trash*), TAPS generates the invariant

$$\mathsf{p} \Rightarrow (\mathsf{g}_\mathsf{p} \wedge (\forall \mathsf{v} \in \mathsf{nv}_\mathsf{p} : \mathit{atom}(\mathsf{v})))$$

[13]Our unicity lemmas are similar to those used by Paulson [21], except that they are generated in a uniform way.

Intuitively, this is an invariant because the implicant holds before $\mathsf{p}$ is first truthified, and the implicant is positive (hence stable).

**example (continued):** NSL generates the following guard lemmas:

$$\begin{array}{rcl} p0 & \Rightarrow & pub(A) \wedge pub(B) \wedge atom(Na) \\ p1 & \Rightarrow & pub(B) \wedge pub(m0) \wedge atom(Nb) \\ p2 & \Rightarrow & p0 \wedge pub(m1) \\ p3 & \Rightarrow & p1 \wedge pub(m2) \\ oopsNa & \Rightarrow & p0 \wedge dk(k(A)) \\ oopsNb & \Rightarrow & p1 \wedge dk(k(B)) \end{array}$$

**end of example**

## 4.2 Unicity Lemmas

Unicity lemmas come in two flavors. The first unicity lemma says that for variable assignments satisfying a given state predicate, the value assigned to any nonce variable of the predicate uniquely determines the other variables in the signature of the predicate (because nonce variables are not reused). Formally, for $\mathsf{v} \in \mathsf{nv}_\mathsf{p}$,

$$\mathsf{p}(\Sigma_\mathsf{p}) \wedge \mathsf{p}({\Sigma_\mathsf{p}}') \wedge \mathsf{v} = \mathsf{v}' \Rightarrow \Sigma_\mathsf{p} = {\Sigma_\mathsf{p}}'$$

TAPS actually uses the following (equivalent) formulation, which is more efficient for most theorem provers: for $\mathsf{v} \in \mathsf{nv}_\mathsf{p}$, $\mathsf{w} \in \Sigma_\mathsf{p}$,

$$\mathsf{p}(\Sigma_\mathsf{p}) \Rightarrow \mathsf{w} = f_{\mathsf{v},\mathsf{w}}(\mathsf{v})$$

where $f_{\mathsf{v},\mathsf{w}}$ is a new Skolem function.

For hand proofs, we want to avoid explicit arguments to state predicates, so we use instead the following schema: for any formula $\mathsf{f}$,

$$\mathsf{p} \wedge (\mathbf{I}\ [\mathsf{v}, \ldots] :\ \mathsf{p} \wedge \mathsf{f})) \Leftrightarrow (\mathbf{I}\ [\Sigma_\mathsf{p}, \ldots] :\ \mathsf{p} \wedge \mathsf{f})$$

This captures the way we use unicity in hand proofs – to widen the scope of an import.

The second unicity lemma says that the same nonce value cannot be assigned to two different nonce variables: for distinct nonce variables $\mathsf{v} \in \mathsf{nv}_\mathsf{p}$ and $\mathsf{w} \in \mathsf{nv}_\mathsf{q}$,

$$\mathsf{p}(\Sigma_\mathsf{p}) \wedge \mathsf{q}({\Sigma_\mathsf{q}}') \Rightarrow \mathsf{v} \neq \mathsf{w}'$$

or, equivalently,

$$\neg(\mathsf{p} \wedge (\mathbf{I}\ [\mathsf{v} : \mathsf{w}] :\ \mathsf{q}))$$

TAPS does not generate explicit unicity lemmas involving the spy-generated atoms (*Trash*), because they are not needed for useful protocol properties. These lemmas are, however, needed to prove the soundness of the nonce lemmas below.

**example (continued):** NSL generates the following unicity lemmas:

$$\begin{array}{rcl} p0(A,B,Na) & \Rightarrow & A = f_{Na,A}(Na) \land B = f_{Na,B}(Na) \\ p1(A,B,Na,Nb) & \Rightarrow & A = f_{Nb,A}(Nb) \land B = f_{Nb,B}(Nb) \land Na = f_{Nb,Na}(Nb) \end{array}$$

$$p0(A,B,Na) \land p1(A',B',Na',Nb') \Rightarrow Na \neq Nb'$$

For hand proofs, we have instead the lemma schemas

$$\begin{array}{rcl} p0 \land (\mathbf{I}\,[Na,\ldots] :\ p0 \land \mathsf{f}) & \Leftrightarrow & (\mathbf{I}\,[A,B,Na,\ldots] :\ p0 \land \mathsf{f}) \\ p1 \land (\mathbf{I}\,[Nb,\ldots] :\ p1 \land \mathsf{f}) & \Leftrightarrow & (\mathbf{I}\,[A,B,Na,Nb,\ldots] :\ p1 \land \mathsf{f}) \\ \neg(p0 \land (\mathbf{I}\,[Na\colon Nb] :\ p1)) & & \end{array}$$

**end of example**

## 4.3 The Secrecy Invariant

We would like to prove an invariant *inv* given by a definition of the form

$$inv \Leftrightarrow (\forall X : pub(X) \Rightarrow ok(X))$$

where *ok* is a stable, unary predicate[14] Intuitively, *ok* is our way of describing the constraints we put on the set of published messages; we will use a hypothetical proof of the invariance of *inv* to generate a suitable definition of *ok*. (Similarly, we will use the predicates *prime* and *primeAt* below before giving their definitions.)

To show that *inv* is an invariant, we must show that it holds initially and that it is stable. *inv* holds initially because *pub* is initially *false*. To show the stability of *inv*, we need to show that it is preserved by every action of the protocol. Because *ok* is stable, *inv* is preserved by actions that do not update *pub*, so we need only consider actions of the form $\mathsf{g} \longrightarrow pub(\mathsf{M})$; to show that such an action preserves *inv*, it suffices to prove the proof obligation

$$inv \land \mathsf{g} \Rightarrow ok(\mathsf{M})$$

(again, because $ok(\mathsf{M})$ is stable). Each of the proof obligations assumes *inv*, and their collective proofs implies the invariance of *inv*, so we simply assume *inv* throughout, keeping in mind that anything we prove along the way is meaningless unless all of the proof obligations are discharged.

Expanding out the proof obligations for all of the relevant program actions gives us the following set of formulas to prove:

(1) $\mathsf{p} \Rightarrow ok(\mathsf{M_p})$

(2) $ok(nil)$

[14]We describe *ok*, *prime* and *primeAt* as predicates to emphasize that their interpretations can depend on the state.

$$\begin{array}{lrcl}
(3) & pub(X) \wedge pub(Y) & \Rightarrow & ok(cons(X,Y)) \\
(4) & pub(X) \wedge pub(Y) & \Rightarrow & ok(enc(X,Y)) \\
(5) & pub(cons(X,Y)) & \Rightarrow & ok(X) \wedge ok(Y) \\
(6) & pub(enc(X,Y)) \wedge dk(X) & \Rightarrow & ok(Y)
\end{array}$$

We satisfy (2–5) as follows. Let *prime* be a stable unary predicate (defined below), and define

$$\begin{array}{ll}
ok(X) \Leftrightarrow & (X = nil) \vee prime(X) \\
& \vee(\mathbf{I}\ [X : cons(Y,Z)] :\ pub(Y) \wedge pub(Z)) \\
& \vee(\mathbf{I}\ [X : enc(Y,Z)] :\ pub(Y) \wedge pub(Z))
\end{array}$$

(2)–(4) then follow immediately. Since

$$\begin{array}{ll}
pub(cons(X,Y)) \wedge \neg prime(cons(X,Y)) & \Rightarrow \{inv\quad\} \\
ok(cons(X,Y)) \wedge \neg prime(cons(X,Y)) & \Rightarrow \{\text{def } ok\} \\
pub(X) \wedge pub(Y) & \Rightarrow \{inv\quad\} \\
ok(X) \wedge ok(Y) &
\end{array}$$

the proof obligation (5) follows from

(7) $prime(cons(X,Y)) \Rightarrow ok(X) \wedge ok(Y)$

Similarly, the obligation (6) follows from

(8) $prime(enc(X,Y)) \wedge dk(X) \Rightarrow ok(Y)$

We are thus left with the obligations (1), (7), and (8).

The obligations (1) are each of the form $\mathsf{f} \Rightarrow ok(\mathsf{X})$. Although there is no induction principle for messages, there is an induction principle for terms, so we repeatedly replace these obligations with simpler ones based on the structure of $\mathsf{X}$:

- drop obligations of the form $\mathsf{f} \Rightarrow ok(nil)$;
- replace the obligation $\mathsf{f} \Rightarrow ok(cons(\mathsf{Y},\mathsf{Z}))$ with the obligations $\mathsf{f} \Rightarrow prime(cons(\mathsf{Y},\mathsf{Z}))$, $\mathsf{f} \Rightarrow ok(\mathsf{Y})$ and $\mathsf{f} \Rightarrow ok(\mathsf{Z})$;
- replace the obligation $\mathsf{f} \Rightarrow ok(enc(\mathsf{Y},\mathsf{Z}))$ with the obligations $\mathsf{f} \Rightarrow prime(enc(\mathsf{Y},\mathsf{Z}))$ and $\mathsf{f} \wedge dk(\mathsf{Y}) \Rightarrow ok(\mathsf{Z})$.

The soundness of each of these rules follows from the definition of *ok*.

**example (continued):** To illustrate this procedure, consider the first transition of the NSL protocol. The corresponding obligation (1) is

$$p0 \Rightarrow ok(m0)$$

Since $m0$ is defined as $enc(k(B), cons(A, cons(Na, nil)))$, this obligation is replaced with the obligations

$$\begin{array}{rcl}
p0 \wedge dk(k(B)) & \Rightarrow & ok(cons(A, cons(Na, nil))) \\
p0 & \Rightarrow & prime(m0)
\end{array}$$

The first of these obligations, in turn, is replaced with

$$\begin{array}{rcl} p0 \wedge dk(k(B)) & \Rightarrow & prime(cons(A, cons(Na, nil))) \\ p0 \wedge dk(k(B)) & \Rightarrow & ok(A) \\ p0 \wedge dk(k(B)) & \Rightarrow & ok(cons(Na, nil)) \end{array}$$

**end of example**

This procedure leaves us with obligations of the forms

- $\mathsf{f}_i \Rightarrow prime(cons(\mathsf{Y}_i, \mathsf{Z}_i))$;
- $\mathsf{g}_j \Rightarrow prime(enc(\mathsf{Y}_j, \mathsf{Z}_j))$; and
- $\mathsf{f}_k \Rightarrow ok(\mathsf{Z}_k)$, where $\mathsf{Z}_k$ is not an application of *nil*,*cons*, or *enc*.

We satisfy obligations of the first two types by defining

$$\begin{array}{rcl} prime(X) & \Leftrightarrow & primeCons(X) \vee primeEnc(X) \vee primeAt(X) \\ primeCons(X) & \Leftrightarrow & (\vee i : (\mathbf{I}\ [X : cons(\mathsf{Y}_i, \mathsf{Z}_i)] :\ \mathsf{f}_i)) \\ primeEnc(X) & \Leftrightarrow & (\vee j : (\mathbf{I}\ [X : enc(\mathsf{Y}_j, \mathsf{Z}_j)] :\ \mathsf{g}_j)) \end{array}$$

where *primeAt* is a stable unary predicate such that $primeAt(X) \Rightarrow atom(X)$. Because we already introduced the obligations $\mathsf{f} \Rightarrow ok(\mathsf{Y}_i)$ and $\mathsf{f} \Rightarrow ok(\mathsf{Z}_i)$ for each proof obligation of the form $\mathsf{f}_i \Rightarrow prime(cons(\mathsf{Y}_i, \mathsf{Z}_i))$, this definition of *prime* satisfies (7). Because we already introduced the obligation $\mathsf{f} \wedge dk(\mathsf{Y}_i) \Rightarrow ok(\mathsf{Z}_i)$ for each proof obligation of the form $\mathsf{f}_i \Rightarrow prime(enc(\mathsf{Y}_i, \mathsf{Z}_i))$, this definition of *prime* satisfies (8).

The proof obligations of the third type are delegated to the resolution theorem prover. To discharge these, we need to generate a suitable definition of *primeAt*. TAPS defines it as

$$primeAt(X) \Leftrightarrow (\vee \mathsf{v}, \mathsf{p} : \mathsf{v} \in \mathsf{nv}_\mathsf{p} \wedge (\mathbf{I}\ [X : \mathsf{v}] :\ \mathsf{p} \wedge \mathsf{L}_\mathsf{v}))$$

where $\mathsf{L}_\mathsf{v}$ (the *label* of $\mathsf{v}$) is a formula giving conditions under which an atom generated as $\mathsf{v}$ might be published.

Because these conditions might be arbitrarily complex, TAPS allows the user to specify $\mathsf{L}_\mathsf{v}$ in the `Labels` section; if absent, TAPS chooses

$$\mathsf{L}_\mathsf{v} \equiv (\vee k : (\mathsf{Z}_k \equiv \mathsf{v}) \wedge \mathsf{f}_k)$$

where $k$ ranges over the obligations of the third type and $\equiv$ is syntactic identity. This choice of $\mathsf{L}_\mathsf{v}$ is heuristic and fragile (it generates different invariants for protocols that are equivalent modulo renaming of variables in a transition), but it has proved very effective in practice. This heuristic leverages information implicit in the choice of variable names used in a protocol: if one protocol transition generates a fresh value, and another transition publishes a message with a component using the same name, then by stripping off encryption surrounding the published component, an adversary should only be able to get information about the generated value. The best defense of this heuristic is that it chooses correct nonce labels for all of the examples we have tried (i.e., it allowed us to simply remove existing nonce labels without further modification to the protocols).

**example (continued):** In NSL (figure 1), we explicitly gave labels for the two nonce types:

$$\mathsf{L}_{Na} = \mathsf{L}_{Nb} = dk(k(A)) \vee dk(k(B))$$

Intuitively, this says that the nonces should only be published only if the private key of one of the two participating principals has been compromised. This annotation generates the definitions

$$primeEnc(X) \Leftrightarrow (\mathbf{I}\,[X : m0] :\ p0) \vee (\mathbf{I}\,[X : m1] :\ p1) \vee (\mathbf{I}\,[X : m2] :\ p2)$$

$$\begin{array}{rll} primeAt(X) \Leftrightarrow & & (\mathbf{I}\,[X : Na] :\ p0 \wedge (dk(k(A)) \vee dk(k(B)))) \\ & \vee & (\mathbf{I}\,[X : Nb] :\ p1 \wedge (dk(k(A)) \vee dk(k(B)))) \\ & \vee & (\mathbf{I}\,[X : Trash] :\ trash) \end{array}$$

and the proof obligations:

$$\begin{array}{lcl} p0 \wedge dk(k(B)) & \Rightarrow & ok(A) \\ p0 \wedge dk(k(B)) & \Rightarrow & ok(Na) \\ p1 \wedge dk(k(A)) & \Rightarrow & ok(B) \\ p1 \wedge dk(k(A)) & \Rightarrow & ok(Na) \\ p1 \wedge dk(k(A)) & \Rightarrow & ok(Nb) \\ p2 \wedge dk(k(B)) & \Rightarrow & ok(Nb) \\ oopsNa & \Rightarrow & ok(Na) \\ oopsNb & \Rightarrow & ok(Nb) \end{array}$$

Had the labels not been given explicitly, TAPS would have defined

$$\begin{array}{lcl} \mathsf{L}_{Na} & \Leftrightarrow & (p0 \wedge dk(k(B))) \vee (p1 \wedge dk(k(A))) \vee\ oopsNa) \\ \mathsf{L}_{Nb} & \Leftrightarrow & (p1 \wedge dk(k(A))) \vee (p2 \wedge dk(k(B))) \vee\ oopsNb) \end{array}$$

**end of example**

## 4.4 Consequences of the secrecy invariant

Instead of working with *inv* directly, TAPS uses the following corollaries:

$$\begin{array}{rcl} & & ok(nil) \\ ok(cons(X,Y)) & \Rightarrow & ok(X) \wedge ok(Y) \\ pub(X) \wedge pub(Y) & \Rightarrow & ok(cons(X,Y)) \\ ok(enc(X,Y)) & \Leftrightarrow & primeEnc(enc(X,Y)) \vee (pub(X) \wedge pub(Y)) \\ \mathsf{p} & \Rightarrow & (ok(\mathsf{v}) \Leftrightarrow (\mathbf{I}\,[\Sigma_{\mathsf{p}}] :\ \mathsf{L}_{\mathsf{v}})) \text{ for } \mathsf{v} \in \mathsf{nv}_{\mathsf{p}} \end{array}$$

The first four corollaries are jointly referred to as *message lemmas*, and corollaries of the last type are called *nonce lemmas*. To see that the nonce lemmas are sound, for $\mathsf{v} \in \mathsf{nv_p}$,

$$\begin{array}{ll}
\mathsf{p} \wedge ok(\mathsf{v}) & \Leftrightarrow \{\text{guard lemma for } \mathsf{p}\} \\
\mathsf{p} \wedge atom(\mathsf{v}) \wedge ok(\mathsf{v}) & \Leftrightarrow \{\text{def of } ok\} \\
\mathsf{p} \wedge primeAt(\mathsf{v}) & \Leftrightarrow \{\text{def of } primeAt\} \\
\mathsf{p} \wedge (\vee\ \mathsf{w}, \mathsf{q} : \mathsf{w} \in \mathsf{nv_q} \wedge (\mathbf{I}\ [\mathsf{v}\colon \mathsf{w}] :\ \mathsf{q} \wedge \mathsf{L_w})) & \Leftrightarrow \{\text{logic}\} \\
(\vee\ \mathsf{w}, \mathsf{q} : \mathsf{w} \in \mathsf{nv_q} \wedge \mathsf{p} \wedge (\mathbf{I}\ [\mathsf{v}\colon \mathsf{w}] :\ \mathsf{q} \wedge \mathsf{L_w})) & \Leftrightarrow \{\text{second unicity lemma}\} \\
\mathsf{p} \wedge (\mathbf{I}\ [\mathsf{v}] :\ \mathsf{p} \wedge \mathsf{L_v}) & \Leftrightarrow \{\text{first unicity lemma}\} \\
(\mathbf{I}\ [\Sigma_\mathsf{p}] :\ \mathsf{L_v}) &
\end{array}$$

**example (continued):** Using these corollaries, we rewrite $pub(m0)$ as follows:

$$\begin{array}{ll}
pub(m0) & \Rightarrow \{inv\} \\
ok(m0) & \Rightarrow \{\text{corollaries of } ok\} \\
primeEnc(m0) \vee (pub(k(B)) \wedge ok(A) \wedge ok(Na)) & \Rightarrow \{\text{def } primeEnc\} \\
(\mathbf{I}\ [m0] :\ p0) \vee (\mathbf{I}\ [m0\colon m1] :\ p1) \vee\ (\mathbf{I}\ [m0\colon m2] :\ p2) & \\
\vee\ (pub(k(B)) \wedge ok(A) \wedge ok(Na)) & \Rightarrow \{m0 \neq m1', m2'\} \\
(\mathbf{I}\ [m0] :\ p0) \vee (pub(k(B)) \wedge ok(A) \wedge ok(Na)) & \Rightarrow \{\text{injectivities}\} \\
p0 \vee (pub(k(B)) \wedge ok(A) \wedge ok(Na)) &
\end{array}$$

By similar reasoning,

$$\begin{array}{lll}
pub(m1) & \Rightarrow & p1 \vee (pub(k(A)) \wedge ok(B) \wedge ok(Na) \wedge ok(Nb)) \\
pub(m2) & \Rightarrow & (\mathbf{I}\ [B, Nb] :\ p2) \vee (pub(k(B)) \wedge ok(Nb))
\end{array}$$

NSL also generates the nonce lemmas

$$\begin{array}{lll}
p0 & \Rightarrow & (ok(Na) \Leftrightarrow (dk(k(A)) \vee dk(k(B)))) \\
p1 & \Rightarrow & (ok(Nb) \Leftrightarrow (dk(k(A)) \vee dk(k(B))))
\end{array}$$

We can then discharge the proof obligations for NSL as follows:

$$\begin{array}{ll}
p0 \wedge dk(k(B)) & \Rightarrow \{p0 \Rightarrow pub(A)\ \text{(guard lemma for } p0)\} \\
pub(A) & \Rightarrow \{inv\} \\
ok(A) &
\end{array}$$

$$\begin{array}{ll}
p0 \wedge dk(k(B)) & \Rightarrow \{\text{nonce lemma for } Na\} \\
ok(Na) &
\end{array}$$

$$\begin{array}{ll}
p1 \wedge dk(k(A)) & \Rightarrow \{p1 \Rightarrow pub(B)\ \text{(guard lemma for } p1);\ inv\} \\
ok(B) &
\end{array}$$

$$\begin{array}{ll}
p1 \wedge dk(k(A)) & \Rightarrow \{p1 \Rightarrow pub(m0)\ \text{(guard lemma for } p1)\} \\
pub(m0) \wedge dk(k(A)) & \Rightarrow \{\text{message lemma for } m0\} \\
(p0 \vee ok(Na)) \wedge dk(k(A)) & \Rightarrow \{\text{logic}\} \\
(p0 \wedge dk(k(A))) \vee ok(Na) & \Rightarrow \{\text{nonce lemma for } Na\} \\
ok(Na) &
\end{array}$$

$$\begin{array}{ll} p1 \wedge \mathit{dk}(k(A)) & \Rightarrow \{\text{nonce lemma for } \mathit{Nb}\} \\ \mathit{ok}(\mathit{Nb}) & \end{array}$$

$$\begin{array}{ll} p2 \wedge \mathit{dk}(k(B)) & \Rightarrow \{p2 \Rightarrow \mathit{pub}(m1) \text{ (guard lemma for } p2)\} \\ \mathit{pub}(m1) \wedge \mathit{dk}(k(B)) & \Rightarrow \{\text{message lemma for } m1\} \\ (p1 \vee \mathit{ok}(\mathit{Nb})) \wedge \mathit{dk}(k(B)) & \Rightarrow \{\text{logic}\} \\ (p1 \wedge \mathit{dk}(k(B))) \vee \mathit{ok}(\mathit{Nb}) & \Rightarrow \{\text{nonce lemma for } \mathit{Nb}\} \\ \mathit{ok}(\mathit{Nb}) & \end{array}$$

$$\begin{array}{ll} \mathit{oopsNa} & \Rightarrow \{\text{guard lemma for } \mathit{oopsNa}\} \\ p0 \wedge \mathit{dk}(k(A)) & \Rightarrow \{\text{nonce lemma for } \mathit{Na}\} \\ \mathit{ok}(\mathit{Na}) & \end{array}$$

$$\begin{array}{ll} \mathit{oopsNb} & \Rightarrow \{\text{guard lemma for } \mathit{oopsNb}\} \\ p1 \wedge \mathit{dk}(k(B)) & \Rightarrow \{\text{nonce lemma for } \mathit{Nb}\} \\ \mathit{ok}(\mathit{Nb}) & \end{array}$$

**end of example**

**example (continued):** The original Needham–Schroeder public key protocol omitted $B$ from the message $m1$; replacing $B$ with *nil*[15] gives the definition

$$m1 = \mathit{enc}(k(A), \mathit{cons}(\mathit{nil}, \mathit{cons}(\mathit{Na}, \mathit{cons}(\mathit{Nb}, \mathit{nil}))))$$

Thus, we would have only the weaker

$$\mathit{pub}(m1) \Rightarrow (\mathbf{I}\ [A, \mathit{Na}, \mathit{Nb}] :\ p1) \vee\ (\mathit{pub}(k(A)) \wedge \mathit{ok}(\mathit{Na}) \wedge \mathit{ok}(\mathit{Nb}))$$

As a result, it is no longer possible to prove the proof obligation

$$p2 \wedge \mathit{dk}(k(B)) \Rightarrow \mathit{ok}(\mathit{Nb})$$

which says that the third protocol message does not release inappropriate information (i.e., a message that is not *ok*) if the recipient is compromised. This is just what happens in Lowe's man-in-the-middle attack [14] — the last message sent by an honest initiator of a session with a compromised responder can leak to the spy the value of a nonce generated by an uncompromised responder, even though neither the uncompromised responder nor the principal he believes to be his initiator are compromised.
**end of example**

[15] To simplify the comparison, we avoid reducing $m1$ to a 2-tuple, which creates additional complication because of the collision with $m0$.

## 4.5 Proving authentication properties

Finally, the guard, unicity, message, and nonce lemmas are used to prove any desired authentication properties (given in the `Goals` section).

**example (continued):** Here are proofs of the authentication properties for the NSL protocol. The first theorem says that, if $A$ completes his final step, then either $A$ or $B$ is compromised, or $B$ has completed his first step, with the same values for $A$,$B$,$Na$ and $Nb$:

$$\begin{array}{ll}
p2 & \Rightarrow \{\text{guard lemma for } p2\} \\
p0 \wedge pub(m1) & \Rightarrow \{\text{message lemma for } m1\} \\
p0 \wedge (p1 \vee ok(Na)) & \Rightarrow \{\text{logic}\} \\
(p0 \wedge ok(Na)) \vee p1 & \Rightarrow \{\text{nonce lemma for } Na\} \\
dk(k(A)) \vee dk(k(B)) \vee p1 &
\end{array}$$

The second theorem says that, if $B$ completes his final step, then either $A$ or $B$ is compromised, or $A$ has completed his final step, with the same values for $A$,$B$,$Na$ and $Nb$:

$$\begin{array}{ll}
p3 & \Rightarrow \{\text{guard of } p3\} \\
p1 \wedge pub(m2) & \Rightarrow \{\text{message } m2\} \\
p1 \wedge ((\mathbf{I}\ [B, Nb] :\ p2) \vee ok(Nb)) & \Rightarrow \{\text{guard of } p2\} \\
p1 \wedge ((\mathbf{I}\ [B, Nb] :\ p2 \wedge pub(m1)) \vee ok(Nb)) & \Rightarrow \{\text{message } m1\} \\
p1 \wedge ((\mathbf{I}\ [B, Nb] :\ p2 \wedge (p1 \vee ok(Nb))) \vee ok(Nb)) & \Rightarrow \{\text{logic}\} \\
p1 \wedge ((\mathbf{I}\ [B, Nb] :\ p2 \wedge p1) \vee ok(Nb)) & \Rightarrow \{\text{logic}\} \\
(p1 \wedge ok(Nb)) \vee (p1 \wedge (\mathbf{I}\ [B, Nb] :\ p2 \wedge p1)) & \Rightarrow \{\text{unicity for } Nb\} \\
(p1 \wedge ok(Nb)) \vee (\mathbf{I}\ [B, \Sigma_{p1}] :\ p2)) & \Rightarrow \{\text{nonce } Nb\} \\
dk(k(A)) \vee dk(k(B)) \vee (\mathbf{I}\ [B, \Sigma_{p1}] :\ p2)) & \Rightarrow \{\Sigma_{p2} \subseteq \Sigma_{p1}\} \\
dk(k(A)) \vee dk(k(B)) \vee p2 &
\end{array}$$

**end of example**

## 4.6 Output Annotations

There is an important case where the secrecy invariant generated by the procedure above is too strong (and TAPS is unable to discharge the resulting proof obligations). The problem arises when one protocol step publishes a nested encryption, and another protocol step strips off the outer encryption and publishes the inner encryption without looking at its structure[16]. The best-known protocols that do this kind of "blind stripping" are the Needham-Schroeder Symmetric Key Protocol and various flavors of Kerberos.

[16]This issue does not arise if the inner encryption is re-packaged and re-encrypted with the original key (e.g. as is done in the final message of the Needham-Schroeder Symmetric Key protocol).

**example:** Consider the following (artificial) protocol, where $m0 = enc(Na, nil)$:

$$\begin{array}{lll} Na: & true & \xrightarrow{p0} & enc(k, m0) \\ & enc(k, Y) & \xrightarrow{p1} & Y \end{array}$$

TAPS generates the following definition of *primeEnc*:

$$primeEnc(X) \Leftrightarrow (\mathbf{I}\ [X : enc(k, m0)] :\ p0) \vee (\mathbf{I}\ [X : m0] :\ p0 \wedge dk(k))$$

However, this invariant is too strong (it isn't an invariant), and TAPS fails to prove the proof obligation $p1 \Rightarrow ok(Y)$. To get a satisfactory invariant, the second disjunct above has to be weakened to

$$(\mathbf{I}\ [X : m0] :\ p0 \wedge (dk(k) \vee (\mathbf{I}\ [m0 : Y] :\ p1)))$$

**end of example**

To cater to such protocols, we allow the user to guide the construction of the secrecy invariant by labeling subterms of outputs of protocol steps with positive formulas. The label is written in square brackets; the formula associated with this label is given in the `Labels` section. Intuitively, the formula gives conditions that necessarily hold if the surrounding encryption is ever stripped away, revealing a previously unpublished message. We add a new rule for breaking up a proof obligation of the form $\mathsf{f} \Rightarrow ok(\mathsf{X})$:

- replace the obligation $\mathsf{f} \Rightarrow ok(\texttt{[S]}\mathsf{X})$ with the obligations $\mathsf{f} \Rightarrow \mathsf{L}_S$ and $\mathsf{L}_S \Rightarrow ok(\mathsf{Y})$, where $\mathsf{L}_S$ is the formula associated with the label `S`.

**example (continued):** To generate an appropriate secrecy invariant for the protocol shown above, we decorate the first transition with the label $S$

$$\begin{array}{lll} Na: & true & \xrightarrow{p0} & enc(k, [S]m0) \\ & enc(k, Y) & \xrightarrow{p1} & Y \end{array}$$

where $\mathsf{L}_S = p0 \wedge (dk(k) \vee (\mathbf{I}\ [m0 : Y] :\ p1))$. The modified proof obligation rules allow TAPS to generate the appropriate definition of *prime*.
**end of example**

For a more realistic example where explicit annotation is necessary, see the Needham-Schroeder shared key protocol (figure 3 in the appendix), which uses explicit annotation on the subterm $m0$ published in step $p1$. In practically all cases, the hints are of precisely the form illustrated in the example above. The generation of these hints could be further automated by introducing notation that makes explicit the correspondence between the variable bindings in actions that send and receive messages. (For example, both the Casper [15] and CAPSL [19] languages provide such notations.)

## 4.7 Transition Labels and Recursive Protocols

Recall that we required the guard of a transition to be positive, in order to guarantee that the guard is stable. TAPS actually allows arbitrary formulas as guards, and allows the user to provide a positive formula $\mathsf{L}(\mathsf{p})$ to use in place of $\mathsf{g_p}$ the guard lemma. (The user provides this formula in the `Labels` section). The guard lemma then becomes

$$\mathsf{p} \Rightarrow (\mathbf{I}\ [\Sigma_{\mathsf{p}}] :\ \mathsf{L}(\mathsf{p})) \wedge (\forall \mathsf{v} \in \mathsf{nv_p} : \mathit{atom}(\mathsf{v})))$$

and TAPS has to prove the additional proof obligation

$$\mathsf{g_p} \wedge \mathit{fresh}(\mathsf{nv_p}) \Rightarrow (\mathbf{I}\ [\Sigma_{\mathsf{p}}] :\ \mathsf{L}(\mathsf{p}))$$

We originally intended these labels to be used in protocols like Woo-Lam, where a principal has to check that incoming nonce challenges do not collide with nonce challenges previously sent out; such a test introduces a negation in the guard. However, it is just as easy (if slightly less honest) to simply rule out such collisions using an axiom. In practice, the critical application of transition labels is in verifying recursive protocols. The procedure is similar to Floyd's method for sequential program verification: we choose a set of "cut" transitions, such that every transition loop is in the cut, and for each cut transition, we add the necessary inductive hypotheses as an explicit label.

**example:** Consider the protocol consisting of the single transition

$$p \xrightarrow{p} \mathit{nil}$$

Defining $\mathsf{L}(p) = \mathit{false}$ in the `Labels` section introduces a new assumption $p \Rightarrow \mathit{false}$ and a new proof obligation $p \Rightarrow \mathit{false}$, which is trivially discharged from the assumption. The conclusion is that $\neg p$ holds in all states.
**end of example**

This approach to recursive protocols is somewhat weaker than what one would wish; there are no recursive functions, and the labels have to be positive. Thus, we can model chains of various sorts, but we have no good way to state conditions such as "$X$'s value is good or one of $X$'s ancestors is compromised". In order to cope with such assertions, we are considering extending the input language to allow users to axiomatize new state predicates.

## 5 Experimental Results

We used TAPS to analyze all but three of the authentication protocols in the Clark & Jacob survey [8] (we did not analyze Diffie-Hellman key exchange (which depends explicitly on exponentiation), Shamir-Rivest-Adelman Three-pass (which uses commutative encryption), or the Gong mutual authentication (which uses XOR)). Such a statement has to be taken with a grain of salt, since the meaning of "verification" depends on how protocols are modeled and what properties are being verified. (For example, TAPS has no built-in notion of "injective authentication", and Casper tests mostly make strong type assumptions, so our results and those obtained using Casper [9] are not completely comparable.) We formalized each of these protocols and tried to prove what we expected the protocols could reasonably achieve. For protocols with known bugs (or bugs that we discovered accidentally), we weakened the specification appropriately and/or fixed the protocol (typically by adding type information to prevent obvious message confusion attacks). Output annotations were needed only for Kerberos, the Needham-Schroeder Symmetric Key protocol, and the Amended Needham-Schroeder Symmetric Key protocol. In addition, a transition label was needed for each of the Woo-Lam protocols (since, in the absence of additional type checks, they have recursive executions). Verification time averaged about half a second per protocol; the longest verification times were for the ISO four-pass protocol and for Kerberos, each of which required about two and a half seconds.

For each protocol, we were either able to prove the desired properties or find anomalous behaviors that prevented verification. The results were generally in agreement with those reported in [9], but we did not detailed comparison[17]. The only definite discrepancy we noticed (i.e., an attack on a protocol that was certified by Casper/FDR, Clark/Jacob, and Brackin) is a self-authentication attack on the ISO Four-Pass protocol, which Casper/FDR could have caught but for a mistake in the checking configuration.

We also used TAPS to verify the examples included in the Isabelle distribution. For each protocol, we tried to reproduce faithfully the Isabelle formalizations and tried to prove those safety properties that seemed to be of interest (as opposed to those that seemed of interest only as a step toward proving other results). The major changes were that (1) we eliminated timestamps from BAN Kerberos and Kerberos (but see section 6.1), (2) we rewrote the Bull-Otway recursive protocol to eliminate the recursive server function, and (3) we eliminated the explicit types and typechecking in those protocols where it seemed unnecessary (basically, wherever it was not needed to disambiguate protocol messages). (The Isabelle proofs would probably also go through without these type assumptions.) For protocols admitting reasonable comparison and for which Paulson provided estimated development times (Yahalom, TLS, Needham-Schroeder public key), the TAPS developments required about an order of magnitude less user time to formalize and verify. (Of course this comparison is somewhat unfair, since we had his protocol models to use as guidance.)

## 6 Detours

In this section, we describe some features that are not currently part of TAPS.

[17]A fair comparison is difficult, for many reasons. For examples, by default, TAPS allows the intruder to copy protocol sessions, but does not require authentication to be injective.

## 6.1 Timestamps

A previous version of TAPS provided built-in support for timestamps, primarily to reproduce the Isabelle analyses of BAN Kerberos [1] and Kerberos [2]. Typically, one wants to show that with appropriate use of timeouts, the authentication properties of the protocol are robust to the addition of "oops" actions that expose nonces or keys to the spy sufficiently long after their generation.

TAPS used *Trash* atoms as timestamps; this guaranteed that they were published and did not collide with atoms generated by other transitions. To reason about timestamp ordering, we introduced a linear ordering $<$ and a binary associative, commutative operation $+$ all values, along with the axioms

$$\begin{array}{rcl} x & < & x + y \\ x < y & \Leftrightarrow & x + z < y + z \end{array}$$

(Intuitively, think of each element of the model being mapped to a positive nonstandard real.)

Timestamps were introduced along with transitions, just like nonces. The introduction of a timestamp $\mathsf{t}$ for a transition $\mathsf{p}$ introduced the following additional proof rules:

- All timestamps are published:

  $$\mathsf{p} \Rightarrow \mathit{pub}(\mathsf{t})$$

- Timestamps uniquely determine the other values in the signature of their transitions:

  $$\mathsf{p} \wedge \mathsf{p}' \wedge \mathsf{t} = \mathsf{t}' \quad \Rightarrow \quad \Sigma_{\mathsf{p}} = \Sigma_{\mathsf{p}}{}'$$

  For $\mathsf{w}$ a nonce or timestamp of $\mathsf{q}$, distinct from $\mathsf{t}$ and *Trash*,

  $$\neg(\mathsf{p} \wedge \mathsf{q}' \wedge \mathsf{t} = \mathsf{w}')$$

- A timestamp is said to be *timely* if it is greater than all other issued timestamps. (Note that timeliness is not stable.) If the transition in which a timestamp is declared is given an explicit label, then the timeliness of the timestamp is added as a conjunct to the guard. Note that the ordering on timestamps is based on the order in which they are used as timestamps, not the order in which they are generated, so the spy can freely forge future or past timestamps.

Using this timestamp mechanism, we were able to reproduce the main Isabelle results on BAN Kerberos and Kerberos. We were also able to add timestamp reasoning to protocols from the Clark and Jacob Survey without much difficulty. However, making use of timestamps required introducing new transition labels on every transition where timestamp freshness is important. This was not surprising; like nonce freshness, timestamp freshness is not stable, but unlike nonce freshness, the content of timestamp freshness cannot be eliminated by introducing simple axioms. However, these hints could have been eliminated by using temporal guard lemmas (see below).

The main reasoning for removing timestamp support from TAPS was that timestamp reasoning slowed theorem proving down substantially (typically a factor of about 3-10; in the extreme case of Kerberos, a factor of 20). The degradation is not entirely surprising; it is well known that transitive relations like $<$ can wreak havoc with theorem provers not specially designed to handle them. The problem could be fixed by changing theorem proving strategies, but this is probably not worth the effort.

## 6.2 General Invariants

Currently, TAPS checks stability of a formula by checking that it is positive. This has not seriously hindered the usefulness of TAPS for cryptographic protocols, because the only nonpositive formulas needed in practice are for protocols with branching behavior; we handle these by simply postulating the necessary exclusion as an axiom. (Axioms like this, which mention only the control state of a principal, are not very risky, and letting TAPS deduce their invariance would not significantly enhance its usefulness.) However, we plan to extend TAPS to allow arbitrary formulas in place of positive ones, checking their stability using standard weakest preconditions. This will make TAPS applicable to a wider variety of systems (e.g., systems where cryptography is used to enforce a dynamically changing security policy).

## 6.3 Temporal Guard Lemmas

The guard lemmas that TAPS generates are not as strong as they might be. If a history predicate is true, TAPS concludes that the guard holds, whereas it could reach the stronger conclusion that the guard held at an earlier time when the history predicate was *false*. This stronger rule would eliminate most of the user-supplied hints needed for recursive protocols and for protocols that use timestamps. For example, applying this to the protocol

$$p \quad \xrightarrow{p} \quad nil$$

TAPS could conclude $\neg p$ without needing a user-supplied transition label. This type of deduction is exploited by several systems, including Athena, Isabelle, and NPA[18]

However, this approach has a drawback: it takes us from first-order reasoning to a form of first-order temporal reasoning, which might have a serious impact in performance. (The reasoning is easier than general temporal reasoning, since all the predicates are stable, but it still involves reasoning in multiple contexts.)

[18]In Athena and NPA, this is built into the search strategy: when searching for how one of a set of terms can be produced, a path is ignored if it requires previous production of a member of the set.

## 7 Acknowledgments

Anish Arora, Steve Brackin, Jay Misra, Larry Paulson, J. R. Rao, Paul Syverson, and several anonymous referees provided valuable feedback on earlier versions of this paper. Dawn Song, James Heather, and Cathy Meadows provided helpful information on Athena, RankAnalyser, and the NPA respectively. The author is grateful to Gavin Lowe for providing access to the verification scripts from the Casper/FDR analysis of the Clark & Jacob survey.

## References

[1] G. Bella and L. Paulson. Mechanising BAN Kerberos by the inductive method. In *CAV 10*, pages 416–427, 1998.

[2] G. Bella and L. C. Paulson. Kerberos version IV: Inductive analysis of the secrecy goals. In *ESORICS 5*, pages 361–375, 1998.

[3] M. Bellaire and P. Rogaway. Entity authentication and key distribution (extended abstract). In *CRYPTO 13*, 1993.

[4] M. Bellare and P. Rogaway. Provably secure session key distribution: the three party case. In *STOC 27*, pages 57–66, 1995.

[5] B. Blanchet. An efficient cryptographic prolotocol verifier based on prolog rules. In *CSFW 14*, 2001.

[6] M. Boreale. Symbolic analysis of cryptographic protocols in the spi-calculus. In *ICALP '01*, 2001.

[7] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. In *Practical Cryptography for Data Internetworks*. IEEE Computer Society Press, 1996.

[8] J. Clark and J. Jacob. A survey of authentication protocol literature: Version 1.0, 1997.

[9] B. Donovan, P. Norris, and G. Lowe. Analyzing a library of security protocols using Casper and FDR. In *Workshop on Formal Methods and Security Protocols*, 1999.

[10] F. Fabrega, J. Herzog, and J. Guttman. Honest ideals in strand spaces. In *CSFW 11*, 1998.

[11] L. Gong, R. Needham, and R. Yahalom. Reasoning About Belief in Cryptographic Protocols. In *Proceedings 1990 IEEE Symposium on Research in Security and Privacy*, pages 234–248, 1990.

[12] J. Heather, G. Lowe, and S. Schneider. How to prevent type flaw attacks on security protocols. In *CSFW 13*, pages 255–268, 2000.

[13] J. Heather and S. Schneider. Towards automatic verification of authentication protocols on an unbounded network. In *CSFW 13*, pages 132–143, 2000.

[14] G. Lowe. Breaking and fixing the needham-schroeder public-key protocol using FDR. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 147–166, 1996.

[15] G. Lowe. Casper: A compiler for the analysis of security protocols. In *CSFW 10*, 1997.

[16] W. McCune. OTTER 3.0 reference manual and guide. Technical report, Argonne National Laboratory, 1994.

[17] C. Meadows. Language generation and verification in the NRL protocol analyzer. In *CSFW 9*, pages 48–61, 1996.

[18] C. Meadows. Invariant generation techniques in cryptographic protocol analysis. In *CSFW 13*, 2000.

[19] J. K. Millen. CAPSL: Common authentication protocol specification language. In *Proceedings on the Workshop on New Security Paradigms*, pages 132–133, 1997.

[20] J. Mitchell, V. Shmatikov, and U. Stern. Finite-state analysis of SSL 3.0. In *Proceedings of the Seventh USENIX Security Symposium*, pages 201–215, 1998.

[21] L. Paulson. The inductive approach to verifying cryptographic protocols. *JCS*, 6:85–128, 1998.

[22] S. Schneider. Verifying authentication protocols with CSP. In *CSFW 10*, 1997.

[23] D. Song. Athena: A new efficient automatic checker for security protocol analysis. In *CSFW 12*, 1999.

[24] T. Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, April 1997.

[25] C. Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In *CADE 16*, 1999.

[26] C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, E. Keen, C. Theobalt, and D. Topić. System description: SPASS version 1.0.0. In *CADE 15*, pages 378–382, 1999.

```
Protocol Yahalom
/* .5 seconds */

Definitions {
        m0 = {A,Na}
        m1 = {A,Na,Nb}_k(B)
        m2 = {B,Kab,Na,Nb}_k(A)
        m3 = {A,Kab}_k(B)
        m4 = {Nb}_Kab
}

Transitions {
/* A->B */  Na : pub(A) /\ pub(B)    -p0-> m0
/* B->S */  Nb : pub(B) /\ pub(m0)   -p1-> m1
/* S->A */  Kab: sk(Kab) /\ pub(m1)
                 /\ pub(A) /\ pub(B) -p2-> {m2,m3}
/* A->B */       p0 /\ pub({m2,X})   -p3-> {X,m4}
/* B    */       p1 /\ pub({m3,m4})  -p4-> {}
/* S->  */       p2                  -oops-> {Na,Nb,Kab}
}

Axioms {
     sk(X) /\ dk(X) => pub(X)
     sk(k(X))
     k injective
}

Goals {
     p3  => pub(k(A))  \/ pub(k(B)) \/ (p1 /\ p2)
     p4  => pub(k(A)) \/ pub(k(B)) \/  (I [A,B,Kab]: p3)
            \/ (I [A,B,Na,Nb]: oops)
}
```

Figure 2: TAPS input for the Yahalom protocol

```
Protocol NeedhamSchroederConventional
/* .6 seconds */

Definitions {
     m0 = {A,B,Na}
     m3 = {Kab,A,{}}_k(B)
     m1 = {Na,B,Kab,[G] m3}_k(A)
     m2 = {Na,B,Kab,X}_k(A)
     m4 = {Nb}_Kab
     m5 = {Nb,Nb}_Kab
}
Transitions {

/* A->S */ Na : pub(A) /\ pub(B)   -p0-> m0
/* S->A */ Kab: pub(m0) /\ sk(Kab) -p1-> m1
/* A->B */      p0 /\ pub(m2)      -p2-> X
/* B->A */ Nb : pub(m3)            -p3-> m4
/* A->B */      p2 /\ pub(m4)      -p4-> m5
/* B    */      p3 /\ pub(m5)      -p5-> {}
                p1 /\ p3           -oops-> {Na,Nb,Kab}
}

Axioms {
     sk(k(X))
     sk(X) /\ dk(X) => X
     k injective
}

Labels {
     G  : p1 /\ (dk(k(A)) \/ (I [m1:m2]:p2))
}

Goals {
p2 => (I [A,B,Kab,Na,X]: p1) \/ ok(Kab)
p3 => (I [A,B,Kab]: p2) \/ pub(k(B)) \/ pub(k(A))
p4 => p3 \/ pub(k(A)) \/ pub(k(B)) \/ (I [A,B,Kab]: oops)
p5 => (I [A,B,Nb,Kab]: p4) \/ pub(k(A)) \/ pub(k(B))
              \/ (I [A,B,Kab]: oops)
}
```

Figure 3: TAPS input for the Needham-Schroeder Shared Key Protocol