

An Automated Approach for Proving PCL Invariants

John C. Mitchell¹ Arnab Roy² Mukund Sundararajan³

*Department of Computer Science
Stanford University
Stanford, U.S.A.*

Abstract

Protocol composition logic, PCL, is a formal approach for proving security properties of a class of network protocols. PCL involves reasoning directly about properties achieved by protocols steps, in a setting that does not require explicit reasoning about attacker actions. The method relies on protocol invariants to combine properties of different roles of a protocol. While some protocol invariants can be complex to identify and prove, many useful PCL invariants are relatively straightforward consequences of the programs (roles) executed by the agents involved in the protocol. We present a logic program based approach for automating proofs of invariants that appears effective for invariants that are required for several standardized, widely deployed protocols. We use the well-known Transport Layer Security Protocol (TLS/SSL) to illustrate the approach.

Keywords: Logic for security protocols, Prolog, automated proofs, Transport Layer Security

1 Introduction

Protocol Composition Logic (PCL) [10,11,3,5,4,7,15] is an evolving formal logical framework for proving security properties of network protocols. The central question addressed by PCL is whether it is possible to prove properties of substantial security protocols compositionally, using reasoning steps that do not mention attacker actions explicitly. In order to reason about protocols compositionally, the proof of properties of one sequence of actions by one agent involves not only local reasoning about the security goal of that component, but also environment conditions that prevent destructive interference from other actions that may use the same certificates or key materials. These environment conditions are generally stated as *protocol invariants*, properties that are true for all of the roles of the protocol at hand, and properties that may be required of any other protocol running in the same environment. Various versions of PCL studied in past work have been proved

¹ Email: mitchell@cs.stanford.edu

² Email: arnab@cs.stanford.edu

³ Email: mukunds@cs.stanford.edu

sound for protocol runs that use any number of principals and sessions, over both symbolic models and (for a subset of the logic at present) over more traditional cryptographic assumptions [8].

The PCL proof method uses extensions of first-order logic with axioms and proof rules for protocol actions, temporal reasoning, knowledge, and a specialized form of invariance rule called the *honesty rule*. The honesty rule is essential for combining facts about one role with inferred actions of other roles, in the presence of attackers. Intuitively, if Alice receives a response from a message sent to Bob, the honesty rule captures Alice’s ability to use properties of Bob’s role to reason about how Bob generated his reply. Roughly, the honesty rule says that if a property ϕ is preserved by all the roles of a protocol, then if Bob executes only roles of the protocol, ϕ is preserved by Bob’s actions. A basic feature of PCL is that the invariants are proved by reasoning only about the actions specified by the protocol, which are the only actions carried out by honest parties to the protocol; the need to consider attacker actions is obviated by the form of the logic. Because honest principals do not have their private keys misused by an attacker (by definition), the honesty rule does not involve reasoning about situations where private keys are compromised.

As we discuss in Sections 2.2 and Section 2.3, while invariants are easy to formulate, checking whether they hold is a tedious process, making invariant checking a perfect target for automation. This is also observed independently by [13]. In this paper, we describe a method for automatically establishing some invariants, using a logic programming formulation of protocol actions, the invariants themselves, and sufficient conditions to establish the stated invariants. In effect, for each protocol and invariant, our method creates a logic program with restricted use of negation that searches for protocol properties that could violate the invariant. This is done in such a way that if the logic program fails to prove a certain goal, it follows that the invariant is provable in PCL. Because we work with conditions that guarantee a PCL proof, and PCL does not require explicit reasoning about actions by an attacker, our logic programs do not need to consider any possible actions by the attacker.

We are primarily interested in invariants that are written as logical implications, since these arise frequently in case studies of practical protocols. However, this form of invariant poses several challenges. One is efficiency and correct termination – our initial approach produced small logic programs that search for violations of sufficient conditions, but take unreasonably long to execute. For soundness, we need to ensure that for *all* possible ways of satisfying the antecedent of an implication, the consequent holds. This leads us to use logic programming with negation as failure, which further exacerbates efficiency issues. The key reasons why we still achieve efficiency are as follows. We use SLD-resolution [17] as the basis of our decision procedure. In PCL, each program variable is defined exactly once. This creates mutual exclusion among the rules of our logic program; at most any one such rule is applicable at any point in the goal search. Further, our encoding of the PCL proof system ensures that unsuccessful search paths terminate quickly. Thus the reason why we get computational efficiency is closely related to the structure of PCL programs and the PCL proof system required to establish invariants. While effective in useful cases, our method is not complete since it

uses sufficient conditions that are not necessary for the existence of a PCL proof. Further, as explained above, reasoning about many important invariants does not involve cryptographic properties and so our technique is applicable to both the symbolic and computational models of cryptography.

While we do not attempt here to address the full range of criticisms and conceptual misunderstandings presented recently in [2], we do note that in the process of providing a translation of portions of PCL into logic programming, this paper does clarify some syntactic issues involving the PCL term algebra and implicit typing restrictions that are discussed in [2].

2 Preliminaries

We first describe Protocol Composition Logic (PCL) in brief to help make the paper accessible to readers unfamiliar with PCL. We then discuss the well-known Transport Layer Security (TLS) protocol, which we use as a running example throughout the paper. Readers familiar with PCL and the analysis of TLS using PCL may skip to Section 3.

2.1 Overview of PCL

The framework of Protocol Composition Logic (PCL) [10,3,5,7] comprises a protocol programming language for defining the roles of a protocol, a proof system for establishing properties of protocols, and soundness theorems relating proof rules to semantics of the protocol programs and logic. Proofs in PCL are sound in the standard symbolic model; in this model, all cryptographic primitives are assumed to be perfect, for instance an attacker can only decrypt an encrypted message only if it has the appropriate key.

A protocol is defined by a set of roles, each specifying a sequence of actions to be executed by an honest agent. In PCL, protocol roles are formally defined using a simple “protocol programming language” based on [10], which we illustrate using the TLS Client and Server roles in the next section. The possible protocol actions include nonce generation, signatures and encryption, communication steps, and decryption and signature verification via pattern matching. Programs can also depend on input parameters (typically determined by context or the result of set-up operations) and provide output parameters to subsequent operations. A thread is an instance of a protocol role.

The state of execution of a protocol at a given time is the sequence events that have occurred till that time. Events can be communication steps (send/receive) or internal actions (encryption, signature, hashing and so on). Most protocol proofs use formulas of the form $\theta[P]_X\phi$, which means that starting from a state where formula θ is true, after actions P are executed by the thread X , the formula ϕ is true in the resulting state. Formulas ϕ typically make assertions about temporal order of actions (useful for stating authentication) and/or the data accessible to various principals (useful for stating secrecy). As a notational convention, the principal executing a thread (sequence of actions) X is written as \hat{X} . In the next section, we list a proof of authentication of TLS.

The proof system extends first-order logic with axioms and proof rules for protocol actions, temporal reasoning, properties of cryptographic primitives, and a specialized form of program invariance rule called the *honesty rule*, described in Section 2.3. We list below the axioms we use in this paper. For the proof of soundness of the axioms and the rules, we refer the reader to [6,11,7].

- AA1** $\top[a]_X a$
AA2 $\text{Start}(X)[\]_X \neg a(X)$
AA3 $\neg \text{Send}(X, t)[b]_X \neg \text{Send}(X, t)$ if $\sigma \text{Send}(X, t) \neq \sigma b$ for all substitutions σ
AA4 $\top[a; \dots; b]_X a(b)$
- AN1** $\text{New}(X, x) \wedge \text{New}(Y, x) \supset X = Y$
AN2 $\top[\text{new } x;]_X \text{Has}(Y, x) \supset (Y = X)$
AN3 $\top[\text{new } x;]_X \text{Fresh}(X, x)$
AN4 $\text{Fresh}(X, x) \supset \text{Gen}(X, x)$
- VER** $\text{Honest}(\hat{X}) \wedge \text{Verify}(Y, \text{SIG}[\hat{X}](x)) \supset \exists X. \text{Sign}(X, x)$
- P1** $\text{Persist}(X, t)[a]_X \text{Persist}(X, t)$, for $\text{Persist} \in \{\text{Has}, \text{FirstSend}, a, \text{Gen}\}$.
P2 $\text{Fresh}(X, t)[a]_X \text{Fresh}(X, t)$, where $t \not\subseteq a$.
- FS1** $\text{Fresh}(X, t)[\text{send } t';]_X \text{Contains}(t', t) \supset \text{FirstSend}(X, t, t')$
FS2 $\text{FirstSend}(X, t, t') \wedge a(Y, t'') \wedge X \neq Y \wedge \text{Contains}(t', t) \wedge \text{Contains}(t'', t)$
 $\supset \text{Send}(X, t') < a(Y, t'')$

Axioms represents general truths applicable to every protocol. For instance, the axiom **VER** encodes the well known property of signatures that if a thread verifies that a message x is signed by a principal \hat{Y} , it must be that \hat{Y} 's signature key was used to generate the signature. Further, if the agent \hat{Y} is honest (the meaning of honesty is discussed further in Section 2.3), no one else has access to this key, implying that there exists a thread of the agent Y that did indeed sign the term x .

2.2 TLS

In this section we discuss the Transport Layer Security (TLS) protocol [9]. Broadly, TLS involves two principals called the TLS client and the TLS server. TLS guarantees mutual authentication and establishes a shared secret between the two principals. We focus on the proof of the authentication property and identify the program invariant that it needs. We also established other security properties of TLS as part of a larger study [12] on IEEE 802.11i.

We first express the TLS protocol in our protocol programming language. Though TLS has several modes of operation, we restrict our attention to the mode where both the server and the client have certificates. The **TLS : Client** and **TLS : Server** programs are described in Table 1. In the programs \hat{X} and \hat{Y} denote the identities of the principals and $(\mathbf{I})[\text{actions}]_X \langle \mathbf{O} \rangle$ specifies a thread X of principal \hat{X} , executing the actions inside brackets and none after that; \mathbf{I} and \mathbf{O} are the input and output parameters of the thread.

| | |
|---|--|
| <pre> TLS : Client = (X, \hat{Y}, V_x)[new n_x; send $\hat{X}.\hat{Y}.n_x.V_x$; receive $\hat{Y}.\hat{X}.n_y.V_y$; new <i>secret</i>; <i>enckey</i> := pkenc <i>secret</i>, \hat{Y}; <i>sigterm</i> := $\hat{X}.\hat{Y}.n_x.V_x \cdot \hat{Y}.\hat{X}.n_y.V_y \cdot$ <i>enckey</i>; <i>sigvx</i> := sign <i>sigterm</i>, \hat{X}; <i>hc2</i> := hash $\hat{X}.\hat{Y}.n_x.V_x \cdot \hat{Y}.\hat{X}.n_y.V_y \cdot$ <i>enckey</i> · <i>sigvx</i> · “client”, <i>secret</i>; send $\hat{X}.\hat{Y}.enckey.sigvx.hc2$; receive $\hat{Y}.\hat{X}.hs''$; <i>hs'</i> := $\hat{X}.\hat{Y}.n_x.V_x \cdot \hat{Y}.\hat{X}.n_y.V_y \cdot \hat{X}.\hat{Y} \cdot$ <i>enckey</i> · <i>sigvx</i> · “server”; verifyhash <i>hs''</i>, <i>hs'</i>, <i>secret</i>;]X(X, \hat{Y}, <i>secret</i>) </pre> | <pre> TLS : Server = (Y, V_y)[receive $\hat{X}.\hat{Y}.n_x.V_x$; new n_y; send $\hat{Y}.\hat{X}.n_y.V_y$; receive $\hat{X}.\hat{Y}.enckey.sig.hc2''$; <i>sigterm</i> := $\hat{X}.\hat{Y}.n_x.V_x \cdot \hat{Y}.\hat{X}.n_y.V_y \cdot$ <i>enckey</i>; verify <i>sig</i>, <i>sigterm</i>, \hat{X}; <i>secret</i> := pkdec <i>enckey</i>, \hat{Y}; <i>hc2'</i> := $\hat{X}.\hat{Y}.n_x.V_x \cdot \hat{Y}.\hat{X}.n_y.V_y \cdot$ <i>enckey</i> · <i>sig</i> · “client”; verifyhash <i>hc2''</i>, <i>hc2'</i>, <i>secret</i>; <i>hs</i> := hash $\hat{X}.\hat{Y}.n_x.V_x \cdot \hat{Y}.\hat{X}.n_y.V_y \cdot \hat{X}.\hat{Y} \cdot$ <i>enckey</i> · <i>sig</i> · “server”, <i>secret</i>; send $\hat{Y}.\hat{X}.hs$;]Y(Y, \hat{X}, <i>secret</i>) </pre> |
|---|--|

Table 1
TLS: Client and Server Programs

2.2.1 Modeling TLS

We briefly describe an execution of the TLS client role. The client starts by sending a nonce n_x and its configuration information V_x to the server. It then receives a nonce n_y and the server’s configuration information, V_y . It now generates a nonce, *secret*, encrypts it under the server’s public key and sends this to the server; this is the secret that is shared between the two principals. It also concatenates all the messages it has seen so far, and sends its signature over this concatenation to the server; this allows the server to verify that the clients view of the protocol execution matches its own. Finally, the client receives from the server a keyed hash (keyed by the term *secret*) of the concatenation of all the messages the server has seen; this allows client to verify that its view of the protocol run matches the server’s view. These steps use the **verify** and **verifyhash** actions to check signatures and keyed hashes respectively.

We now discuss the authentication guarantee for the TLS server. There is a symmetric guarantee for the TLS client; we focus on the server guarantee as an example. The guarantee states that on completion of the server role, principal \hat{Y} is sure that it is communicating with its intended client \hat{X} , further, the principals agree on each other’s identity, protocol completion status, the values of the protocol version, cryptographic suite, and the *secret* that the client sends to the server. The authentication property $\phi_{tls,auth}$ is formulated in terms of matching conversations [1]. Informally, the property states that on execution of the server role, there exists a role of the intended client with a corresponding view of the interaction; more precisely the patterns in the sends and receives of the two roles match and the actions are ordered in the sequence implied by an intended protocol execution.

$$\begin{aligned}
\phi_{tls,auth} ::= & \text{Honest}(\hat{X}) \wedge \text{Honest}(\hat{Y}) \supset \exists X. \\
& (\text{Send}(X, tsm1) < \text{Receive}(Y, tsm1)) \wedge (\text{Receive}(Y, tsm1) < \text{Send}(Y, tsm2)) \\
& \wedge (\text{Send}(Y, tsm2) < \text{Receive}(X, tsm2)) \wedge (\text{Receive}(X, tsm2) < \text{Send}(X, tsm3)) \\
& \wedge (\text{Send}(X, tsm3) < \text{Receive}(Y, tsm3)) \wedge (\text{Receive}(Y, tsm3) < \text{Send}(Y, tsm4))
\end{aligned}$$

We use $tsm1$, $tsm2$, $tsm3$, $tsm4$ as abbreviations for various TLS messages:
 $tsm1 := \hat{X}.\hat{Y}.n_x.V_x$, $tsm2 := \hat{Y}.\hat{X}.n_y.V_y$, $tsm3 := \hat{X}.\hat{Y}.ENC_Y(secret).sigvx.hc2$, $tsm4 := \hat{Y}.\hat{X}.hs$.

The terms $sigvx$, $hc2$, hs are defined in Table 1. The formula $a < b$ asserts that action a occurred before action b . Note that the receive action corresponding to the last message sent by the server is not part of the guarantee as the server receives no acknowledgment for this message. We now use $\phi_{tls,auth}$ to state the authentication guarantee for a server \hat{Y} , communicating with client \hat{X} .

TLS Authentication Guarantee:

- (i) On execution of the server role, *session authentication* is guaranteed if $\Gamma_{tls,1}$, defined below holds.

Formally, $\Gamma_{tls,1} \vdash [\text{TLS:Server}]_X \phi_{tls,auth}$.

- (ii) The formula $\Gamma_{tls,1}$ (defined below) is an invariant of TLS.

$$\begin{aligned}
\Gamma_{tls,1} : & \text{Honest}(\hat{X}) \wedge \text{Sign}(X, sigterm) \supset \\
& (\text{Send}(X, tsm1) < \text{Receive}(X, tsm2) < \text{Send}(X, tsm3)) \wedge \\
& \text{FirstSend}(X, n_x, tsm1) \wedge \text{FirstSend}(X, secret, tsm3).
\end{aligned}$$

The formula $\text{FirstSend}(X, n, m)$ states that thread X generated nonce n and sent it out first time in the message m . The invariant requires that any honest principal which sends out a signature of a certain form also performs actions consistent with the TLS client role. We discuss the proof of the second part of the guarantee in Section 2.4. We now describe how the invariant is used in the security proof (Part (i) of the guarantee above).

2.2.2 Proof of the Authentication Guarantee

The first part of the argument uses the authentication property of signatures along with the protocol invariant $\Gamma_{tls,1}$ to argue that there must be a thread of client \hat{X} , the intended peer of server \hat{Y} , which must have performed certain actions corresponding to the TLS client role. The proof is summarized by the following steps below. The first line of the proof uses the axioms **AA1**, **P1**, **AA4** to conclude that the TLS Server has performed a certain sequence of actions. The next three steps use the invariant $\Gamma_{tls,1}$ and the server's signature verification action to conclude that a thread X^0 of server's intended peer \hat{X} must have performed certain actions.

$$\begin{aligned}
\mathbf{AA1, P1, AA4} \quad & [\mathbf{TLS : Server}]_Y (\text{Receive}(Y, tsm1) < \text{Send}(Y, tsm2) \\
& < \text{Receive}(Y, tsm3) < \text{Send}(Y, tsm4)) \quad (1) \\
\mathbf{AA1, P1} \quad & [\mathbf{TLS : Server}]_Y \text{Verify}(Y, \text{SIG}_{\hat{X}}\{\text{sigterm}\}) \quad (2) \\
(-1), \mathbf{VER} \quad & [\mathbf{TLS : Server}]_Y \text{Honest}(\hat{X}) \wedge \hat{X} \neq \hat{Y} \supset \exists X. \text{Sign}(X, \text{sigterm}) \quad (3) \\
(-1), \Gamma_{tls,1} \quad & [\mathbf{TLS : Server}]_Y \text{Honest}(\hat{X}) \wedge \hat{X} \neq \hat{Y} \supset \\
\text{and Inst } X \text{ to } X^o \quad & (\text{Send}(X^o, \hat{X}, \hat{Y}, m1) < \text{Receive}(X^o, \hat{Y}, \hat{X}, m2) \\
& < \text{Send}(X^o, \hat{X}, \hat{Y}, m3)) \wedge \text{New}(X^o, \text{secret}) \\
& \wedge \text{FirstSend}(X^o, n_x, tsm1) \wedge \text{FirstSend}(X^o, \text{secret}, tsm3) \quad (4)
\end{aligned}$$

The second part of the proof uses the first part of the proof along with properties of nonces to order the actions of the client thread X^o and the server thread Y ; we also conclude that the client thread must have the secret as we are guaranteed that it generated it. This proves the authentication guarantee.

$$\begin{aligned}
\mathbf{AN3, FS1, P1} \quad & [\mathbf{TLS : Server}]_Y \text{FirstSend}(Y, n_y, tsm2) \quad (5) \\
(-1), (4), \mathbf{FS2} \quad & [\mathbf{TLS : Server}]_Y \text{Honest}(\hat{X}) \wedge \hat{X} \neq \hat{Y} \supset \text{Send}(Y, tsm2) < \text{Receive}(X^o, tsm2) \quad (6) \\
(4), (1), \mathbf{FS2} \quad & [\mathbf{TLS : Server}]_Y \text{Honest}(\hat{X}) \wedge \hat{X} \neq \hat{Y} \supset \\
& (\text{Send}(X^o, tsm1) < \text{Receive}(Y, tsm1)) \wedge (\text{Send}(X^o, tsm3) < \text{Receive}(Y, tsm3)) \quad (7) \\
(1), (4), (6), (7) \quad & [\mathbf{TLS : Server}]_Y \text{Honest}(\hat{X}) \wedge \hat{X} \neq \hat{Y} \supset \\
& \exists X. (\text{Send}(X, tsm1) < \text{Receive}(Y, tsm1)) \wedge (\text{Receive}(Y, tsm1) < \text{Send}(Y, tsm2)) \\
& \wedge (\text{Send}(Y, tsm2) < \text{Receive}(X, tsm2)) \wedge (\text{Receive}(X, tsm2) < \text{Send}(X, tsm3)) \\
& \wedge (\text{Send}(X, tsm3) < \text{Receive}(Y, tsm3)) \wedge (\text{Receive}(Y, tsm3) < \text{Send}(Y, tsm4)) \quad (8)
\end{aligned}$$

2.3 Honesty Rule and Establishing Invariants

The honesty rule is an invariance rule for proving properties about the actions of principals that execute roles of a protocol, similar in spirit to the basic invariance rule of LTL [14] and invariance rules in other logics of programs. The honesty rule is used to combine facts about one role with inferred actions of other roles. For example, suppose Alice receives a signed response from a message sent to Bob. Alice may use facts about Bob’s role to infer that Bob must have performed certain actions before sending his reply. This form of reasoning may be sound if Bob is honest, since honest, by definition in our framework, means “follows one or more roles of the protocol.” The term “honest” is not meant in any deep philosophical sense – a principal could fail to be honest in some run either through dishonest intent, or as a result of some compromise that reveals the principal’s key to an attacker. The assumption that Bob is honest is essential because the intruder may perform arbitrary actions with any key that has been compromised. An example property that can be proved by this method as we shall soon see is the invariant required by the proof from the previous section, $\Gamma_{tls,1}$.

Recall from the previous section that a protocol \mathcal{Q} is a set of roles $\{\rho_1, \rho_2, \dots, \rho_k\}$, each executed by zero or more honest principals in any run of \mathcal{Q} . A sequence P of actions is an *initial segment* of role ρ , written $P \in IS(\rho)$, if P is a contiguous prefix of ρ such that (P starts at the beginning of ρ) (i) P ends with the last action before a receive, or (iii) P ends with the last action of the role. A principal is *honest* in a run of a protocol if the actions performed by that principal or with that principals keys are precisely the interleaving of the actions of some number of initial segments

of specified roles of the protocol. We now formally state *the honesty rule*:

$$\frac{\forall \rho \in \mathcal{Q}. \forall P \in IS(\rho). \text{Start}(X) [P]_X \phi}{\text{Honest}(\hat{X}) \supset \phi} \mathbf{HON}_{\mathbf{Q}} \quad \begin{array}{l} \text{no free variable in } \phi \\ \text{except } X \text{ bound in} \\ [P]_X \end{array}$$

Informally, the honesty rule says that if a certain property is a post condition of every sequence of actions that an honest agent may execute, then the property holds for the honest principal. We are guaranteed that honest principals follow the protocol. Further, we assume that programs are scheduled non-preemptively and halt only when they need input, for instance at receive actions. This allows us, in the antecedent of the rule, to quantify only over action sequences that are initial segments of the protocol \mathcal{Q} . A slightly different form of the honesty rule (based on induction over so called *basic sequences*) occurs in [6]; the form stated above is strictly more powerful and easier to automate using the method explored here.

2.4 Proving Invariants by Hand

We now briefly describe the process of proving that the formula $\Gamma_{tls,1}$ is an invariant of TLS. In this case, \mathcal{Q} is TLS and consists of two roles, the TLS:Client role and the TLS:Server role. Further, the TLS:Client role has three initial segments (as it has two receive actions); call these $TLS : Client_0$, $TLS : Client_1$, $TLS : Client_2$ in increasing order of length. The TLS:Server role has two initial segments (as it has two receive actions, but one of them begins the role); call these $TLS : Server_0$, $TLS : Server_1$, in increasing order of length.

We must prove that $\Gamma_{tls,1}$ is a post condition of each of these initial segments. Some of the arguments are trivial: The initial segments $TLS : Client_0$, $TLS : Server_0$ and $TLS : Server_1$ do not contain signature actions and in these cases, the antecedent of the formula fails trivially and the invariant holds.

The non-trivial cases are the initial segments $TLS : Client_1$, $TLS : Client_2$ as they contain signature actions of the appropriate form and the antecedent of the formula $\Gamma_{tls,1}$ holds. We now prove the invariant, by hand, for the initial segment $TLS : Client_1$. The proof for $TLS : Client_2$ is similar.

The proof has two parts. We first argue that there is only one binding for which the antecedent is true (signature is generated). We do this by establishing that the antecedent does not hold for all other bindings.

$$\mathbf{AA2, AA3, P1} \quad [\mathbf{TLS : Client}_1]_X \text{Sign}(X, x) \supset x = \text{sigterm} \quad (9)$$

Next we argue that for the one binding that we cannot rule out, the consequent is true. First we show that three actions happened in a certain sequence. We have to be careful enough to show that this is true for precisely the same binding for which the antecedent holds. From the TLS security standpoint, it is important that client's signature matches its view of the session.

$$\mathbf{AA4} \quad [\mathbf{TLS : Client}_1]_X (\text{Send}(X, \text{tism1}) < \text{Receive}(X, \text{tism2}) < \text{Send}(X, \text{tism3})) \quad (10)$$

Next we show that the nonce n_x and the shared secret *secret* are first sent out in the two messages generated by the TLS client. Again, we have to be careful enough

to show that this is true for precisely the same binding for which the antecedent holds. From the TLS security standpoint, the client’s signature is an explicit commitment on the part of the client that it generated fresh nonces and sent them out for the first time in certain TLS messages. The proof from the previous section used this to relatively order the messages of the client with respect to the server’s actions.

$$\mathbf{AN3, P2, FS1} \quad [\mathbf{TLS : Client}_1]_X \text{FirstSend}(X, n_x, tsm1) \quad (11)$$

$$\mathbf{AN3, P2, FS1} \quad [\mathbf{TLS : Client}_1]_X \text{FirstSend}(X, secret, tsm3) \quad (12)$$

Throughout the process, we needed to carefully prove that the consequent is true for precisely the binding of *sigterm* for which the antecedent is true. Though invariants are easy to formulate, the process of checking that they hold is fairly tedious and thus invariant checking is a perfect candidate for automation.

To finish this section we list a program for which the TLS invariant does not hold and describe an attack on TLS security. Consider the following program, which receives a term, signs it and sends it out.

$$\mathbf{SignatureOracle} = (X, \hat{Y})[\text{receive } \hat{Y}.\hat{X}.x; m = \text{sign } x.\hat{X}; \text{send } \hat{X}.\hat{Y}.m; \quad]_X$$

Suppose that a TLS client also executes the above program, then the attacker can break the authentication property of TLS as follows. It starts a session with a server *S* pretending to be a client *C*. It sends the first message and receives the second message. The one thing it cannot do is produce the client’s signature; at this stage it makes the client *C* execute the signature oracle protocol by sending it an appropriately constructed term to sign. It can now complete TLS protocol with the server *S*, which believes that it has communicated with client *C* and shares a secret with it.

From our proof’s standpoint, we would not be able to show that $\Gamma_{tls,1}$ is an invariant of a principal that executes the above signature oracle program: The antecedent is possibly true for some binding of *x*, while the consequent of the invariant is definitely not. Our approach in the next section ensures all such flaws are identified automatically.

3 Checking Invariants via Logic Programs

We now describe our logic program based approach to automatically check invariants. We informally state the main result as follows.

Theorem 3.1 (Informal) *We can construct a logic program consisting of a set of rules that encode part of the PCL proof system, a set of facts that encode the PCL program and a query that encodes a PCL invariant having a certain general structure, such that the logic program failing on the query implies that the invariant holds for the PCL program.*

We assert that the encoding of the PCL proof system is independent of the program and the invariant, and the encoding of the invariant is independent of the protocol under consideration.

3.1 Encoding PCL programs

We start by describing our encoding of PCL programs. Recall from Section 2.3 that we must show that the invariants holds at the conclusion of every initial segment of every protocol that an honest principal may execute; we describe invariant checking process for one such initial segment, the general case is simply a conjunction of such steps.

PCL programs are modeled as a set of facts involving the action predicate. The action predicate has the form $action(thread, actionnumber, output, actionname, input)$. For instance, the first initial segment of the TLS Client role ($TLS : Client_0$ from Section 2.4) is written as the following set of facts. The second fact $action('X', 1.1, t1, cat, [nx, vx])$ asserts that thread X concatenates terms nx and vx to generate the term $t1$. The $actionnumber$ field is ordinal and helps assert the ordering of actions within a thread.

```
action('X', 1, nx, new, _).
action('X', 1.1, t1, cat, [nx, vx]).
action('X', 1.2, t2, cat, ['Y', t1]).
action('X', 1.3, t3, cat, ['X', t2]).
action('X', 2, _, send, t3).
```

This initial segment picks a nonce and sends out a message containing the chosen nonce and the client's selection of the cryptosuite, to the intended server \hat{Y} . The translation of PCL programs into the logic program encoding is largely straightforward. A few issues that came up: First, we model *send* actions as having no outputs, the action on the network is treated as a side-effect. In general, the number of inputs to an action depends on the type of action. The main difference between the PCL program listing and the logic program encoding is the explicit modeling of the concatenation actions. An alternative encoding could have used the list construct, but we found that this leads to computational inefficiencies in practice.

3.2 Encoding the PCL proof system

Next we describe our encoding of the PCL proof system related to invariant checking. Recall from Section 2.4 that invariant checking involves checking for the presence of certain actions that act on terms with certain structures. Thus, the fragment PCL proof system that we are concerned with expresses the construction of terms by the PCL program actions. Our logic program encoding defines the structure of terms using two term equality predicates, $may(symbol, pattern)$ and $must(symbol, pattern)$, which roughly state the the program symbol X may have the structure $pattern$ and must have the structure $pattern$, respectively.

```
may(M, M) :- action(Y, N, M, new, []).
may(M, enc(R1, R2)) :- action(Y, N, M, pkenc, [P1, P2]), may(P1, R1), may(P2, R2).
may(M, pair(R1, R2)) :- action(Y, N, M, cat, [P1, P2]), may(P1, R1), may(P2, R2).
may(M, pair(R1, R2)) :- action(Y, N, [P1, P2], uncat, M), may(P1, R1), may(P2, R2).
may(P1, enc(M1, R2)) :- action(Y, N, M, pkdec, [P1, P2]), may(M, M1), may(P2, R2).
may(P1, R2) :- action(Y, N, _, match, [P1, P2]), may(P2, R2).
may(M, sig(R1, R2)) :- action(Y, N, M, sign, [P1, P2]), may(P1, R1), may(P2, R2).
may(P1, sig(M1, R2)) :- action(Y, N, _, verify, [P1, M, P2]), may(M, M1), may(P2, R2).
may(M, hash(R1, R2)) :- action(Y, N, M, hash, [P1, P2]), may(P1, R1), may(P2, R2).
may(P1, hash(M1, R2)) :- action(Y, N, _, verifyhash, [P1, M, P2]), may(M, M1), may(P2, R2).
```

There is an identical set of rules with *must* substituted for *may*. The consequent (left hand side) of each rule is a predicate representing a term equality. For instance the second rule says that the output of a public key encryption has a term structure that inductively depends on the inputs to the encryption. Each rule can also be viewed as an axiom in PCL. For instance, the second rule is based on the PCL axiom $[m := \text{pkenc } p1, p2;]_X m = E_{pk}[p2](p1)$, where $E_{pk}[p2](p1)$ denotes the encryption of $p1$ with the public-key $p2$.

While the *may* and *must* have an identical set of rules associated with them, the only dissimilarity arises in the handling of terms whose structures are not completely determined; for instance when a term is received and not parsed further, we cannot conclude anything about its structure. Soundness of invariant checking requires that we handle this lack of information differently in the antecedent and the consequent of the invariant. Roughly, soundness is always preserved when the antecedent is deemed true more frequently and the consequent is deemed false more frequently. As we shall see in the encoding of invariants, the encoding of the antecedent uses only the *may* predicate while that of the consequent uses only the *must* predicate.

We now formally define unconstrained symbols, symbols that we do not know the structure of.

Definition 3.2 A symbol m is said to be *constructed* by a certain action if it occurs in the appropriate position as described below:

| | | |
|---------------------------------|---------------------------------|------------------------------------|
| <code>new m;</code> | <code>m := pkenc p1, p2;</code> | <code>m := hash p1, p2;</code> |
| <code>match m as p;</code> | <code>p1 := pkdec m, p2;</code> | <code>verifyhash m, p1, p2;</code> |
| <code>m := cat p1, p2;</code> | <code>m := sign p1, p2;</code> | |
| <code>p1, p2 := uncat m;</code> | <code>verify m, p1, p2;</code> | |

If symbol m corresponds to the term t , then we define action *constructing* t to be the action constructing m ; A symbol m is said to be *unconstrained* if it is not constructed by any action and neither is it in the input list. Note that send and receive actions do not construct terms.

We assume that each symbol in a PCL program is constructed by a unique action: There is certainly no need to reconstruct terms redundantly or use the the same symbol to represent different terms (we can handle this by alpha renaming). More formally, the predicate $\text{may}(X, t)$ means that the symbol X in the PCL program corresponds to a pattern t , with unconstrained symbols corresponding to any pattern; and $\text{must}(X, t)$ means that the symbol X in the PCL program corresponds to a pattern t ; with unconstrained symbols essentially not substituted for. We can determine unconstrained symbols by inspecting the program alone. The soundness of invariant checking depends on faithfully adding appropriate may and must facts for all unconstrained symbols in the programs. For instance, if a term t is received and is not parsed further, nothing is known about its structure. We then add $\text{may}(t, X)$ and $\text{must}(t, t)$ as facts in the logic program; this asserts that on the one hand, term t is possibly equal to any structure; and on the other hand, we cannot confirm that it has a specific structure.

| | |
|--------------------------|---|
| <code>may(m, X).</code> | <code>m is an unconstrained symbol</code> |
| <code>must(m, m).</code> | <code>m is an unconstrained symbol</code> |

For the TLS client program, we regard n_y and V_y as unconstrained symbols. This amounts to conservatively assuming that the server nonce n_y and configuration

information V_y are not type checked at the client's end. We would like to prove invariants even in the absence of such type information.

3.3 Encoding Invariants

We conclude this section by describing how we model invariants. From our experience of looking at a number of protocols in current industrial usage (e.g. IEEE802.11i, Kerberos V5, IKEv2 and so on), most invariants used to prove authentication properties are of the form:

$$\Gamma : \forall \bar{a}. (\theta(\bar{a}) \supset \exists \bar{b}. \phi(\bar{a}, \bar{b})),$$

where θ and ϕ are conjunctions of PCL action predicates and term equality predicates. The invariants state that if an honest participant did certain action(s) on terms of a certain form, then it also did some other action(s) on terms of a related form. As we will use a standard decision procedure which focuses on finding satisfying assignments to a query, the above, universally quantified form of the invariant is not appropriate—If we ask the query by encoding $\theta(\bar{a}) \supset \exists \bar{b}. \phi(\bar{a}, \bar{b})$ in some way, the program will return all instantiations of the free variables that satisfy the non-universally quantified formula; however that will not imply that the universally quantified formula holds because there could be *some* instantiation which makes the antecedent true but not the consequent. Therefore, we ask the negated query:

$$\neg \Gamma : \exists \bar{a}. (\theta(\bar{a}) \wedge \forall \bar{b}. \neg \phi(\bar{a}, \bar{b})),$$

If the invariant holds, the program should fail to come up with an instantiation in this case. We encode invariants as a conjunction of action predicates or derivatives of action predicates along with *may* or *must* predicates that define patterns of symbols used in the actions; more precisely, each action predicate is paired with a *may* or a *must* predicate which is conjunction where the first parameter of the term predicate is bound in the action predicate. As discussed in the previous section, our definitions of *may* and *must* handle lack of information differently, *may* allows matches to be liberal while *must* is conservative. We will ensure that θ uses only *may* while ϕ only uses *must*, making invariants checking conservative and hence sound.

In this paper we will take $\Gamma_{tls,1}$ as the running example. The query corresponding to $\Gamma_{tls,1}$ can be stated as follows. This is the negated version of the PCL query $\Gamma_{tls,1}$ from the previous section. We declare the invariant holds if the query fails.

```
? action(X, N, M, sign, [M1,J]), may(M, sig(pair(AX, pair(AY,
  pair(Vnx, pair(Vvx, pair(AY, pair(AX, pair(Vny, pair(Vvy,
    enc(Vsecret, AY))))))), AX)),
\+ (
  action(X, N1, _, send, M2), must(M2, pair(AX, pair(AY,
    pair(Vnx, Vvx))),
  action(X, N2, M3, receive, _), must(M3, pair(AY,
    pair(AX, pair(Vny, Vvy))),
  action(X, N3, _, send, M4), must(M4, pair(AX, pair(AY,
    pair(enc(Vsecret, AY), pair(sig(pair(AX, pair(AY, pair(Vnx,
    pair(Vvx, pair(AY, pair(AX, pair(Vny, pair(Vvy,
    enc(Vsecret, AY))))))), AX), hash(pair(AX, pair(AY,
    pair(Vnx, pair(Vvx, pair(AY, pair(AX, pair(Vny, pair(Vvy,
    pair(enc(Vsecret, AY), pair(sig(pair(AX, pair(AY, pair(Vnx,
    pair(Vvx, pair(AY, pair(AX, pair(Vny, pair(Vvy, enc(Vsecret,
    AY))))))), AX), client))))))), Vsecret))))),
```

```

N1 < N2, N2 < N3,
firstsend(X, Vnx, M2),
firstsend(X, Vsecret, M4),
action(X, N4, Vnx, new, _)
).

```

Note that this invariant has action ordering and the predicate ‘FirstSend’ in addition to basic action predicates. The action ordering is taken care of by the numbering of the actions in the encoding of the program and using the numbers appropriately in the query (in the above query, the line that asks if $N1 < N2$ and if $N2 < N3$.)

The predicate Firstsend holds for some nonce n and a term t if the nonce was sent out for the first time in the term t . This is modeled in logic program as follows by the following additional rule, which essentially says that the predicate holds for a specific term and a nonce if there was no term sent out earlier that contains the nonce.

```

firstsend(X, Nu, M) :- action(X, _, Nu, new, []),
action(X, N1, _, send, M), contains(M, Nu),
\+ (action(X, N2, _, send, M1), contains(M1, Nu), N2 < N1).

```

The description of FirstSend depends on whether a nonce is *contained* in a particular term or not. Contains is defined as follows:

```

tcontains(S, S) :- action(Y, N, S, new, []).
tcontains(pair(T1, T2), S) :- tcontains(T1, S).
tcontains(pair(T1, T2), S) :- tcontains(T2, S).
tcontains(enc(T1, T2), S) :- tcontains(T1, S).
tcontains(sig(T1, T2), S) :- tcontains(T1, S).
tcontains(hash(T1, T2), S) :- tcontains(T1, S).

contains(M, S) :- must(M, T), tcontains(T, S).

```

The full encoding is given in Appendix A.

3.4 Proving Soundness of Invariant Checking

Theorem 3.3 *Given the above encoding of the PCL protocol Q and the PCL proof system (\vdash_{PCL}) as a logic program, failure of the query that encodes the negation of the Invariant Γ implies that $Q \vdash_{PCL} \Gamma$.*

We now discuss the proof of the main theorem. We show that invariant checking is sound for a standard decision procedure on logic programs. Recall from Section 2.3 that the invariant must be deemed to be true for every initial segment of every protocol that the honest principal executes; we describe the process for one such initial segment and show that, if our decision procedure outputs failure on the query for our logic program, then the invariant holds for that initial segment.

In the following proofs, we prove that a decision procedure employing SLD-resolution [17] in a left to right, depth first manner is sound for checking invariants. We also use the fact that SLD-resolution tries every rule in the logic program before failing. Most Prolog interpreters implement SLD-resolution in a left-to-right, depth first manner [16] which means that we can use standard tools to realize the decision procedure. In particular, we used SWI-Prolog as our prototype tool [18].

Recall from the previous section that we work with invariants that have a specific structure; the antecedent is a conjunction of (action, may) predicate pairs, where the first parameter of the may predicate is bound in the corresponding action predicate. Similarly, the consequent is a conjunction of (action, must) predicate pairs. The proof of the above theorem follows from the completeness of checking sub queries of the form (action, may) (Lemma 3.8) and the soundness of consequent checking

sub queries of the form $(\text{action}, \text{must})$ (Lemma 3.4). We emphasize here that the soundness and completeness lemmas are stated with respect to the correspondence between the logic program encoding and PCL. For now we ignore the `FirstSend` and ordering predicates. We will return to comment on these near the end of this section.

Lemma 3.4 *If the decision procedure proves the query $\text{action}(\text{Thread}, \text{Number}, \text{Output}, \text{Action}, \text{Input}), \text{must}(X, \tau)$, where X occurs in Input or Output, then there exists an action in the initial segment with output ρ and a substitution σ , which does not substitute any unconstrained symbols in the program, satisfying $\sigma(\tau) = \sigma(\rho)$.*

Substituting for unconstrained program symbols by constants in the consequent of an invariant is unsound in the as we cannot definitely assert that the program constant has a specific structure. We now prove the lemma.

Proof. First note that we must use facts and not rules to match the `action` predicate as there are no rules with this predicate at the head. Further, based on our logic program and as our decision procedure works left-to-right, once the action predicate in the query has matched an action fact, the X in the second predicate of the query is bound to a constant. Effectively we are left with a query $\text{must}(x, \tau)$, for some constant x . We prove the lemma by induction on the number of rules that the decision procedure applies.

For the base case, suppose that the decision procedure does not apply any rules (uses only facts). The decision procedure binds the action query to an action fact with input/output that is either an unconstrained variable or a member of the program input list; as these are the only symbols with an associated `must` fact. Further, each such term t only has a fact $\text{must}(t, t)$ associated with it and therefore the query pattern τ must either be precisely t , in which case soundness is trivial, or a variable V , in which case, V is existentially quantified in the query and there indeed exists a program action that matches the query; in the first case it is important that we ensure that in encoding programs ground terms other than constant strings do not match query constants. Finally, in either case, unconstrained symbols are not substituted for.

Induction Hypothesis (IH): Suppose we have soundness for all queries of the form $\text{must}(x, \tau)$ which the decision procedure proves by the application of $n \geq 0$ rules.

We now prove soundness for a query $\text{must}(x, \tau)$ that the decision procedure proves by application of $n + 1$ rules. The first rule that the decision procedure applies must have one of the following two forms:

- $\text{must}(M, M) :- \text{action}(Y, N, M, \text{new}, [])$. In this case, we have the binding $\sigma(\tau) = m$, where $\text{action}(y, n, m, \text{new}, [])$ is a fact, and we can argue soundness as in the base case.

- $\text{must}(M, \text{enc}(R1, R2)) :- \text{action}(Y, N, M, \text{pkenc}, [P1, P2]), \text{must}(P1, R1), \text{must}(P2, R2)$. As the `must`s on the RHS must have been proved using $\leq n$ rule application. So, by IH, there must exist a substitution σ_1 , which does not substitute unconstrained symbols in the program, such that there is a term ρ_1 occurring in the program satisfying $\sigma_1(P1) = \sigma_1(\rho_1)$. Similarly we have $\sigma_2(P2) = \sigma_2(\rho_2)$.

Because of matching of the `pkenc` action which takes P1 and P2 as inputs, there must be a term ρ such that $\rho = E_{pk}[\rho_1](\rho_2)$ occurs in the program. Since both the musts occur in the same conjunct, σ_1, σ_2 must be consistent wrt common variables, and hence we can construct the composite substitution $\sigma = \sigma_1 \cup \sigma_2$. As $\sigma_1(P1) = \sigma_1(\rho_1)$, $\sigma_2(P2) = \sigma_2(\rho_2)$ and $\rho = E_{pk}[\rho_1](\rho_2)$, we have $\sigma(\tau) = \sigma(\rho)$ and σ does not substitute unconstrained variables.

- the proof is similar for all other actions. □

We now discuss completeness of antecedent checking. This is the more involved part of the proof that uses properties of our logic program as well as assumptions about the decision procedure. The next lemma shows that any query of the form $may(x, \tau)$ has exactly one way to be resolved as in PCL every program symbol is defined exactly once.

Definition 3.5 A fact or a rule is said to be *effective* for a goal of the form $may(x, \tau)$, if it is either a fact of the form $may(x, \dots)$ or a rule with the predicate may at the head, whose first subgoal is satisfied by an action fact which constructs the term x .

Recall from section 3.2 that each symbol is constructed at most once. If it was constructed by a program action, then there is exactly one action fact that has this symbol as an output. Further there is exactly one rule corresponding to each action type. Further, no $may(\dots)$ fact is directly effective for the goal. The argument for the case when the symbol x is either unconstrained or part of the input list, is similar; in this case, a unique fact $may(x, \tau')$ matches the query and no action constructs the symbol x . So we have the following fact.

Fact 3.6 *For any goal of the form $may(x, \tau)$, there is at most one fact or a rule that is effective.*

We now use the above fact to show completeness of queries of the type $may(x, \tau)$ for some constant x .

Lemma 3.7 *Fix a query $may(x, \tau)$. If there exists a substitution σ , which may substitute unconstrained symbols in the program to any term, such that term ρ corresponding to symbol x occurring in the program satisfies $\sigma(\tau) = \sigma(\rho)$, then the query returns true. Further, there is no other substitution σ' that matches the query upto most general unification.*

Proof. Our proof is by induction on the number of steps in the PCL program used to define x , $steps(x)$. Formally, let I denote the set of inputs to the action that constructs a symbol m . Define the value of the function $steps(m)$ to be $1 + \sum_{i \in I} steps(i)$, with the following additional definitions that complete the base case:

For unconstrained and input list symbol m , $steps(m) = 0$
 For new m ; $steps(m) = 1$;

Base case: Here x must correspond to an unconstrained symbol or a symbol in the input list. By Fact 3.6, there is a unique effective ground fact $may(x, \cdot)$, which the decision procedure will find after rejecting all other rules. If x is unconstrained then it makes the appropriate substitution $\sigma(Y) = \tau$, where the ground fact is

$\text{may}(x, Y)$. Otherwise, σ must correspond to the identity substitution. In either case, the substitution is unique.

Induction Hypothesis (IH): If x was constructed in $n \geq 0$ steps and there is a substitution σ such that $\sigma(\tau) = \sigma(\rho)$, then query $\text{may}(x, \tau)$ returns true in finite time and σ is the unique such substitution.

Induction: Suppose x was constructed in $n + 1$ steps and there exists a substitution σ , which may substitute unconstrained symbols in the program to any term, such that $\sigma(\tau) = \sigma(\rho)$. Since x was constructed by a unique action, we have exactly one of the following cases:

- by action `[new x;]`: There is only one effective rule `may(M, M) :- action(Y, N, M, new, [])`. As we argued in the base case, the decision procedure must reject all other choices and find this rule in finite time. Also, clearly the substitution σ is unique, with τ bound to the constructed nonce.
- by action `[x := cat x1, x2;`] where x_1, x_2 correspond to terms ρ_1, ρ_2 respectively: It is given that there is a substitution σ such that $\sigma(\tau) = \sigma(\rho) = \sigma(\rho_1.\rho_2)$. Now, this could occur in one of two ways:
 - (i) τ is `pair(τ_1, τ_2)` where τ_i 's are themselves term patterns - in this case $\sigma(x_1) = \sigma(\tau_1)$ and $\sigma(x_2) = \sigma(\tau_2)$.

At some point, the decision procedure considers the one effective rule `may(M, pair(R1, R2)) :- action(Y, N, M, cat, [P1, P2]), may(P1, R1), may(P2, R2)` and matches the first subgoal against the correct fact `action(y, n, x, cat, [x1, x2])`. By IH, since x_1, x_2 were constructed in $\leq n$ steps and there exists substitution σ such that $\sigma(x_1) = \sigma(\tau_1)$ and $\sigma(x_2) = \sigma(\tau_2)$, therefore the decision procedure is able to return true for `may(x1, τ_1), may(x2, τ_2)`.

- (ii) τ is a single variable. This case is dealt with similarly as above, except that the decision procedure uses x_1, x_2 directly instead of τ_1, τ_2 .

In both cases finite time is ensured as there is only one effective `may` rule whose sub-goals are satisfied in finite time. Further, the substitution σ is unique.

- other actions are reasoned similarly. □

Lemma 3.8 *The decision procedure returns all matches to the query `action(Thread, Number, Output, a, Input), may(X, τ)`, where X occurs in either Output or Input.*

Proof. the decision procedure adds a goal `may(x, τ)` for each fact `action('X', n, o, a, i)`, of which there are a finite number, that matches the first predicate of the query. By Lemma 3.7, there is at most one match for such a query and it is found in finite time. □

Proof. [Theorem 3.3] We prove the following statements, that together imply the theorem:

- The decision procedure explores all query variable bindings that make the antecedent true.
- For each such binding, if the consequent is proved by the decision procedure, then

the consequent holds with that binding at the conclusion of the initial segment.

We first prove that all antecedent matches are explored. The proof is by induction on the number of `action(...)`, `may(...)` conjuncts in the antecedent. The base case, when there is exactly one conjunct follows from Lemma 3.8.

Induction Hypothesis (IH): Assume that for $n \geq 0$ `action(..., M, Action, ...)`, `may(M, τ)` conjuncts, the decision procedure explores all possible successful bindings.

Induction: Suppose there are $n + 1$ such conjuncts. Consider the first `action(..., M, Action, ...)`, `may(M, τ)` conjunct. The decision procedure is going to match the `action(..., M, Action, ...)` against all `Action` actions in the protocol program and return successfully if corresponding bindings are found for τ . Each such binding for τ imposes a binding for common variables on τ_i 's on the rest of the `may(Mi, τ_i)` conjuncts. So rest of the conjuncts are explored with τ_i 's with some variables instantiated, which in essence are patterns again. So by induction hypothesis, the decision procedure will explore all possible bindings for each of the succeeding conjuncts for each binding of the first conjunct.

Now we prove the second part of the sufficient condition. If the antecedent holds with a certain binding, the common variable bindings are transmitted to the consequent as well. The variables that are not bound yet still remain part of the τ_i patterns. We use a similar induction. The base case, if there is only one conjunct follows from Lemma 3.4.

Induction Hypothesis (IH): Assume that for $n \geq 0$ `action(..., M, Action, ...)`, `may(M, τ)` conjuncts, if the decision procedure matches successfully then they are indeed satisfied by the initial segment.

Induction: Suppose there are $n + 1$ such conjuncts. Consider the first `action(..., M, Action, ...)`, `may(M, τ)` conjunct. By Lemma 3.4, since this is matched successfully, the initial segment must satisfy it. This binding for τ imposes a binding for common variables on τ_i 's on the rest of the `may(Mi, τ_i)` conjuncts. So rest of the conjuncts are explored with τ_i 's with some variables instantiated. So by induction hypothesis, if the decision procedure matches rest of the conjuncts, then they must be satisfied by the initial segment. \square

The soundness of the implementation with `firstsend` in the consequent follows if: if `action(X, -, Nu, new, [])`, `action(X, N1, -, send, M)`, `contains(M, Nu)`, `\+ (action(X, N2, -, send, M1), contains(M1, Nu), N2 < N1)`. succeeds then the initial segment satisfies `FirstSend(X, Nu, M)` for the same binding.

Now this rule succeeds if there exists a binding for which if `action(X, -, Nu, new, [])`, `action(X, N1, -, send, M)`, `contains(M, Nu)` is matched by the decision procedure then the initial segment does not satisfy `(Send(X, M1) \wedge Contains(M1, Nu) \wedge N2 < N1)`. The proof now involves a soundness proof for the first part and a completeness proof for the second part. Details can be completed using a similar proof approach to above.

Completeness

We finish the paper with a brief discussion about completeness of invariant checking. We first list a toy example that shows that our invariant check-

ing method is not complete. Consider the following program and the formula $\text{Receive}(X, \text{"Alice"}) \supset \text{Send}(X, \text{"Alice"})$.

$$\text{Gossip} = (X, \hat{Y})[\text{receive } \hat{Y}.\hat{X}.x; \text{send } \hat{X}.\hat{Y}.x;]_X$$

It is easy to see that the formula is an invariant of the above protocol as the string “Alice” is sent whenever it is received by the program. However our invariant checking method proves *unhappy* and declares the invariant false. The precise query is:

```
unhappy :- action(X,N,_, receive,M1), may(M1, "Alice")
          \+ (action(X,N,M2,send, _), must(M2, "Alice"))
```

Our invariant checking method finds that the antecedent is true for the binding $x = \text{"Alice"}$, but cannot prove the consequent for this binding: The must fact in the Prolog encoding of the program is overly restrictive in this context.

We now briefly discuss why this lack of completeness is not a severe issue in practice. In practice the antecedent of the invariant encodes that the principal signs, encrypts or constructs a keyed hash over a term that is very protocol specific. For instance, our running example $\Gamma_{tls,1}$, has the pattern *sigterm* that is specific to the TLS protocol as it is a concatenation of messages that have a TLS specific structure. We expect that most well designed real world protocols that are not TLS, would have a sufficiently distinct structure so as not to satisfy the antecedent for any binding. On the other hand if the antecedent of $\Gamma_{tls,1}$ is satisfied for some binding by an initial segment of a protocol other than TLS, the protocol’s actions are sufficiently similar to TLS; this can often be leveraged to construct an attack as in the example from Section 2.4.

References

- [1] M. Bellare and P. Rogaway. Entity authentication and key distribution. In *Advances in Cryptology - Crypto'93 Proceedings*. Springer-Verlag, 1994.
- [2] C.J.F. Cremers. On the protocol composition logic PCL. In M. Abe and V. Gligor, editors, *Proc. of the ACM Symposium on Information, Computer & Communication Security (ASIACCS '08)*, pages 66–76, Tokyo, March 2008. ACM Press.
- [3] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic. A derivation system for security protocols and its logical formalization. In *Proceedings of 16th IEEE Computer Security Foundations Workshop*, pages 109–125. IEEE, 2003.
- [4] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic. Abstraction and refinement in protocol derivation. In *Proceedings of 17th IEEE Computer Security Foundations Workshop*, pages 30–45. IEEE, 2004.
- [5] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic. Secure protocol composition. In *Proceedings of 19th Annual Conference on Mathematical Foundations of Programming Semantics*, volume 83 of *Electronic Notes in Theoretical Computer Science*, 2004.
- [6] A. Datta, A. Derek, J. C. Mitchell, and A. Roy. Protocol composition logic. In *Electronic Notes in Theoretical Computer Science (ENTCS), Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin*, 172:311–358, 2007.
- [7] Anupam Datta, Ante Derek, John C. Mitchell, and Dusko Pavlovic. A derivation system and compositional logic for security protocols. *Journal of Computer Security*, 13(3):423–482, 2005.
- [8] Anupam Datta, Ante Derek, John C. Mitchell, Vitaly Shmatikov, and Mathieu Turuani. Probabilistic polynomial-time semantics for a protocol security logic. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP '05)*, Lecture Notes in Computer Science, pages 16–29. Springer-Verlag, 2005.
- [9] T. Dierks and E. Rescorla. The TLS Protocol — Version 1.0. IETF RFC 2246, January 1999.

- [10] Nancy Durgin, John C. Mitchell, and Dusko Pavlovic. A compositional logic for protocol correctness. In *Proceedings of 14th IEEE Computer Security Foundations Workshop*, pages 241–255. IEEE, 2001.
- [11] Nancy Durgin, John C. Mitchell, and Dusko Pavlovic. A compositional logic for proving security properties of protocols. *Journal of Computer Security*, 11:677–721, 2003.
- [12] Changhua He, Mukund Sundararajan, Anupam Datta, Ante Derek, and John C. Mitchell. A modular correctness proof of iee 802.11i and tls. In *ACM Conference on Computer and Communications Security*, pages 2–15, 2005.
- [13] Doug Kuhlman, Ryan Moriarty, Tony Braskich, Steve Emeott, and Mahesh V. Tripunitara. A correctness proof of a mesh security architecture. In *CSF*, pages 315–330. IEEE Computer Society, 2008.
- [14] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.
- [15] Catherine Meadows and Dusko Pavlovic. Deriving, attacking and defending the gdoi protocol. In *ESORICS*, pages 53–72, 2004.
- [16] Ulf Nilsson and Jan Maluszynski. *Logic, Programming, and PROLOG*. John Wiley & Sons, Inc., New York, NY, USA, 1995.
- [17] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.
- [18] Jan Wielemaker. SWI-prolog. <http://www.swi-prolog.org/>.

A Prolog Encoding of TLS and an Invariant

```

% Initial Segment Consisting of the First Two Basic Sequences of the TLS Client Program

% BASIC SEQUENCE #1
% This basic sequence contains two actions, a new and a send.
% The message sent is called 'Client Hello' TLS parlance.

action(y,1,nx,new, []).
action(y,1.1,t1,cat,[nx,vx]).
action(y,1.2,t2,cat,['Y',t1]).
action(y,1.3,t3,cat,['X',t2]).
action(y,2,dummy,send,t3).

% BASIC SEQUENCE #2
% This basic sequence contains 5 parts. First the client checks the server's response to
% the client hello. Next, it creates a 'secret' and encrypts it using the server's
% public key. Then it constructs a signature over the first message and the encryption.
% Then it constructs keyed hash, with 'secret' as key, over the first message, the
% encryption and the signature. Finally it sends out the encryption, the signature and
% the hash.

% Checking the server's response.
action(y,3,t4,receive, []).
action(y,3.1,[t5,t6],uncat,t4).
action(y,3.2,[t7,t8],uncat,t6).
action(y,3.3,[ny,vy],uncat,t8).
action(y,3.4, dummy, match, [t5, 'Y']).
action(y,3.5, dummy, match, [t7, 'X']).

% Creating a shared secret
action(y,4,secret,new, []).
action(y, 5, encky, pkenc, [secret,'Y']).

% Constructing the signature
action(y, 5.1, t9, cat, [vy,encky]).
action(y, 5.2, t10, cat, [ny,t9]).
action(y, 5.3, t11, cat, ['X',t10]).
action(y, 5.4, t12, cat, ['Y',t11]).
action(y, 5.5, t13, cat, [vx,t12]).
action(y, 5.6, t14, cat, [nx,t13]).
action(y, 5.7, t15, cat, ['Y',t14]).
action(y, 5.8, sigterm, cat, ['X',t15]).
action(y, 6, sigvx, sign, [sigterm, 'X']).

% Constructing the keyed hash.
action(y, 6.1, t16, cat, [sigvx, client]).
action(y, 6.2, t17, cat, [encky,t16]).
action(y, 6.3, t18, cat, [vy,t17]).
action(y, 6.4, t19, cat, [ny,t18]).
action(y, 6.5, t20, cat, ['X',t19]).
action(y, 6.6, t21, cat, ['Y',t20]).
action(y, 6.7, t22, cat, [vx,t21]).
action(y, 6.8, t23, cat, [nx,t22]).
action(y, 6.9, t24, cat, ['Y',t23]).
action(y, 6.91, t25, cat, ['X',t24]).
action(y, 7, hc2, hash, [t25, secret]).

% The Client sends the third message.
action(y, 7.1, t26, cat, [sigvx, hc2]).
action(y, 7.2, t27, cat, [encky,t26]).
action(y, 7.3, t28, cat, ['Y',t27]).
action(y, 7.5, t29, cat, ['X',t28]).
action(y, 8, dummy, send, t29).

% (Trivial) term equalities needed for atoms used in the program

may('X', 'X').
may('Y', 'Y').
may(vx, vx).
may(client, client).
must('X', 'X').
must('Y', 'Y').
must(vx, vx).
must(client, client).

% Term equalities needed for unparsed variables that occur in the program

may(vy, X).
may(ny, X).
must(vy, vy).
must(ny, ny).

```

```

% Checking that the invariant is satisfied by the initial segment.
% Note that the invariant is negated: we are unhappy if the antecedent occurs and the
% consequent is not satisfied.

unhappy :- action(X, N, M, sign, [M1,J]), may(M, sig(pair(AX, pair(AY, pair(Vnx,
pair(Vvx, pair(AY, pair(AX, pair(Vny, pair(Vvy, enc(Vsecret,
AY))))))))), AX)),
\+ (
  action(X, N1, _, send, M2), must(M2, pair(AX, pair(AY, pair(Vnx, Vvx))))),
  action(X, N2, M3, receive, _), must(M3, pair(AY, pair(AX, pair(Vny, Vvy))))),
  action(X, N3, _, send, M4), must(M4, pair(AX, pair(AY, pair(enc(Vsecret, AY),
pair(sig(pair(AX, pair(AY, pair(Vnx, pair(Vvx, pair(AY, pair(AX,
pair(Vny, pair(Vvy, enc(Vsecret, AY))))))))), AX), hash(pair(AX,
pair(AY, pair(Vnx, pair(Vvx, pair(AY, pair(AX, pair(Vny, pair(Vvy,
pair(enc(Vsecret, AY), pair(sig(pair(AX, pair(AY, pair(Vnx, pair(Vvx,
pair(AY, pair(AX, pair(Vny, pair(Vvy, enc(Vsecret, AY))))))))), AX),
client))))))))), Vsecret)))))),
  N1 < N2, N2 < N3,
  firstsend(X, Vnx, M2),
  firstsend(X, Vsecret, M4),
  action(X, N4, Vnx, new, _)
).

% The proof system for proving invariants

may(M, M) :- action(Y,N,M,new, []).
may(M, enc(R1,R2)) :- action(Y,N,M,pkenc, [P1,P2]), may(P1,R1), may(P2,R2).
may(M, pair(R1,R2)) :- action(Y,N,M,cat, [P1,P2]), may(P1,R1), may(P2,R2).
may(M, pair(R1,R2)) :- action(Y,N,[P1, P2], uncat, M), may(P1,R1), may(P2,R2).
may(P1, enc(M1,R2)) :- action(Y,N,M,pkdec, [P1,P2]), may(M, M1), may(P2,R2).
may(P1,R2) :- action(Y,N,_,match, [P1,P2]), may(P2, R2).
may(M, sig(R1,R2)) :- action(Y,N,M,sign, [P1,P2]), may(P1,R1), may(P2,R2).
may(P1, sig(M1,R2)) :- action(Y,N,_,verify, [P1,M,P2]), may(M, M1), may(P2,R2).
may(M, hash(R1,R2)) :- action(Y,N,M,hash, [P1,P2]), may(P1,R1), may(P2,R2).
may(P1, hash(M1,R2)) :- action(Y,N,_,verifyhash, [P1,M,P2]), may(M, M1), may(P2,R2).

must(M, M) :- action(Y,N,M,new, []).
must(M, enc(R1,R2)) :- action(Y,N,M,pkenc, [P1,P2]), must(P1,R1), must(P2,R2).
must(M, pair(R1,R2)) :- action(Y,N,M,cat, [P1,P2]), must(P1,R1), must(P2,R2).
must(M, pair(R1,R2)) :- action(Y,N,[P1, P2], uncat, M), must(P1,R1), must(P2,R2).
must(P1, enc(M1,R2)) :- action(Y,N,M,pkdec, [P1,P2]), must(M, M1), must(P2,R2).
must(P1,R2) :- action(Y,N,_,match, [P1,P2]), must(P2, R2).
must(M, sig(R1,R2)) :- action(Y,N,M,sign, [P1,P2]), must(P1,R1), must(P2,R2).
must(P1, sig(M1,R2)) :- action(Y,N,_,verify, [P1,M,P2]), must(M, M1), must(P2,R2).
must(M, hash(R1,R2)) :- action(Y,N,M,hash, [P1,P2]), must(P1,R1), must(P2,R2).
must(P1, hash(M1,R2)) :- action(Y,N,_,verifyhash, [P1,M,P2]), must(M, M1), must(P2,R2).

tcontains(S, S) :- action(Y,N,S,new, []).
tcontains(pair(T1, T2), S) :- tcontains(T1, S).
tcontains(pair(T1, T2), S) :- tcontains(T2, S).
tcontains(enc(T1, T2), S) :- tcontains(T1, S).
tcontains(sig(T1, T2), S) :- tcontains(T1, S).
tcontains(hash(T1, T2), S) :- tcontains(T1, S).

contains(M, S) :- must(M, T), tcontains(T, S).

firstsend(X, Nu, M) :- action(X, _, Nu, new, []), action(X, N1, _, send, M),
contains(M, Nu), \+ (action(X, N2, _, send, M1),
contains(M1, Nu), N2 < N1).

% ----- %

```