A HOL Formalization of CAPSL Semantics

Stephen H. Brackin [*]
Arca Systems, Inc.
303 E. Yates St.
Ithaca, NY 14850

Abstract

This paper describes a state-based Higher Order Logic theory of protocol failure that formalizes the semantics of the Common Authentication Protocol Specification Language, a specification language being developed for use by all protocol designers and all protocol-analysis tools. This theory gives the basis for a new, fast and thorough, protocol-analysis tool based on automatically constructing proofs.

1 Introduction

Cryptographic protocols are short sequences of message exchanges, usually involving encryption, intended to establish secure communication over insecure networks. They are central to any activities that require such communication, such as commercial transactions over the Internet.

Protocol failure [6] occurs when an active wiretapper, without performing successful code-breaking, can obtain restricted information (a *nondisclosure* failure) or trick legitimate protocol participants into thinking that they are communicating with each other when they are actually communicating with the wiretapper (an *authentication* failure). The wiretapper does this by cleverly intercepting, blocking, modifying, and replaying messages, tricking legitimate protocol participants into performing any decryptions needed by the wiretapper.

This paper considers only the worst case, that in which an active wiretapper is in complete control of the network — i.e., every message sent by a legitimate protocol participant goes to the wiretapper, and every message received by a legitimate protocol participant comes from the wiretapper. A sequence of wiretapper actions that results in a nondisclosure or authentication failure under these conditions is called an *attack*. The remainder of this paper will call the active wiretapper the *attacker*.

Current tools for detecting protocol failure use one or more of the following approaches:

- attempting to construct attacks, using algebraic properties of the algorithms in the protocols [11, 10, 8];
- attempting to construct proofs, using specialized logics based on a notion of "belief", that any attacks are of limited effect — i.e., that protocol participants can confidently reach desired conclusions [2, 1]; or
- attempting to construct proofs, using formal models of the actual computations performed in protocols, that attacks are impossible [16, 13, 15, 14].

The first approach is thorough, but slow and labor-intensive, because it involves searches over a space of possible attacker actions that grows exponentially with the size of the protocol. The second approach is fast and automatic, but it misses many failures [5] — e.g., it detects only authentication failures that occur when there are no nondisclosure failures. The third approach has yet to be fully automated.

This paper describes a step toward automating the third approach. It describes PDL, for "Protocol Description Logic", a state-based Higher Order Logic (HOL) theory of protocol failure that is simpler and more expressive than earlier, trace-based HOL theories of protocol failure. PDL formalizes the low-level details of the actions actually performed by processes executing a protocol.

Higher Order Logic (HOL) [7] is a collection of tools for producing formal proofs. One of its central ideas is defining "theorem" as a *type* over a strongly typed metalanguage that is a powerful, general-purpose programming language in its own right [12]. Defining "the-

[*]This work was supported by the Advanced Research Projects Agency through Rome Laboratory contract F30602-97-C-0303.

orem" as a type makes type checking in the metalanguage's compiler the mechanism for deciding whether a theorem has been proved, and allows unconstrained metalanguage programs for automatically constructing proofs.

PDL is sufficiently expressive to formalize all protocols specified in the latest version of the *Common Authentication Protocol Specification Language* (CAPSL). CAPSL is being developed by Millen, with the cooperation of an international group of researchers, and is intended to become a standard specification language for use by all protocol designers and all protocol-analysis tools. CAPSL is still evolving; its latest revision is available at `www.csl.sri.com/~millen/capsl`. It is intended to be as similar as possible to the informal notation normally used to describe protocols, but to also have a formal semantics expressive enough to include all possible sources of protocol failure.

Some of the possible sources of protocol failure, all giving failures missed by a belief-based protocol-analysis tool [5], follow:

- Inadequate type and equality checking — protocol processes need not perform type and equality checks that they could perform, checks that prevent many attacks;
- Concurrent sessions — an attacker can replay messages from different, possibly concurrent, executions of the protocol;
- Misinterpretations — attacker interference can cause the actions taken by legitimate protocol participants to not mean what they are supposed to mean;
- Accidental disclosure — attacker interference can cause legitimate protocol participants to accidentally give away their secrets; and
- Algebraic properties attacks — the attacker can use algebraic properties of the protocol's encryption functions to make meaningful modifications to cryptotext without knowing the plaintext.

PDL models all these possibilities.

The remainder of this paper is organized as follows: Section 2 gives a high-level overview of PDL. Section 3 gives PDL formalizations of the agents, roles, data objects, and functions involved in a protocol, and defines three PDL languages as concrete recursive types:

- an *action* language defining the actions carried out by individual processes;
- a *state* language describing the possible states of a network with agents creating processes carrying out the protocol, but with an attacker completely controlling the network; and
- a *belief* language expressing the conditions that a protocol analyst can either initially assume or receive compelling evidence for on the basis of the computations performed by a protocol process.

Section 4 defines the formal semantics for the action language and process creation by giving an inductively defined function `Possibly` that identifies the possible network states that can arise for particular initial conditions. Finally, Section 5 sketches how PDL can be used to prove belief-inference theorems analogous to the inference rules of the belief logics.

See [4, 3] for earlier versions of PDL. Report [3] corrects and extends [4]. This paper significantly extends PDL by adding the possibility of concurrent sessions, and it significantly changes PDL's interpretation of beliefs by making beliefs functions of process' states rather than parts of these states.

2 High-Level Overview

PDL is a theory of communicating sequential processes. PDL defines these processes with a simple, imperative programming language intended to be convenient for describing cryptographic protocols, but also Turing-complete, hence able to describe arbitrary computations. This language is derived from the imperative subset of CAPSL.

PDL identifies protocol participants with CAPSL *agents*, where each agent is able to act in one or more CAPSL *roles*. PDL determines a process performing a role as a function of the data available to the agent creating that process. PDL uses a combination of *slots* and *actions* to describe the states of the different processes.

A *slot* is a family of abstract storage locations, each capable of holding any finite amount of data of any type. Every slot has exactly one member for each process. All slot names are known to all processes, but for each process a slot simply names the unique storage location member of that slot that belongs to this process. There are no "global" values; each process can only access its own storage locations. For simplicity, the remainder of this paper will refer to the single member of a slot that belongs to a particular process as a "slot", since there is no ambiguity.

An *action* is a continuation, a program describing the computation a process has yet to perform. A process' execution state is completely determined by the contents of its slots and the action it has yet to perform. PDL formally defines the meanings of its actions, programs in an imperative programming language, by inductively defining how the various processes' slots' contents change if their computations advance and the actions they have yet to perform change. PDL makes no assumptions about the speed of computations, and does not require that computations advance.

In PDL, every process also has an *address* consisting of the name of the agent that created the process, the name of the role the process is performing, and an integer *session number*. Every message in PDL is a triple consisting of the addresses of its source and destination processes and a "message body" field.

PDL only allows processes to change each others' slots through "send" and "receive" actions. A process executing a "send" hangs, repeatedly sending out a signal to the agent for the intended recipient, until this signal is acknowledged by what the "send" presumes is the intended recipient. The "send" then makes what it presumes is a transfer of the intended message, and terminates. A process executing a "receive" hangs, repeatedly testing to see if it has received a signal. If it receives such a signal, the "receive" acknowledges this signal, receives what it presumes is the intended message from the sender, and terminates.

Unlike most theories of communicating processes, PDL assumes that the network over which the processes communicate is under the complete control of an attacker who can do everything except violate the intended cryptographic properties of the encryption, hash, and nonce-generation functions used.

As in CAPSL, PDL assumes that message transfers take place through the agents of the sending and receiving processes. If an agent receives a message addressed to a process that does not exist, the agent first creates this process, then forwards the message to it. The agent checks that the session numbers of sending and receiving processes are equal. PDL also has a special "start" message causing the agent to create the process the message is addressed to, without considering the session number of the sender, but to then discard the "start" message. The "start" messages can be thought of as coming from the attacker who controls the network, modeling that the attacker might know exactly when a legitimate agent will start a protocol session.

A PDL "state" element is a *partial* description of a *possible* network state. While it is possible in PDL to express full network states — the slot contents and remaining actions of all processes, and all pieces of information obtained by the attacker — PDL is defined so that this is not necessary for making deductions.

Time in PDL is like sex was in Victorian England: it is always there, and always of great interest, but never explicitly mentioned. The PDL conjunction operator for "state" elements is interpreted as "and simultaneously", so that if "s_1 and simultaneously s_2" is a possible state, then s_1 and s_2 are both possible states, but not necessarily vice-versa.

The next two sections describe the HOL implementation of PDL.

3 Types, Terms, and Languages

PDL defines the polymorphic concrete recursive types `:Term`, `:Expr`, `:Action`, `:Proc`, `:State` and `:Belief`. `:Term` defines the data objects exchanged or computed during a protocol session. `:Expr` defines the computations protocol processes perform to compute values. `:Action` and `:Expr` together define a simple programming language giving processes' actions. `:Proc` defines the processes carrying out protocol roles. `:State` elements give partial descriptions of the network state. `:Belief` elements form a language for expressing network properties that analysts can either assume or become compelled to believe on the basis of computations performed by processes.

In its actual implementation, all identifiers in PDL either end with two underscores, or contain two underscores and a semicolon or ampersand, to avoid possible conflicts with names in standard HOL tools or in user-supplied protocol specifications. To avoid clutter, this paper will eliminate these underscores except in the names of the various pairing operators; these names would otherwise be semicolons or ampersands, which have other meanings.

3.1 Primitive Types

In PDL, the following type variables are primitive, instantiated with different types for different protocols:

- `'agent` — person or machine performing one or more protocol roles.
- `'data` — arbitrary data, including keys, nonces, and timestamps.
- `'function` — code for a hash, encryption, or other function, including equality tests and user-defined type checks.
- `'location` — slot name; each principal has distinct, disjoint slots for each pair of distinct locations.
- `'role` — protocol role name.

3.2 Type `:Term`

`:Term` elements can be thought of as meaningful names for bit strings. They give pieces of data that are exchanged or computed during a protocol. Descriptions of the `:Term` constructors and their intended meanings follow.

- `Ta` — `'agent` element
- `Tc` — applied to a `'function` and a `:Term`, it denotes the result of applying this function to this term;
- `Td` — `'data` element.
- `Tf` — `'function` element.
- `Tn` — nonnegative integer, often a session number.
- `Tr` — `'role` element, it makes this role into a `:Term`.
- `Ts` — "start" value causing creation of an initiator process.
- `Tx` — nonexistent term, the "no value" value.
- `__;` — infix pairing operator for `:Term` values.

3.3 Type `:Expr`

`:Expr` elements identify the actual computations performed by principals. In an `:Expr`, a `'location` element represents the value stored in the slot with this name for the principal performing the computation. Descriptions of the `:Expr` constructors follow.

- `Ec` — applied to a `'function` and an `Expr`, it denotes the result of applying the function given by the code to the value given by the expression.
- `El` — applied to a `'location`, it denotes the value stored in the slot named by this `'location` for the principal evaluating the expression.
- `Es` — expression for a "sizeof" term, computed at compile time, used in some of the CAPSL list operations.
- `Ez` — zero expression; used for the CAPSL "forgets" operation.
- `;_` — infix pairing operator for `:Expr` values.

3.4 Type `:Action`

`:Action` elements name actions taken by protocol principals. The `:Action` constructors and informal descriptions of their meanings follow. Their formal meanings are given via the definition of function `Possibly` in Section 4.

- `Assign` — applied to a `'location` and an `:Expr`, it stores the value given by the expression into the slot named by the location.
- `Done` — stop.
- `IfThenElse` — applied to an `:Expr` and two `:Actions`, it tests the boolean value given by the expression, and, if this value is true, does the first action, and otherwise does the second action.
- `Nop` — do nothing.
- `Receive` — applied to a `'location`, it waits, repeatedly testing for a signal that a message is ready. If there is such a signal, it acknowledges it, receives the message, and stores it into the slot named by the location.
- `Send` — applied to an `:Expr`, it waits, repeatedly sending the agent in the "to" address in the message given by the expression a signal that a message is ready. If this signal is acknowledged, supposedly by a process created by this agent, it sends this process the message.
- `Test` — applied to an `:Expr`, it tests the boolean value given by this expression. If this value is true, it does nothing. Otherwise, it aborts all the processes of the agent executing it in the same session.
- `While` — applied to an `:Expr` and an `:Action`, it repeatedly tests the boolean value given by the expression, and does the action if this value is true.
- `_;_` — infix pairing operator for `:Action` values.

3.5 Type `:Proc`

`:Proc` elements name either null processes or processes with remaining actions and functions mapping slots to data values. Descriptions of the `:Proc` constructors follow:

- `Pr` — applied to a function mapping `'location` values to `:Term` values, and to an `:Action`, it names the process whose slots contain the values given by the function and which has yet to perform the action.

- `Px` — the null process.

The main use of the `:Proc` type is having a null process, the "process" being run by an agent for a role and session number for which that agent does not have a process running.

3.6 Type Abbreviations

The remaining PDL type and function definitions use the following type abbreviations:

A `:Mem` is a function mapping `'location` values to `:Term` values. It gives the contents of the slots for an agent or process.

An `:InitMem` is a function mapping `'agent` values to the `:Mem` values giving these agents' initial data possessions.

A `:RunState` is a function mapping `'agent`, `'role`, and session number values to `:Proc` elements. It gives all agents' current or initial processes performing all roles in all sessions.

3.7 Type `:State`

`:State` elements give potentially full descriptions of possible network execution states. The `:State` constructors and informal descriptions of their meanings follow. Their formal meanings are given via the definition of function `Possibly` in Section 4, which identifies possible state values as a function of the assumed algebraic properties of the protocol's functions and constants, the initial data possessions of the protocol's agents, and the means by which agents create processes to perform the protocol's roles.

- `AgentState` — applied to a `:RunState`, it says that each agent is currently running for each role and session number the processes that is the value of this run state for that agent, role, and session number. This completely determines the state of the agents.

- `MsgReceive` — applied to a message `:Term`, it says that the process identified by the "to" address in this message has received the message, will receive it after this process' agent creates it, or, if the "data" field is the "start" message, will be created and the message discarded.

- `MsgSend` — applied to a message `Term`, it says that some process has sent the message to the process identified in the "to" address in the message. Whether that process actually receives it, of course, depends on the attacker controlling the network.

- `NetHas` — applied to a `:Term`, it says that an attacker in complete control of the network either has the term in its possession or could compute it from other terms that are in its possession.

- `_&_` — infix conjunction operator for `:State` values. This operator is interpreted as "and simultaneously" so not all properties of ordinary conjunction hold for it.

3.8 Type `:Belief`

`:Belief` elements name network conditions that analysts can assume or come to believe, including beliefs about other process' actions, on the basis of the computations performed by a process. In the comments that follow, this process is called "the process being examined". Informal definitions of the `:Belief` constructors follow. Section 5 sketches how these constructors can be formalized in a future extension of PDL.

- `Believes` — applied to an address `:Term` and a `:Belief`, it says that the computations performed by the process with this address provide compelling reason to hold this belief.

- `Fresh` — applied to a `:Term`, it says that this term denotes a value that was created for the first time by a process having the same session number as the process being examined.

- `Holds` — applied to an address `Term` and an arbitrary second `Term`, it says that the process with the address given by the first `:Term` either had the value given by the second `:Term` when this process was created, received this value since its creation, or could compute this value from other values that it holds.

- `KnownOnlyTo` — applied to an address `:Term` list and an arbitrary `:Term`, it says that only the processes whose addresses are in this list can possibly compute the value denoted by this term.

- `PrivateKey` — applied to an `'agent`, `'function`, and `:Term`, it says that this agent, for encryptions using this function, has the value given by this term as one of its private keys.

- `PublicKey` — applied to an `'agent`, `'function`, and `:Term`, it says that this agent, for encryptions using this function, has the value given by this term as one of its public keys.

- `Received` — applied to an address `Term` and an arbitrary second `Term`, it says that the process with the address given by the first `:Term` either had the value given by the second `:Term` when this process was created or received this value since its creation.

- `Recognizes` — applied to an address `Term` and an arbitrary second `Term`, it says that the process with the address given by the first `:Term` can identify the value given by the second `:Term` as meaningful information.

- `SameValue` — applied to an address `:Term`, an `:Action`, and two `Expr` values, it says that for the process with the address given by the term, when it had yet to perform the `:Action`, the value given for it by the first expression was the same as the current value given by the second expression for the process currently being examined.

- `Trustworthy` — applied to an `'agent`, it says that all processes created by this agent can be trusted to follow the protocol and perform all the tests that the protocol calls for them to perform.

- `&_` — infix conjunction operator for `:Belief` elements.

The `:Belief` constructs are modeled after corresponding constructs in the BGNY belief logic [2], in the hope that theorems analogous to the rules in this belief logic will be true.

4 Language Meanings

PDL assigns meanings to its `:Action` and `:State` languages via its inductive definition of the function `Possibly`, which identifies all valid partial descriptions of all network states that are possible for particular initial conditions. `Possibly` has the following five arguments:

1. a conjunction of hypotheses about the functions and constants used in the protocol;

2. a `:Term` giving values initially held by an attacker in control of the network;

3. a function mapping agents to the `:Mem` elements that give these agents' initial data possessions;

4. a function defining the processes agents create to play particular roles in particular sessions for the protocol; and

5. a `:State` element.

`Possibly` has value "true" if its fifth argument is a true partial description of a possible network state under the initial conditions given by its other four arguments.

The definition of function `Possibly` uses several subfunctions, as well as constants used to name unevaluated abstract functions.

4.1 Function Constants

The definition of `Possibly` uses the following function constants:

- `SizeOf` is an unevaluated function mapping `:Term` elements to `:num` (i.e., non-negative integer) values.

- `Vs` is an unevaluated function mapping `:State` elements to truth values.

- `Vt` is an unevaluated function mapping `:Term` elements to bit-string values.

The `SizeOf` value of a term is the bit size of the amount of memory the term occupies. PDL uses `SizeOf` to model the `sizeof` CAPSL construct for describing possibly ambiguous concatenation operations.

The `Vs` value of a state is the truth value of the assertion that this state holds, and the `Vt` value of a term is the binary value of this term. PDL uses `Vs` and `Vt` to make inferences using assumed algebraic and logical properties of the intended values of `:State` and `:Term` elements.

4.2 Subfunctions

The definition of `Possibly` uses several subfunctions, but all except one of these subfunctions are trivial. These trivial subfunctions check that terms sent or received as messages are of the appropriate form, and if so extract particular parts of these messages.

A *PDL address* is a `:Term` that is a triple (i.e., a pair whose second element is a pair) consisting of an agent (i.e., `Ta`) term, a role (i.e., `Tr`) term, and a session number (i.e., `Tn`) term. A *PDL message* is a `:Term` that is a triple consisting of two PDL addresses — a "to" address and a "from" address — followed by the body of the message, which is an arbitrary `:Term`.

This paper will not give the definitions of the `Possibly` subfunctions for extracting parts of PDL messages, but their names and descriptions of what they do follow:

- `Fst` gives the first element of a pair, or `Tx` when applied to a term that is not a pair.

- `Snd` gives the second element of a pair, or `Tx` when applied to a term that is not a pair.
- `AgTr` maps an agent term to itself, or `Tx` when applied to a term that is not an agent term.
- `RlTr` maps a role term to itself, or `Tx` when applied to a term that is not a role term.
- `SnTr` maps a session number term to itself, or `Tx` when applied to a term that is not a session number term.
- `TsTr` tells whether a term is `Ts`, the "start" value.
- `ToAgMsg` gives the "to" agent term in a PDL message, or `Tx` when applied to a term that is not of the appropriate form.
- `ToRlMsg` gives the "to" role term in a PDL message, or `Tx` when applied to a term that is not of the appropriate form.
- `ToSnMsg` gives the "to" session number term in a PDL message, or `Tx` when applied to a term that is not of the appropriate form.
- `FrAgMsg` gives the "from" agent term in a PDL message, or `Tx` when applied to a term that is not of the appropriate form.
- `FrRlMsg` gives the "from" role term in a PDL message, or `Tx` when applied to a term that is not of the appropriate form.
- `FrSnMsg` gives the "from" session number term in a PDL message, or `Tx` when applied to a term that is not of the appropriate form.
- `BdMsg` gives the body of a PDL message, or `Tx` when applied to a term that is not of the appropriate form.

The only non-trivial `Possibly` subfunction is `Eval`, which finds the `:Term` value of an `:Expr` with respect to a `:Mem` element associating values with slots. Its definition follows. The pairing operator for expressions (`__;`) changes into the pairing operator for terms (`;__`). In this definition, to avoid clutter, all variables that begin with lower-case letters are implicitly universally quantified.

```
(Eval mm (Ec f ex) = Tc f (Eval mm ex)) /\
(Eval mm (El l) = mm l) /\
(Eval mm (Es tr) = Tn (SizeOf tr)) /\
(Eval mm Ez = Tn 0) /\
(Eval mm (ex1 __; ex2) =
  Eval mm ex1 ;__ Eval mm ex2)
```

4.3 Rules

The 22 rules that inductively define the function `Possibly` follow. Each rule is preceded by a short phrase that summarizes the event or property that the rule defines and is followed by comments that note any special features of the rule. In all of these rules, to avoid clutter, variables that begin with lower-case letters are implicitly universally quantified. The rules use standard HOL notation; the `!` symbol stands for the "for all" quantifier, and the `\` symbol stands for the lambda, "function of", operator. This paper gives a detailed description of only the first rule, which defines the `Assign` action. The other rules are either simpler than, or similar to, the first rule.

Rule `A1` — a process performs an assignment:

```
Possibly hyp nhas initmem initproc
  (AgentState ags _&_ st) /\
(ags ag rl sn =
  Pr mm (Assign l ex _;_ ac)) /\
(Eval mm ex = tr) /\
(hyp ==> (Vt tr = Vt tr')) ==>
Possibly hyp nhas initmem initproc
  (AgentState
     (\ag' rl' sn'.
       ((ag' = ag) /\
        (rl' = rl) /\
        (sn' = sn))
       => (Pr (\l'. (l' = l)
                    => tr'
                    | (mm l')) ac)
       | (ags ag' rl' sn')) _&_
   st)
```

Rule `A1` says that if it is possible that

1. the agents are running the processes given by the "agent state" function `ags`, which maps every agent, role, and session number to a process, and
2. simultaneously, the state `st` holds, and
3. for agent `ag`, role `rl`, and session number `sn`, the value of `ags` is the process having memory `mm` and remaining action `(Assign l ex _;_ ac)`, and
4. for memory `mm`, expression `ex` evaluates to term `tr`, and
5. the conjunction of hypotheses about the functions and constants used in the protocol shows that `tr` and `tr'` have the same value,

then it is possible that

1. the agents are running the processes given by an "agent state" function equal to `ags` except for agent `ag`, role `rl`, and session number `sn`, for which the process is changed to one in which the value stored in slot `l` is `tr'`, the values stored in all the other slots are the same as those for memory `mm`, and the process' remaining action is `ac`, and

2. simultaneously, the state `st` holds.

Rule `A2` — a process takes a "true" if-then-else branch:

```
Possibly hyp nhas initmem initproc
  (AgentState ags _&_ st) /\
(ags ag rl sn =
  Pr mm (IfThenElse ex ac1 ac2 _;_ ac3)) /\
(Eval mm ex = tr) /\
(hyp ==> (Vt tr = [T])) ==>
Possibly hyp nhas initmem initproc
  (AgentState
     (\ag' rl' sn'.
       ((ag' = ag) /\
        (rl' = rl) /\
        (sn' = sn))
       => (Pr mm (ac1 _;_ ac3))
       | (ags ag' rl' sn')) _&_
   st)
```

Rule `A3` — a process takes a "false" if-then-else branch:

```
Possibly hyp nhas initmem initproc
  (AgentState ags _&_ st) /\
(ags ag rl sn =
  Pr mm (IfThenElse ex ac1 ac2 _;_ ac3)) /\
(Eval mm ex = tr) /\
(hyp ==> (Vt tr = [F])) ==>
Possibly hyp nhas initmem initproc
  (AgentState
     (\ag' rl' sn'.
       ((ag' = ag) /\
        (rl' = rl) /\
        (sn' = sn))
       => (Pr mm (ac2 _;_ ac3))
       | (ags ag' rl' sn')) _&_
   st)
```

Rule `A4` — a process performs a null operation:

```
Possibly hyp nhas initmem initproc
  (AgentState ags _&_ st) /\
(ags ag rl sn = Pr mm (Nop _;_ ac)) ==>
Possibly hyp nhas initmem initproc
  (AgentState
     (\ag' rl' sn'.
       ((ag' = ag) /\
        (rl' = rl) /\
        (sn' = sn))
       => (Pr mm ac)
       | (ags ag' rl' sn')) _&_
   st)
```

Rule `A5` — a process receives a message:

```
Possibly hyp nhas initmem initproc
  (AgentState ags _&_
   MsgReceive tr _&_ st) /\
(hyp ==> (Vt tr = Vt tr')) /\
~(TsTr (BdMsg tr')) /\
(ToAgMsg tr' = Ta ag) /\
(ToRlMsg tr' = Tr rl) /\
(ToSnMsg tr' = Tn sn) /\
(FrSnMsg tr' = Tn sn) /\
(ags ag rl sn =
  Pr mm (Receive l _;_ ac)) ==>
Possibly hyp nhas initmem initproc
  (AgentState
     (\ag' rl' sn'.
       ((ag' = ag) /\
        (rl' = rl) /\
        (sn' = sn))
       => (Pr (\l'. (l' = l)
                    => tr'
                    | (mm l')) ac)
       | (ags ag' rl' sn')) _&_
   st)
```

Rule `A5` requires a process to have the same session number as a message's putative sender before it allows this process to receive that message. It also excludes the "start" message. Otherwise, it assigns the message to the slot into which the process is waiting to receive a message.

Rule `A6` — a process sends a message:

```
Possibly hyp nhas initmem initproc
  (AgentState ags _&_ st) /\
(ags ag rl sn =
  Pr mm (Send ex _;_ ac)) /\
(Eval mm ex = tr) /\
(hyp ==> (Vt tr = Vt tr')) ==>
Possibly hyp nhas initmem initproc
  (AgentState
     (\ag' rl' sn'.
       ((ag' = ag) /\
        (rl' = rl) /\
        (sn' = sn))
       => (Pr mm ac)
       | (ags ag' rl' sn')) _&_
   MsgSend tr' _&_
   st)
```

Rule `A7` — a process performs a test that passes:

```
Possibly hyp nhas initmem initproc
  (AgentState ags _&_ st) /\
(ags ag rl sn =
  Pr mm (Test ex _;_ ac)) /\
(Eval mm ex = tr) /\
(hyp ==> (Vt tr = [T])) ==>
Possibly hyp nhas initmem initproc
  (AgentState
     (\ag' rl' sn'.
       ((ag' = ag) /\
        (rl' = rl) /\
        (sn' = sn))
       => (Pr mm ac)
       | (ags ag' rl' sn')) _&_
   st)
```

Rule `A8` — a process performs a test that fails:

```
Possibly hyp nhas initmem initproc
  (AgentState ags _&_ st) /\
(ags ag rl sn =
  Pr mm (Test ex _;_ ac)) /\
(Eval mm ex = tr) /\
(hyp ==> (Vt tr = [F])) ==>
Possibly hyp nhas initmem initproc
  (AgentState
     (\ag' rl' sn'.
       (sn' = sn)
       => (Pr (\l. Tn 0) Abort)
       | (ags ag' rl' sn')) _&_
   st)
```

Rule `A8` aborts and zeros the memory of every process in a session that involves a test that fails.

Rule `A9` — a process continues a "while" loop:

```
Possibly hyp nhas initmem initproc
  (AgentState ags _&_ st) /\
(ags ag rl sn =
  Pr mm (While ex ac1 _;_ ac2)) /\
(Eval mm ex = tr) /\
(hyp ==> (Vt tr = [T])) ==>
Possibly hyp nhas initmem initproc
  (AgentState
     (\ag' rl' sn'.
       ((ag' = ag) /\
        (rl' = rl) /\
        (sn' = sn))
       => (Pr mm (ac1 _;_
                  While ex ac1 _;_
                  ac2))
       | (ags ag' rl' sn')) _&_
   st)
```

Rule `A10` — a process exits a "while" loop:

```
Possibly hyp nhas initmem initproc
  (AgentState ags _&_ st) /\
(ags ag rl sn =
  Pr mm (While ex ac1 _;_ ac2)) /\
(Eval mm ex = tr) /\
(hyp ==> (Vt tr = [F])) ==>
Possibly hyp nhas initmem initproc
  (AgentState
     (\ag' rl' sn'.
       ((ag' = ag) /\
        (rl' = rl) /\
        (sn' = sn))
       => (Pr mm ac2)
       | (ags ag' rl' sn')) _&_
   st)
```

Rule `A11` — a process performs a pair of actions:

```
Possibly hyp nhas initmem initproc
  (AgentState ags _&_ st) /\
(ags ag rl sn =
  Pr mm ((ac1 _;_ ac2) _;_ ac3)) ==>
Possibly hyp nhas initmem initproc
  (AgentState
     (\ag' rl' sn'.
       ((ag' = ag) /\
        (rl' = rl) /\
        (sn' = sn))
       => (Pr mm (ac1 _;_ ac2 _;_ ac3))
       | (ags ag' rl' sn')) _&_
   st)
```

Rule `C1` — operator `_&_` is commutative, associative, and idempotent:

```
Possibly hyp nhas initmem initproc st /\
((!stx sty. Vs (stx _&_ sty) =
            Vs (sty _&_ stx)) /\
 (!stx sty stz.
   Vs ((stx _&_ sty) _&_ stz) =
   Vs (stx _&_ sty _&_ stz)) /\
 (!stx. Vs (stx _&_ stx) = Vs stx) ==>
 (Vs st = Vs st')) ==>
Possibly hyp nhas initmem initproc st'
```

Rule `C1` is the only rule involving `Vs`. It uses `Vs` to state several general properties of the "and simultaneously" operator at the same time.

Rule `C2` — part of a possible state is a possible state:

```
Possibly hyp nhas initmem initproc
  (st1 _&_ st2) ==>
Possibly hyp nhas initmem initproc st1
```

Rule `I1` — a state with all agents running no processes is possible:

```
Possibly hyp nhas initmem initproc
  (AgentState (\ag rl sn. Px))
```

Rule `N1` — the attacker possesses everything that is sent by every process:

```
Possibly hyp nhas initmem initproc
  (MsgSend tr _&_ st) /\
(hyp ==> (Vt tr = Vt tr')) ==>
Possibly hyp nhas initmem initproc
  (NetHas tr' _&_ MsgSend tr' _&_ st)
```

Rule `N2` — the attacker can send anything that it possesses:

```
Possibly hyp nhas initmem initproc
  (NetHas tr _&_ st) /\
(hyp ==> (Vt tr = Vt tr')) ==>
Possibly hyp nhas initmem initproc
  (MsgReceive tr' _&_ NetHas tr' _&_ st)
```

Rule `N3` — the attacker can pair any two things that it possesses:

```
Possibly hyp nhas initmem initproc
  (NetHas tr1 _&_ NetHas tr2 _&_ st) /\
(hyp ==> (Vt (tr1 __; tr2) = Vt tr3)) ==>
Possibly hyp nhas initmem initproc
  (NetHas tr1 _&_
   NetHas tr2 _&_
   NetHas tr3 _&_
   st)
```

Rule `N4` — the attacker can take apart any pairs that it possesses:

```
Possibly hyp nhas initmem initproc
  (NetHas (tr1 __; tr2) _&_ st) ==>
Possibly hyp nhas initmem initproc
  (NetHas tr1 _&_
   NetHas tr2 _&_
   NetHas (tr1 __; tr2) _&_
   st)
```

Rule `N5` — the attacker can apply any functions whose code it possesses to any terms that it possesses:

```
Possibly hyp nhas initmem initproc
  (NetHas (Tf f) _&_ NetHas tr1 _&_ st) /\
(hyp ==> (Vt (Tc f tr1) = Vt tr2)) ==>
Possibly hyp nhas initmem initproc
  (NetHas (Tf f) _&_
   NetHas tr1 _&_
   NetHas tr2 _&_
   st)
```

Rule `N6` — the attacker has the `nhas` and "start" terms, and all agent names, role names, and session numbers:

```
Possibly hyp nhas initmem initproc st ==>
Possibly hyp nhas initmem initproc
  (NetHas (nhas __; Ts __;
          Ta ag __; Tr rl __; Tn sn) _&_
   st)
```

Rule `P1` — an agent starts a new process and discards the "start" message:

```
Possibly hyp nhas initmem initproc
  (AgentState ags _&_
   MsgReceive tr _&_
   st) /\
(hyp ==> (Vt tr = Vt tr')) /\
TsTr (BdMsg tr') /\
(ToAgMsg tr' = Ta ag) /\
(ToRlMsg tr' = Tr rl) /\
(ToSnMsg tr' = Tn sn) /\
(ags ag rl sn = Px) /\
(initproc ag rl sn = Pr mm ac) /\
(!l.
  (mm l = Tn 0) \/
  (mm l = initmem ag l) \/
  (mm l = Ta ag) \/
  (mm l = Tr rl) \/
  (mm l = Tn sn)) ==>
Possibly hyp nhas initmem initproc
  (AgentState
     (\ag' rl' sn'.
       ((ag' = ag) /\
        (rl' = rl) /\
        (sn' = sn))
       => (Pr mm ac)
       | (ags ag' rl' sn')) _&_
   st)
```

A "start" message need not come from a putative sender with the same session number as the process this message initiates. Every slot for a new process must contain zero, or a value initially held by the agent starting the new process, or some part of the new process' address.

Rule `P2` — an agent starts a new process and forwards a message to it:

```
Possibly hyp nhas initmem initproc
  (AgentState ags _&_
   MsgReceive tr _&_
   st) /\
(hyp ==> (Vt tr = Vt tr')) /\
~(TsTr (BdMsg tr')) /\
(ToAgMsg tr' = Ta ag) /\
(ToRlMsg tr' = Tr rl) /\
(ToSnMsg tr' = Tn sn) /\
(FrSnMsg tr' = Tn sn) /\
(ags ag rl sn = Px) /\
(initproc ag rl sn = Pr mm ac) /\
(!l.
  (mm l = Tn 0) \/
  (mm l = initmem ag l) \/
  (mm l = Ta ag) \/
  (mm l = Tr rl) \/
  (mm l = Tn sn)) ==>
Possibly hyp nhas initmem initproc
  (AgentState
     (\ag' rl' sn'.
       ((ag' = ag) /\
        (rl' = rl) /\
        (sn' = sn))
       => (initproc ag rl sn)
       | (ags ag' rl' sn')) _&_
   MsgReceive tr _&_
   st)
```

An ordinary message to a non-existent process must come from a putative sender with the same session number as the process this message initiates. Every slot for a new process must contain zero, or a value initially held by the agent starting the new process, or some part of the new process' address.

5 Belief Inferences

This section sketches how belief inferences analogous to the rules of the belief logics can later be added to PDL as proved theorems.

The first step will be defining "beliefs are correct" interpretations of PDL's belief constructs by formalizing their informal meanings. After that, it will be possible to prove theorems of the form that if the "beliefs are correct" interpretations of belief hypotheses hold, then the "beliefs are correct" interpretations of belief conclusions also hold. These theorems can then be used to reason directly from assumed initial beliefs to desired belief conclusions, as in belief-logic inferences, but in a completely rigorous way.

Proving such theorems will probably be difficult, because of the large number of possibilities that must be considered, but the "beliefs are correct" interpretations, and the exact statements of the theorems, can be adjusted to make the theorems true. Paulson's work on proving analogous theorems for his models [13, 14, 15] should be helpful in dealing with these complexities, as will work by Lowe on proving that searches find all possible attacks [9], and work by Thayer, Herzog, and Guttman on proving properties of abstract characterizations of possible protocol executions [17].

Unlike the belief logics, PDL will not consider justifiable beliefs to be *parts of* process' states, but *functions of* these states. This will guarantee that PDL belief inferences are made only on the basis of the computations actually performed by processes.

PDL will address the inference limitations identified in Section 1 as follows:

- Type and equality checking — all belief inferences will be functions of the tests and computations processes have actually performed, not the tests and computations they *could* perform.

- Concurrent sessions — CAPSL and PDL model arbitrary numbers of concurrent sessions, including sessions in which the same agent plays multiple roles.

- Misinterpretations — PDL will make belief inferences from computations performed by one process, if these inferences depend on computations performed by another process, only if the computations performed by the first process justify `SameValue` conclusions relating the data the first process holds to data released by the second process, and will then make these inferences only from the computations performed by the second process before it released this data.

- Accidental disclosure — PDL will address this possibility with its `KnownOnlyTo` construct and conservative inferences based on all the information that *might* have been released to the network in an arbitrary number of previous or concurrent sessions.

- Algebraic properties attacks — The `hyp` argument to PDL's `Possibly` function will make it possible for PDL to model all of these attacks, though it will only be possible to consider some of them in a fast, automatic, proof-construction algorithm.

The great advantage of this approach is that the complexities that arise in proving desired belief inferences will not need to be considered again once these inferences are proved. This gives the possibility that PDL-based protocol analyses can be comparable in thoroughness to the attack-construction analyses, but still be comparable in speed to the belief-logic analyses.

References

[1] S. Brackin. Deciding cryptographic protocol adequacy with HOL: The implementation. In *Theorem Proving in Higher Order Logics*, number 1125 in Lecture Notes in Computer Science, pages 61–76, Turku, Finland, August 1996. Springer-Verlag.

[2] S. Brackin. A HOL extension of GNY for automatically analyzing cryptographic protocols. In *Proceedings of Computer Security Foundations Workshop IX*, County Kerry, Ireland, June 1996. IEEE.

[3] S. Brackin. A State-Based HOL Theory of Protocol Failure. Technical Report 98007, Arca Systems, Inc., Ithaca, NY, October 1997. Available at `www.arca.com`.

[4] S. Brackin. A state-based HOL theory of protocol failure. In *Supplementary Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics: TPHOLs '97*, pages 35–47, Murray Hill, NJ, August 1997.

[5] S. Brackin. Evaluating and improving protocol analysis by automatic proof. In *Proceedings of Computer Security Foundations Workshop XI*, Rockport, MA, June 1998. IEEE.

[6] D. Dolev and A. Yao. On the security of public key protocols. Technical Report STAN-CS-81-854, Stanford University, Stanford, CA, May 1981.

[7] M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, Cambridge, 1993.

[8] G. Lowe. Some new attacks upon security protocols. In *Proceedings of Computer Security Foundations Workshop IX*, County Kerry, Ireland, June 1996. IEEE.

[9] G. Lowe. Towards a completeness result for model checking of security protocols. In *Proceedings of Computer Security Foundations Workshop XI*, Rockport, MA, June 1998. IEEE.

[10] C. Meadows. A system for the specification and analysis of key management protocols. In *Proceedings of the Symposium on Security and Privacy*, pages 182–195, Oakland, CA, May 1991. IEEE.

[11] J. Millen. The Interrogator model. In *Proceedings of the Symposium on Security and Privacy*, pages 251–260, Oakland, CA, May 1995. IEEE.

[12] L. Paulson. *ML for the Working Programmer.* Cambridge University Press, Cambridge, 1993.

[13] L. Paulson. Proving properties of security protocols by induction. Technical Report 409, CUCL, December 1996.

[14] L. Paulson. Mechanized proofs for a recursive authentication protocol. Technical Report 418, CUCL, Cambridge, UK, April 1997.

[15] L. Paulson. Mechanized proofs of security protocols: Needham-Schroeder with public keys. Technical Report 413, CUCL, Cambridge, UK, January 1997.

[16] E. Snekkenes. *Formal Specification and Analysis of Cryptographic Protocols.* PhD thesis, University of Oslo, Oslo, Norway, January 1995.

[17] J. Thayer, J. Herzog, and J. Guttman. Honest ideals on strand spaces. In *Proceedings of Computer Security Foundations Workshop XI*, Rockport, MA, June 1998. IEEE.