CPSA Design

John D. Ramsdell Joshua D. Guttman
The MITRE Corporation
CPSA Version 2.2.7

February 1, 2012

Contents

Chapter 1

Introduction

The Cryptographic Protocol Shapes Analyzer (CPSA) attempts to enumerate all essentially different executions possible for a cryptographic protocol. We call them the *shapes* of the protocol. Naturally occurring protocols have only finitely many, indeed very few shapes. Authentication and secrecy properties are easy to determine from them, as are attacks and anomalies.

The shapes analysis is performed within a pure Dolev-Yao model. The CPSA program reads a sequence of problem descriptions, and prints the steps it used to solve each problem. For each input problem, CPSA is given some initial behavior, and it descovers what shapes are compatible with it. Normally, the initial behavior is from the point of view of one participant. The analysis reveals what the other participants must have done, given the participant's view.

Ideally, if there is a finite number of shapes associated with a problem statement, CPSA will find them given enough resources. In other words, the search is complete, i.e. every shape can in fact be found in a finite number of steps. The Completeness of CPSA [4] contains a proof that the search algorithm is complete.

A CPSA release includes two other documents, The CPSA Specification [7] and the CPSA Primer [6]. The specification describes the CPSA algorithm in a form that is closely related to its implementation.

There are many design decisions that are not reflected in the specification. Including these decisions in the specification would clutter the document. The purpose of this document is to describe the key omitted design decisions and provide a link between the specification and the source code. It assumes the specification has been thoroughly read. Definitions are not reproduced,

so the specification should be accessible when reading this document. The CPSA Primer provides an overview of CPSA, and may be worth reading before this document is approached.

1.1 Notation

Originally, the specification and the design were one and the same. Everything was specified in the design formalism. After the split, the description of protocols and preskeletons diverged by omitting details in the design formalism in what is used for the specification formalism.

The key difference between the two formalisms is the design formalism more directly models the Haskell data structures used in the CPSA program. An instance of a Haskell data structure is modeled as an element in an order-sorted term algebra [3]. The reduction systems in the specification translate to term reduction systems in the design.

Unlike the specification, zero-based indexing is used though out this document and in the source code it describes. In what follows, a finite sequence is a function from an initial segment of the natural numbers. The length of a sequence f is $|f|$, and sequence $f = \langle f(0), \ldots, f(n-1) \rangle$ for $n = |f|$. Alternatively, $\langle x_0, x_1, \ldots, x_{n-1} \rangle = x_0 :: x_1 :: \ldots :: x_{n-1} :: \langle\rangle$. If S is a set, then S^* is the set of finite sequences of S, and S^+ is the non-empty finite sequences of S. The concatenation of sequences f and f' is $f \frown f'$. The prefix of sequence f of length n is $f|_n$.

Those familiar with literature on strand spaces might wonder why lowercase k is used for skeletons rather than blackboard bold $\mathbb{A}$. The notation used in the design document is motivated by the code, and k with and without decoration somehow became associated with preskeletons, probably because p, r, t, and s, were already in use. Many other notational conventions are directly inspired by the code.

In this document, lowercase Latin letters usually stand for terms, and uppercase Latin letters stand for sequences or sets of terms.

Chapter 2

Messages

The formalism used in the design and the specification for message algebras is the same, an order-sorted term algebra. This chapter describes the relation between terms and the external syntax used by the CPSA program for the Basic Crypto Algebra, and then describes the interface between the algebra module and the rest of the program.

Table 2.1 presents a slightly modified signature for the Basic Crypto Algebra. It specifies a syntax for operations that follows mathematical tradition, such as writing K_A for $\mathsf{pubk}(A)$. Tag constants are quoted strings.

For pairing, parentheses are omitted when the context permits, and comma is right associative. Pairing was once called concatenation, hence the use of the symbol `cat` for pairing.

In the actual implementation, binary operations for pubk and privk have been added. The first argument is a name and the second argument is a tag. This extension was added so as to model key usage, such as associating a signing and encryption key with a name.

In the S-expression syntax used by the program, the simplest term is a variable, which syntactically is a SYMBOL as described in Appendix A. Internally, each variable has a sort, so the sort of each variable in the input must be declared in a `vars` form, such as:

```
(vars (t text) (n name) (k akey)).
```

The translation of S-expression terms is given in Table 2.2. Figure 3.2 on Page 13 contains examples of BCA message terms. Also see TERM in Table A.1, Appendix A.

Base sort symbols: **name**, **text**, **data**, **skey**, **akey**
Non-base sort symbol: **mesg**

Subsorts: **name**, **text**, **data**, **akey**, **skey** $<$ **mesg**

$\{\!\lvert \cdot \rvert\!\}_{(\cdot)}$: **mesg** $\times$ **mesg** $\to$ **mesg**	Encryption
$\#$: **mesg** $\to$ **mesg**	Hashing
$(\cdot,\cdot)$: **mesg** $\times$ **mesg** $\to$ **mesg**	Pairing
"$\ldots$": **mesg**	Tag constants
$K_{(\cdot)}$: **name** $\to$ **akey**	Public key of name
$(\cdot)^{-1}$: **akey** $\to$ **akey**	Inverse of asymmetric key
ltk: **name** $\times$ **name** $\to$ **skey**	Long term shared key

Axiom: $(x^{-1})^{-1} \approx x$ for x: **akey**

Table 2.1: Basic Crypto Signature

$$
\begin{aligned}
[\![\texttt{(pubk } t\texttt{)}]\!] &= K_{[\![t]\!]} \\
[\![\texttt{(privk } t\texttt{)}]\!] &= K^{-1}_{[\![t]\!]} \\
[\![\texttt{(invk } t\texttt{)}]\!] &= [\![t]\!]^{-1} \\
[\![\texttt{(ltk } t_0\ t_1\texttt{)}]\!] &= \mathsf{ltk}([\![t_0]\!], [\![t_1]\!]) \\
[\![\texttt{"}\ldots\texttt{"}]\!] &= \text{“}\ldots\text{”} \\
[\![\texttt{(enc } t_0\ \ldots\ t_{n-1}\ t_n\texttt{)}]\!] &= \{\!\lvert [\![\texttt{(cat } t_0\ \ldots\ t_{n-1}\texttt{)}]\!] \rvert\!\}_{[\![t_n]\!]} \\
[\![\texttt{(hash } t_0\ \ldots\ t_{n-1}\texttt{)}]\!] &= \#[\![\texttt{(cat } t_0\ \ldots\ t_{n-1}\texttt{)}]\!] \\
[\![\texttt{(cat } t\texttt{)}]\!] &= [\![t]\!] \\
[\![\texttt{(cat } t_0\ t_1\ \ldots\texttt{)}]\!] &= ([\![t_0]\!], [\![\texttt{(cat } t_1\ \ldots\texttt{)}]\!])
\end{aligned}
$$

Table 2.2: S-expression Terms

The code that implements the Basic Crypto Algebra does not directly implement an order-sorted algebra. Instead, it implements a many-sorted algebra and exports an order-sort algebra based on the implementation. Appendix B provides the complete details of the implementation.

2.1 Algebra Interface

The details of each implementation of a CPSA message algebra are hidden by an interface. This section presents the view of a term algebra exposed by the interface. Some aspects of the interface are omitted from this discussion. For example, each implementation of an algebra must provide a means to read a term from an S-expression, and write a term as an S-expression. Also omitted are functions in the interface that are specializations of a more general function added to enhance performance.

Each algebra provides a predicate to determine if a term is a variable, and another to determine if a term is an atom. A fresh variable generator is in the interface. Given a generator state and a term, it produces a clone of the term in which the variables have been replaced with freshly generated ones. It also returns the new generator state.

2.1.1 Equations

An algebra reports answers to unification and matching problems by returning a sequence of order-sorted substitutions. A different data structure is used for each problem, in this document indicated by using σ for answers to the unification problem of $\sigma(t_0) \equiv \sigma(t_1)$, and using σ_E for answers to the matching problem of $\sigma_E(t_0) \equiv t_1$. As the typical case is for sets of equations to be solved, the unification and match functions have been designed to allow an incremental approach to solving the members of the set, by extending a substitution for one pair of equated terms. They have the following signatures:

$$\begin{aligned} \mathit{unify} &: \mathcal{T}_\top(X) \times \mathcal{T}_\top(X) \times (X \to \mathcal{T}_\top(X)) \to (X \to \mathcal{T}_\top(X))^* \\ \mathit{match} &: \mathcal{T}_\top(X) \times \mathcal{T}_\top(Y) \times (X \to \mathcal{T}_\top(Y)) \to (X \to \mathcal{T}_\top(Y))^* \end{aligned}$$

An answer to the matching problem is called an *environment*. An environment differs from a substitution produced as an answer to a unification problem in that it may explicitly specify identity mappings, thus forbidding

extensions to the environment that conflicts with these mappings. This distinction is crucial for correctly answering matching problems by iteratively extending an environment.

To support checks to see if terms are isomorphic via the match function, the algebra interface includes the *match variable renaming* predicate that tests an environment to see if it is a one-to-one variable-to-variable order sorted substitution.

To support pruning, there is a function that given an environment and a term, determines if there are variables in the term that are in the domain of the environment.

2.1.2 Term Internals

The interface includes a function that returns the set of variables in a term, and a function that returns the terms carried by a term. Other subterms are accessed via position oriented functions. Recall that a position is a finite sequence of natural numbers, and the message in t that occurs at p, is written $t @ p$. The interface includes a data type for a position that hides its implementation. The interface also includes the ancestors function $anc(t, p)$ and the carried positions function $carpos(t, t')$ as defined in the specification.

Each algebra provides a way to obtain a set of positions at which a subterm occurs within a term, and a way to replace the subterm at a given position with another term. These functions are used to generalize by variable separation.

Definition 2.1 (All Positions)**.** Given a term t, the set of positions at which t occurs in t' is $allpos(t, t')$, where

$$allpos(t, t') = \begin{cases} \{\langle\rangle\} & \text{if } t' \equiv t, \text{ else} \\ \{\langle i\rangle \frown p \mid p \in allpos(t, t_i), i < n\} & \\ & \text{if } t' = f(t_0, \ldots, t_{n-1}), \text{ else} \\ \{\} & \text{otherwise.} \end{cases}$$

Definition 2.2 (Replace)**.** Given terms t and t', and position p, the term that results from replacing the term at p with t in t', is $replace(t, p, t')$, where

$$replace(t, \langle\rangle, t') = t;$$
$$replace(t, \langle i\rangle \frown p, f(t_0, \ldots, t_{n-1})) = f(t'_0, \ldots, t'_{n-1}) \text{ where}$$
$$t'_j = \begin{cases} replace(t, p, t_i) & \text{if } i = j; \\ t_j & \text{otherwise.} \end{cases}$$

2.1.3 Encryptions and Derivations

Finally, the remaining functions in the interface are the ones that expose the encryption oriented properties of terms. The *decryption key* function returns the key used to decrypt a term if it is an encryption, otherwise it returns an error indicator. The *encryptions* function returns the set of encryption terms carried by a term, each one paired with its encryption key. CPSA treats a hashed term as if it were an encryption in which the term that is hashed is the encryption key, so hashes with their content is also returned by this function. The penetrator derivable function from the section in the specification of the same name is in the interface. Given a derivable predicate that has been specialized with a given set of supported terms and a set of atoms to avoid, a target term, and a source term, the *protectors* function returns an error indicator if the target is carried by the source outside of an encryption, where the derivable predicate is used to determine if a decryption key can be used to expose the target. Otherwise, it returns a set of encryptions in the source that carry the target and have underivable decryption keys. If two encryptions protect the target, only the outside one is returned. The inside encryption is the one that is carried by the outside encryption. Pseudo code for the decryption key and the protectors functions is in the specification.

Chapter 3

Protocols and Preskeletons

Terms over an order-sorted signature extended from a message signature describe key data structures in the CPSA program. Given a message signature that defines the sort **mesg** and the atoms, the additional sorts and operations are in the CPSA Signature in Table 3.1. The signature uses the sort s **list** for sequences of terms of sort s, and the sort s **set** for injective sequences of terms of sort s.

Definition 3.1 (CPSA Algebra)**.** Algebra $\mathfrak{A}(X)$ is a CPSA *algebra* if it is the order-sorted quotient term algebra generated by variable set X, and X has the following property. For each sort s, X_s is empty when s is **atom**, **evt**, **role**, **maplet**, **instance**, **node**, **ordering**, and **preskel**.

Some of the terms over a CPSA signature are not well-formed, and omitted from interpretation. The text describing a term of a sort includes the conditions for it being well-formed.

In what follows, the external syntax for protocols is presented, and later, its translation into terms over a CPSA signature. For preskeletons, the internal representation is presented first, followed by its external syntax.

3.1 Protocols

A protocol defines the patterns of allowed behavior for non-adversarial participants, called the *regular* participants. The behavior of each regular participant is an instance of a protocol template, called a role. Figure 3.1 displays the roles that make up the Needham-Schroeder protocol.

Additional sort symbols: **atom**, **evt**, **role**, **maplet**, **instance**, **node**, **ordering**, and **preskel**

Subsorts: for each base sort s, $s < \mathbf{atom} < \mathbf{mesg}$

$+$: $\mathbf{mesg} \to \mathbf{evt}$ $\quad -$: $\mathbf{mesg} \to \mathbf{evt}$ $\quad \mathsf{r}$: $\mathbf{evt\,list} \times \mathbf{atom\,set} \times \mathbf{atom\,set} \to \mathbf{role}$

m: $\mathbf{mesg} \times \mathbf{mesg} \to \mathbf{maplet}$ $\quad \mathsf{i}$: $\mathbf{role} \times \mathbf{nat} \times \mathbf{maplet\,set} \to \mathbf{instance}$

n: $\mathbf{nat} \times \mathbf{nat} \to \mathbf{node}$ $\quad \mathsf{o}$: $\mathbf{node} \times \mathbf{node} \to \mathbf{ordering}$

k: $\mathbf{role\,set} \times \mathbf{instance\,list} \times \mathbf{ordering\,set} \times \mathbf{atom\,set} \times \mathbf{atom\,set} \to \mathbf{preskel}$

mesg	the sort of all messages (implementation of $\top$)
atom	the sort of all base sorted messages
evt	a transmission or reception event
trace	a sequence of events used in a role
role	a trace, a non-originating set, and a uniquely-originating set
protocol	a set of roles
nat	a natural number
maplet	a map from a role variable to a preskeleton term
instance	a strand's trace and inheritance as instantiated from a role
node	a pair of numbers, a strand identifier and a strand position
ordering	a causal ordering between a pair of nodes
preskel	a preskeleton

Table 3.1: CPSA Signature

Figure 3.1: Needham-Schroeder Initiator and Responder Roles

```
(defrole resp (vars (b a name) (n2 n1 text))
  (trace (recv (enc n1 a (pubk b)))
         (send (enc n1 n2 (pubk a)))
         (recv (enc n2 (pubk b)))))
```

Figure 3.2: Needham-Schroeder Responder Role

In S-expression syntax, a protocol is a named set of roles and is defined by the `defprotocol` form. See PROTOCOL in Table A.1, Appendix A.

```
(defprotocol ns basic
  (defrole init ...)
  (defrole resp ...))
```

The name of this protocol (ID) is `ns`, and the second identifier (ALG) names the message algebra in use. The identifier for the Basic Crypto Algebra is `basic`.

During the reading process, the appropriate algebra is implicitly bound to the internal representation of a protocol and many data structures derived from it. The protocol name is used at read time to bind it with its usages, and for output and error messages, but is otherwise unused and thus omitted from the design specification. The internal representation of a protocol is simply a set of roles—as a term of sort **role set** in Table 3.1.

The S-expression syntax for a role has a name, a declared set of variables, and a trace that provides a template for the behavior of its instances. A trace is a non-empty sequence of events, either a message transmission or a reception. An outbound term is `(send` t`)` and an inbound message with term t is `(recv` t`)`. The translations of events are $+[\![t]\!]$ and $-[\![t]\!]$ respectively, where $[\![t]\!]$ is the translation of the S-expression t into an term of sort **mesg** in Table 3.1. Needham-Schroeder responder's role in S-expression syntax is in Figure 3.2.

Some atoms in a role have special properties. The atoms listed in the `non-orig` form are assumed to be non-originating, and those in the `uniq-orig` form are assumed to be uniquely originating. The implications of these assumption is as in the specification.

Internally, role $\mathsf{r}(C, N, U)$ has a trace C, and two sets of atoms, N and U. The atoms in N are assumed to be non-originating, and the atoms in U are

assumed to be uniquely originating. As with protocols, the name is used during input and output, but omitted from this specification.

A role is well-formed if it satisfies the conditions listed for a role in the specification. A protocol is well-formed if no variable occurs in more than one role. The external syntax used by CPSA uses variable renaming to create the illusion that the same variable may occur in two roles. In the external syntax, two roles may share the same identifier.

Associated with each protocol is an implicit role. For some variable x of sort **mesg**, that does not occur in any role in the protocol, there is a *listener role* of the form $lsn = \mathsf{r}(\langle -x, +x \rangle, \langle\rangle, \langle\rangle)$. A listener role is used to assert that a term is not a secret. In the implementation, the only difference between a listener role and non-listener roles is its name is the empty string, a fact used when printing.

3.2 Preskeletons

The other key CPSA data structure is a preskeleton—see the k operation in Table 3.1. A preskeleton is used to encode classes of protocol executions, including its shapes, the answers produced by CPSA. One component of a preskeleton is its protocol, and one component is a set of strands. There are more components, but the set's representation is presented next.

As in the specification, a sequence of instances represents a set of strands. The instance $\mathsf{i}(r, h, E)$ contains a role r, a positive number h called its height, the length of the trace associated with the instance, and an environment E, a term of sort **maplet set**.

The environment E is well-formed if it represents the order-sorted substitution σ_E such that for every maplet $\mathsf{m}(x, y)$, $\sigma_E x = y$. Note that x is always a variable, unlike its analog in the external syntax. An instance is well-formed if its role is well-formed, its environment is well-formed, and its height is not greater that the length of its role's trace.

The set of strands in a preskeleton is represented by a sequence of instances. The identity of a strand is its position in the sequence, which is where the description of its trace is located. The node $\mathsf{n}(s, p)$ is associated with the event at position p in strand s's trace. In other words, if I is a sequence of instances, the event at $\mathsf{n}(s, p)$, written $evt(I, \mathsf{n}(s, p))$, is $\sigma_E(C(p))$, where $I(s) = \mathsf{i}(\mathsf{r}(C, N, U), h, E)$ and $p < h \leq |C|$. A node associated with an inbound term is a *reception node*, and a node associ-

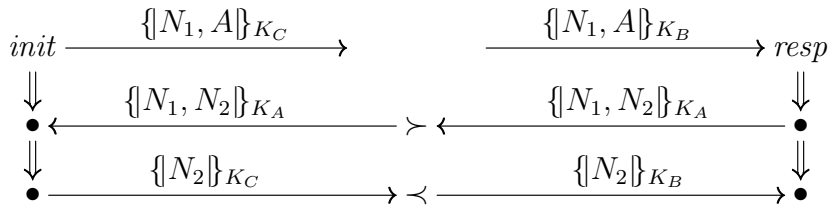

Figure 3.3: Needham-Schroeder Shape (K_A^{-1} uncompromised, N_2 fresh)

ated with an outbound term is a *transmission node*. The term stripped of its direction is written $msg(I, \mathsf{n}(s,p))$. The set of nodes in sequence I is $\{\mathsf{n}(s,p) \mid s < |I|, I(s) = \mathsf{i}(r,h,E), p < h\}$.

The preskeleton $\mathsf{k}(P, I, O, N, U)$ contains a protocol P, a non-empty sequence of instances I, a set of communication orderings O, a set of non-originating terms N, and a set of uniquely originating terms U. The node ordering $o(n_0, n_1)$ asserts that n_0 precedes n_1, that the event at n_0 is outbound, the event at n_1 is inbound, and n_0 and n_1 are on different strands. The atoms in N are assumed to be non-originating, and the atoms in U are assumed to be uniquely originating.

Members of the set of communication orderings O relate nodes in differing strands. There is an implied ordering of nodes within the same strand. Strand succession orderings of the form $\mathsf{o}(\mathsf{n}(s,p-1), \mathsf{n}(s,p))$, where $0 < p < h$ and h is the height of strand s are implicit, and must not be in O.

Associated with each preskeleton k is a graph. The vertices of the graph are the nodes of the instance sequence I, and the edges are the reverse of the both communication ordering O and the implied strand succession orderings. The edges are reversed because events in a node's past are of interest when analyzing a node. When the graph is acyclic, the transitive asymmetric relation $\prec_k$ of k is the transitive closure of the graph, and $n_0 \prec_k n_1$ asserts that the message event at n_0 precedes the one at n_1. (A preskeleton with a graph that contains cycles is not well-formed.)

To be well-formed, in addition to the requirements on communication orderings listed above, a preskeleton must satisfy the same conditions listed for a preskeleton in the specification.

A Needham-Schroeder shape in traditional Strand Space notation is in Figure 3.3, and its representation using order-sorted terms is given in Figure 3.4.

With $A, A', A'', B, B', B'', C$: **name**, $N_1, N_1', N_1'', N_2, N_2', N_2''$: **text**:

$$
\begin{array}{l}
resp = \mathsf{r}(resp_t, \langle\rangle, \langle\rangle) \text{ where} \\
resp_t = \langle -\{\!|N_1, A|\!\}_{K_B}, +\{\!|N_1, N_2|\!\}_{K_A}, -\{\!|N_2|\!\}_{K_B} \rangle
\end{array}
$$

$$
\begin{array}{l}
init = \mathsf{r}(init_t, \langle\rangle, \langle\rangle) \text{ where} \\
init_t = \langle +\{\!|N_1', A'|\!\}_{K_{B'}}, -\{\!|N_1', N_2'|\!\}_{K_{A'}}, +\{\!|N_2'|\!\}_{K_{B'}} \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathsf{k}(\langle resp, init\rangle, I, O, \langle K_{A''}^{-1}\rangle, \langle N_2''\rangle) \text{ where} \\
\quad I = \langle \mathsf{i}(resp, 3, E), \mathsf{i}(init, 3, E')\rangle \\
\quad E = \langle \mathsf{m}(A, A''), \mathsf{m}(B, B''), \mathsf{m}(N_1, N_1''), \mathsf{m}(N_2, N_2'')\rangle \\
\quad E' = \langle \mathsf{m}(A', A''), \mathsf{m}(B', C), \mathsf{m}(N_1', N_1''), \mathsf{m}(N_2', N_2'')\rangle \\
\quad O = \langle \mathsf{o}(\mathsf{n}(0,1), \mathsf{n}(1,1)), \mathsf{o}(\mathsf{n}(1,2), \mathsf{n}(0,2))\rangle
\end{array}
$$

Figure 3.4: Needham-Schroeder Preskeleton

3.2.1 Preskeleton S-Expression Syntax

The `defskeleton` form in Table A.1, Appendix A is used to specify a preskeleton in S-expression syntax. (With the exception of the initial problem statement, a preskeleton is always a skeleton.) On output, a preskeleton Referring to Table A.1, the ID in the preskeleton form names a protocol. It refers to the most recent protocol definition of that name which precedes the preskeleton form in the input. The ID in the `defstrand` form names a role. The integer in the strand form gives the height of the strand. The sequence of pairs of terms in the strand form specify an environment used to construct the events in a strand from its role's trace. The first term is interpreted using the role's variables and the second term uses the preskeleton's variables. The environment used to produce the strand's trace is derived by matching the second term using the first term as a pattern. The `deflistener` form creates an instance of a listener role for the given term.

The `precedes` form specifies members of the node relation. The first integer in a node identifies the strand using the order in which strands are defined in the `defskeleton` form.

A variable may occur in more then one role within a protocol. The reader performs a renaming so as to ensure these occurrences do not overlap. Furthermore, the maplets used to specify a strand need not specify how to map every role variable. The reader inserts missing mappings, and renames

```
(defskeleton ns (vars (n2 n1 text) (a b b-0 name))
  (defstrand resp 3 (n2 n2) (n1 n1) (b b) (a a))
  (defstrand init 3 (n1 n1) (n2 n2) (a a) (b b-0))
  (precedes ((0 1) (1 1)) ((1 2) (0 2)))
  (non-orig (privk a))
  (uniq-orig n2))
```

Figure 3.5: Needham-Schroeder `defskeleton`

every preskeleton variable that also occurs in a role of its protocol. The sort of every preskeleton variable that occurs in the `non-orig` or `uniq-orig` list or in a maplet must be declared, using the `vars` form.

Needham-Schroeder shape in S-expression syntax is displayed in Figure 3.5. The effect of reader renaming is shown in Figure 3.4 by adding primes to variables.

The PROT-ALIST, ROLE-ALIST, and SKEL-ALIST productions in Table A.1 are Lisp style association lists, that is, lists of key-value pairs, where every key is a symbol. Key-value pairs with unrecognized keys are ignored, and are available for use by other tools. On output, unrecognized key-value pairs are preserved when printing protocols, but elided when printing preskeletons.

See the CPSA Primer for more examples of CPSA external syntax.

3.3 Source Code

Protocols, roles, and preskeletons have additional fields not documented here. For example, each role and preskeleton keeps track of the variable set used to generate its message algebra, and the variable generator used to create fresh variables. There is a name associated with each protocol and role. Preskeletons have addition fields associated with generalization, as discussed in Section 4.3.

3.4 Specification

There is a direct relationship between the protocols and roles in this specification, and protocols and roles as members of a CPSA algebra (Definition 3.1).

The relationship between preskeletons in the two documents requires explanation. There are two significant differences.

In this document and in the code, a strand is represented by the instance $\mathsf{i}(r, h, E)$, but in the specification, a strand is represented by a trace and a role. These two representations are mathematically equivalent, as the trace of a strand can be computed from r, h, and E. The code adds the trace to the fields of an instance.

The second significant difference is that the precedes relation is represented by a list of ordered pairs in this document, but in the specification, the precedes relation is always asymmetric and transitive. In the code, the transitive reduction is performed on the precedes relation, and the output always contains the fewest number of ordered pairs possible without changing the transitive closure of the precedes relation.

Chapter 4

Reductions

This chapter describes the implementation-oriented refinements made to support reduction that are considered too detailed to be included in the specification.

In the CPSA implementation and the design formalism, every preskeleton includes a link to its protocol. Two preskeletons are not related by a homomorphism unless they specify the same protocol.

For each preskeleton k, the implementation maintains an *origination map*, $\mathcal{O}(k, t)$. It maps each of the preskeleton's uniquely originating terms to the set of nodes at which it originates. For hulled preskeletons, the range of this map must contain singleton sets or the empty set. The origination map returns an error indicator when given a term not assumed to be uniquely originating, a feature used to check the implementation's consistency.

Each preskeleton contains the state of a variable generator. It's used to ensure a source of fresh variables for any preskeleton derived from it.

4.1 Preskeleton Reductions

Given a well-formed preskeleton, an attempt is made to convert it into a set of skeletons. This section describes a few implementation details omitted from the specification.

The implementation uses sequences to represent some sets. The function *nub* removes duplicates from a sequence.

4.1.1 Substitution

The function $\mathbb{S}_\sigma$ applies the order-sorted substitution σ to a preskeleton.

$$\begin{aligned}\mathbb{S}_\sigma(\mathsf{k}(P,I,O,N,U)) &= \mathsf{k}(P,\mathbb{S}_\sigma \circ I, O, nub(\sigma \circ N), nub(\sigma \circ U))\\ \mathbb{S}_\sigma(\mathsf{i}(r,h,E)) &= \mathsf{i}(r,h,\mathbb{S}_\sigma \circ E)\\ \mathbb{S}_\sigma(\mathsf{m}(x,y)) &= \mathsf{m}(x,\sigma(y))\end{aligned}$$

The substitution specifies a homomorphism as long as it preserves the nodes at which each uniquely originating term originates. In other words, its a homomorphism only if for each uniquely originating term t in k, $\mathcal{O}(k,t) \subseteq \mathcal{O}(k',\sigma(t))$, where $k' = \mathbb{S}_\sigma(k)$. The implicit homomorphism is $(\phi_{\mathrm{id}},\sigma)$, where ϕ_{id} is the identity strand map for k.

4.1.2 Compression

The function $\mathbb{C}_{s,s'}$ compresses s into s' in a preskeleton.

$$\begin{aligned}\mathbb{C}_{s,s'}(\mathsf{k}(P,I,O,N,U)) &= \mathsf{k}(P,I \circ \phi'_s, \mathbb{C}_\phi \circ O, N, U)\\ \mathbb{C}_\phi(\mathsf{o}(n_0,n_1)) &= \mathsf{o}(\mathbb{C}_\phi(n_0),\mathbb{C}_\phi(n_1))\\ \mathbb{C}_\phi(\mathsf{n}(s,p)) &= \mathsf{n}(\phi(s),p)\\ \phi(j) = \phi_{s,s'}(j) &= \begin{cases}\phi_s(s') & \text{if } j = s\\ \phi_s(j) & \text{otherwise}\end{cases}\\ \phi_s(j) &= \begin{cases}j-1 & \text{if } j > s\\ j & \text{otherwise}\end{cases}\\ \phi'_s(j) &= \begin{cases}j+1 & \text{if } j \geq s\\ j & \text{otherwise}\end{cases}\end{aligned}$$

where the trace of $I(s)$ is a prefix of the trace of $I(s')$. Although not shown, orderings of the form $\mathsf{o}(\mathsf{n}(s,p),\mathsf{n}(s,p'))$ are removed from the ordering when $p < p'$, so they do not cause the output preskeleton to fail to be well-formed. The implicit homomorphism is $(\phi_{s,s'},\sigma_{\mathrm{id}})$, where σ_{id} is the identity substitution. Note that $\phi_{s,s'} \circ \phi'_s = \sigma_{\mathrm{id}}$.

4.1.3 Order Enrichment

The function $\mathbb{O}$ performs order enrichment by adding a node orderings between the node at which a uniquely originating atom originates and the nodes at which it is gained. A message is *gained* by a trace if it is carried by some

event and the first event in which it is carried is inbound. Let $\mathcal{G}(k,t)$ be the set of nodes in k at which t is gained.

$$\mathbb{O}(\mathsf{k}(P,I,O,N,U)) = \mathsf{k}(P,I,nub(O \frown O'),N,U)$$

where $O' = \{\mathsf{o}(n_0,n_1) \mid t \in U, n_0 \in \mathcal{O}(k,t), n_1 \in \mathcal{G}(k,t)\}$. Order Enrichment specifies a homomorphism that is an embedding.

4.1.4 Transitive Reduction

The function $\mathbb{R}$ performs a transitive reduction on a preskeleton's node relation. The transitive reduction of an ordering is the minimal ordering such that both orderings have the same transitive closure. Here, communication orderings implied by transitive closure are removed.

The transitive reduction of a skeleton is isomorphic to the skeleton. The reduction is performed to speed up the code that checks for isomorphisms. When two skeletons are transitively reduced, and isomorphic, they have the same number of communication orderings.

In the implementation, transitive reduction is the last operation applied to a preskeleton during the process of converting a preskeleton into a pruned skeleton. Isomorphism testing is only performed on pruned skeletons.

4.2 Augmentation

The function $\mathbb{A}_{i,n}$ augments a preskeleton with a new strand. It appends the instance i to the sequence of instances, adds a node ordering, and adds atoms as specified by the role. The function orders the last node in the strand before some node in the preskeleton.

$$\begin{aligned}
\mathbb{A}_{i,n}(\mathsf{k}(P,I,O,N,U)) &= \mathsf{k}(P, I \frown \langle i \rangle, \langle \mathsf{o}(\mathsf{n}(|I|, h-1), n) \rangle \frown O, N_1, U_1) \\
i &= \mathsf{i}(\mathsf{r}(C, N_0, U_0), h, E) \\
N_1 &= nub(N \frown \sigma_e \circ N_0) \\
U_1 &= nub(U \frown \sigma_e \circ U_0)
\end{aligned}$$

Although not shown, elements in N_1 that contain a variable that does not occur in some event in the constructed preskeleton are dropped, as are elements in U_1 that are not carried in some event in the constructed preskeleton. As with the preskeleton reductions in Section 4.1, it is straightforward to derive the homomorphism associated with a successful augmentation.

4.3 Generalization

The generalization process is the only part of the algorithm that requires non-isomorphic homomorphisms to be explicitly represented. To make this possible, every preskeleton contains two additional fields not yet described, a link to the point-of-view skeleton, and a POV strand map. The POV strand map is the second component of the homomorphism from the point-of-view skeleton to the preskeleton. The first component can be generated via matching using the point-of-view skeleton.

The POV strand map is maintained by every reduction applied to a preskeleton. The compression reduction is the only one that requires careful thought.

Chapter 5

Search Strategy

The top-level loop maintains two sequences of skeletons, a to do list, and a set of skeletons that have already been seen. When the to do list is empty, the loop exits.

For each problem statement, CPSA attempts the convert the preskeleton into a skeleton. If the conversion fails, an error is signaled. Otherwise, a search is started with the skeleton as the POV skeleton of Section 4.3. The top-level loop starts with the POV skeleton as the single member of the to do list and the seen set.

The following steps constitute one iteration of the top-level loop. The first skeleton on the to do list is removed and is the subject of the iteration. If the skeleton is unrealized, contraction and augmentation are used to compute its cohort. Otherwise, generalization reductions are tried in an effort to make one generalization step. The generated skeletons are the subject's children. Each child that is isomorphic to a member of the seen set is dropped. The other children are added to the seen set and to the end of the to do list. The final step in the iteration is to print the subject skeleton.

When the subject in unrealized, the test node is selected as follows. The strands are considered in reverse order, and the first unrealized node in one of the strands is used as the test node. By default, the program tries encryptions for critical messages before it tries atoms. The encryptions considered as critical messages are obtained using the *encryptions* function in the algebra interface.

The search order can be modified by using command-line arguments or the `herald` form. The option `--check-nonces` cause CPSA to try atoms before encryptions. Option `--try-old-strands` tries the strands in order. Option `--reverse-nodes` tries nodes later in the strand first.

Chapter 6

Haskell Modules

An overview of the modules that make up a `cpsa` program follows.

CPSA.Lib.Utilities: Contains generic list functions and a function that determines if a graph has a cycle.

CPSA.Lib.SExpr: A data structure for S-expressions, the ones that are called proper lists.

CPSA.Lib.Pretty: A simple pretty printer.

CPSA.Lib.Printer: A CPSA specific pretty printer using S-expressions.

CPSA.Lib.Notation: A CPSA specific pretty printer using infix notation.

CPSA.Lib.Algebra: Defines the interface to CPSA algebras.

CPSA.Lib.CPSA: Exports the types and functions used to construct an implementation of an algebra.

CPSA.Basic.Algebra: Basic Crypto Algebra implementation.

CPSA.Lib.Entry: Provides a common entry point for programs based on the CPSA library.

CPSA.Lib.Protocol: Protocol data structures.

CPSA.Lib.Strand: Instance and preskeleton data structures and support functions.

CPSA.Lib.Cohort: Computes the cohort associated with a skeleton or its generalization.

CPSA.Lib.Reduction: Term reduction for the CPSA solver.

CPSA.Lib.Expand: Expands macros using definitions in the input.

CPSA.Lib.Loader: Loads protocols and preskeletons from S-expressions.

CPSA.Lib.Displayer: Displays protocols and preskeletons as S-expressions.

Chapter 7

Visualization

This section describes the Causally Intuitive Preskeleton Layout algorithm used to generate visualizations of preskeletons. The algorithm is simple to implement and explain, and because it is designed for preskeletons, it produces better results than is available from generic graph layout algorithms, such as the ones used by Graphviz [2].

The preskeleton is prepared by performing a transitive reduction on its ordering relation. Communication edges implied by transitive closure are removed. The result is called a Hasse diagram.

The Hasse diagram is created by considering each communication edge. If there is a path from its source to its destination that does not traverse the edge, the edge is deleted.

To simplify the task, each strand is laid out vertically with early nodes above later ones. The strands are horizontally placed in the same order as they appear in the preskeleton. The spacing between successive nodes on a strand is the same, as is the horizontal spacing between strands. The vertical position of a node is called its rank.

Within this framework, the simplest layout algorithm is to use the position of the node in its strand as its rank. When using this layout strategy, the only difficultly occurs when a node ordering arrow crosses over a node in an unrelated strand. To avoid ambiguity, arrows that cross strands are curved.

When using position based ranking, the result often contains upward sloping arrows. Within a strand, no node that is after a node is above the node, but with upward sloping arrows, this property no longer holds. The motivation for the Causally Intuitive Preskeleton Layout algorithm is that

eliminating upward sloping arrows makes causal relations easier to grasp.

The layout algorithm has two phases. The first phase stretches strands so as to eliminate upwardly sloping arrows, and the second phase compresses them so as to eliminate some unnecessary stretching. Without phase two, some nodes early in a strand appear to be oddly separated from others in the strand.

Each phase starts with a to do list containing every node in the preskeleton. For each node in the to do list, if conditions are met, it updates the current node ranking and adds nodes to the to do list. The phase is finished when the to do list is empty.

Phase I starts with the position based ranking $r(s, p) = p$. Let $P(n)$ be the set the predecessors of node n, excluding the nodes on the strand of n. Let $P_r(n) = \{r(n') \mid n' \in P(n)\}$ be the ranks of $P(n)$. The stretch rule is considered for each element in the to do list.

The stretch rule applies to node n_1 if $r(n_1) < h$, where $h = \max(\{r(n_1)\} \cup P_r(n_1))$. In that case, the ranking is updated so that $r(n_1) = h$, and the linearize rule is applied to the next strand node if it exists.

The linearize rule applies to node n_1 if $r(n_1) \leq r(n_0)$, where n_0 is the previous strand node. In that case, the ranking is updated so that $r(n_1) = r(n_0)+1$, the to do list is augmented with elements in $S(n_1)$, and the linearize rule is applied to the next strand node if it exists, where $S(n)$ is the set the successors of node n, excluding the nodes on the strand of n.

In phase II, the compress rule is considered for each element in the to do list. It applies to node n_1 with a next strand node of n_2 if $r(n_1) < h$, where $h = \min(\{r(n_2) - 1)\} \cup S_r(n_1))$ and $S_r(n)$ is the ranks of $S(n)$. In that case, the ranking is updated so that $r(n_1) = h$, and the to do list is augmented with elements in $P(n_1)$ and the previous strand node of n_1 if it exists.

Acknowledgement

Carolyn Talcott provided valuable feedback on drafts of this document.

Appendix A

Basic Crypto Algebra Syntax Reference

The complete syntax for the analyzer using the Basic Crypto Algebra is shown in Table A.1. The start grammar symbol is FILE, and the terminal grammar symbols are: (,), SYMBOL, STRING, INTEGER, and the constants set in typewriter font.

The ALIST, PROT-ALIST, ROLE-ALIST, and SKEL-ALIST productions are Lisp style association lists, that is, lists of key-value pairs, where every key is a symbol. Key-value pairs with unrecognized keys are ignored, and are available for use by other tools. On output, unrecognized key-value pairs are preserved when printing protocols, but elided when printing skeletons.

The contents of a file can be interpreted as a sequence of S-expressions. The S-expressions used are restricted so that most dialects of Lisp can read them, and characters within symbols and strings never need quoting. Every list is proper. An S-expression atom is either a SYMBOL, an INTEGER, or a STRING. The characters that make up a symbol are the letters, the digits, and the special characters in "`-*/<=>!?:$%_&~^+`". A symbol may not begin with a digit or a sign followed by a digit. The characters that make up a string are the printing characters omitting double quote and backslash. Double quotes delimit a string. A comment begins with a semicolon, or is an S-expression list at top-level that starts with the `comment` symbol.

FILE	←	HERALD? FORM+
HERALD	←	(`herald` TITLE ALIST)
TITLE	←	SYMBOL | STRING
FORM	←	COMMENT | PROTOCOL | SKELETON
COMMENT	←	(`comment` ...)
PROTOCOL	←	(`defprotocol` ID ALG ROLE+ PROT-ALIST)
ID	←	SYMBOL
ALG	←	SYMBOL
ROLE	←	(`defrole` ID VARS TRACE ROLE-ALIST)
VARS	←	(`vars` DECL∗)
DECL	←	(ID+ SORT)
SORT	←	`text` | `data` | `name` | `skey` | `akey` | `mesg`
TRACE	←	(`trace` EVENT+)
EVENT	←	(`send` TERM) | (`recv` TERM)
TERM	←	ID | (`pubk` ID) | (`privk` ID) | (`invk` ID)
	|	(`pubk` ID STRING) | (`privk` ID STRING)
	|	(`ltk` ID ID) | STRING | (`cat` TERM+)
	|	(`enc` TERM+ TERM) | (`hash` TERM+)
ROLE-ALIST	←	(`non-orig` HT-TERM∗) ROLE-ALIST
	|	(`uniq-orig` TERM∗) ROLE-ALIST | ...
HT-TERM	←	TERM | (INTEGER TERM)
PROT-ALIST	←	...
SKELETON	←	(`defskeleton` ID VARS STRAND+ SKEL-ALIST)
STRAND	←	(`defstrand` ID INTEGER MAPLET∗)
	|	(`deflistener` TERM)
MAPLET	←	(TERM TERM)
SKEL-ALIST	←	(`non-orig` TERM∗) SKEL-ALIST
	|	(`uniq-orig` TERM∗) SKEL-ALIST
	|	(`precedes` NODE-PAIR∗) SKEL-ALIST | ...
NODE-PAIR	←	(NODE NODE)
NODE	←	(INTEGER INTEGER)

Table A.1: CPSA Syntax

Appendix B

The Basic Crypto Many-Sorted Algebra

The implementation uses a many-sorted algebra. The many-sorted message algebra described here is a reduction of the order-sorted message algebra in Table 2.1 using the method described in [3, Section 4]. The order-sorted message signature is reproduced in Table B.1 in a form that uses prefix notation for every term formed using an operation. In the related many-sorted signature in Table B.2, the inclusion function symbols are text, data, name, skey, and akey. Section 4 of the paper describes the sense in which algebras that model the many-sorted signature are essentially the same as the ones that model the order-sorted message signature.

Terms are constructed from a set I of identifiers and a set of functions symbols. The symbols of arity one are text, data, name, skey, akey, pubk,

Base sort symbols: **name**, **text**, **data**, **skey**, **akey**
Non-base sort symbol: **mesg**

Subsorts: **name**, **text**, **data**, **akey**, **skey** $<$ **mesg**

$\mathsf{pubk}\colon \mathbf{name} \to \mathbf{akey} \quad \mathsf{invk}\colon \mathbf{akey} \to \mathbf{akey} \quad \mathsf{ltk}\colon \mathbf{name} \times \mathbf{name} \to \mathbf{skey}$
$\mathsf{enc}\colon \mathbf{mesg} \times \mathbf{mesg} \to \mathbf{mesg} \quad \mathsf{hash}\colon \mathbf{mesg} \to \mathbf{mesg}$
$\mathsf{cat}\colon \mathbf{mesg} \times \mathbf{mesg} \to \mathbf{mesg} \quad C_i\colon \mathbf{mesg}$

Axiom: $\mathsf{invk}(\mathsf{invk}(x)) \approx x$ for $x\colon \mathbf{akey}$

Table B.1: Basic Crypto Order-Sorted Signature

Sort symbols: **name**, **text**, **data**, **skey**, **akey**, and **mesg**

$\mathsf{pubk}\colon \mathbf{name} \to \mathbf{akey} \quad \mathsf{invk}\colon \mathbf{akey} \to \mathbf{akey} \quad \mathsf{ltk}\colon \mathbf{name} \times \mathbf{name} \to \mathbf{skey}$
$\mathsf{enc}\colon \mathbf{mesg} \times \mathbf{mesg} \to \mathbf{mesg} \quad \mathsf{hash}\colon \mathbf{mesg} \to \mathbf{mesg}$
$\mathsf{cat}\colon \mathbf{mesg} \times \mathbf{mesg} \to \mathbf{mesg} \quad C_i\colon \mathbf{mesg}$
$\mathsf{name}\colon \mathbf{name} \to \mathbf{mesg} \quad \mathsf{text}\colon \mathbf{text} \to \mathbf{mesg} \quad \mathsf{data}\colon \mathbf{data} \to \mathbf{mesg}$
$\mathsf{skey}\colon \mathbf{skey} \to \mathbf{mesg} \quad \mathsf{akey}\colon \mathbf{skey} \to \mathbf{mesg}$

Axiom: $\mathsf{invk}(\mathsf{invk}(x)) \approx x$ for $x\colon \mathbf{akey}$

Table B.2: Basic Crypto Many-Sorted Signature

invk, and hash. The symbols of arity two are ltk, cat, and enc. The signature is given in Table B.2. Grammar rules define the terms used by this algebra.

The set of asymmetric keys K is defined as follows.

$$K \leftarrow I \mid \mathsf{pubk}(I) \mid \mathsf{invk}(I) \mid \mathsf{invk}(\mathsf{pubk}(I))$$

The key $\mathsf{invk}(x)$ is the inverse of the asymmetric key x, and $\mathsf{pubk}(x)$ is principal x's public key.

Each occurrence of an identifier in a term is associated with a sort symbol. The context in which an identifier occurs determines the sort. The sort symbols are **text**, **data**, **name**, **akey**, **skey**, and **mesg**, where a name refers to a principal. An identifier occurrence in an asymmetric key of the form $\mathsf{pubk}(x)$ has sort **name**, otherwise it has sort **akey**.

The set of atoms B is defined as follows.

$$B \leftarrow \mathsf{text}(I) \mid \mathsf{data}(I) \mid \mathsf{name}(I) \mid \mathsf{skey}(I) \mid \mathsf{skey}(\mathsf{ltk}(I, I)) \mid \mathsf{akey}(K)$$

The atom $\mathsf{skey}(\mathsf{ltk}(x, y))$ is a symmetric, long term key shared between two principals x, and y. The occurrence of x in $\mathsf{text}(x)$ has sort **text**, sort **data** for $\mathsf{data}(x)$, sort **name** for $\mathsf{name}(x)$, and sort **skey** for $\mathsf{skey}(x)$. The occurrences of x and y in $\mathsf{skey}(\mathsf{ltk}(x, y))$ both have sort **name**.

The set of terms T is defined as follows.

$$T \leftarrow I \mid B \mid Q \mid \mathsf{cat}(T, T) \mid \mathsf{enc}(T, T) \mid \mathsf{hash}(T)$$

where Q is the set of tags, represented by quoted string literals. The second argument in enc is a term for a key. A term of the form x is called an indeterminate, and the identifier occurrence has sort **mesg**.

The terms of interest are well-formed. A term is *well-formed* if every occurrence of each identifier has the same sort. An example of a non-well-formed term is $\mathsf{cat}(x, \mathsf{akey}(x))$ because the identifier x occurs with two sorts, **mesg** and **akey**. A pair of well-formed terms are *compatible* if every identifier that occurs in both terms occurs with the same sort.

A term is a variable if it specifies an identifier and its sort.

$$V \leftarrow I \mid \mathsf{text}(I) \mid \mathsf{data}(I) \mid \mathsf{name}(I) \mid \mathsf{skey}(I) \mid \mathsf{akey}(I)$$

When a term is well-formed, the same variable is associated every occurrence of an identifier in a term.

There are efficient ways of implementing unification for this algebra because there are efficient ways for implementing for unification in equational theories representable by a convergent term rewrite system [1]. As long as terms are compatible, substitutions produced by unifiers map an identifier that occurs with a given sort to a term of the same sort. Depending on the sort symbol, a substitution is limited to the following forms:

mesg	$I \mapsto I$	$I \mapsto B$	$I \mapsto \mathsf{cat}(T, T)$	$I \mapsto \mathsf{enc}(T, T) \ldots$
akey	$I \mapsto I$	$I \mapsto \mathsf{pubk}(I)$	$I \mapsto \mathsf{invk}(I)$	$I \mapsto \mathsf{invk}(\mathsf{pubk}(I))$
skey	$I \mapsto I$	$I \mapsto \mathsf{ltk}(I, I)$		
text	$I \mapsto I$			
data	$I \mapsto I$			
name	$I \mapsto I$			

The current implementation uses an algorithm for unification without equations described by Laurence Paulson [5, Page 381] with modifications to the unification functions as shown in Figure B.1, where $e :: \ell$ means $\langle e \rangle \frown \ell$. The function unify calls unify_aux in the unmodified version.

$$\mathsf{unify}(\ell, t, t') = \mathsf{unify\_aux}(\ell, \mathsf{chase}(\ell, t), \mathsf{chase}(\ell, t'))$$

$$\begin{array}{l}
\mathsf{chase}(\ell, x) = \\
\quad \text{let } t = \mathsf{lookup}(x, \ell) \text{ in} \\
\quad \text{if } x = t \text{ then } x \text{ else } \mathsf{chase}(\ell, t) \\
\mathsf{chase}(\ell, \mathsf{invk}(t)) = \mathsf{chase_invk}(\ell, t) \quad (!) \\
\mathsf{chase}(\ell, t) = t
\end{array}$$

$$\begin{array}{ll}
\mathsf{chase_invk}(\ell, x) = & (!) \\
\quad \text{let } t = \mathsf{lookup}(x, \ell) \text{ in} & (!) \\
\quad \text{if } x = t \text{ then } \mathsf{invk}(x) \text{ else } \mathsf{chase_invk}(\ell, t) & (!) \\
\mathsf{chase_invk}(\ell, \mathsf{invk}(t)) = \mathsf{chase}(\ell, t) & (!) \\
\mathsf{chase_invk}(\ell, t) = \mathsf{invk}(t) & (!)
\end{array}$$

$$\begin{array}{l}
\mathsf{lookup}(x, \langle\rangle) = x \\
\mathsf{lookup}(x, (y, t) :: \ell) = \text{if } x = y \text{ then } t \text{ else } \mathsf{lookup}(x, \ell)
\end{array}$$

$$\begin{array}{ll}
\mathsf{unify_aux}(\ell, x, x) = \ell & \\
\mathsf{unify_aux}(\ell, x, t) = \text{if } \mathsf{occurs}(x, t) \text{ then raise failure else } (x, t) :: \ell & \\
\mathsf{unify_aux}(\ell, t, x) = \mathsf{unify_aux}(\ell, x, t) & \\
\mathsf{unify_aux}(\ell, \mathsf{invk}(x), \mathsf{pubk}(y)) = \mathsf{unify_aux}(\ell, x, \mathsf{invk}(\mathsf{pubk}(y))) & (!) \\
\mathsf{unify_aux}(\ell, \mathsf{pubk}(x), \mathsf{invk}(y)) = \mathsf{unify_aux}(\ell, y, \mathsf{invk}(\mathsf{pubk}(x))) & (!) \\
\mathsf{unify_aux}(\ell, f(t, \ldots), f(t', \ldots)) = \mathsf{unify_list}(\ell, \langle t, \ldots\rangle, \langle t', \ldots\rangle) & \\
\mathsf{unify_aux}(\ell, t, t') = \text{raise failure} &
\end{array}$$

$$\begin{array}{l}
\mathsf{unify_list}(\ell, \langle\rangle, \langle\rangle) = \ell \\
\mathsf{unify_list}(\ell, t :: u, t' :: u') = \mathsf{unify_list}(\mathsf{unify}(\ell, t, t'), u, u') \\
\mathsf{unify_list}(\ell, u, u') = \text{raise failure}
\end{array}$$

Figure B.1: Unifier

$$
\begin{array}{l}
\mathsf{match}(\ell, x, t) = \\
\quad \text{if } \neg\, \mathsf{bound}(x, \ell) \text{ then } (x, t) :: \ell \\
\quad \text{else if } \mathsf{lookup}(x, \ell) = t \text{ then } \ell \\
\quad \text{else raise failure} \\
\mathsf{match}(\ell, f(t, \ldots), f(t', \ldots)) = \mathsf{match\_list}(\ell, \langle t, \ldots \rangle, \langle t', \ldots \rangle) \\
\mathsf{match}(\ell, \mathsf{invk}(t), t') = \mathsf{match}(\ell, t, \mathsf{invk}(t')) \qquad (!) \\
\mathsf{match}(\ell, t, t') = \text{raise failure}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{bound}(x, \langle\rangle) = \mathsf{false} \\
\mathsf{bound}(x, (y, t) :: \ell) = x = y \text{ or } \mathsf{bound}(x, \ell)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{match\_list}(\ell, \langle\rangle, \langle\rangle) = \ell \\
\mathsf{match\_list}(\ell, t :: u, t' :: u') = \mathsf{match\_list}(\mathsf{match}(\ell, t, t'), u, u') \\
\mathsf{match\_list}(\ell, u, u') = \text{raise failure}
\end{array}
$$

Figure B.2: Matcher

Appendix C

Diffie-Hellman

Table C.1 contains the signature used for Diffie-Hellman analysis. In this algebra, there is an equation for the commutativity of exponents, but there is no equation for associativity. Therefore, this algebra cannot be used to analyze group Diffie-Hellman protocols.

The equation for exponentiation is $(h^x)^y \approx (h^y)^x$ for $h\colon$ **base** and $x, y\colon$ **expn**. Unification and matching in this algebra produce a finite number of most general unifiers, that is, the unification type is finitary. For example, $\{x \mapsto u, y \mapsto v\}$ and $\{x \mapsto G^v, u \mapsto G^y\}$ and unify x^y and u^v. Unification is finitary because of an approximation. To unify a^{xy} and b^{wz}, we unify a^{xy} with G^{uv} and G^{vu} with b^{wz}, where u and v are freshly generated variables. In other words, for the purpose of unification only, the equation for exponentiation is $(G^x)^y \approx (G^y)^x$.

Base sort symbols: **name**, **text**, **data**, **skey**, **akey**, **expn**
Non-base sort symbols: **mesg**, **base**

Subsorts: **name**, **text**, **data**, **akey**, **skey**, **expn**, **base** $<$ **mesg**

$\{\!\lvert \cdot \rvert\!\}_{(\cdot)}\colon$ **mesg** $\times$ **mesg** $\to$ **mesg**	Encryption
$\#\colon$ **mesg** $\to$ **mesg**	Hashing
$(\cdot,\cdot)\colon$ **mesg** $\times$ **mesg** $\to$ **mesg**	Pairing
"$\ldots$"$\colon$ **mesg**	Tag constants
$K_{(\cdot)}\colon$ **name** $\to$ **akey**	Public key of name
$(\cdot)^{-1}\colon$ **akey** $\to$ **akey**	Inverse of asymmetric key
$\mathsf{ltk}\colon$ **name** $\times$ **name** $\to$ **skey**	Long term shared key
$G\colon$ **base**	Generator constant
$(\cdot)^{(\cdot)}\colon$ **base** $\times$ **expn** $\to$ **base**	Exponentiation

Axioms: $(x^{-1})^{-1} \approx x$ for $x\colon$ **akey**
$(h^x)^y \approx (h^y)^x$ for $h\colon$ **base** and $x, y\colon$ **expn**

Table C.1: Diffie-Hellman Signature

Bibliography

[1] M. Fay. First-order unification in an equational theory. In *Proc. 4th Workshop on Automated Deduction*, 1979.

[2] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.

[3] Joseph A. Goguen and Jose Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992.

[4] Moses D. Liskov, Paul D. Rowe, and F. Javier Thayer. Completeness of CPSA. Technical Report MTR110479, The MITRE Corporation, 2011.

[5] Laurence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[6] John D. Ramsdell and Joshua D. Guttman. *CPSA Primer*. The MITRE Corporation, 2009. In `http://hackage.haskell.org/package/cpsa` source distribution, `doc` directory.

[7] John D. Ramsdell, Joshua D. Guttman, Moses D. Liskov, and Paul D. Rowe. *The CPSA Specification: A Reduction System for Searching for Shapes in Cryptographic Protocols*. The MITRE Corporation, 2009. In `http://hackage.haskell.org/package/cpsa` source distribution, `doc` directory.

Index