CPSA Primer

John D. Ramsdell Joshua D. Guttman
The MITRE Corporation
CPSA Version 2.2.9

May 24, 2012

Analyzing a cryptographic protocol means finding out what security properties—essentially, authentication and secrecy properties—are true in all its possible executions. Protocol analysis is hard because an adversary can often manipulate the regular, law-abiding participants. The adversary may be able to manipulate the regular participants into an unexpected execution, breaking a secrecy or authentication property that the protocol was intended to ensure.

CPSA, The Cryptographic Protocol Shapes Analyzer, is a software tool. Given a protocol definition and some assumptions about executions, it attempts to produce descriptions of all possible executions of the protocol compatible with the assumptions. Naturally, there are infinitely many possible executions of a useful protocol, since different participants can run it with varying parameters, and the participants can run it repeatedly.

However, for many naturally occurring protocols, there are only finitely many of these runs that are essentially different. Indeed, there are frequently very few, often just one or two, even in cases where the protocol is flawed. We call these essentially different executions the *shapes* of the protocol. Authentication and secrecy properties are easy to "read off" from the shapes, as are attacks and anomalies.

The purpose of this document is to provide the background required to

© 2010 The MITRE Corporation. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, this copyright notice and the title of the publication and its date appear, and notice in given that copying is by permission of The MITRE Corporation.

make effective use of a CPSA software distribution. In particular, the advice in Section 12 is essential reading.

The CPSA program reads a sequence of problem descriptions, and prints the steps it used to solve each problem. Each input problem contains some initial behavior, together with assumptions about some uncompromised keys and freshly chosen values. CPSA discovers what shapes are compatible with this problem description. Normally, the initial behavior is a local run of one participant, so that the problem is to see what possible executions exist from that participant's "point of view." The analysis reveals what the other participants must have done, given the first participant's view.

The shapes analysis is performed within a pure Dolev-Yao model [3]. CPSA's search is based on a high-level algorithm that claims to be complete, i.e. every shape can in fact be found in a finite number of steps [2]. CPSA's search has not been shown to be complete, a deficiency we are committed to repair.

1 Overview

A CPSA release includes several programs, an analyzer, and various tools used to interpret the results. The analyzer, `cpsa`, provides support for several algebras, one of which is the Basic Crypto Algebra. Programs that assist in the interpretation of results are `cpsashape` and `cpsagraph`. The analyzer prints the steps it used to solve each problem. The `cpsashapes` program extracts the shapes discovered by an analyzer run. The `cpsagraph` program graphs both forms of output using Scalable Vector Graphics (SVG). A standards-compliant browser such as FireFox or Safari displays the generated diagrams.

The expected work flow follows. An analysis problem is entered using an ordinary text editor, preferably one with support for Lisp syntax. Problem statement errors in the input are detected by running the analyzer. Many error reports are of the form that allow editors such as Emacs to move its cursor to location of the problem.

There are two classes of problem statement errors: syntax and semantic errors. Correcting syntax errors is straightforward, but correcting semantic errors requires an understanding of the core data structures. Section 9.1 describes their correction.

Once the problem statement errors have been eliminated, the analyzer should produce useful output as a text document. The text document con-

tains each step used to derive a shape from a problem statement. It is common to filter the output using the `cpsashapes` program, and look only at the computed shapes associated with each problem statement.

The `cpsagraph` program is applied to the output to produce a more readable, hyperlinked XHTML document that can be displayed in a standards-compliant web browser. The CPSA User Guide contains the up-to-date description of `cpsagraph` generated documents. The guide is also the place to find command-line usage information for all programs in a release. The user guide is an XHTML document delivered with the software.

The `cpsa` program uses S-expressions for both input and output. S-expression is an abbreviation for a Symbolic Expression of Lisp fame, and is described in Appendix A.

The input may optionally start with a `herald` form. The form contains a title for the run and an association list. The association list allows options normally specified on the command line to be specified within an input file. In the following example, the herald form specifies a strand bound of 12 in a way that is equivalent to the command line option `--bound=12`.

```
(herald Needham-Schroeder (bound 12))
```

The body of the input consists of two forms: protocol definitions and initial behavior descriptions. The exact details of both forms depend on the message algebra specified by the protocol. Protocols that specify `basic` as their algebra get an implementation of the Basic Crypto Algebra (BCA) described in the next section. A complete grammar for `cpsa` input with BCA protocols is displayed in Table 1 on Page 27.

2 BCA Messages

Each message exchanged in a protocol is represented by a term. Terms represent atomic values such text objects, principal names, and asymmetric and symmetric keys. They also represent values composed from other values via encryption and concatenation.

A sort system is used to classify terms. A CPSA message algebra is an order-sorted algebra [4] with restrictions, hence the use of the word sort instead of type. Other aspects of CPSA's use of order-sorted algebras is beyond the scope of this paper.

The sorts that correspond to the atomic values are the base sorts—for BCA they are *text*, *data*, *name*, *akey*, and *skey*. The non-base sort is *mesg*. Every term is of sort *mesg*, and every one of the other sorts is a subsort of *mesg*.

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)).
```

Asymmetric keys come in pairs related by the `invk` operator. If t is a term of sort *akey*, so is `(invk` t`)`. Furthermore, `(invk (invk` t`))` is equated with t. A name can be used to identify an asymmetric key pair using the `pubk` and `privk` operators, as in `(pubk n)` and `(privk n)`—the latter is interpreted as `(invk (pubk n))`. A name may be associated with more than one key using the binary form of `pubk`, where the first argument is a quoted constant, as in `(pubk "sig" n)` and `(pubk "enc" n)`. Two names can be used to identify a long term symmetric key with the `ltk` operator.

Terms formed from base sorted variables, and the operators `invk`, `pubk`, `privk`, and `ltk` are called atoms, because each one represents an atomic value. A key property of an atom is that the receiver of an atom carried in a message term cannot decompose the atom into parts. For example, the reception of a message that consists of the atom `(invk k)` does not allow its receiver to deduce `k`. Within this document, S-expression syntax will often, for the sake of readability, be replaced by the traditional notation for terms. Thus `(invk k)` will be written as K^{-1}, and `(pubk n)` as K_N.

The terms in this algebra are freely generated from the atoms, tags, concatenation, encryption, and hashing. A tag is a quoted constant. The concatenation of terms t and t' is t, t' in traditional infix notation and `(cat` t t'`)` in S-expression syntax. The comma operator is right associative and `(cat` a b c d`)` is equivalent to `(cat` a `(cat` b `(cat` c d`)))`. Given t, the term to be protected, and key term k, the encryption of t using k is $\{|t|\}_k$ in traditional notation and `(enc` t k`)` in S-expression syntax. The term represents asymmetric encryption when the key is of sort **akey**, otherwise it represents symmetric encryption. Additionally, `(enc` a b c d k`)` is equivalent to `(enc (cat` a b c d`)` k`)`. Given t, the term to be hashed, its hash is $\#t$ in traditional notation and `(hash` t`)` in S-expression syntax. CPSA treats a hashed term as if it were an encryption in which the term that is hashed is the

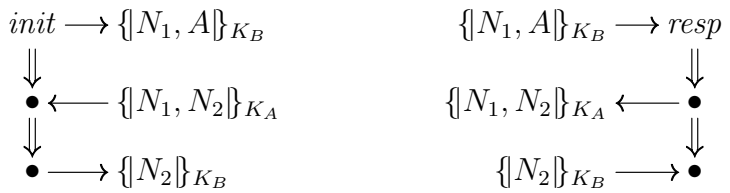


Figure 1: Needham-Schroeder Initiator and Responder Roles

encryption key. As with encryption, `(hash` *a b c d*`)` is equivalent to `hash (cat` *a b c d*`))`. Figure 2 on Page 6 contains examples of BCA message terms. Also see TERM in Table 1, Appendix A.

A message term *carries* a subterm of the message if the possession of the right set of keys allows the extraction of the subterm. The carries relation is the least relation such that (1) t carries t, (2) $\{|t_0|\}_{t_1}$ carries t if t_0 carries t, and (3) t_0, t_1 carries t if t_0 or t_1 carries t. As noted above, the message k^{-1} does not carry k. Also, $\{|t|\}_k$ does not carry k unless (anomalously) t carries k.

3 Protocols

A *protocol* defines the patterns of allowed behavior for non-adversarial participants. In other words, the behavior of each participant must be an instance of some protocol template, called a *role.* Figure 1 displays the roles that make up the Needham-Schroeder protocol.

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

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 message events, either a message reception or a transmission. An inbound message with term t is $-t$ in text and `(recv` *t*`)` in S-expression syntax. An

```
(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 2: Needham-Schroeder Responder Role

outbound term is $+t$ in text and `(send` t`)` in S-expression syntax. The Needham-Schroeder responder's role in S-expression syntax is in Figure 2.

Zero-based indexing is used though out this document and in the source code it describes. Within the document, a finite sequence is a function from an initial segment of the natural numbers. Angle brackets are used for sequence construction. Thus $\langle 3, 2, 99 \rangle = \{0 \mapsto 3, 1 \mapsto 2, 2 \mapsto 99\}$, and the responder's trace is $\langle -\{|N_1, A|\}_{K_B}, +\{|N_1, N_2|\}_{K_A}, -\{|N_2|\}_{K_B} \rangle$. The length of a sequence x is $|x|$.

A term *originates* in a trace if it is carried in some event and the first term in which it is carried is an outbound term. A term is *acquired* by a trace if it first occurs in an inbound term and is also carried by that term.

Some atoms in a role have special properties. An atom may be declared to be non-originating with the `non-orig` form or the `pen-non-orig` form and uniquely originating with the `uniq-orig` form. The declarations make assertions about instances of a role, assertions that will be defined after role instantiation is explained.

Every variable that occurs in each term declared to be non-originating must occur in some term in the trace, and the term must not be carried by any term in the trace. Every variable that occurs in each term declared to be penetrator non-originating must occur in some term in the trace, but the term may be carried by some term in a trace. Each term declared to be uniquely originating must originate in the trace. Each variable of sort *mesg* must be acquired in the trace.

4 Executions

A protocol analysis problem is specified by a skeleton. Some background information is presented before details of a problem specification is given.

A skeleton describes a set of executions of a protocol. They specify the

local behavior of participants and their interactions via message-passing. A definition of an execution is presented for the case in which protocols declare no terms to be non-originating or uniquely originating, and then is later amended. The executions are a representation of the strand space notion of bundle.

A strand represents a principal executing a single local session of a protocol role. The sequence of events that describes the session is patterned after a prefix of its role's trace. In the context of a protocol, a strand's trace is represented by an *instance*, a triple consisting of a role name, a height, and a map from role variables to terms. The length of the described sequence is the instance's *height*, and must be positive and no greater than the length of the associated role's trace. The map is the instance's *environment.* The domain of the map is the set of variables that occur in the events in the prefix of the role's trace of the same length as the instance's height.

An example of an instance of the Needham-Schroeder responder role is:

```
(defstrand resp 2 (b a) (a b) (n1 n1) (n2 n2))
```

The trace associated with this instance is:

$$\langle -\{\!|N_1, B|\!\}_{K_A}, +\{\!|N_1, N_2|\!\}_{K_B} \rangle.$$

The *position* of a event in the instance's trace is its index in the sequence, and in this example, the position of $-\{\!|N_1, B|\!\}_{K_A}$ is zero.

In addition to a protocol, a component of an execution is a collection of local sessions. An instance cannot be used to identify one session, because two principals may engage in the same pattern of message passing. Instead, a sequence of instances is used as a component of an execution, and a *strand* is represented by an index that selects an instance from the sequence. In other words, each principal executing a local session is represented by a natural number less than the length of the sequence of instances, and its behavior is described by the instance found using its representation.

A *node* is a pair of natural numbers that identifies an event within a sequence of instances. The first integer is the strand, and the second is the position of the event within the strand's trace. A node with an inbound term is a reception node, and one with an outbound term is a transmission node. Given a sequence i, its nodes are:

$$\{(s, p) \mid s < |i|, p < \mathsf{height}(i(s))\}.$$

The remaining component of an execution is a binary relation between transmission nodes and reception nodes, where each pair of nodes in the relation agree on their message term. The relation specifies message transmissions between strands. The nodes of the graph associated with an execution are the nodes in the sequence of instances, and the edges include the ones in the relation. Additionally, the following strand succession edges are included:

$$\{((s,p),(s,p+1)) \mid s < |i|, p+1 < \mathsf{height}(i(s))\}.$$

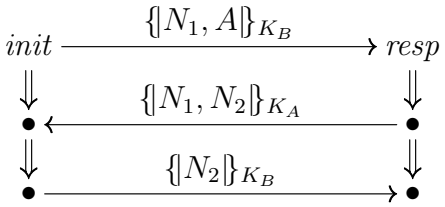
Executions with cyclic graphs are omitted from consideration, because they violate causality. Relation R on set S is *asymmetric* iff $x\ R\ y$ implies not $y\ R\ x$ for all distinct $x, y \in S$. The transitive asymmetric relation $\prec$ is the transitive closure of the graph's edges, and $n_0 \prec n_1$ asserts that the message event at n_0 precedes the one at n_1.

The node relation of an execution satisfies one additional property. For each reception node in its sequence, there exists a unique transmission node, related to it by the communication ordering. In other words, the relation must be a function. Informally, this property ensures that every message reception is accounted for by the activity of a principal that is part of the execution. Figure 3 shows the intended execution of the Needham-Schroeder Protocol. The node graph of an execution is a bundle.

A set of runs of the protocol is associated with each execution. A variable is associated with an execution if the variable occurs in the range of an environment in some instance. To derive a run from an execution, a substitution that maps each variable associated with the execution to a ground term is applied to the execution. CPSA message algebras do not contain the constants required to name all the ground terms. For most sorts, the identity of a particular ground term is irrelevant to the analysis. Message tags are the only exception to this rule. For this reason, CPSA message algebras are free algebras, but not initial algebras.

Strands in executions represent both adversarial and non-adversarial behaviors. A strand that is an instance of a protocol role is non-adversarial, and is called *regular*. A strand that represents adversarial behavior is called a *penetrator* strand.

The roles that define adversary behavior codify the basic abilities that make up the Dolev-Yao model. They include transmitting an atom such as a name or a key; transmitting a tag; transmitting an encrypted message after receiving its plain text and the key; and transmitting a plain text after




```
(vars (n1 n2 text) (a b name))
(defstrand init 3 (n1 n1) (n2 n2) (a a) (b b))
(defstrand resp 3 (n2 n2) (n1 n1) (b b) (a a))
(precedes ((0 0) (1 0)) ((1 1) (0 1)) ((0 2) (1 2)))
```

Figure 3: Needham-Schroeder Intended Run

receiving ciphertext and its decryption key. The adversary can also concatenate two messages, or separate the pieces of a concatenated message. Since a penetrator strand that encrypts or decrypts must receive the key as one of its inputs, keys used by the adversary—compromised keys—have always been transmitted by some participant.

Figure 4 shows a penetrated execution of the Needham-Schroeder Protocol. Strand space theory would decompose the penetrator behavior into multiple strands; instead the description of the penetrator has been simplified by the use of an artificially constructed role *pen*.

A *non-originating term* is an atom that is carried by no message, and it or its inverse is the key of an encryption in some message. Each non-originating term is a key that is not compromised, as the penetrator has no access to the key.

A *penetrator non-originating term* is an atom that is forbidden from being originated on a penetrator strand in an execution.

A *uniquely originating term* is an atom that originates on exactly one strand in the execution. The correct behavior of some protocols depends on the fact that its executions include only ones in which some terms are uniquely originating—each term's provenance is one node. Implementations of these protocols can ensure unique origination of a term by freshly generating a nonce for the component of the message it represents.

Occasionally, protocol roles designate some terms to be non-originating or uniquely originating. Consider the prefix of the trace of a role associated

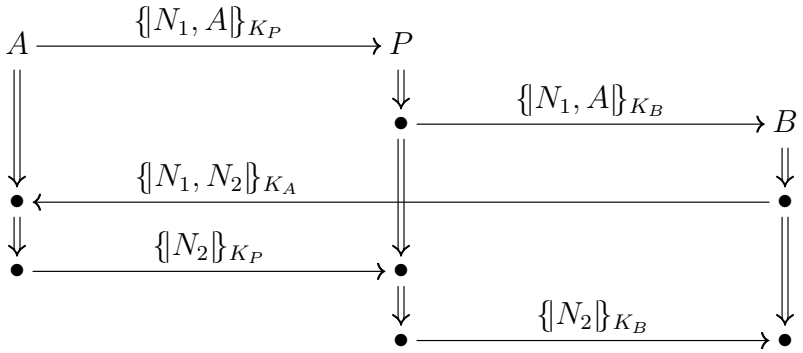


Figure 4: Needham-Schroeder Penetrated

with some instance. When a uniquely originating role term is carried in the prefix, the instance's environment maps it to a term that must originate in the instance's strand. Furthermore, when the variables in a non-originating role term occur in the prefix, the instance's environment maps that term to one that must not be carried by any message term in the execution. The mapping of a non-originating role term can be conditioned on the height of the instance. A role non-origination assumption of the form `(3 a)` asserts that `a` will not be mapped into an instance unless its height is at least three. Penetrator non-originating atoms are similar. Section 12 provides advice on when to add non-origination or unique origination assumptions to roles.

5 Skeletons

A *skeleton* represents regular behavior that might make up part of an execution. The components of a skeleton are similar to an execution. The components of a skeleton include a protocol, a sequence of instances, and a binary relation between transmission nodes and reception nodes within the sequence. Unlike an execution, the strands in a skeleton specify only regular behavior. Furthermore, the pair of nodes in the relation need not agree on their message term. Two nodes are related if the transmitting node precedes the reception node, as an execution it represents may include nodes between the related transmission and reception nodes.

The final three additional components of a skeleton are a set of non-originating terms, a set of penetrator non-originating terms, and a set of

```
(defskeleton PROTOCOL VARIABLES
   ...                  ; Instance sequence
   (precedes ...)       ; Node orderings
   (non-orig ...)       ; Non-originating terms
   (pen-non-orig ...) ; Penetrator non-originating terms
   (uniq-orig ...))     ; Uniquely originating terms
```

Figure 5: Components of Skeletons

uniquely originating terms. To be a skeleton, each uniquely originating term must originate in at most one strand in the skeleton, and each non-originating term must never be carried by some event in the skeleton and every variable that occurs in the term must occur in some event. For a penetrator non-originating term, is suffices that every variable that occurs in the term must occur in some event. Furthermore, for each uniquely originating term that originates in the skeleton, the node relation must ensure that reception nodes that carry the term follow the node of its origination. Figure 5 shows the components in the S-expression representation of a skeleton.

Two skeletons are equivalent if there is a permutation of the strands in one skeleton that when applied to its sequence and its node relation, produces the other skeleton. Two skeletons are also equivalent if they differ only by a systematic, sort-preserving renaming of their variables, excluding variables that occur in the domains of environments—the role variables. The trace of a strand is used for the comparison, so role names need not match.

A skeleton is normalized by performing a transitive reduction on its 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. Two skeletons are equivalent if their normalized forms are equivalent.

One special skeleton is associated with each execution. It summarizes the regular behavior of the execution. It is derived from the execution by enriching its node relation to contain all node orderings implied by transitive closure, deleting all strands and nodes that refer to penetrator behavior, and then performing the transitive reduction on the resulting node relation. The set of uniquely originating terms is the set of terms that originate on exactly one strand in the execution, and are carried in a term of a regular strand. The set of non-originating terms is the union of two sets. One set contains

each term that is used as an encryption or decryption key in some term in the execution, but is not carried by any term. The other set contains the terms specified by non-origination assumptions in roles. If a realized skeleton instance maps all of the variables that occur in one of its non-originating role terms, the mapped term is a member of the skeleton's set of non-originating terms. A skeleton is *realized* if it summarizes the behavior of some execution.

Skeletons are a central concept in the CPSA algorithm because they capture the notion that the details of penetrator behavior are irrelevant to the analysis. From the perspective of regular behavior, all that is needed is a description of the messages that can be derived by the penetrator at a given reception node of a regular strand. In a realized skeleton, some combination of penetrator behavior and regular behavior derives the message at every reception node. The skeleton's node relation specifies the transmission nodes that provide messages available to the penetrator for message derivations. The rules for message derivation are algebra specific. If the message derivation rules imply a message at a reception node in a skeleton is derivable, the node is *realized.*

A skeleton with an unrealized node might be related to another skeleton with additional regular behavior that makes the original node realized. A skeleton with additional regular behavior is called a refinement.

A skeleton A structurally refines B if the transitive closure of the graph associated with B is a subgraph of the one associated with A, the event at corresponding nodes agree, the set of uniquely originating terms of B is a subset of the ones in A, the set of non-originating terms of B is a subset of the ones in A, and a uniquely originating term that originates in B originates at the corresponding node in A. Penetrator non-originating assumptions are similar to non-originating assumptions.

A skeleton A message refines B if A and B agree on all but the terms in the range of each strand's environment and its non-originating and uniquely originating terms, and there is a substitution that maps each term in B to its related term in A. Similar to structural refinement, a uniquely originating term that originates in B originates at the same node in the image of B.

A skeleton A refines B if A is equivalent to a skeleton that structurally or message refines B.[1] Each skeleton describes the realized skeletons that

[1] A skeleton A refines B if there is a homomorphism from B to A as defined in [2]. The implementation avoids the complexities of directly representing homomorphisms by composing structural and message refinement with equivalence checks, as is done in this chapter.

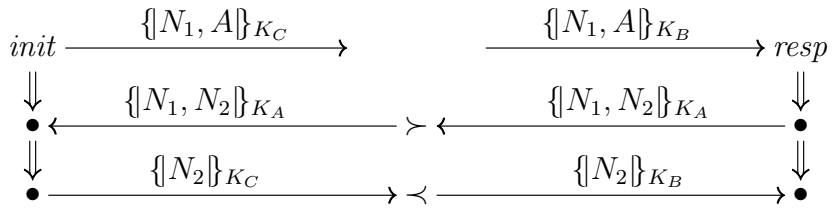


Figure 6: Needham-Schroeder Shape (K_A^{-1} uncompromised, N_2 fresh)

refine it. A skeleton is *dead* if no realized skeleton refines it. A diagram of a skeleton is in Figure 6.

The CPSA algorithm computes realized skeletons from unrealized skeletons by identifying an unrealized node, and computing the skeletons that refine the unrealized skeleton by making the target node realized.

6 Listeners

In addition to the roles specified in a protocol, for each term t, a regular strand may be an instance of the listener role with the trace $lsn(t) = \langle -t, +t \rangle$. There are no non-originating or uniquely originating terms associated with a listener role.

A listener strand is used in a skeleton to assert that an atom t is available on its own to the adversary, unprotected by encryption. For example, to test if the protocol keeps a term t from the adversary, one adds a strand that listens for t. The term is protected if the resulting skeleton is dead. Otherwise, the CPSA analyzer program will find a refined realized skeleton that shows how the adversary accesses t.

A listener instance in S-expression syntax follows. See STRAND in Table 1, Appendix A.

```
(deflistener TERM)
```

The CPSA programs generate the associated listener role and hide it on output. Listener role names are absent in all forms of output, one indication that a strand is an instance of a listener role.

7 Authentication Tests

Authentication tests guide the search for skeletons that refine one with an unrealized node into ones in which it is realized. There are two types of authentication tests, nonce and encryption tests. In both cases, an unrealized node is selected, called the *test node.* A term carried by the inbound message at the test node is identified as the *critical term.* A critical term is one that occurs in a message context, the construction of which cannot be explained by the regular behavior in the current skeleton or by penetrator behavior. An authentication test determines the additional regular behavior required to refine a skeleton into ones in which the test node is realized.

The critical term in a nonce test is a uniquely originating term, the nonce. It is freshly generated by one regular participant in each run of the protocol. A nonce is unguessable by both regular and adversarial participants except when it is received in an unprotected context.

A reception node's *outbound predecessors* is the set of messages sent by transmission nodes that precede the reception node, as given by the transitive closure of its skeleton's node ordering relation. Suppose every occurrence of the nonce in the outbound predecessors of the test node is within a context protected by encryption, but the critical term in the test node occurs outside of all of those encryptions. Clearly, another participant was able to decrypt one of the test node's outbound predecessors. The set of encryptions that protects a critical term in a test node's outbound predecessors is called its *escape set.*

There are three ways to refine a skeleton to account for the decryption: regular augmentation, listener augmentation, and contraction. For regular augmentation, an instance of a protocol role is added to the skeleton. The final transmission node of the strand is called a *transforming node.* The strand is selected for augmentation because relative to the test node's outbound predecessors, the transforming node's message shows the strand performed the decryption.

The penetrator can expose the critical term if it has access to any of the decryption keys used to protect it in the escape set. For listener augmentation, an instance of a listener role listening to one decryption key used to protect the critical term is added, and the skeleton's communication ordering relation is updated to record the fact that the listener's transmission node precedes the test node. If the resulting skeleton is not dead, the decryption can be explained by penetrator behavior.

For a contraction, a message refining substitution is found that equates two or more atoms. Sometimes, the key used to protect the critical term can be equated with one used in previous messages, thus vacuously explaining the decryption of the critical term.

The critical term in an encryption test is an encryption. When the encryption key is unavailable, the encryption is unguessable. Whenever the unavailability of the encryption key can be established, the methods used to refine skeletons with nonce tests apply. Additionally, an instance of a listener role listening for the critical term's encryption key is added. If the resulting skeleton is not dead, the encryption can be explained by penetrator behavior.

8 Generalization

Repeated use of authentication tests either produce realized skeletons or show that a skeleton is dead. The next step in the algorithm is to make each realized skeleton into a shape, using the process of generalization.

Realized skeleton A *generalizes* realized skeleton B if A refines the origin problem specification, and B refines A. Furthermore, A may not combine strands in B. The *shape* associated with a realized skeleton is its maximally generalized realized skeleton. The shapes of a protocol capture all the essentially different executions possible for the protocol consistent with the initial behavior specification. Figure 6 shows the shape associated with the Needham-Schroeder Protocol from the point of view of a responder strand.

A different fixed set of operations is used to transform a realized skeleton into a shape: deletion, weakening, separation, and forgetting. If an operation succeeds, it produces a more general realized skeleton, one that is not equivalent to the starting skeleton. If no operations succeeds, the skeleton is a shape.

For deletion, a node and all nodes that follow it in a strand are deleted, and the resulting skeleton is checked to see if it generalized the starting skeleton. The operation is tried for each node in the starting skeleton until there is a success.

If deletion fails, a skeleton is weakened by deleting one element of the communication ordering, then checking the result to see if it generalized the starting skeleton. The operation is tried for each communication ordering in the starting skeleton until there is a success.

If weakening fails, origination assumption forgetting is tried by deleting

each term in the non-originating set that is not specified by a role. This is followed by deleting each term in the uniquely originating set that is not specified by a role.

If origination assumption forgetting fails, variable separation is tried. Sometimes a more general skeleton can be found by replacing some occurrences of one variable by a fresh variable. To separate a variable, the collection of places at which the variable occurs in the range of all environments is generated, and a fresh variable is substituted for the variable at a subset of these places. All possibilities are tried until a more general skeleton is found.

Sometimes a shape is derived from another shape by collapsing two strands in a shape. Collapsing might produce an unrealized skeleton, so authentication tests apply.

9 Skeletons

With this background, the `defskeleton` form in Table 1, Appendix A is explained. The key object in CPSA input and output is a skeleton, but an object with weaker properties is allowed for the initial problem statement. A *preskeleton* is a skeleton except that terms in the uniquely originating set may originate in more than one strand. Furthermore, the node relation of a preskeleton need not imply that a node that carries a uniquely originating term is after the node of its origination. A preskeleton that cannot be immediately converted into a skeleton is erroneous, and an error message is issued.

Referring to SKELETON in Table 1, the ID in the skeleton form names a protocol. It refers to the most recent protocol definition of that name which precedes the skeleton form. The ID in the strand 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 messages in a strand from its role's trace. The first term is interpreted using the role's variables and the second term uses the skeleton'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 `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 every skeleton variable that also occurs in a role of its protocol. The sort of every skeleton variable that occurs in the `non-orig`, `pen-non-orig`, or `uniq-orig` list or in a maplet must be declared, using the `vars` form.

The 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, with the exception of the `comment` key.

9.1 Semantic Errors in the Input

The error messages generated for syntax errors are informative, however the ones generated for semantic errors are less so. A role might be rejected because it is not well-formed. A role is not well-formed if (1) there is a term declared to be uniquely originating that does not originate in the trace, (2) there is a term declared to be non-originating that is carried by some term in the trace, a variable occurs in the term that does not occur in the trace, or the declaration of the term included a height, and a variable occurs in the term that does not occur in the prefix of the trace of the given height, or (3) a variable of sort **mesg** is not acquired in the trace. The error message might not indicate which condition caused the rejection.

Similarly, a skeleton might be rejected because it is not well-formed. A skeleton is not well-formed if (1) the first node in a node pair refers to an inbound term, or the second node refers to an outbound term, (2) the node ordering contains cycles, (3) a term declared to be uniquely originating, is not carried by any term, (4) an instance maps a uniquely originating role term to a term that does not originate in the instance's strand, or (5) a term declared to be non-originating is carried by a term in some strand, or a variable occurs in the term that does not occur in any strand. Once again, the error message might not indicate which condition caused the rejection.

9.2 Needham-Schroeder Input

This section contains the verbatim input of the running example used throughout this paper. The use of an editor that pretty-prints S-expressions is recommended.

```
;;; Hey Emacs, use -*- mode:scheme -*-
(herald "Needham-Schroeder Public-Key Protocol"
        (comment "This protocol contains a man-in-the-middle"
                 "attack discovered by Galvin Lowe."))
```

An S-expression version of Figure 1 follows.

```
(defprotocol ns basic
  (defrole init
    (vars (a b name) (n1 n2 text))
    (trace
     (send (enc n1 a (pubk b)))
     (recv (enc n1 n2 (pubk a)))
     (send (enc n2 (pubk b)))))
  (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))))))
```

The protocol is analyzed from the point of view of a complete run of one instance of an initiator role.

```
(defskeleton ns
  (vars (a b name) (n1 text))
  (defstrand init 3 (a a) (b b) (n1 n1))
  (non-orig (privk b) (privk a))
  (uniq-orig n1))
```

10 Output

The CPSA output format has been designed so that it can be reused as input. All skeletons in the output are normalized skeletons, with the possible exception of an initial preskeleton, the one used to state a problem. If an initial preskeleton cannot be converted into a skeleton, an error is immediately signaled.

For each skeleton, a CPSA analyzer computes a set of skeletons that refine it using a fixed set of operations based on authentication tests. The immediate descendants of a skeleton is called its *cohort.* A member of the cohort that is equivalent to a previously seen skeleton is replaced by that skeleton. Thus the skeletons in an analysis form a directed acyclic graph. All but one skeleton has a single parent, and one skeleton can be a member of several cohorts.

The operations above either produce realized skeletons or show that a skeleton is dead. The next step in the algorithm is to make each realized skeleton into a shape, using the process of generalization. Although the set of shapes is in some sense the answer to the problem, an understanding of the operations used to generate the shapes can be very informative. The remainder of this section describes the annotations in the output that allow for an understanding of each step of the analysis.

On output, key-value pairs are added to each skeleton's association list, SKEL-ALIST. Every skeleton in the output is labeled with a unique identifier with `(label` INTEGER`)`. A skeleton has `(parent` INTEGER`)` if it is a member of the cohort of the identified parent. A skeleton has `(seen` INTEGER+`)` when members of its cohort are equivalent to previously seen skeletons. A skeleton lists its unrealized nodes with `(unrealized` NODE∗`)`. The traces associated with each strand is given by the `(traces ...)` form.

Figure 7 shows a skeleton generated during an analysis of the Needham-Schroeder Protocol from the point of view of an initiator strand. It is labeled as 1. It's parent is labeled 0. It has one child, and that child has not been seen before. It is unrealized.

The operation used to derive a skeleton is recorded with `(operation` TEST KIND TERM NODE TERM∗`)`, where TEST is the authentication test `encryption-test` or `nonce-test`, KIND is `(added-strand` ID INTEGER`)`, `(contracted` MAPLET∗`)`, or `(added-listener` TERM`)`, TERM is the critical term, NODE in the test node, and the remaining terms specify the escape set. When the operation kind is added strand, the instance's role

```
(defskeleton ns
  (vars (n1 n2 n2-0 text) (a b name))
  (defstrand init 3 (n1 n1) (n2 n2) (a a) (b b))
  (defstrand resp 2 (n2 n2-0) (n1 n1) (b b) (a a))
  (precedes ((0 0) (1 0)) ((1 1) (0 1)))
  (non-orig (privk a) (privk b))
  (uniq-orig n1)
  (operation nonce-test (added-strand resp 2) n1 (0 1)
    (enc n1 a (pubk b)))
  (traces
    ((send (enc n1 a (pubk b))) (recv (enc n1 n2 (pubk a)))
      (send (enc n2 (pubk b))))
    ((recv (enc n1 a (pubk b)))
      (send (enc n1 n2-0 (pubk a)))))
  (label 1)
  (parent 0)
  (unrealized (0 1))
  (comment "1 in cohort - 1 not yet seen"))
```

Figure 7: Annotated CPSA Output

name and height are provided. For kind added-listener, a term is provided. For kind contracted, the substitution is provided. When a substitution refers to a variable not in the skeleton, its name is unpredictable. For generalization, the operation is recorded as `(operation generalization` METHOD`)`, where METHOD is one of `deleted` NODE, `weakened` NODE-PAIR, `separated` TERM, or `forgot` TERM. Shapes can be collapsed leading to new shapes. For shape collapsing, the operation is recorded as `(operation collapsed` INTEGER INTEGER`)`, where the two INTEGERs identify the strands merged.

The skeleton in Figure 7 was generated as a result of a nonce test, by augmenting the starting skeleton with a responder strand of length two. The critical term is `n1`, the test node is `(0 1)`, and the escape set has one element.

When the operation kind is added strand, it is possible that the number of strands in the skeleton and its parent are the same. In this case, CPSA has found a way to produce a more concise representation of the skeleton by merging two strands.

10.1 Needham-Schroeder Output

This section contains the verbatim output of the running example used throughout this paper. A run starts by displaying the program's version number.

```
(herald "Needham-Schroeder Public-Key Protocol"
  (comment "This protocol contains a man-in-the-middle"
    "attack discovered by Galvin Lowe."))

(comment "CPSA 2.2.9")
(comment "All input read")
```

An S-expression version of Figure 1 follows.

```
(defprotocol ns basic
  (defrole init
    (vars (a b name) (n1 n2 text))
    (trace (send (enc n1 a (pubk b)))
      (recv (enc n1 n2 (pubk a))) (send (enc n2 (pubk b)))))
  (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))))))
```

The protocol is analyzed from the point of view of a complete run of one instance of an initiator role.

```
(defskeleton ns
  (vars (n1 n2 text) (a b name))
  (defstrand init 3 (n1 n1) (n2 n2) (a a) (b b))
  (non-orig (privk a) (privk b))
  (uniq-orig n1)
  (traces
    ((send (enc n1 a (pubk b))) (recv (enc n1 n2 (pubk a)))
      (send (enc n2 (pubk b)))))
  (label 0)
  (unrealized (0 1))
  (comment "1 in cohort - 1 not yet seen"))
```

A nonce test justifies adding an instance of part of a reponder role.

```
(defskeleton ns
  (vars (n1 n2 n2-0 text) (a b name))
  (defstrand init 3 (n1 n1) (n2 n2) (a a) (b b))
  (defstrand resp 2 (n2 n2-0) (n1 n1) (b b) (a a))
  (precedes ((0 0) (1 0)) ((1 1) (0 1)))
  (non-orig (privk a) (privk b))
  (uniq-orig n1)
  (operation nonce-test (added-strand resp 2) n1 (0 1)
    (enc n1 a (pubk b)))
  (traces
    ((send (enc n1 a (pubk b))) (recv (enc n1 n2 (pubk a)))
      (send (enc n2 (pubk b))))
    ((recv (enc n1 a (pubk b)))
      (send (enc n1 n2-0 (pubk a)))))
  (label 1)
  (parent 0)
  (unrealized (0 1))
  (comment "1 in cohort - 1 not yet seen"))
```

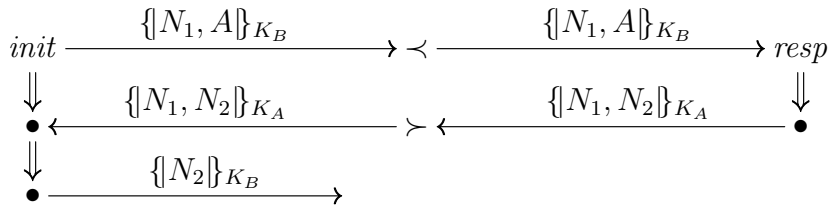


Figure 8: Needham-Schroeder Shape (Initiator Point of View)

A nonce test justifies a contraction that produces the one and only shape. The shape is also displayed in Figure 8.

```
(defskeleton ns
  (vars (n1 n2 text) (a b name))
  (defstrand init 3 (n1 n1) (n2 n2) (a a) (b b))
  (defstrand resp 2 (n2 n2) (n1 n1) (b b) (a a))
  (precedes ((0 0) (1 0)) ((1 1) (0 1)))
  (non-orig (privk a) (privk b))
  (uniq-orig n1)
  (operation nonce-test (contracted (n2-0 n2)) n1 (0 1)
    (enc n1 n2 (pubk a)) (enc n1 a (pubk b)))
  (traces
    ((send (enc n1 a (pubk b))) (recv (enc n1 n2 (pubk a)))
      (send (enc n2 (pubk b))))
    ((recv (enc n1 a (pubk b)))
      (send (enc n1 n2 (pubk a)))))
  (label 2)
  (parent 1)
  (unrealized)
  (shape))
```

The following phrase means CPSA is finished with this problem—its exhausted its to do list.

```
(comment "Nothing left to do")
```

11 Macros

After reading the input, CPSA expands macros before in analyzing the results. A macro definition is a top-level form.

(`defmacro` (NAME ARG*) BODY)

The CPSA program expands all calls to macros in non-macro defining S-expressions using the macros that have been defined previously. A macro definition can be used to expand a call if the first element of a list matches the name of the macro, and the length of the remaining elements in the list matches the length of the macro's argument list. When two macros definitions are applicable, the last definition takes precedence. The CPSA program omits macro definitions from its output.

12 Advice

This section contains advice derived from using CPSA. When specifying CPSA input, one must decide when to specify terms as uniquely originating or non-originating in a role, and when to specifying them in the initial skeleton. If a flesh value is generated by all programs that implement a role in a protocol, the term that represents the fresh value should be assumed to be uniquely originating in the role. Otherwise, unique origination assumptions should be specified in the initial skeleton.

Adding non-origination assumptions to a role can lead to an excessively weak protocol analysis, i.e. an analysis relative to an unrealistically narrow assumption. Placing non-origination assumptions in the initial skeleton is preferred whenever possible.

The Basic Crypto Algebra has two text-like sorts, **text** and **data**. For some protocols, message fields that carry uniquely originating data cannot be carried by other text-like fields. To ensure CPSA does not explore skeletons in which uniquely originating data is carried in fields with predictable values, the convention is to use the sort **data** for uniquely originating data, and sort **text** for the other fields.

When looking at the output, try extracting the shapes first. If the shapes only version of the output does not answer your questions, try studying the output that contains intermediate skeletons.

When CPSA generates an unexpected intermediate skeleton, study its operation field (see Page 10). Usually, the unexpected intermediate skeletons of interest have been generated as a result of an authentication test. Section 7 explains how to interpret an operation field for an authentication test.

When using CPSA for protocol design, focus on authentication tests. For each iteration of the design, search for the most informative unexpected intermediate skeleton. That skeleton is likely to suggest a missing origination assumption, or the redesign of some message term included in the operation field.

There are situations in which origination assumptions are not justified for initial segments of runs of a protocol, but are required to show that complete runs of the protocol have certain expected shapes. In this case, the progressive refinement analysis technique is used. The initial segments of runs are analyzed with only the origination assumptions justified initially. A shape associated with a partial run may justify additional origination assumptions. For example, a strand in a partial run may send its next message only after a trust decision is made, and the implication of the decision is that the strand infers that some key is uncompromised. In this case, the shape with the new origination assumption and additional regular behavior is supplied as input to CPSA, thereby refining the original problem.

As there is no guarantee that CPSA is bug free, you may come upon input that causes non-termination. As a result, whenever you run the program unattended, you should limit its memory usage. To get output that can be visualized, specify a step count and/or a strand bound so that CPSA has the chance to abort the run in a fashion that generates graphable output. Of course, sending us input that causes erroneous behavior will help us improve CPSA.

13 Formula Extraction

The `cpsalogic` program extracts a formula in the language of order-sorted first-order logic for each problem and its shapes from a CPSA run. The formula is called a shape analysis sentence. The formula is satisfied in all realized skeletons when CPSA finds all the shapes for the problem. The details of formula extraction are presented in Appendix B of The CPSA Specification [7].

14 Annotations

The `cpsaannotations` program uses protocol annotations to annotate shapes and generate protocol soundness obligations for use with the rely-guarantee method of trust management [5]. The language of formulas is order-sorted first-order logic extended with a modal says operator. Formula terms may include function symbols that are not part of a protocol's message signature. The syntax of a formula is specified in the CPSA Overview [6].

15 Parameter Analysis

The parameters of a role are the atoms that are not acquired by the role's trace, but must be available before a complete execution of the trace is possible. The `cpsaparameters` program computes sets of sets of parameters consistent with the role. If the expected parameter set is not a member, a specification error is indicated. The details of parameter analysis are presented in Appendix A of The CPSA Specification [7].

Acknowledgement

Jonathan K. Millen provided valuable feedback as our first CPSA user and on a draft of this document.

A BCA Syntax Reference

The complete syntax for the analyzer using the Basic Crypto Algebra is shown in Table 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

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
	|	(`pen-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
	|	(`pen-non-orig` TERM∗) SKEL-ALIST
	|	(`uniq-orig` TERM∗) SKEL-ALIST
	|	(`precedes` NODE-PAIR∗) SKEL-ALIST | ...
NODE-PAIR	←	(NODE NODE)
NODE	←	(INTEGER INTEGER)

Table 1: CPSA Syntax

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.

References

[1] Edsger W. Dijkstra. Why numbering should start at zero. `http://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html`, August 1982.

[2] Shaddin F. Doghmi, Joshua D. Guttman, and F. Javier Thayer. Searching for shapes in cryptographic protocols. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, number 4424 in LNCS, pages 523–538. Springer, March 2007. Extended version at `http://eprint.iacr.org/2006/435`.

[3] Daniel Dolev and Andrew Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 29:198–208, 1983.

[4] 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.

[5] Joshua D. Guttman, F. Javier Thayer, Jay A. Carlson, Jonathan C. Herzog, John D. Ramsdell, and Brian T. Sniffen. Trust management in strand spaces: A rely-guarantee method. In *In Proc. of the European Symposium on Programming (ESOP 04), LNCS*, pages 325–339. Springer-Verlag, 2004.

[6] John D. Ramsdell and Joshua D. Guttman. *CPSA Overview*. 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