CPSA Overview

John D. Ramsdell Joshua D. Guttman
The MITRE Corporation

July 29, 2010

Enclosed is a brief overview of CPSA, along with a description of CPSA's support for the rely-guarantee method. The message terms $\langle term \rangle$ used by CPSA are a straightforward representation of terms using Lisp-style, prefix notation.

A subset of the terms are called *atoms*. Atoms belong to the *base sorts* `name, text, data, skey, akey`. Syntactically, atomic terms may be either symbols (i.e., identifiers) or atomic-sorted function applications such as `(pubk a)`. Even though an atom as a term may have terms within it, a receiver of an atom is not allowed to extract terms that occur in it. This reflects the fact that the reception of the atom `(invk k)`, the inverse of some asymmetric key `k`, does not allow the receiver to construct `k`.

Non-atomic terms are constructed by applications of encryption (`enc`) and pairing (`cat`), where n-ary concatenation is parsed right-associatively. The second argument of an encryption is the key. Encryption may also be written in an n-ary form where the last argument is the key and the arguments preceding it are implicitly concatenated.

A term carries one of its subterms 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) `(enc` t_0 t_1`)` carries t if t_0 carries t, and (3) `(cat` t_0 t_1`)` carries t if t_0 or t_1 carries t. Note that `(enc` t_0 t_1`)` does not carry t_1 unless (anomalously) t_0 carries t_1.

1 Protocols

A protocol is a set of roles.

`(defprotocol` ⟨*sym*⟩ `basic` ⟨*role*⟩$^+$`)`

The symbol ⟨*sym*⟩ names the protocol. The symbol `basic` identifies the term algebra used to specify messages in roles.

A role has the form:

`(defrole` ⟨*sym*⟩ `(vars` ⟨*decl*⟩*`)`
`(trace` ⟨*event*⟩$^+$`)`
`(non-orig` ⟨*non*⟩*`)`$^?$
`(uniq-orig` ⟨*atom*⟩*`)`$^?$
⟨*annos*⟩`)`

⟨*non*⟩ `::=` ⟨*atom*⟩ `|` `(`⟨*height*⟩ ⟨*atom*⟩`)`

Non-terminal ⟨*sym*⟩ is an S-expression symbol that names the role. A ⟨*decl*⟩ is a list of variable symbols followed by a sort symbol. The `trace` is a sequence of message events, each indicating a message to be transmitted or received. The syntax used for a message event ⟨*event*⟩ has one of two forms, `(send` ⟨*term*⟩`)` or `(recv` ⟨*term*⟩`)`. The length of a role is the length of its trace, and must be positive. The remaining components of a role will be described later.

A term originates in a trace if it is carried in some event and the first event in which it is carried is a sending term. A term is acquired by a trace if it first occurs in a receiving term and is also carried by that term.

2 Executions

An execution of a protocol may involve any number of strands, each conveying either regular or adversarial behavior. Thus, each strand is an instance of some role. For CPSA input and output, a strand is specified by the following form:

`(defstrand` ⟨*sym*⟩ ⟨*int*⟩ ⟨*maplet*⟩*`)`

The symbol names the role, ⟨*int*⟩ is the height which must be positive and no greater than the role's length, and the remainder determines a substitution from role variables to terms.

⟨*maplet*⟩ `::=` `(`⟨*sym*⟩ ⟨*term*⟩`)`

The trace associated with the specified behavior is the result of truncating the role's trace so it agrees with the height, and applying the substitution `(`⟨*maplet*⟩*`)`.

A strand's behavior includes inherited origination assumptions. When a role assumes atom a is uniquely originating using the `uniq-orig` form, applying the substitution `(`⟨*maplet*⟩*`)` to a produces an inherited uniquely originating atom. Role atoms assumed to be non-originating using the `non-orig` form are inherited similarly. For a non-originating assumption, a strand height may be associated with an atom. In this case, a non-originating assumption is inherited by strands that meet or exceed the height constraint. Note that the definition of a uniquely originating atom and a non-originating atom in an execution is still to come.

A strand in an execution is identified by a natural number. To describe an execution, the behavior of each participant is listed sequentially, and position of the `defstrand` form in the list determines the strand's identifier. Zero-based indexing is used, so zero identifies the first strand.

A messaging event in an execution occurs at a node, which is a pair of natural numbers. The first number is the strand's identifier. The second number is the position of an event in the trace of the strand, once again using zero-based indexing. Thus node `(1 1)` in

```
(defstrand r1 3 (a b) (b a))
(defstrand r2 2 (x a) (y a) (z b))
```

names the last event in the last strand. The term is the result of instantiating the second event in role `r2`'s trace using the substitution `((x a) (y a) (z b))`.

Message exchanges are part of an execution. Each exchange is described by a pair of nodes. The first node must name a sending term, and the second node must name a receiving term. In an execution, the two terms are the same. Furthermore, for each receiving term in a strand's trace, there is a unique node that transmits its term. In other words, all message receptions are explained by transmissions within the execution.

In an execution, a *uniquely originating atom* originates in the trace of exactly one strand. An inherited uniquely originating atom must originate in the trace of its strand. CPSA uses uniquely originating atoms to model freshly generated nonces used in many protocols.

A *non-originating atom* is carried by no trace of any strand in an execution, and it or its inverse is the key of an encryption in one of those traces. The inherited non-origination atoms must satisfy this property too.

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 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 strand. The basic adversary roles are built into CPSA.

3 Skeletons

CPSA never directly represents adversarial behavior. Instead, a skeleton is used. A skeleton represents regular behavior that might make up part of an execution. A skeleton is specified in CPSA output using a `defskeleton` form.

```
(defskeleton ⟨sym⟩ (vars ⟨decl⟩*)
   ⟨defstrand⟩+
   (precedes ⟨pair⟩*)?
   (non-orig ⟨atom⟩*)?
   (uniq-orig ⟨atom⟩*)?)
```

The symbol names the protocol used by its participants. The regular strands are specified as they are in an execution. The precedes form defines a binary relation on nodes (⟨*pair*⟩ `::=` (⟨*node*⟩ ⟨*node*⟩)). As in an execution, the first node names a sending term and the second term names a receiving term. Unlike an execution, the pair of nodes in the relation need not agree on their message term. Two nodes are related if the sending event precedes the

reception reception event, as an execution it represents may include events in between.

The final two additional components of a skeleton are a set of non-originating atoms, and a set of uniquely originating atoms. To be a skeleton, each uniquely originating atom must originate in at most one strand in the skeleton, and each non-originating atom must never be carried by some event in the skeleton and every variable that occurs in the atom must occur in some event. Furthermore, for each uniquely originating atom that originates in the skeleton, the node relation must ensure that reception nodes that carry the atom follow the node of its origination.

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 atoms 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 atoms 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.

3.1 Preskeletons

Preskeletons are used to pose problems for CPSA to solve. A preskeleton is similar to a skeleton except atoms assumed to uniquely originate may originate in more than one strand, and the node relation need not ensure that reception nodes that carry the atom follow some node of origination. Experience has shown that it is more natural to specify some problems in a form that doesn't satisfy all the properties of a skeleton. If CPSA cannot immediately convert its input into a skeleton, an error is signal. With the exception of the restatement of the original problem, all preskeletons printed by CPSA are skeletons. A problem statement is called a *scenario*, and the converted skeleton is called the *scenario skeleton*.

3.2 Shapes

Given a scenario skeleton, CPSA determines whether there is an execution containing the strands in the skeleton, and satisfying its origination assumptions. Usually an execution contains additional regular strands, as well as adversary behavior. A major part of what CPSA does is to find all additional regular strands that are necessary to extend the scenario to an execution—a realized skeleton. If a realized skeleton is most-general, in the sense that there is no other realized skeleton that can be instantiated to it by merging nodes or atoms, then it is called a *shape*. CPSA finds all shapes for a scenario.

4 Listeners

In addition to the roles specified in a protocol, for each term t, a regular strand may be an instance of a so-called *listener* role with the trace `(recv` t`)` `(send` t`)`. There are no non-originating or uniquely originating atoms associated with a listener role. Listener behavior is specified with:

`(deflistener` ⟨*term*⟩`)`

A listener strand is used in a skeleton to assert that a term t is derivable by the adversary, unprotected by encryption. Hence it is used to test for compromise of a term. The term is protected if the resulting skeleton is unrealizable. Otherwise, CPSA will find a shape that shows how the adversary accesses t.

5 Annotations

To be analyzed, each role in a protocol must include an `annotations` form, as defined in Table 1. The ⟨*term*⟩ just after the `annotations` symbol is a role atom that, when instantiated, is the principal associated with the strand in the shape. A principal may be a key.

What follows is sequences of pairs. The integer gives the position of the event in the trace that is annotated by the formula, using zero-based indexing. Thus, each formula is associated with a node. Nodes for which no formula is specified are implicitly defined to be the trivial formula `(and)` for truth. Use `(or)` for falsehood.

⟨*annos*⟩	::=	`(annotations` ⟨*term*⟩ `(`⟨*int*⟩ ⟨*form*⟩`)`*`)`
⟨*form*⟩	::=	`(`⟨*sym*⟩ ⟨*fterm*⟩*`)` `|` `(not` ⟨*form*⟩`)`
	`|`	`(and` ⟨*form*⟩*`)` `|` `(or` ⟨*form*⟩*`)`
	`|`	`(implies` ⟨*form*⟩* ⟨*form*⟩`)`
	`|`	`(iff` ⟨*form*⟩ ⟨*form*⟩`)`
	`|`	`(says` ⟨*term*⟩ ⟨*form*⟩`)`
	`|`	`(forall (`⟨*decl*⟩*`)` ⟨*form*⟩`)`
	`|`	`(exists (`⟨*decl*⟩*`)` ⟨*form*⟩`)`
⟨*fterm*⟩	::=	⟨*term*⟩ `|` `(`⟨*sym*⟩ ⟨*fterm*⟩*`)`

Table 1: Annotation Syntax

The language of formulas is 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.

On output, each shape contains an `annotations` form and an `obligations` form. The annotations form presents every non-trivial formula derived from the protocol. The obligations form presents every non-trivial formula that must be true if the shape is sound.