One Picture is Worth a Dozen Connectives:

A Fault-Tree Representation of NPATRL Security Requirements

Iliano Cervesato[1][⋆] and Catherine Meadows[2]

[1] Carnegie Mellon University, Qatar Campus

P.O. Box 42866, Doha — Qatar

`iliano@cmu.edu`

[2] Center for High Assurance Computer Systems, Naval Research Laboratory

Washington, DC 20375 — USA

`meadows@itd.nrl.navy.mil`

Abstract. *In this paper we show how we can increase the ease of reading and writing security requirements for cryptographic protocols at the Dolev-Yao level of abstraction by developing a visual language based on fault trees. We develop such a semantics for a subset of NPATRL, a temporal language used for expressing safety requirements for cryptographic protocols, and show that the subset is sound and complete with respect to the semantics. We also show how the fault trees can be used to improve the presentation of some specifications that we developed in our analysis of the Group Domain of Interpretation (GDOI) protocol. Other examples involve a property of Kerberos 5, and a visual account of the requirements in Lowe's authentication hierarchy.*

Keywords: C.2.0.f Network-level security and protection, C.2.2.c Protocol verification, F.4.3 Formal Languages.

[⋆] This work was supported by ONR. Cervesato was partially supported by NRL under contract N00173-00-C-2086. This work was completed while this author was visiting Princeton University and later Tulane University.

1 Introduction

Like cryptographic protocols themselves, requirements for cryptographic protocols are not easy to get right. Although it is possible to divide cryptographic protocol requirements into broad classes such as secrecy, authentication, freshness, and so forth, requirements within the classes can vary in subtle, but nevertheless crucial ways. And, since many security problems arise from a misunderstanding of the requirements rather than a problem with the design of the protocol itself, it is important to get this right.

One of the first steps in supporting the development of sound requirements is to find a way of stating them precisely. Since the goal of most cryptographic protocols is to guarantee that certain events do not occur unless certain other events have or have not taken place, some sort of temporal logic seems like the most likely candidate. With that in mind, we have developed the NRL Protocol Analyzer Temporal Requirements Language (NPATRL) [14], for specification of requirements at the Dolev-Yao level of abstraction. In the Dolev-Yao model, cryptographic operations are specified as black box operations. This allows the analyst to capture the problems that arise in cryptographic protocol design even with the cryptographic algorithms themselves work perfectly.

Although, as its name suggests, NPATRL was originally s intended to be used the NRL Protocol Analyzer (NPA) [7], it can be used independently. It has been used to specify different types of requirements not only for generic key distribution and agreement protocols, as in [14], but also for complex protocols such as Secure Electronic Transaction (SET) [9] and the Group Domain of Interpretation (GDOI) Protocol [8].

It is important that requirements be easy to read, write, and understand. This, unfortunately, is where temporal languages fall short. Even for relatively simple languages like NPATRL, which includes only one temporal operator, requirements can quickly become complex and difficult to understand. For example, the *strong secrecy* requirement for the GDOI protocol (the requirement that temporary keys remain secret in the face of the compromise of certain temporary and longterm keys — see also Section 5) assumes the form in NPATRL displayed below:.

$$
\begin{array}{ll}
 & \mathsf{learn}(P, (), (K_G), \_) \\
\Rightarrow & \diamondminus \mathsf{gcks\_losegroupkey}(G, (), (K_G), \_) \\
 & \vee (\quad \diamondminus (\quad \mathsf{gcks\_sendpushkey}(G, (), (K_G, K'_G), N) \\
 & \qquad\qquad \wedge \diamondminus \mathsf{gcks\_sendpullkey}(G, M_{\mathsf{d}}, (N_{GM}, K''_G, K_{GM}), \_)) \\
 & \quad \wedge \neg \diamondminus (\quad \mathsf{gcks\_sendpushkey}(G, (), (K_G, K'_G), N) \\
 & \qquad\qquad \wedge \diamondminus \mathsf{gcks\_cancel}(G, M_{\mathsf{d}}, (N_{GM}), \_))) \\
 & \vee \diamondminus \mathsf{gcks\_sendpullkey}(G, M_{\mathsf{d}}, (N_{GM}, K_G, K_{GM}), \_) \\
 & \vee (\quad \diamondminus \mathsf{gcks\_losepairwisekey}(G, (), (M, K_{GM}), \_) \\
 & \quad \wedge \diamondminus \mathsf{gcks\_sendpullkey}(G, M, (\_, K_G, K_{GM}), \_))
\end{array}
$$

Even after the meaning of the various symbols has been explained (in Section 2), this requirement is hard to read and hard to compare with other similar requirements of GDOI. Our attempt at pretty-printing this formula does not help. As this example demonstrates, it is important to develop some more user-friendly forms of representation for requirements. This is particularly true as protocols themselves become larger and more complex.

NPATRL requirements, as they are used in conjunction with the NPA, have a tree-like structure. One gives an end goal, such as the intruder's learning a key like in the example above, or a principal accepting a key as genuine, and then defines the sequences of actions that either should or should not precede that event (the formula following $\Rightarrow$). These conditions on sequences of events are defined in terms of the usual logical connectives, "and", "or", and "not", and one temporal operator ($\diamondminus$), which denotes "happened before". Given this treelike structure, and the use of logical connectives, it appears that fault trees [16] could provide a natural candidate for a graphical representation. Fault trees, which were originally developed for use in the analysis of safety-critical systems, have also proved popular as a graphical language for formally specifying software safety requirements. Formal semantics have been provided for fault trees in terms of various logical systems (see [4] for an example and a discussion of the literature). More recently, fault trees have been used for the security analysis of systems [10, 12], although no attempt has been made to provide a formal semantics in this context yet.

Our intended use of fault trees is somewhat different from the above. In most applications to system safety and security, the root node is assumed to be an undesired event: either a system fault or a security violation, while the branches of the tree describe the conditions and events leading up to the fault. In our case, the root node represents an event which may or may not be desirable; what is important is that the event should not occur unless the conditions specified in the branches have previously been satisfied. Thus, while most fault tree semantics interpret precedence in the tree in terms of causality, we will be interpreting precedence in terms of a temporal relationship. Moreover, since we will also be interested in events that should *not* occur before a given event (e.g., if a principal accepts a session key as

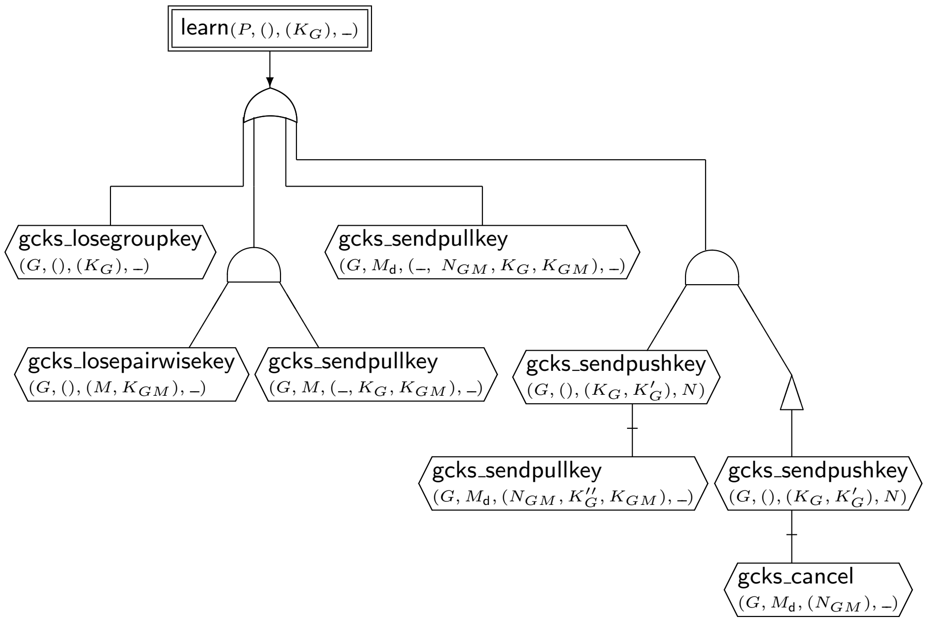

learn$(P, (), (K_G), _)$
gcks_losegroupkey $(G, (), (K_G), \_)$
gcks_sendpullkey $(G, M_{\mathsf{d}}, (\_, N_{GM}, K_G, K_{GM}), \_)$
gcks_losepairwisekey $(G, (), (M, K_{GM}), \_)$
gcks_sendpullkey $(G, M, (\_, K_G, K_{GM}), \_)$
gcks_sendpushkey $(G, (), (K_G, K'_G), N)$
gcks_sendpullkey $(G, M_{\mathsf{d}}, (N_{GM}, K''_G, K_{GM}), \_)$
gcks_sendpushkey $(G, (), (K_G, K'_G), N)$
gcks_cancel $(G, M_{\mathsf{d}}, (N_{GM}), \_)$

Fig. 1. Fault Tree Specification of Strong Secrecy in GDOI

genuine, it should not have accepted that key before), we will need NOT-gates as well as the mainstream AND-gates and OR-gates. While NOT-gates are included in some variations of fault trees, they are not a standard component.

Figure 1 gives a representation inspired by fault trees to the above GDOI requirement. Its meaning will be explained in the body of this paper. For now, it is enough to note that the two-dimensional layout makes the requirements easier to read than the equivalent NPATRL formula given earlier. It consequently simplifies the task of designing requirements and comparing variants.

The remainder of this paper is organized as follows. In Section 2, we recall the NPATRL language and characterize the fragment that is used within the NPA. In Section 3, we give a brief introduction to fault trees and present the variant we will be using. We give an NPATRL semantics for it in Section 4. In Section 5, we present some example NPATRL requirements with their fault tree interpretations. We similarly give a simple pictorial representation of a complex theorem about Kerberos 5 in Section 6. In Section 7, we use precedence trees to schematically highlight the differences between various definitions of authentication and secrecy. Section 8 concludes the paper.

2 The NPATRL Language

We recall the NPATRL specification logic in Section 2.1 and give a natural model semantics for it in Section 2.2. We precisely identify the sublanguage used for writing security requirements in the NPA in Section 2.3, specializing our model semantics to it in Section 2.4.

2.1 NPATRL

The *NRL Protocol Analyzer Temporal Requirements Language* [14], better known as NPATRL (and pronounced "N Patrol"), was originally designed to be a requirements language for the NPA. This formalism makes available the abstract expressiveness of a logical language to specify generic requirements at a high enough level to capture intuitive goals precisely, and yet it can be interpreted in the NPA search engine.

NPATRL requirements are logical expressions whose atomic formulas are *event statements*. An event describes an action by a principal. It either denotes an action by an honest principal or the special learn event that indicates the acquisition of information by the adversary. In NPATRL, an event is represented by a predicate symbol defined on four arguments. The name of the predicate symbol is the name of the event. The first argument is the principal executing the event. The second is a list of names of other principals relevant to the event (for example, the intended recipient of a message). The third argument is a list of other information relevant to the event (for example a key being exchanged). The last argument is the local round number that identifies the sequence of events executed by a principal who is participating in the protocol. Round numbers are unique to each instantiation of a role. Since a principal may engage in a protocol more than once, the round number serves to distinguish different rounds. The use of round numbers thus guarantees that no event occurs more than once.

For example, we consider the case in which an initiator A accepts a key K as good for communicating with a responder B. This would be represented as follows:

$$\mathsf{initiator\_accept\_key}(A, B, K, N)$$

Here and below, symbols starting with a capital letter stand for a variable. For convenience, we omit the parentheses from singleton lists, writing B for (B) and K for (K) in this example.

The logical infrastructure of NPATRL consists of the usual connectives $\neg$, $\wedge$, $\vee$, and $\Rightarrow$, and the temporal modality $\Diamond\!\!\!\!-$ which is interpreted as "happened at some time before" or "previously". $\Diamond\!\!\!\!-$ is *strict* in the sense that the formula it qualifies is expected to hold in the past, but not necessarily in the present. NPATRL is then defined by the following grammar:

$$E \quad ::= \quad a \quad | \quad \neg E \quad | \quad E_1 \wedge E_2 \quad | \quad E_1 \vee E_2 \quad | \quad E_1 \Rightarrow E_2 \quad | \quad \Diamond\!\!\!\!- E$$

where a stands for an event. The variables in an NPATRL specification are implicitly quantified at the front of the clause.

For example, we may have the following requirement involving the $\mathsf{initiator\_accept\_key}$ just defined:

If an initiator A accepts a key K for communicating with another principal B, then a server must have previously sent this key with the idea that it should be used for communications between A and B.

We can construct an expression of the above requirement as follows:

$$\mathsf{initiator\_accept\_key}(A, B, K, \_) \quad \Rightarrow \quad \diamondminus\, \mathsf{svr\_send\_key}(\mathsf{server}, (A, B), K, \_)$$

Here and below, we adopt the Prolog convention of writing "_" for single-occurrence variables whose actual name is unimportant.

For more discussion of event statements and their relation to NPA specifications, see [14].

2.2 A Model Semantics for NPATRL

We will now develop a trace semantics for NPATRL that describes when a given formula is satisfied relative to a sequence of recorded events. This is expressed through the judgment $T \models E$ where E is the formula and T is the sequence of events or *trace*, formally defined as follows:

$$T \quad ::= \quad \cdot \quad | \quad T, a$$

Here, "$\cdot$" stands for the empty trace, a is an event as defined in the previous section but with the requirement that it contains no variables, and "," extends a trace with an event. As suggested earlier, NPATRL traces never contain repeated events. We assume that time flows from left to right so that a is the most recent event of the trace T, a. While this grammar specifies finite length traces, the definitions below apply also to infinite traces.

The above satisfiability judgment is defined by the following rules:

$$\frac{T \models [t/x]E \quad \textit{for all ground } t}{T \models \forall x.E}\,\forall$$

$$\frac{T \models E_1 \qquad T \models E_2}{T \models E_1 \wedge E_2}\,\wedge \qquad \frac{T \models E_i}{T \models E_1 \vee E_2}\,\vee_{i=1,2} \qquad \frac{T \not\models E}{T \models \neg E}\,\neg$$

$$\frac{}{T, a \models a}\,\textbf{atom} \qquad\qquad \frac{T \models E}{T, T', a \models \diamondminus E}\,\diamondminus$$

The rules in the two upper rows reduce the satisfiability of a formula with a traditional logical symbol as its main operator to the satisfiability (or lack thereof) of its immediate subformulas, relative to the same trace. In the rule for the (implicit and extensional) universal quantifier, t shall be a ground term. Here, we denote with $T \not\models E$ the unsatisfiability of E w.r.t. T which we interpret as a meta-theoretic notion (although this judgment could be defined via proper rules). The interpretation of implication is as usual derived from that of $\neg$ and either $\wedge$ or $\vee$.

The last two rules define the temporal interpretation of NPATRL. An atomic event is satisfiable only if it appears as the last event in the trace at hand. A temporal formula $\diamondminus E$ is satisfiable in the current trace (T, T', a) only if E is satisfiable in some trace T obtained by stripping a non-empty suffix (T', a) from the trace at hand. Notice that this non-emptiness requirement realizes the strict interpretation of $\diamondminus$ [8].

2.3 NPA-Acceptable NPATRL Expressions

When NPATRL is used in the context of the NPA, negations of the NPATRL requirements are used to specify states that the NPA attempts to prove unreachable. Although NPATRL was originally designed to be used with the NPA, it is actually much more expressive than the set of specifications whose negations are accepted by the tool. Indeed, the NPA is optimized for assessing two important classes of requirements within a security protocol, secrecy, and authentication. In this section, we identify the sublanguage of NPATRL actually used by NPA while performing these tasks, and outline the reasons for this symbiosis. Characterizing this linguistic fragment, and motivating it on model-theoretic grounds, is a secondary contribution of this paper. This is the language we will be focusing on in the remainder of this paper.

The language of NPA-acceptable NPATRL expressions is given by the following grammar. We will refer to this language as $\mathbf{NPATRL_{NPA}}$.[3]

$$\begin{array}{lcl}
R & ::= & a \Rightarrow F \\
F & ::= & E \;\mid\; \neg E \;\mid\; F_1 \wedge F_2 \;\mid\; F_1 \vee F_2 \\
E & ::= & \diamondminus a \;\mid\; \diamondminus(a \wedge F)
\end{array}$$

There are two main restrictions with respect to NPATRL: first a top-level $\mathbf{NPATRL_{NPA}}$ formula shall always have the structure $a \Rightarrow F$. This derives from the fact that secrecy and authentication requirements almost invariably have this form (see above), and therefore it has been built into the NPA. This class may need to be extended to accommodate the specification of other types of protocols.

The second main difference w.r.t. NPATRL is the layering of the language over the non-terminals F and E. In particular, notice that every occurrence of the $\diamondminus$ operator is bound to an unshakable instance of an atomic event a. This restriction is due to the fact that the NPA uses these formulas for model-checking a protocol specification: upon encountering a $\diamondminus$ operator, it will skip a portion of the (dynamically generated) trace. The event a in E identifies a place in the current trace from which to continue with the verification of the subformula F (if present). This is a huge efficiency gain as it may otherwise have to verify F after skipping each event of the trace. A more formal explanation is provided in Section 2.4.

[3] This language generalizes and rationalizes the initial attempt in [2]. In particular, it fully characterizes the set of NPATRL expressions that can be processed by NPA.

Next, we will show how $\mathbf{NPATRL_{NPA}}$ is powerful enough to express a number of recurring linguistic constructions for protocol analysis.

Many commercial protocols implement optional behaviors: a principal can either run a streamlined version of the protocol, or choose to take optional steps, typically to ensure that the exchange satisfy stronger requirements. This calls for a subformula F of the form

$$\diamondminus\, opt \Rightarrow \diamondminus(opt \wedge F_{opt})$$

where opt the optional event, and F_{opt} is the corresponding optional property. Using simple propositional equivalences, we can transform this formula into

$$\neg\diamondminus\, opt \vee \diamondminus(opt \wedge F_{opt}),$$

which belongs to $\mathbf{NPATRL_{NPA}}$. Observe that this formula is not equivalent to $\diamondminus\,(opt \Rightarrow F_{opt})$, which is not expressible in $\mathbf{NPATRL_{NPA}}$.

Note that an alternative way to represent a requirement R with an optional behavior $\diamondminus\, opt \Rightarrow \diamondminus(opt \wedge F_{opt})$ (written $R[\diamondminus\, opt \Rightarrow \diamondminus(opt \wedge F_{opt})]$ where we indicate a formula R with subformula F as $R[F]$) is to split it into two requirements: one that describes situations in which opt does not occur (this is $R[\neg\diamondminus\, opt]$) and one that spells out the consequences of opt taking place (that is $R[\diamondminus\,(opt \wedge F_{opt})]$). However having a single requirement is often more self-contained, and therefore easier to read.

Although more rarely used, we may be interested in disjunctive targets, *i.e.*, subformulas of the form $\diamondminus\,((a \vee b) \wedge F_{a\vee b})$. Since $\vee$ distributes over $\wedge$, and $\diamondminus$ distributes over $\vee$ (a simple exercise), it is easy to transform this expression into the $\mathbf{NPATRL_{NPA}}$ formula $\diamondminus(a \wedge F_{a\vee b}) \vee \diamondminus(b \wedge F_{a\vee b})$.

2.4 Model Checking NPA-Acceptable NPATRL Expressions

On the basis of these considerations, we can specialize the trace semantics of NPATRL to the specifics of $\mathbf{NPATRL_{NPA}}$ formulas. The three lines below describe the behavior of each of the three productions in the grammar of this language, with the addition of the implicit universal quantifiers possibly prefixing the top-level R non-terminal. For simplicity, we keep the judgment form and the name of the rules the same, to the extent possible.

$$\frac{T \models [t/x]R \;\; \textit{for all ground } t}{T \models \forall x.R}\,\forall \qquad \frac{T, a \models F}{T, a \models a \Rightarrow F}\,{\Rightarrow_1} \qquad \frac{(T{=}\cdot \text{ or } T{=}T',a' \text{ with } a{\neq}a')}{T \models a \Rightarrow F}\,{\Rightarrow_2}$$

$$\frac{T \models F_1 \qquad T \models F_2}{T \models F_1 \wedge F_2}\,\wedge \qquad \frac{T \models F_i}{T \models F_1 \vee F_2}\,\vee_{i=1,2} \qquad \frac{T \not\models F}{T \models \neg F}\,\neg$$

$$\frac{}{T, a, T', a' \models \diamondminus a}\,\diamondminus_1 \qquad \frac{T, a \models F}{T, a, T', a' \models \diamondminus(a \wedge F)}\,\diamondminus_2$$

It can easily be verified that these rules are derivable from the inferences in Section 2.2, guided by the grammatical productions of $\mathbf{NPATRL_{NPA}}$.

Observe that the last two rules are free of the inherent non-determinism of rule $\diamondminus$ in Section 2.2 as they anchor every jump in the past to the specific event a that, as we remarked, can occur at most once in a trace. This turns this simple semantics into an effective model-checking procedure, whose only remaining form of non-determinism is the "don't know" branching in rule $\vee_i$. This is essentially the procedure that is implemented in the NPA.

3 Fault Trees Representation of NPATRL Requirements

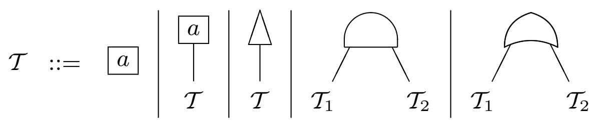
We will now introduce fault trees in Section 3.1 and then define the specialized notion of precedence tree in Section 3.2 that we will later use for specifying requirements for security protocols.

3.1 Fault Trees

Fault trees [16] were originally introduced to facilitate the safety analysis of system designs. A fault tree has as its root the description of a failure situation the system designer wishes to avoid. Inner nodes represent the conditions or events that enable the fault. These children will be roots of subtrees that define the conditions and events that enable them, and so on. Structures isomorphic to fault trees have also been used to specify conditions that should be met in order for an event to take place. For example, a simple tree will describe the constraint "*A passenger needs a ticket and a photo ID to board a plane, but should not carry a weapon*" as soon as we have defined these objects.

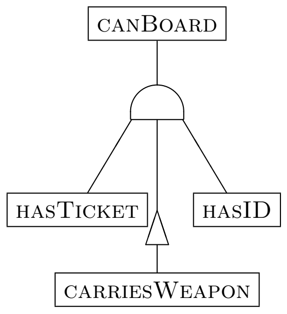
The nodes in a fault tree can be either basic events (defined below) or *logical gates*. The most widely used combinators are the AND-gate (⌓) and the OR-gate (⌒). Edges link each of them them to a top node and to two children. They specify that the event in the top node can occur only when all (resp., at least one) of the situations expressed by the children nodes takes place. Another combinator node we will use is the NOT-gate (△), which specifies that the event in the top node can occur only when the situation described by the single child node does not. For our purposes, *events* will be first-order atomic formulas akin to those already used in NPATRL (*e.g.*, canBoard). We enclose them in a box in our graphical representation: the last example becomes $\boxed{\text{canBoard}}$.

We can formalize the above definition of a fault tree by means of the following tree grammar:



Edges have a double meaning in a fault tree: when the parent node is a gate, they act as simple connectors. However, when the parent node is an event, they typically denote either implication or equality (actually if-and-only-if), depending on the semantics chosen.

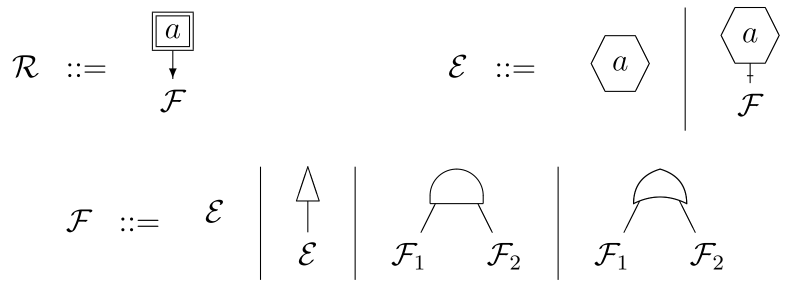
Sequences of AND-gates are often represented as an n-ary AND-gate (with $n \geq 2$). This can be viewed as a canonical form that abstracts associativity and commutativity issues away. An analogous convention is applied to OR-gates. We rely on this simplified syntax in the tree adjacent to this text, that specifies the naive requirement about boarding a plane stated above.



3.2 Precedence Trees

A direct, syntax-oriented, translation of NPATRL in the language of fault trees is an easy exercise, as the precedence operator $\diamondminus$ could be handled by adding a dedicated gate. This simple solution has drawbacks when focusing on the language of NPA-acceptable NPATRL: first, the prevalent E-formulas would be clumsily rendered by long chains of AND-gates and the new $\diamondminus$-gate; second, we would be able to draw trees that do not correspond to formulas in the restricted language. For these reasons, we will define a notion of fault trees specialized to the language of NPA-acceptable NPATRL expressions

We introduce *precedence trees*, a variant of fault trees aimed at giving a graphical representation to NPA-acceptable NPATRL expression. Similarly to $\mathbf{NPATRL_{NPA}}$ expressions, we will layer the tree grammar for precedence trees in three classes, that we label with the calligraphic letter corresponding to the syntactic categories R, F, and E of these objects (the translations in Section 4 will actually define a direct mapping between homonymous classes). The following grammatical productions define precedence trees:



In [2], we rendered precedence trees as a subclass of fault trees, keeping the syntax unaltered. This led to three interpretations of edges, which depended on the nature of their origin and destination. This also led to confusion among first-time readers. For this reason, we break with the tradition and graphically distinguish objects on the basis of their interpretation. In particular, precedence trees are drawn with three types of edges, and two shapes of nodes (besides the connectives).

The root of a precedence tree is enclosed in a double square box (for emphasis) and the single edge departing from it is decorated with an arrow, suggestive of an implication. Since all other actions are introduced by the $\diamondminus$ modality, we write them in an hexagonal box. For conciseness, we display formulas of the form $\diamondminus(a \wedge F)$ by having an edge with a horizontal bar (suggestive of a +) connect the box containing a to the subtree representing F. Connectives and the edges radiating out of them are left unchanged.

4 NPATRL Semantics

In this section, we define the translation from $\mathbf{NPATRL_{NPA}}$ to precedence trees. The inverse mapping is straightforward and therefore left as an exercise to the reader. These translations are important components of systems that use human-readable precedence trees to enter and visualize machine-efficient NPATRL requirements.

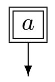
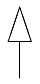
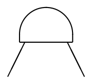
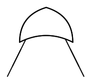
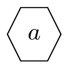
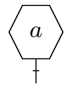
$\ulcorner R \urcorner^r$	$\ulcorner F \urcorner^f$				$\ulcorner E \urcorner^e$	
$a \Rightarrow F$	E	$\neg E$	$F_1 \wedge F_2$	$F_1 \vee F_2$	$\diamondminus a$	$\diamondminus(a \wedge F)$
 $\ulcorner F \urcorner$	$\ulcorner E \urcorner$	 $\ulcorner E \urcorner$	 $\ulcorner F_1 \urcorner$ $\ulcorner F_2 \urcorner$	 $\ulcorner F_1 \urcorner$ $\ulcorner F_2 \urcorner$		 $\ulcorner F \urcorner$

Fig. 2. Translating $\mathbf{NPATRL_{NPA}}$ to Precedence Trees

We first define a family of three representation functions $\ulcorner \_ \urcorner^c$, with $c \in \{r, f, e\}$, that transform an $\mathbf{NPATRL_{NPA}}$ R-, F- and E-expression N into a precedence tree $\ulcorner N \urcorner^c$ of the appropriate type. We will generally keep the superscript c implicit as it will typically be possible to deduce it from the context. These translations are displayed in Figure 2: for every syntactic category defining $\mathbf{NPATRL_{NPA}}$ (top row), we give a mapping between each grammatical production in this category (middle row) and a precedence tree fragment (bottom row). By construction, $\ulcorner \_ \urcorner^c$ is total for each c. It is then easy to prove that it is an adequate encoding of $\mathbf{NPATRL_{NPA}}$ into precedence trees (and vice versa), as formalized by the following property, which can be easily derived:

Property 1.

The translation $\ulcorner \_ \urcorner^c$ is a bijection between precedence trees and $\mathbf{NPATRL_{NPA}}$ expressions of category $c \in \{r, f, e\}$.

5 Expressing GDOI Requirements as Precedence Trees

In this section, we describe some requirements similar to those that came up in our analysis of the Group Domain of Interpretation (GDOI) protocol [8] to demonstrate the benefits of fault trees over NPATRL for communicating non-trivial requirements. This protocol had some extremely complex requirements, especially for secrecy, that could be stated very precisely in NPATRL, but that we sometimes had difficulties reading back and explaining to others. However, even these requirements turned out to be fairly straightforward to understand once we reduced them to a visual representation.

GDOI is a protocol intended to support secure many-to-one communication. A typical application would be subscription to some digital service such as movies. In GDOI, a group controller distributes keys and key updates to group members. The group controller is also responsible for enrolling new members and giving them the current group key, as well as expunging members that leave the group.

A member joins the group by initiating a handshake protocol with the controller using a *pairwise key* shared between them. As a result of the handshake, the controller sends the current *group key* to the member. We wish to be sure that the member always gets the most current group key, but as we will see the exact meaning of this requirement is subtle. Another type of message is used to send new group keys to current group members. This is known as the *Groupkey Push* message.

We will consider two types of requirements: freshness requirements and secrecy requirements. The first we will set forth in some detail; the second we will describe in broad outline.

For the freshness requirements, the following types of data are relevant: the controller name, denoted by G, the member name, denoted by M, the pairwise key, denoted by K_{GM}, and group keys, denoted by K_{new},K_{old}, K, K', and K''. All terms are variables universally quantified over their respective types. Again, the symbol _ is a "don't-care" symbol. We have four types of events:

1. $\mathsf{member\_acceptkey}(M, G, (K_{GM}, K), N)$ describes a new member M accepting a key K as a result of the handshake protocol.
2. $\mathsf{gcks\_losepairwisekey}(G, (), (M, K_{GM}), N)$ describes the compromise of the pairwise key K_{GM} of G and M.
3. $\mathsf{gcks\_createkey}(G, (), (K_{new}, K_{old}), N)$ describes the controller creating and distributing new key K_{new} and making an old key K_{old} obsolete.
4. $\mathsf{member\_requestkey}(M, G, (), N)$ describes a member M requesting to join a group.

We consider two anti-replay requirements for the handshake protocol. The first says that,

if a member accepts a key from the controller in a protocol run, no newer key should have been distributed prior to the member's request.

This we call *recency freshness*, since it says that the member should accept the most recently generated key. We express it as the formatted NPATRL statement below.

$$\begin{aligned}
&\mathsf{member\_acceptkey}(M, G, (K_{GM}, K_{old}), N)\\
&\Rightarrow \quad \Diamond\mathsf{gcks\_losepairwisekey}(G, (), (M, K_{GM}), \_)\\
&\quad \vee \neg(\Diamond\ (\quad \mathsf{member\_requestkey}(M, G, (), N)\\
&\qquad\qquad \wedge \Diamond\mathsf{gcks\_createkey}(G, (), (K_{new}, K_{old}), \_)))
\end{aligned}$$

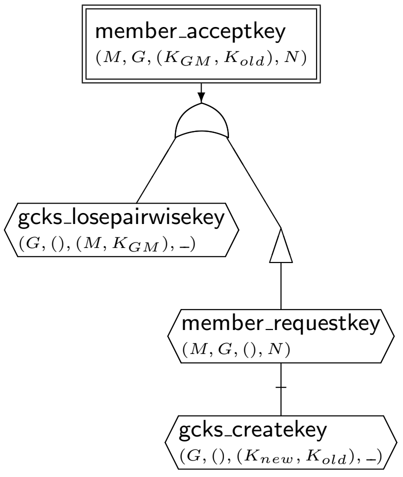
The second says that,

if a member accepts a key from the group controller in a protocol run, then it should not have previously accepted a later key.

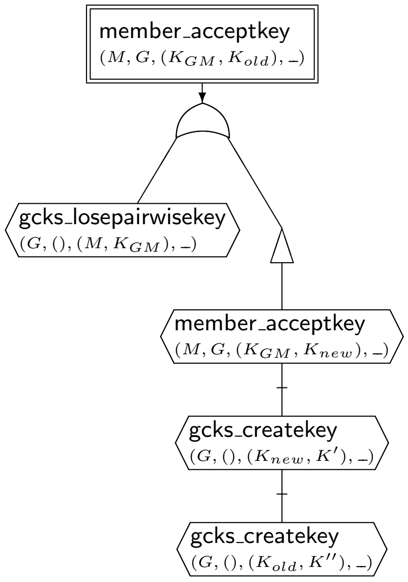
This we call *sequential freshness*, since it is a requirement on the order in which keys are accepted. It is formalized by the following NPATRL expression [8]:

$$
\begin{aligned}
&\mathsf{member\_acceptkey}(M, G, (K_{GM}, K_{old}), \_) \\
\Rightarrow\ &\diamondminus \mathsf{gcks\_losepairwisekey}(G, (), (M, K_{GM}), \_) \\
&\vee \neg(\diamondminus\ (\ \mathsf{member\_acceptkey}(M, G, (K_{GM}, K_{new}), \_) \\
&\qquad \wedge \diamondminus\ (\ \mathsf{gcks\_createkey}(G, (), (K_{new}, K'), \_) \\
&\qquad\qquad \wedge \diamondminus \mathsf{gcks\_createkey}(G, (), (K_{old}, K''), \_))))
\end{aligned}
$$

Figure 3 displays the precedence trees corresponding to these two forms of freshness.


member_acceptkey
$(M, G, (K_{GM}, K_{old}), N)$
gcks_losepairwisekey
$(G, (), (M, K_{GM}), \_)$
member_requestkey
$(M, G, (), N)$
gcks_createkey
$(G, (), (K_{new}, K_{old}), \_)$

(a) Recency Freshness


member_acceptkey
$(M, G, (K_{GM}, K_{old}), \_)$
gcks_losepairwisekey
$(G, (), (M, K_{GM}), \_)$
member_acceptkey
$(M, G, (K_{GM}, K_{new}), \_)$
gcks_createkey
$(G, (), (K_{new}, K'), \_)$
gcks_createkey
$(G, (), (K_{old}, K''), \_)$

(b) Sequential Freshness

Fig. 3. Freshness Requirements for the GDOI Pull Protocol

We note that each tree describes two possible conditions under which the final event (the member's accepting a key) should be reachable. One describes a safety condition; if the final event occurs, then a certain sequence of events should not have occurred in the past. But the other describes a condition, the compromise of a pairwise key, under which we can make no guarantees. In our analysis of the GDOI protocol, especially of the secrecy requirements, we found many such conditions, under which the protocol could either make no guarantees or only partial guarantees. For example, the *perfect forward secrecy* condition says that, if the pairwise key is compromised, then the intruder can learn group keys generated after the compromise, but not before. Many of these conditions interacted with each other, making it difficult to specify them correctly. We found that a graphical representation made it much easier to keep these conditions straight, and to mix and match the different conditions, thus providing a useful tool for requirement composition.

Figure 4 visually illustrates the discussion in the previous paragraph by showing a tree representing a requirement consisting of five such conditions that cropped up in our analysis of GDOI. Of particular interest are the properties of backward and forward access control. Backward access control means that a group member does not learn previous keys upon joining a group; forward access control means that she does not learn new keys after leaving it. These can be generalized to mean that possession of a group key does not imply possession of keys generated before that key (backward access control) or after it (backward access control).

The conditions in the tree are expressed in terms of the ways in which intruder P can learn a group key K_G. We will not go into as much detail as for the freshness requirements, but will give the general outline of the requirement here.

1. The first condition says that if the intruder learns a new key K_G', then it can learn K_G if it was generated previously to K_G'; this is the absence of backward access control.

$$\mathsf{learn}(P, (), (K_G), \_) \Rightarrow \mathsf{BI}(K_G) \vee \mathsf{BC1}(K_G) \vee \mathsf{BC2b}(K_G) \vee \mathsf{BC2a}(K_G) \vee \mathsf{BC3a}(K_G) \qquad \textit{where}$$

$\mathsf{BI}(K_G) =$
$\quad \Diamonddot\, \mathsf{learn}(P, (), (K'_G), \_)$
$\quad \wedge \Diamonddot\, (\; \mathsf{gcks\_createkey}(G, (), (K'_G), \_)$
$\qquad \wedge \Diamonddot\, \mathsf{gcks\_createkey}(G, (), (K_G), \_))$

$\mathsf{BC1}(K_G) =$
$\quad \Diamonddot\, \mathsf{gcks\_losegroupkey}(G, (), (K_G), \_)$

$\mathsf{BC2a}(K_G) =$
$\quad \Diamonddot\, (\; \mathsf{gcks\_sendpushkey}(G, (), (K_G, K'_G), N)$
$\qquad \wedge \Diamonddot\, \mathsf{gcks\_sendpullkey}(G, M_{\mathsf{d}}, (N_{GM}, K''_G, K_{GM}), \_))$
$\quad \wedge \neg \Diamonddot\, (\; \mathsf{gcks\_sendpushkey}(G, (), (K_G, K'_G), N)$
$\qquad \wedge \Diamonddot\, \mathsf{gcks\_cancel}(G, M_{\mathsf{d}}, (N_{GM}), \_))$

$\mathsf{BC2b}(K_G) =$
$\quad \Diamonddot\, \mathsf{gcks\_sendpullkey}(G, M_{\mathsf{d}}, (N_{GM}, K_G, K_{GM}), \_)$

$\mathsf{BC3a}(K_G) =$
$\quad \Diamonddot\, \mathsf{gcks\_losepairwisekey}(G, (), (M, K_{GM}), \_)$
$\quad \wedge \Diamonddot\, \mathsf{gcks\_sendpullkey}(G, M, (\_, K_G, K_{GM}), \_)$

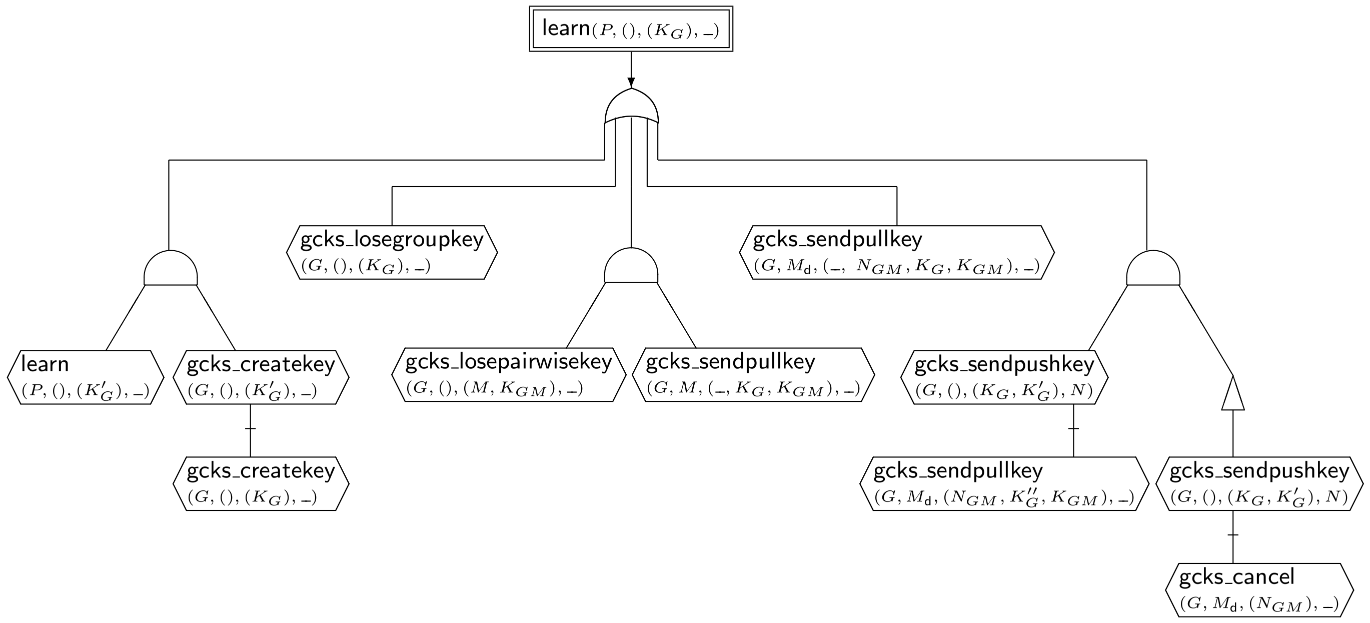

learn$(P, (), (K_G), _)$
learn $(P, (), (K'_G), \_)$
gcks_createkey $(G, (), (K'_G), \_)$
gcks_createkey $(G, (), (K_G), \_)$
gcks_losegroupkey $(G, (), (K_G), \_)$
gcks_losepairwisekey $(G, (), (M, K_{GM}), \_)$
gcks_sendpullkey $(G, M, (\_, K_G, K_{GM}), \_)$
gcks_sendpullkey $(G, M_{\mathsf{d}}, (\_, N_{GM}, K_G, K_{GM}), \_)$
gcks_sendpushkey $(G, (), (K_G, K'_G), N)$
gcks_sendpullkey $(G, M_{\mathsf{d}}, (N_{GM}, K''_G, K_{GM}), \_)$
gcks_sendpushkey $(G, (), (K_G, K'_G), N)$
gcks_cancel $(G, M_{\mathsf{d}}, (N_{GM}), \_)$

Fig. 4. Forward Access Control without Backward Access Control in GDOI

2. The second condition says that the intruder can learn the key if it is compromised (expressed using the gcks_losegroupkey event).

3. The third condition says that the intruder can learn the key if a pairwise key between the controller and an honest group member is compromised and the key is distributed to the group member using the pairwise key when the member joins the group.

4. The fourth condition says that the intruder can learn the key if it is distributed to a dishonest member M_{d} (who is assumed to share all information with the intruder) when it joins the group. Joining the group is designated by the gcks_sendpullkey event, in which the key is sent to the joining member.

5. The fifth condition says that the intruder can learn the key if it is distributed to the group as a whole (designated by the gcks_sendpushkey event) after a dishonest member has joined the group (designated by the gcks_sendpullkey event) but before it has left (designated by the gcks_cancel event).

Note that we could modify the tree in a number of ways if we wanted to change the requirements. If we wanted to introduce backward access control, we could eliminate the first condition. If we wanted to replace backward access control by forward access control, we could replace the first condition by one in which the intruder's learning K'_G precedes the learning of K_G. If we wanted to introduce perfect forward secrecy, we could alter the third condition so that the compromise of the pairwise key must precede its use to distribute the group key to the new member. The tree structure makes it easy for us to do this. By contrast, the textual representation often proved error-prone.

Our last example illustrates the use of optional behaviors: the normal sequence of actions that leads a member to accept a key in the pull protocol consists of a request by this member, and the prior creation of the key by the group controller. Optionally, the group controller can make the disclosure of the group key contingent on the member submitting a certificate known as proof-of-possession (PoP) [8]. In this case, the member must submit the PoP before he can accept the key.

In order to model this optional action, we introduce two more events:

1. $\mathsf{gcks\_requestpop}(G, M, K_{GM}, N)$ describes the group controller G requesting a proof-of-possession from a requesting member M with shared key K_{GM}.
2. $\mathsf{member\_sendpop}(M, G, K_{GM}, N)$ describes a candidate member M sending a proof-of-possession as part of the handshake protocol.

The NPATRL statement describing the handshake in the pull protocol with the optional proof-of-possession exchange is given in in the following:

$$\begin{array}{ll}
& \mathsf{member\_acceptkey}(M, G, (K_{GM}, K), \_) \\
\Rightarrow & \diamond(\ \mathsf{member\_requestkey}(M, G, (), N) \\
& \quad \wedge \diamond \mathsf{gcks\_createkey}(G, (), (K, \_), \_)) \\
\wedge & \left(\begin{array}{l}
\diamond \quad \mathsf{member\_sendpop}(M, G, K_{GM}, \_) \\
\Rightarrow \diamond(\quad \mathsf{member\_sendpop}(M, G, K_{GM}, \_) \\
\qquad \wedge \diamond(\ \ \mathsf{gcks\_requestpop}(G, M, K_{GM}, \_) \\
\qquad\qquad \wedge \diamond \mathsf{member\_requestkey}(M, G, (), N)))
\end{array} \right)
\end{array}$$

where the embedded implication describes the optional PoP request and transmission. Note that the round number N ensures that both mentions of the event member_requestkey refer to the same instance. As described in Section 2, a simple logical transformation allows rewriting this expression into the fragment of NPATRL accepted by the NPA. The precedence tree corresponding to the resulting formula is given in Figure 5.

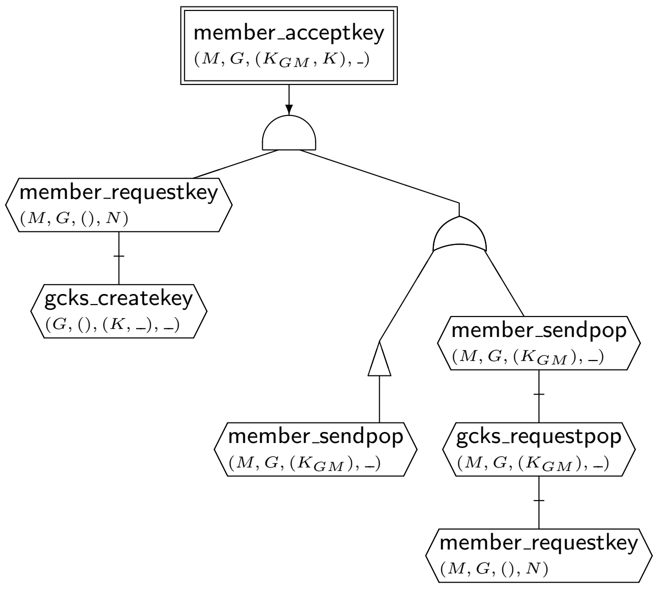


Fig. 5. Accepting a Key with Optional Proof-of-Possession

We propose an abbreviated representation for this requirement in Figure 6: the dotted line introduces the optional behavior and is followed by the embedded arrow originating from the member_sendpop node; the two references to the common member_requestkey have been collapsed in just one node.

6 Expressing Kerberos 5 Requirements as Precedence Trees

In this section, we examine a second example, a fragment of the specification of the expected behavior of the Kerberos 5 protocol, as described in [1]. While the GDOI requirements

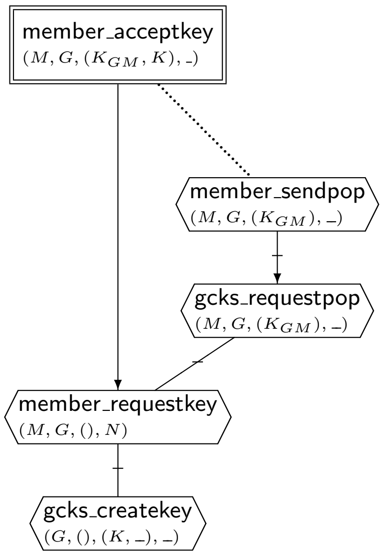


Fig. 6. Accepting a Key with Optional Proof-of-Possession

studied in the previous section drew from a model checking context, we will now be working with expressions meant for mathematical proofs, possibly using a theorem prover. We will see that it makes little difference at the level of specification. The complexity here derives not from the subtleties of a little-explored class of protocols (as with GDOI), but from a highly detailed formalization that tries not to leave anything undefined.

The central result in that paper is the following *data origin authentication* theorem, reported verbatim from [1]:

Theorem 1. *For* C : client, T : TGS, $C, T \neq \mathsf{I}$, S : server, k_C : $\mathsf{dbK}\ C$, k_T : $\mathsf{dbK}\ T$, $AKey$: $\mathsf{shK}\, C T$, *and* n_2 : nonce, *if the beginning state of a finite trace does not contain* $\mathsf{I}(k_C)$, $\mathsf{I}(k_T)$, *or any fact* F *with* $\rho_{k_T}(F; AKey, C) > 0$ *or* $\rho_{AKey}(F; C) > 0$, *and at some point in the trace* T *fires rule* $\alpha_{4.1}$, *consuming the fact* $\mathsf{N}(\{AKey, C\}_{k_T}, \{C\}_{AKey}, C, S, n_2)$, *then earlier in the trace, some* K : KAS *fired rule* $\alpha_{2.1}$, *existentially generating* $AKey$ *and producing the fact* $\mathsf{N}(C, \{AKey, C\}_{k_T}, \{AKey, n, T\}_{k'})$ *for some* n : nonce *and* k' : $\mathsf{dbK}\, C$. *Also, after* K *fired this rule and before* T *fired the rule in the hypothesis,* C

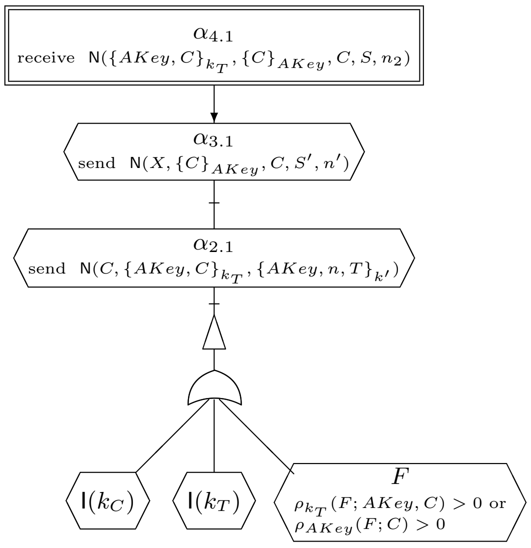

$\alpha_{4.1}$
receive $\mathsf{N}(\{AKey, C\}_{k_T}, \{C\}_{AKey}, C, S, n_2)$
$\alpha_{3.1}$
send $\mathsf{N}(X, \{C\}_{AKey}, C, S', n')$
$\alpha_{2.1}$
send $\mathsf{N}(C, \{AKey, C\}_{k_T}, \{AKey, n, T\}_{k'})$
$\mathsf{I}(k_C)$
$\mathsf{I}(k_T)$
F
$\rho_{k_T}(F; AKey, C) > 0$ or
$\rho_{AKey}(F; C) > 0$

Fig. 7. Data Origin Authentication in the Ticket Granting Exchange of Kerberos 5

fired rule $\alpha_{3.1}$ *to create the fact* $\mathsf{N}(X, \{C\}_{AKey}, C, S', n')$ *for some* X : msg, S' : server, *and* n' : nonce.

A diligent reading of [1] gives meaning to the symbology, but this will be only a small step towards digesting this bulky theorem. The precedence tree in Figure 7 captures the essence of this property in a much more direct way.

A close inspection of this statement classifies its lexical forms into three categories:

– About half of the text is typing information of the form $x : \tau$, where x is a parameter and τ its requested type. These declarations not only list all the parameters that are used in the statement, but constrain their admissible values to objects of the given type.

  This corresponds, and slightly generalizes, the implicit quantification of variables in an NPATRL expression or a precedence tree. We will not take any particular action to model this aspect, although nothing would prevent us from captioning the tree in Figure 7 with a legend that systematically listed all such declarations.

- Theorem 1 mentions three rules, $\alpha_{4.1}$, $\alpha_{3.1}$ and $\alpha_{2.1}$, that are required to be executed in sequence. It also reports on three messages, $\mathsf{N}(\{AKey, C\}_{k_T}, \{C\}_{AKey}, C, S, n_2)$, $\mathsf{N}(X, \{C\}_{AKey}, C, S', n')$ and $\mathsf{N}(C, \{AKey, C\}_{k_T}, \{AKey, n, T\}_{k'})$, that are either produced or consumed by the execution of these rules.

  Rules, together with the parameters appearing in the relevant messages, immediately correspond to NPATRL events. We display them slightly informally as such in Figure 7. Observe that precedence trees are particularly well suited for expressing sequencing constraints, as in this case.

- The first few lines of Theorem 1 lists three initial conditions that should be satisfied in any trace worth considering: that it does not contain $\mathsf{I}(k_C)$, $\mathsf{I}(k_T)$, or any fact F with $\rho_{k_T}(F; AKey, C) > 0$ or $\rho_{AKey}(F; C) > 0$. Facts of the form $\mathsf{I}(\_)$ closely corresponds to the learn events of NPATRL, while a constraint of the form $\rho_k(F; m)$ counts the number of encryptions with key k are needed to access message m in fact F. The interested reader is referred to [1] for further details.

  We simply require that none of the events that can be excised from these constraints takes place before $\alpha_{2.1}$ in the trace at hand. Notice that, by doing so, we are considering a stronger statement. An alternative way of handling initial conditions is outlined in Appendix A,

Although it omits some information, the precedence tree in Figure 7 provides an immediate visual aid to understanding the statement of Theorem 1. We believe, without taking this idea any further in this paper, that a tool that can generate a precedence tree from the formalization of a postulated result, and vice versa, would be very useful.

7 A Hierarchy of Authentication Specifications

We conclude our endeavor in presenting the usefulness of precedence trees with an example that highlights the intuitive discriminating power of a graphical representation, even when the individual specification schemes are rather simple. We will be considering a few of the definitions of *authentication* surveyed in [13], many of which were originally stated in [5]. Authentication has many faces, we are told, with names such as *aliveness*, *weak/non-injective/full agreement*, *correspondence assertions*, *matching histories*, and more. Qualifiers, *e.g.*, *recent* or *mutual*, may add a twist to the meaning. It is difficult to even for an expert to keep all these straight. We have charted schematic precedence trees for five of these definitions in Figures 8 and 9, adding a splash of gray wherever one admits a "recent" variant.

Before going any further, we shall lay down our conventions and assumptions. We consider two-party protocols (although this can easily be generalized to three or more). We will take the point of view of an initiator A authenticating a responder B (the symmetric question, where the responder authenticates the initiator, is treated similarly). We model the initiator's role as a sequence of events $\boldsymbol{x} = (x_0, \ldots, x_n)$ and the responder's role by a dual sequence $\bar{\boldsymbol{x}} = (\bar{x}_0, \ldots, \bar{x}_n)$. An event can be a send or receive step, but also, as in [5], a comment to indicate that the protocol has reached a certain point. Finally, as protocols are parametric, we will often write an initiator event as $x_i(A, B, \boldsymbol{t}_i, N)$, where A is the principal executing the role, B is the principal A believes she is talking to, $\boldsymbol{t}_i$ lists a subset of the other parameters of the exchange (*e.g.*, keys, nonces, etc.), and N is the round number. These are exactly NPATRL events. Responder events will be similarly parameterized as $\bar{x}_i(B, A, \boldsymbol{t}_i, N)$, with

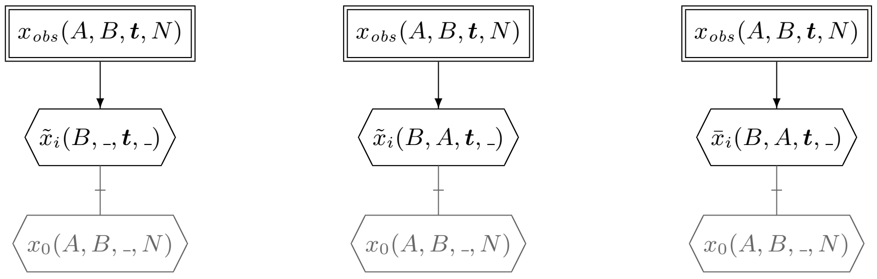

$x_{obs}(A,B,\boldsymbol{t},N)$
$\tilde{x}_i(B,\_,\boldsymbol{t},\_)$
$x_0(A,B,\_,N)$
$x_{obs}(A,B,\boldsymbol{t},N)$
$\tilde{x}_i(B,A,\boldsymbol{t},\_)$
$x_0(A,B,\_,N)$
$x_{obs}(A,B,\boldsymbol{t},N)$
$\bar{x}_i(B,A,\boldsymbol{t},\_)$
$x_0(A,B,\_,N)$

Fig. 8. Recent Aliveness, Weak Agreement, and Non-Injective Agreement

an obvious swap of roles. Sometimes, we will work with events which may belong to either role. We will indicate them with $\tilde{x}_i$.

The most basic definition of authentication is that of *aliveness*, that reads as follows [5]:

We say that a protocol guarantees to an initiator A aliveness *of another agent B if, whenever A (acting as initiator) completes a run of the protocol, apparently with responder B, then B has previously been running the protocol.*

It is observed in [5] that, while B is expected to have run the protocol, it may not have done so with A, and not necessarily as a responder. The recent variant requires that B has been active "recently", for example after A started executing this run of the protocol.

Our rendering of this notion in Figure 8 (left) models a slightly more flexible definition. First, we position ourselves right after A has completed some action x_{obs}, which may be the last event of her role, but may also be an intermediate observation point. Second, we model B's participation by some action $\tilde{x}_i$ chosen to indicate that he did run the protocol. Note that we make no assumption as to whom he believes he is talking to. Rather than committing to a specific action, we could ask that B had executed one out of a subset $\tilde{X}$ of the admissible actions among $\boldsymbol{x}$ or $\bar{\boldsymbol{x}}$, or even, in the multi-protocol case, that B had executed any action F. Third, we allow that A's and B's runs may agree on some data $\boldsymbol{t}$.

Aliveness simply requires that whenever the current trace contains the event $x_{obs}(A, B, \boldsymbol{t}, N)$, then the event $\tilde{x}_i(B, \_, \boldsymbol{t}, \_)$ has previously occurred in this trace. The recent variant adds the further restriction that the initial event of A's run, $x_0(A, B, \_, N)$ precedes B's activity.

The next definition of authentication in [5] is *weak agreement.* Its definition is as follows:

> *We say that a protocol guarantees to an initiator A* weak agreement *with another agent B if, whenever A (acting as an initiator) completes a run of the protocol, apparently with responder B, then B has previously been running the protocol, apparently with A.*

For clarity, we have underlined the differences from aliveness. This definition eliminates the possibility that B was running the protocol with some third agent.

The corresponding precedence tree in figure 8 (center) differs from the case of aliveness only by the presence of A instead of “_” in the second argument of the central node.

Some more assurance is provided by *non-injective agreement* [5]:

> *We say that a protocol guarantees to an initiator A* non-injective agreement *with a responder* B *on a set of data items* $\boldsymbol{t}$ *(where* $\boldsymbol{t}$ *is a set of free variables appearing in the protocol description) if, whenever A (acting as an initiator) completes a run of the protocol, apparently with responder B, then B has previously been running the protocol, apparently with A, and B was acting as responder in his run, and the two agents agreed on the data values corresponding to the variables in* $\boldsymbol{t}$.[4]

The associated precedence tree (Figure 8, right) differs from weak agreement by forcing B to use a responder action $\bar{x}_i$ rather than a generic protocol move $\tilde{x}_i$.

The next step will enrich this pattern. *Agreement* is defined as follows in [5]:

[4] For consistency with our notation, we have renamed the shared data items $\boldsymbol{t}$, while they were called ds in [5].

We say that a protocol guarantees to an initiator A agreement *with a responder B on a set of data items $\boldsymbol{t}$ (where $\boldsymbol{t}$ is a set of free variables appearing in the protocol description) if, whenever A (acting as an initiator) completes a run of the protocol, apparently with responder B, then B has previously been running the protocol, apparently with A, and B was acting as responder in his run, and the two agents agreed on the data values corresponding to the variables in $\boldsymbol{t}$, $\underline{\text{and each such run of } A \text{ corresponds to a unique run of } B}$.*[4]

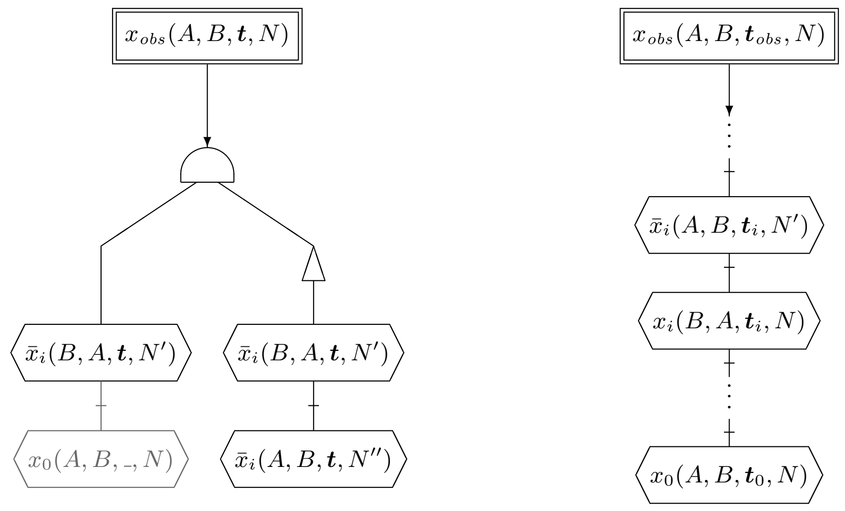
It differs from non-injective agreement by forcing each run of initiator A to correspond to a single run of responder B, over the shared variables $\boldsymbol{t}$. This requirement is expressed in the negated branch of the tree in Figure 9 (left): no two occurrences of $\bar{x}_i$ with identical arguments for the role owner (B), the interlocutor (A), and the shared terms t are allowed.

A related but different form of authentication is Roscoe's *intentional specification* [11]:

No node can believe a protocol run has completed unless a correct series of messages has occurred (consistent as to all the various parameters) up to and including the last message the given node communicates.

This is trivially captured in Figure 9 (right).

While the tree for agreement has clearly something in common with the trees shown in Figure 8, intentional specification is different. It is all the more surprising that agreement is a stronger notion than intentional specification, as proved in [5].



$x_{obs}(A, B, \boldsymbol{t}, N)$
$\bar{x}_i(B, A, \boldsymbol{t}, N')$
$\bar{x}_i(B, A, \boldsymbol{t}, N')$
$x_0(A, B, \_, N)$
$\bar{x}_i(A, B, \boldsymbol{t}, N'')$
$x_{obs}(A, B, \boldsymbol{t}_{obs}, N)$
$\bar{x}_i(A, B, \boldsymbol{t}_i, N')$
$x_i(B, A, \boldsymbol{t}_i, N)$
$x_0(A, B, \boldsymbol{t}_0, N)$

Fig. 9. Recent Agreement and Intentional Specification

8 Conclusions

In this paper we have developed a visual semantics based on fault trees for the subset of the NPATRL language that is used with the NRL Protocol Analyzer (NPA). We have also shown how this language can be used to display complex requirements as in the case of our analysis of the Group Domain of Interpretation Protocol, dense theorems as those that validate Kerberos 5, and subtle definitional differences as in the case of Lowe's authentication hierarchy. Indeed, we found the ability to express requirements in terms of precedence trees very helpful. For example, as the analysis of GDOI progressed, we often found it easier to write a precedence tree specification first. Furthermore, if we came to a difference of opinion about what a particular NPATRL requirement should say, we would often find it helpful to translate the requirement back into the precedence tree language to resolve ambiguities.

Visual representations of temporal languages are not new, of course. But in general they have taken the form of either timeline representations or state machines. Our approach, by limiting itself to the specification of properties of processes instead of the processes themselves, gives a representation of required temporal ordering of events that could be used in concert with these other graphical representations. Likewise, we believe that our language could be used in concert with other visual representations of protocols as well such as strand spaces [15] and GSMPL [6].

Our language right now is limited in that it can only capture specifications that are acceptable by the NPA. We do intend, however, to investigate how much further it can be extended. In particular, there is a new tool under development, Maude-NPA [3], that extends the capabilities of NPA in a number of ways. In particular, since it is based on strand spaces, it allows a richer representation of past and future events within a single state than NPA did, which in turn may support a richer requirements specification language.

Acknowledgments

We would like to thank Paul Syverson for enlightening conversations on the logical interpretation of NPATRL, and to the reviewers, whose comments helped improve the presentation of this paper. A preliminary version of this work appeared as [2].

A On the dual of $\diamondminus$

In this section we explore the dual of the operator $\diamondminus$, which we will denote $\boxminus$.

First, if we read $\diamondminus E$ in the abbreviated form "*previously* E", then $\boxminus E$ shall also be read as "*previously* E", which corresponds to emphasizing $\diamondminus$ as a temporal qualifier, at the detriment of its possibilitic interpretation. On the other hand, if we read $\diamondminus E$ as "*sometimes in the past,* E", then $\boxminus E$ shall stand for "*at all times in the past,* E". This allows a more traditional relation between the two operators, and we can define $\boxminus$ so that $\boxminus E = \neg \diamondminus \neg E$.

When adhering to this definition, the derived satisfiability rules for $\boxminus$ are the following:

$$\frac{}{\cdot \models \boxminus E}\,\boxminus_1 \qquad\qquad \frac{\textit{For all prefix } T' \textit{ of } T \qquad T' \models E}{T, a \models \boxminus E}\,\boxminus_2$$

In particular, for any atomic event a, the strictness restriction of NPA imposes that $T \models \boxminus a$ iff $T = \cdot$ (indeed, for all $T \neq \cdot$, we have that $T \models \diamondminus \neg a$ since $\cdot \models \neg a$). If we now introduce a second event, b, it is easy to show that $T \models (\boxminus \boxminus a) \wedge b$ iff $T = (b)$, *i.e.*, T is the trace that consists of the single event b.

Using this observation, it is possible to express requirements subject to initialization constraints. For example, we can require that a trace begins with a permutation of some initial actions $a_1, ..., a_n$ followed by some "ready" token r for a formula E to hold. We write

$$T \models \diamondminus \left((\underbrace{\boxminus \ldots \boxminus}_{n+2} a) \wedge (\bigwedge_{i=1}^{n} \diamondminus a_i) \wedge r \right) \wedge E$$

which will hold only if the trace T has the form

$$T = a_{\pi(1)} \ \ldots \ a_{\pi(1)} \, r \, T'$$

for some permutation π, and $T' \models E$ (assuming E refers to none of the events $a_1, \ldots, a_n, r$).

References

1. F. Butler, I. Cervesato, A. Jaggard, and A. Scedrov. A Formal Analysis of Some Properties of Kerberos 5 Using MSR. In *Proceedings of the* 15th *Computer Security Foundations Workshop*. IEEE Computer Society, 2002.
2. I. Cervesato and C. Meadows. A Fault-Tree Representation of NPATRL Security Requirements. In R. Gorrieri, editor, *Third Workshop on Issues in the Theory of Security — WITS'03*, pages 1–10, Warsaw, Poland, 2003.
3. S. Escobar, C. Meadows, and J. Meseguer. A rewriting-based inference system for the NRL Protocol Analyzer and its meta-logical properties. *Theoretical Computer Science*, 2007. To appear. Available at http://www.dsic.upv.es/users/elp/sescobar/papers.

4. K. Hansen, A. Ravn, and V. Stavridou. From safety analysis to software requirements. *IEEE Transactions on Software Engineering*, 24(7):573–584, July 1998.

5. G. Lowe. A hierarchy of authentication specifications. In *Proceedings of 10th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 1997.

6. J. McDermott and G. Allwein. A formalism for visual security protocol modeling. *Journal of Visual Languages and Computing*, 2007. To appear.

7. C. Meadows. The NRL Protocol Analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, Feb. 1996.

8. C. Meadows, I. Cervesato, and P. Syverson. Formal Specification and Analysis of the Group Domain of Interpretation using NPATR and the NRL Protocol Analyzer. *Journal of Computer Security*, 12(6):893–932, 2004.

9. C. Meadows and P. Syverson. A formal specification of requirements for payment transactions in the SET protocol. In *Financial Cryptography*, pages 122–140, 1998.

10. A. P. Moore, R. J. Ellison, and R. C. Linger. Attack modeling for information security and survivability. Technical Note CMU/SEI-2001-TN-001, CMU Software Engineering Institute, March 2001.

11. A.W. Roscoe. Intentional specification of security protocols. In *Ninth IEEE Computer Security Foundations Workshop*, pages 28–38, 1996.

12. B. Schneier. Attack trees : Modeling security threats. *Dr. Dobb's Journal*, Dec. 1999.

13. P. Syverson and I. Cervesato. The logic of authentication protocols. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, pages 63–136. Springer-Verlag LNCS 2171, 2001.

14. P. Syverson and C. Meadows. A formal language for cryptographic protocol requirements. *Designs, Codes, and Cryptography*, 7(1 and 2):27–59, January 1996.

15. F. J. Thayer, J. C. Herzog, and J. D. Guttman. Strand spaces: When is a security protocol correct. In *1998 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 1998.

16. W. E. Vesely, F.F. Goldberg, N. H. Roberts, and D. F. Haasl. Fault tree handbook. Technical Report NUREG-0492, U.S. Nuclear Regulatory Commission, January 1981. available at `http://www.nrc.gov/`.