A Fault-Tree Representation of NPATRL Security Requirements

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

[1] Advanced Engineering and Sciences Division, ITT Industries, Inc.
Alexandria, VA 22303 — USA
`iliano@itd.nrl.navy.mil`

[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 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.

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) [10], that is intended to be used with our analysis tool, the NRL Protocol Analyzer (NPA). NPATRL has been used to specify different types of requirements not only for generic key distribution and agreement protocols, as in [10], but also for complex protocols such as Secure Electronic Transaction (SET) [6] and the Group Domain of

* Cervesato was partially supported by NRL under contract N00173-00-C-2086. This work was completed while this author was visiting Princeton University.

Interpretation Protocol [7]. Others have developed similar temporal languages for expressing requirements for cryptographic protocols; see for example Marrero's logic for Brutus [5]. Yet others, for example [1, 2], have developed generic requirements that can be applied to any protocol, for example the correspondence requirement developed in [1, 2]. But even here it may make sense, as Lowe has argued in [4], to develop families of correspondence requirements suited to different applications rather than a single generic requirement.

The next step is to make sure that requirements are 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. Thus, it is necessary to develop some more user-friendly forms of representation for them. This is particularly true as protocols themselves become larger and more complex.

NPATRL requirements, as they are used in conjunction with NPA, have a tree-like structure. One gives an end goal, such as the intruder's learning a key, or a principal accepting a key as genuine, and then defines the sequences of actions that either should or should not precede that event. 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 [11] 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 [3] for an example and a discussion of the literature). More recently, fault trees have been used for the security analysis of systems[8, 9], 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 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.

The remainder of this paper is organized as follows. In Section 2, we recall the NPATRL language. 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 give some example NPATRL requirements with their fault tree interpretations. Section 6 concludes the paper.

2 The NPATRL Language

We define and motivate the NPATRL specification logic in Section 2.1, give a model semantics for it in Section 2.2, and then identify the sublanguage currently used for writing security requirements in Section 2.3.

2.1 NPATRL

The *NRL Protocol Analyzer Temporal Requirements Language*, better known as NPATRL (and pronounced "N Patrol"), was designed to address the shortcomings mentioned in the introduction [10]. 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 the names of other principals relevant to the event (for example, the intended recipient of a message). The third argument is a list of other terms 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.

The logical infrastructure of NPATRL consists of the usual connectives $\neg$, $\wedge$, $\Rightarrow$, etc, and the temporal modality $\diamondminus$ which is interpreted as "happened at some time before" or "previously". NPATRL is then defined by the following grammar:

$$E ::= a \quad | \quad \neg E \quad | \quad E_1 \wedge E_2 \quad | \quad E_1 \vee E_2 \quad | \quad E_1 \Rightarrow E_2 \quad | \quad \diamondminus E$$

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:

If a principal A accepts a key K for communicating with another principal B, then a server must have previously generated and 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:

$$\mathsf{initiator\_accept\_key}(A, B, K, N) \;\Rightarrow\; \diamondminus\, \mathsf{svr\_send\_key}(\mathsf{server}, (A, B), K, M)$$

For more discussion of event statements and how they relate to NPA specifications, see [10].

2.2 A Model Semantics for NPATRL

We will now endow NPATRL with a trace semantics 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:

$$T ::= \cdot \;\mid\; T, a$$

Here, "$\cdot$" stands for the empty trace, a is an event as defined in the previous section but with requirement that it contains no variable, and "," extends a trace with an event. We assume that times flows from left to right so that a is the most recent event of the trace T, a. While this grammar suggests that traces shall be finite in length, the definitions below apply also to infinite traces.

The above satisfiability judgment is defined by the following inference rules:

$$\frac{T \models E_1 \qquad T \models E_2}{T \models E_1 \wedge E_2}\,\wedge \qquad \frac{T \models E_i}{T \models E_1 \vee E_2}\,\vee_{i=1,2} \qquad \frac{T \not\models E}{T \models \neg E}\,\neg \qquad \frac{T \models [t/x]E}{T \models \forall x.E}\,\forall$$

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

The rules in the upper row unsurprisingly 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. 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 inference rules). In rule for the (implicit) universal quantifier, t shall be a ground term. The interpretation of implication is as usual derived from that of $\neg$ and either $\wedge$ or $\vee$.

The two lower 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 $\diamondminus$ [7].

2.3 NPA-Acceptable NPATRL Expressions

Although NPATRL was originally designed to be used with the NRL Protocol Analyzer, it is actually much more expressive than the set of specifications whose negations are accepted by the tool. Thus, in order to make NPATRL usable with NPA, it is necessary to identify a subset of NPATRL requirements that are acceptable by NPA, and to put them into a normal form that is parsable by NPA.

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

$$\begin{aligned} R &::= a \Rightarrow F \\ F &::= E \quad | \quad \neg E \quad | \quad F_1 \wedge F_2 \quad | \quad F_1 \vee F_2 \\ E &::= \diamondminus a \quad | \quad \diamondminus(a \wedge E) \end{aligned}$$

Note that the grammar requires that any negation come before any occurrence of the temporal operator, which has not always been the case with the examples we have examined. However, we can always transform a formula where a negation occurs further down into an $\mathbf{NPATRL_{NPA}}$ expression. Thus, for example, we can express $a \Rightarrow \diamondminus(b \wedge \neg\diamondminus c)$ as $a \Rightarrow \diamondminus b \wedge \neg\diamondminus(b \wedge \diamondminus c)$. Note, however, that this has the effect of making complex requirements somewhat less readable, since occurrences of events must be replicated.

3 Fault Trees

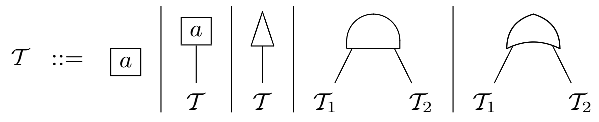
We will now introduce fault trees in Section 3.1 and then define the specialized notion of precedence tree in Section 3.2 that we will later use for specifying requirements for security protocols.

3.1 Fault Trees

Fault trees [11] 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.

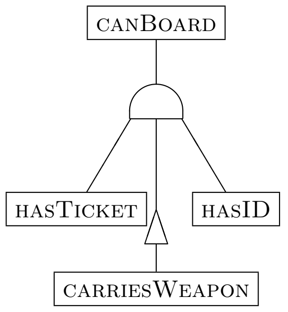
The nodes in a fault tree can be either basic events (defined below) or *logical gates*. The most common such combinators are the AND-gate (⌓) and the OR-gate (⌔). Edges link them to a top node and to two children. They specify that the event in the top node can occur only when all (resp., at least one) 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.

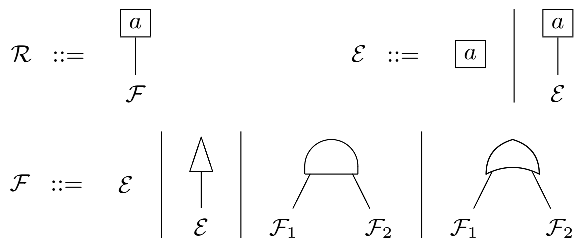
Sequences of AND-gates are often represented as an n-ary AND-gate (with $n \geq 2$). This can be viewed as a canonical form that abstracts associativity and commutativity issues away. An analogous convention is applied to OR-gates. We rely on this simplified syntax in the tree on the left, that specifies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



3.2 Precedence Trees

A direct, syntax-oriented, translation of the non-temporal fragment of NPATRL in the language of fault trees is an easy exercise, and even the precedence operator $\diamondminus$ could be handled by adding a dedicated gate, or a special type of edge. In the reverse direction, giving an elegant translation that spans over subtrees rooted at an event node is however problematic. For this reason, we will concentrate on the language of NPA-acceptable NPATRL expressions and define a fragment of the language of fault trees that allows a sufficiently well-behaved translation.

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:



While the edges of a generic fault tree had a double meaning (connectors and implication/iff), the edges of a precedence tree have a unique meaning within each syntactic category. However, the interpretation differs from category to category. The single edge of an $\mathcal{R}$-tree represents an implication as every NPA-acceptable NPATRL requirement starts with this connective. The edges of $\mathcal{F}$-trees are essentially connectors. The edges of $\mathcal{E}$-trees have instead a temporal interpretation: each event in a chain happens before its successor. This effectively implements a PRIORITY AND without the need of introducing an additional gate. We have considered using different lines for the various interpretations of an edge in a precedence tree, but have come to the conclusion that the benefits of such a departure from the syntax of fault trees would be fairly minimal.

4 NPATRL Semantics

In this section, we define the translation from $\mathbf{NPATRL_{NPA}}$ to precedence trees. These translations are important components of systems that uses human-readable precedence trees to enter and visualize machine-efficient NPATRL requirements.

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 $\ulcorner N\urcorner^c$. 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 1: 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.

The translation functions $\ulcorner\_\urcorner$ explicitly attribute the intuitive meaning we ascribed to the edges of a precedence tree in Section 3.2. Indeed, the single edge of an $\mathcal{R}$-tree derives from the implication of R-expressions, the edges of $\mathcal{F}$-trees are connectors, and the edges of an $\mathcal{E}$-subtree embed the precedence operator $\diamond$ of E-expressions.

The construction of the inverse translation is straightforward, and we omit it in this abstract.

5 Example

In this section, we describe some requirements similar to those that came up in our analysis of the Group Domain of Interpretation (GDOI) protocol [7] 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.

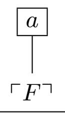
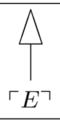
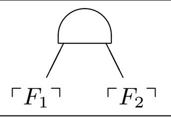
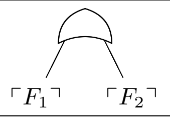
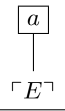
$\ulcorner R \urcorner^r$	$\ulcorner F \urcorner^f$				$\ulcorner E \urcorner^e$	
$a \Rightarrow F$	E	$\neg E$	$F_1 \wedge F_2$	$F_1 \vee F_2$	$\diamondminus a$	$\diamondminus(a \wedge E)$
	$\ulcorner E \urcorner$				$\boxed{a}$	

Fig. 1. Translating $\mathbf{NPATRL_{NPA}}$ to Precedence Trees

We describe a protocol that has two types of principals: group controllers and group members. A member joins the group by initiating a handshake protocol with the controller using a pairwise key shared between it and the controller. 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.

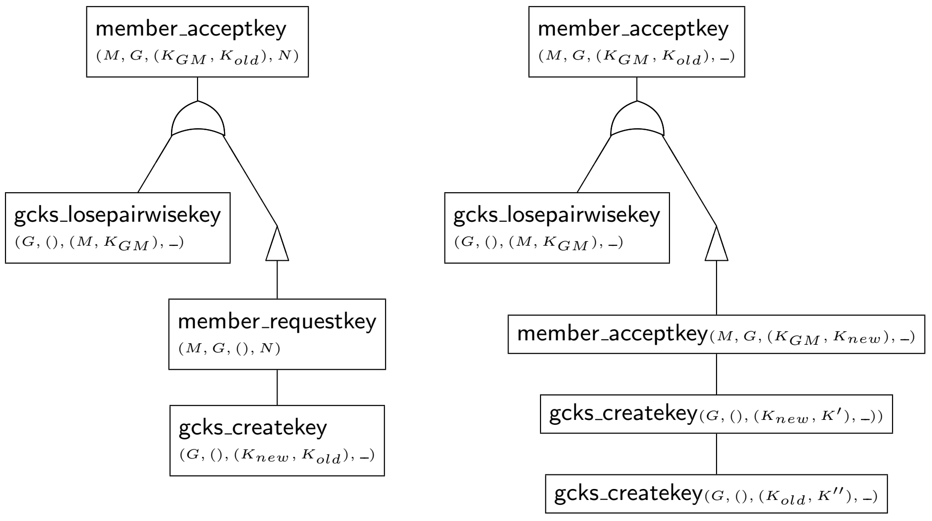
The following types of data are relevant: the controller name, denoted by G, the member name, denoted by M, the pairwise key, denoted by K_{GM}, and group keys, denoted by K_{new},K_{old}, K, K', and K''. All terms are variables universally quantified over their respective types. The symbol _ is a "don't-care"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

1. $\mathsf{member\_acceptkey}(M, G, (K_{GM}, K), N)$ describes a new member accepting a key as a result of the handshake protocol.
2. $\mathsf{gcks\_losepairwisekey}(G, (), (M, K_{GM}), N)$ describes the compromise of the pairwise key.
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 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 new key should have been distributed prior to the member's request. This we call *recency freshness*, since says that the member should accept the most recently generated key. We express it as the formatted NPATRL statement below.

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

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


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

(a) Recency Freshness (b) Sequential Freshness

Fig. 2. Freshness Requirements for the GDOI Pull Protocol

are accepted. It is formalized by the following NPATRL expression [7]:

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

Figure 2 displays the precedence trees corresponding to these two forms of freshness.

We note that both trees describe 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.

6 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. We have also shown how this language can be used to display complex requirements in the case of our analysis of the Group Domain of Interpretation Protocol. Indeed, we found the ability to express requirements in terms of precedence trees very helpful. As the analysis progressed, we often found it easier to write a precedence tree specification first, and then translate it into NPATRL and subsequently the NPA query language. Furthermore, whenever we came to a difference of opinion about what a particular requirement should say, we would often find it helpful to translate the requirement back into the precedence tree language to resolve ambiguities.

Our language right now is limited in that it can only capture specifications that are acceptable by the NRL Protocol Analyzer. We do intend, however, to investigate how much further it can be extended, both within the NPATRL framework and possibly beyond it.

References

1. M. Bellare and P. Rogaway. Entity authentication and key distribution. In D. Stinson, editor, *Advances in Cryptology - CRYPTO 93*. Springer-Verlag, 1994.
2. W. Diffie, P. C. van Oorschot, and M. J. WIener. Authentication and authenticated key esxchange. *Designs, Codes, and Cryptography*, 2:107–125, 1992.
3. K. Hansen, A. Ravn, and V. Stavridou. From safety analysis to software requirements. *IEEE Transactions on Software Engineering*, 24(7):573–584, July 1998.
4. G. Lowe. A hierarchy of authentication specifications. In *Proceedings of 10th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 1997.
5. Wilfredo Marrero. *BRUTUS: A Model Checker for Security Protocols*. PhD thesis, Carnegie Mellon University, 2001.
6. C. Meadows and P. Syverson. A formal specification of requirements for payment transactions in the SET protocol. In *Financial Cryptography*, pages 122–140, 1998.
7. C. Meadows, P. Syverson, and I. Cervesato. Formalizing GDOI group key management requirements in NPATRL. In *Eighth ACM Conference on Computer and Communication Security — CCS'01*, pages 235–244. ACM Press, 2001.
8. 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.
9. B. Schneier. Attack trees : Modeling security threats. *Dr. Dobb's Journal*, Dec. 1999.
10. P. Syverson and C. Meadows. A formal language for cryptographic protocol requirements. *Designs, Codes, and Cryptography*, 7(1 and 2):27–59, January 1996.
11. 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/`.