Verifying Security Protocols with Brutus

E.M. CLARKE
Carnegie Mellon University
S. JHA
University of Wisconsin
and
W. MARRERO
DePaul University

Due to the rapid growth of the "Internet" and the "World Wide Web" security has become a very important concern in the design and implementation of software systems. Since security has become an important issue, the number of protocols in this domain has become very large. These protocols are very diverse in nature. If a software architect wants to deploy some of these protocols in a system, they have to be sure that the protocol has the right properties as dictated by the requirements of the system. In this article we present BRUTUS, a tool for verifying properties of security protocols. This tool can be viewed as a special-purpose model checker for security protocols. We also present reduction techniques that make the tool efficient. Experimental results are provided to demonstrate the efficiency of BRUTUS.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*; D.4.6 [**Operating Systems**]: Security and Protection—*Verification*

General Terms: Security, Verification

Additional Key Words and Phrases: Model-checking, authentication and secure payment protocols, formal methods

This research is sponsored by the National Science Foundation (NSF) under Grant No. CCR-9505472 and the Defense Advanced Research Projects Agency (DARPA) under Contract No. DABT63-96-C-0071. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF, DARPA, or the United States Government.

Authors' addresses: E. Clarke, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213; S. Jha, Computer Science Department, University of Wisconsin, 1210 West Dayton Street, Madison, WI 53706; email: jha@cs.wisc.edu; W. Marrero, Department of Computer Science, DePaul University, Chicago, IL 60604.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 2001 ACM 1049-331X/00/1000–0443 $5.00

1. INTRODUCTION

Security for early computers was provided by their physical isolation. Unauthorized access to these machines was prevented by restricting physical access. The importance of sharing computing resources led to systems where users had to authenticate themselves, usually by providing a name-password pair. This was sufficient if the user was physically at the console or connected to the machine across a secure link. However, the efficiency to be gained by sharing data and computing resources has led to computer networks, in which the communication channels cannot always be trusted. In this case, authentication information such as the name-password pairs could be intercepted and even replayed to gain unauthorized access. When such networks were local to a certain user community and isolated from the rest of the world, many were willing to take this risk and to place their trust in the community. However, in order to be able to share information with those outside the community, this isolation had to be removed. The benefits obtained by such sharing have been enormous, and the gains are demonstrated by the growth of such entities as the "Internet" and the "World Wide Web." As more and more people use these resources, and as more services are offered on-line, the importance of being able to provide security guarantees becomes paramount. Typically, these guarantees are provided by means of protocols that make use of security primitives such as encryption, digital signatures, and hashing. These protocols are notoriously hard to design. In fact, errors have been found in many protocols several years after they were published. Due to our increased dependence on these protocols, it is extremely important that we should have high assurance or confidence in these protocols. For example, compromise of a secure payment protocol used between a customer and an on-line brokerage company can result in huge losses for the customers and brokerage houses. Moreover, negative publicity caused by intrusions can erode customer confidence and affect the entire electronic commerce sector. Since the protocols in the domain of security are widely used, and flaws in these protocols can in some cases result in grave losses, we believe that applying formal methods for the verification of these protocols can be very beneficial. We realize that verifying protocols is one of many steps in building a complete system. However, by having automatic verification tools for performing such tasks, we ease the burden on the software architect so that they can concentrate on other issues. Our main goal was to build a tool that is easy to use and is as automatic as possible. We think any tool that can be readily used by system architects building secure systems should have these characteristics.

A question that immediately comes to mind is "why is it not enough to verify a few existing standards?" In this case verifying security protocols using some *heavyweight techniques* such as theorem proving or manual proofs might be acceptable. In the ensuing paragraph we argue that frequently architects modify standard protocols or design new protocols. Hence, an architect designing a secure system needs to test properties of protocols quickly and effortlessly. Architects design new protocols because of several reasons. First, existing protocols might be too complex and might have a severe performance impact on the system. Depending on the requirements of the system, a software architect might simplify an existing protocol. In this case, one needs to check properties of this simplified protocol quickly. In our experience, this is quite common, especially in the domain of secure payment protocols. Moreover, because of the specific nature of the system being designed a software architect might only need a very simple protocol. For example, if the customer and the merchant communicate directly (e.g., through an on-line brokerage system), the secure payment protocol can be greatly simplified because the communication is occurring between two parties, i.e., there is no need to involve a clearing house or third party. Therefore, in order for a tool to effortlessly blend into the software process a verification tool needs to be *lightweight* and *push button* in nature.

A second requirement for a useful tool is that it should have an expressive specification language for stating properties about the protocol under consideration. In our view, "hardwiring" properties into a tool is not the right approach. One can never anticipate properties that a system architect may want to check. This is primarily because protocols run in the context of a system. In fact, a single protocol might be required to have different properties depending on the larger system it is deployed in. For example, if a secure payment protocol runs between two parties that are untrusted, one might want to check for authentication-type properties, i.e., only authorized parties take part in the protocol. In a secure payment protocol that runs on top of a secure link one can assume that the parties are trusted. Hence, in this case the context of the system imposes different kinds of properties on the secure payment protocol.

Earlier verification work in the domain of security was limited to authentication protocols. These protocols are generally small. Now the verification effort has shifted to secure payment protocols and other protocols related to electronic commerce. With the emergence of electronic commerce these protocols have become quite important. In general, secure payment protocols are much larger than authentication protocols. This brings the efficiency issue to the forefront. Algorithms for verifying these protocols should be efficient in space and time. Moreover, as protocols become complex, the flaws they exhibit have also become more complicated in nature. For example, there are protocols with flaws that only occur if multiple instances or sessions of that protocol are active at the same time. This means that the verification tool should be able to handle multiple sessions. Unless a verification algorithm is designed carefully, multiple sessions can cause an explosion in the explored state space.

BRUTUS was developed to satisfy the requirements outlined above. BRUTUS supports a wide class of security protocols and is completely *push button* in nature. If the protocol under consideration is incorrect or flawed, the tool generates a counterexample or an attack demonstrating the flaw. This feature is invaluable to the designer for fixing the protocol. We have also designed a very expressive logic for specifying properties about the protocol. In our experience, most security properties we have encountered can be easily expressed in our logic. We have also developed reduction techniques that make the verification process efficient in both time and space. However, we believe that efficiency is an area where our tool could still be improved.

A model of computation for protocols is discussed in Section 2. Section 3 describes the logic for expressing properties about cryptographic protocols. Three protocols and some relevant properties are described in Section 4. We have provided several examples of properties required of cryptographic protocols. These can serve as design patterns for the architect using our system. The basic algorithm and reductions used in BRUTUS are briefly discussed in Section 5. Section 6 provides experimental results demonstrating the effectiveness of the reduction techniques. A detailed account of other approaches and their comparison with BRUTUS appears in Section 7. In discussing the related work, we have tried to be comprehensive, so Section 7 is almost a minisurvey of the area of formal methods in security. Finally, in Section 8 we conclude with future directions.

2. THE MODEL

We describe how we model security protocols in BRUTUS. As with other model checkers, having an operational description of the behavior of agents or principals participating in a protocol is desirable.[1] We begin by describing the messages involved in a protocol model, specifically what kinds of messages there are and how they can be constructed. We then give a mathematical model for the agents in the protocol. Finally, we discuss the different actions allowed during the execution of a protocol and describe how they change the state of the system.

2.1 Messages

Typically, the messages exchanged during a run of a protocol are constructed from smaller submessages using concatenation and encryption. The smallest such submessages (i.e., they contain no submessages themselves) are called *atomic messages*. There are four kinds of *atomic messages*.

—*Keys* are used to encrypt messages. Keys have the property that every key k has an inverse k^{-1}. Note that for symmetric cryptography the decryption key is the same as the encryption key, so $k = k^{-1}$.

—*Principal names* are used to refer to the participants in a protocol.

—*Nonces* can be thought of as randomly generated numbers. The intuition is that no one can predict the value of a nonce; therefore, any message containing a nonce can be assumed to have been generated after the nonce was generated, i.e., it is not an "old" message.

[1] Principals and agents will be used synonymously throughout the article.

—*Data* messages play no role in how the protocol works but are intended to be communicated between principals.

Let $\mathcal{A}$ denote the space of *atomic messages*. The set of all messages $\mathcal{M}$ over some set of atomic messages $\mathcal{A}$ is inductively defined as follows:

—If $a \in \mathcal{A}$ then $a \in \mathcal{M}$. (Any *atomic message* is a message.)

—If $m_1 \in \mathcal{M}$ and $m_2 \in \mathcal{M}$ then $m_1 \cdot m_2 \in \mathcal{M}$. (Two messages can be paired together to form a new message.)

—If $m \in \mathcal{M}$ and key $k \in \mathcal{A}$ then $\{m\}_k \in \mathcal{M}$. (A message m can be encrypted with key k to form a new message.)

We would also like to generalize the notion of messages to *message templates*. A message template can be thought of as a message containing one or more message variables. To extend messages to message templates we add the following rule to the inductive definition of messages:

—If v is a message variable, then $v \in \mathcal{M}$.

Because keys have inverses, we always rewrite a message of the form $\{\{m\}_k\}_{k^{-1}}$ as m, or we always keep the messages in a "reduced" form. It is also important to note that we make the following *perfect encryption assumption*: the only way to generate $\{m\}_k$ is from m and k. In other words, for all messages m, m_1, and m_2 and keys k, $\{m\}_k \neq m_1 \cdot m_2$, and $\{m\}_k = \{m'\}_{k'} \Rightarrow m = m' \wedge k = k'$.

We also need to consider how new messages can be created from already known messages by encryption, decryption, pairing (concatenation), and projection. For this we define a derivation relation "$\vdash$" which captures how a message m can be derived from some initial set of information I.

(1) If $m \in I$ then $I \vdash m$.

(2) If $I \vdash m_1$ and $I \vdash m_2$ then $I \vdash m_1 \cdot m_2$ (pairing)

(3) If $I \vdash m_1 \cdot m_2$ then $I \vdash m_1$ and $I \vdash m_2$ (projection)

(4) If $I \vdash m$ and $I \vdash k$ for key k then $I \vdash \{m\}_k$ (encryption)

(5) If $I \vdash \{m\}_k$ and $I \vdash k^{-1}$ then $I \vdash m$ (decryption)

We will use the notation $\overline{I}$ to denote the *closure* of the set I under the rules given above. In other words $m \in \overline{I}$ if and only if $I \vdash m$.

These rules define the most common derivability relation used to model the capabilities of the adversary or the intruder in the literature. We will assume that when trying to subvert a protocol, the adversary can only use messages it can derive using these rules from some initial set of information and from overheard messages. For example, if I_0 is some finite set of messages overheard by the adversary and possibly some initially known messages, then $\overline{I_0}$ represents the set of all messages known to the adversary, and the adversary will be allowed to send any message in $\overline{I_0}$ to any honest agent in an attempt to subvert the protocol.

In general, $\overline{I}$ is infinite, but researchers have taken advantage of the fact that one need not actually compute $\overline{I}$. It suffices to check whether $m \in \overline{I}$ for some finite number of messages m. However, checking if $m \in \overline{I}$ must still be decidable. We briefly discuss this question in Section 5. For a detailed discussion of this question, see Clarke et al. [1998].

2.2 Instances

We model a protocol as an asynchronous composition of a set of named communicating processes which model the honest agents and the adversary. We also model an insecure and lossy communication medium, in which a principal has no guarantees about the origin of a message, and where an adversary is free to eavesdrop on all communications and interfere with fake messages. Therefore, in the model, we insist that all communications go through the adversary. In other words, all messages sent are intercepted by the adversary, and all messages received by honest agents were actually sent by the adversary. In addition, the adversary is allowed to create new messages from the information it gains by eavesdropping, in an attempt to subvert the protocol.

In order to make the model finite, we must place a bound on the number of times a principal may attempt to execute the protocol. Each such attempt will be called a *session*. Each session will be modeled as a principal *instantiating* some role in the protocol (i.e., initiator or responder). For this reason we will call the formal model of an individual session an *instance*. Each instance is a separate copy or instantiation of a principal and consists of a single execution of the sequence of actions that make up that agent's role in the protocol, along with all the variable bindings and knowledge acquired during the execution. A principal can have multiple instances, but each instance is executed once. When we combine these with a single instance of the adversary, we get the entire model for the protocol.

Each *instance* of an honest principal is modeled as a 5-tuple $\langle N, S, I, B, P\rangle$ where

—$N \in \mathit{names}$ is the name of the principal,

—S is a unique *instance ID* for this instance,

—$B : \mathit{vars}(N) \to \mathcal{M}$ is a set of bindings for $\mathit{vars}(N)$, the set of variables appearing in principal N, which are bound for a particular instance as it receives messages,

—$I \subseteq \mathcal{M}$ is the set of messages known to the principal executing this instance, and

—P is a process description given as a sequence of actions to be performed. These actions include the predefined actions **send** and **receive**, as well as user-defined internal actions such as **commit** and **debit**.

The model of the adversary, Z, has some similarities; however, the adversary is not bound to follow the protocol, and so it does not make sense to include either a sequence of actions P_Z or a set of bindings B_Z for the adversary. Instead, at any time, the adversary can receive any message, or it can send any message it can generate from its set of known messages I_Z. The instance corresponding to the adversary will be denoted by S_Z.

The global model is then the asynchronous composition of the models for each instance, including the adversary. Each possible execution of the model corresponds to a *trace*, a finite, alternating sequence of global states and actions $\pi = \sigma_0 \alpha_1 \sigma_1 \alpha_1 \cdots \alpha_n \sigma_n$ for some $n \in \mathbb{N}$, such that $\sigma_{i-1} \xrightarrow{\alpha_i} \sigma_i$ for $0 < i \leq n$ and for the transition relation $\rightarrow$ as defined in the next subsection.

2.3 Actions

The actions allowed during the execution of a protocol include the two predefined actions **send** and **receive** as well as possibly some user-defined actions. The model transitions between global states as a result of actions taken by the individual components, or instances. More formally, we define a transition relation $\rightarrow \subseteq \Sigma \times S \times A \times \mathcal{M} \times \Sigma$ where Σ is the set of global states, where S again is the set of instance IDs, where A is the set of action names (which includes **send** and **receive**), and where $\mathcal{M}$ is the set of all possible messages. We will use the notation $\sigma \xrightarrow{s \cdot a \cdot m} \sigma'$ in place of $(\sigma, s, a, m, \sigma') \in \rightarrow$ when it is more convenient. In the definition of the transition relation given below, we will denote the adversary as $Z = \langle N_Z, S_Z, \emptyset, I_Z, \phi \rangle$ and the honest instances as $H_i = \langle N_i, S_i, B_i, I_i, P_i \rangle$. We will use $\sigma = \langle Z, H_1, \ldots, H_n \rangle$ to denote the global state before the transition and $\sigma' = \langle Z', H'_1, \ldots, H'_n \rangle$ to denote the global state after the transition. In addition, we will use the notation $\hat{B}$ to denote the natural extension of a set of bindings B from the domain of variables to the domain of message templates. In other words, $\hat{B}(m)$ is the result of substituting $B(v)$ for v in the message template m for all the variables v appearing in m. Next we describe all the legal transitions of the model.

—$\sigma \xrightarrow{s \cdot \mathbf{send} \cdot m} \sigma'$: An instance with ID s can send message m in the global state σ, and the new global state is σ' if and only if

(1) $I_{Z'} = I_Z \cup m$. (The adversary adds m to the set of messages it knows.)

(2) There is an instance $H_i = \langle N_i, s, B_i, I_i, \mathbf{send}(s\text{-}msg) \cdot P'_i \rangle$ in σ such that in σ', $H'_i = \langle N_i, s, B_i, I_i, P'_i \rangle$ and $m = \hat{B}_i(s\text{-}msg)$. (There is an instance that is ready to send message m, and this action is removed from its sequence of actions in the next state.)

(3) $H_j = H'_j$ for all $j \neq i$. (All other instances remain unchanged.)

Notice that the adversary intercepts all messages. In order for the intended recipient to receive the message, the adversary must forward it.

—$\sigma \xrightarrow{s\cdot\mathbf{receive}\cdot m} \sigma'$: An instance with ID s can receive message m in global state σ, and the new global state is σ' if and only if

(1) $m \in \overline{I_Z}$. (The adversary can generate the message m.)

(2) There is an instance $H_i = \langle N_i, s, B_i, I_i, \mathbf{receive}(r\text{-}msg) \cdot P'_i\rangle$ in σ such that in σ', $H'_i = \langle N_i, s, B'_i, I'_i, P'_i\rangle$, $I'_i = I_i \cup m$, and B'_i is the smallest extension of B_i such that $\hat{B}'_i(r\text{-}msg) = m$. (There is an instance ready to receive a message of the form of m; in the next state its bindings are updated correctly, and the receive action is removed from its sequence of actions.)

(3) $H_j = H'_j$ for all $j \neq i$. (All other instances remain unchanged.)

Notice that all messages come from the adversary, although as stated above they may simply have been forwarded unchanged. Because all messages come from the adversary, we can also model when an adversary modifies, misdirects, or suppresses messages.

—$\sigma \xrightarrow{s\cdot A\cdot m} \sigma'$: An instance with ID s can perform some user-defined internal action A with argument m in global state σ, and the new global state is σ' if and only if

(1) there is an instance $H_i = \langle N_i, s, B_i, I_i, A(msg) \cdot P'_i\rangle$ in σ such that in σ', $H'_i = \langle N_i, s, B_i, I_i, P'_i\rangle$ and $m = \hat{B}_i(msg)$ (there is an instance s that is ready to perform action A with argument m and this action is removed from its sequence of actions in the next state) and

(2) $H_j = H'_j$ for all $j \neq i$ (all other instances remain unchanged).

3. LOGIC

In order to specify the requirements or the desired properties of the protocol, we will use a first-order logic where quantifiers range over the finite set of instances in a model. In addition, the logic will include the past-time modal operator so that we can talk about things that happened in the history of a particular protocol run or trace. The atomic propositions of the logic will allow us to refer to the bindings of variables in the model, to actions that occur during execution of the protocol, and to the knowledge of the different agents participating in the protocol. Generally, our logic is a variant of the linear-time temporal logic with the past-time operator where at the atomic level one can express actions and knowledge. We will begin with the syntax of the logic, followed by the formal semantics.

3.1 Syntax

As stated above, we will use a first-order logic where quantifiers range over the finite set of instances. The atomic propositions are used to characterize states, actions, and knowledge in the model. The arguments to the atomic propositions are terms expressing instances or messages. We begin by a formal description of terms.

—If S is a instance ID, then S is an instance term.

—If s is an instance variable, then s is an instance term.

—If M is a message, then M is a message term.

—If m is a message variable, then m is a message term.

—If s is an instance term, then $pr(s)$ is a message term. Intuitively, $pr(s)$ represents the principal that is executing instance s.

—If s is an instance term and m is a message variable, then $s.m$ is a message term representing the binding of m in the instance s.

—If m_1 and m_2 are message terms, then $m_1 \cdot m_2$ is a message term.

—If m_1 and m_2 are message terms, then $\{m_1\}_{m_2}$ is a message term. Of course we are assuming that m_2 has the correct type to be used as a key.

—We use "." as a scoping operator. If s an instance term and v is a message variable, then $s.v$ is a message term and intuitively refers to the variable v bound in instance s.

As in standard first-order logic, atomic propositions are constructed from terms using relation symbols. The predefined relation symbols are "=" and "**Knows**." The user can also define other relation symbols which would correspond to user-defined actions in the model. The syntax for atomic propositions is as follows (all relation symbols are used in the infix notation):

—If m_1 and m_2 are message terms, then $m_1 = m_2$ is an atomic proposition. Examples of an atomic proposition would be checking if a customer and merchant agree on the price of a purchase ($C_0.price = M_0.price$), or to check if a particular instance of A believes its authenticating with B ($A_0.partner = B$).

—If s is an instance term and m is a message term, then s **Knows** m is an atomic proposition which intuitively means that instance s knows the message m. This proposition can be used to check if the adversary has compromised the session key (Z **Knows** K).

—If s is an instance term, m is a message term, and A is a user-defined action, then $s\ A\ m$ is an atomic proposition which intuitively means that instance s performed action A with message m as an argument. For example, this could be used to check if a customer C_0 has committed to a transaction with identifier *TID*(C_0 **commit** *TID*).

Finally, *well-formed formulas (WFFs)* are built up from atomic propositions with the usual connectives from first-order and modal logic.

—if f is an atomic proposition, then f is a WFF.

—if f is a WFF, then $\neg f$ is a WFF.

—if f_1 and f_2 are WFFs, then $f_1 \wedge f_2$ is a WFF.

—if f is a WFF and s is an instance variable, then $\exists s.f$ is a WFF.

—if f is a WFF, then $\Diamond_P f$ is a WFF.

The formula $\exists s.f$ means that there exists some instance s_0 such that f is true when you substitute s_0 for s in f, while $\Diamond_P f$ means that at some point in the past f was true. We also use the following common shorthands:

—$f_1 \vee f_2 \equiv \neg(\neg f_1 \wedge \neg f_2)$

—$f_1 \rightarrow f_2 \equiv \neg f_1 \vee f_2$

—$f_1 \leftrightarrow f_2 \equiv (f_1 \rightarrow f_2) \wedge (f_2 \rightarrow f_1)$

—$\forall s.f \equiv \neg \exists s.\ \neg f$ (for all instances s_0, f is true when you substitute s_0 for s)

—$\Box_P f \equiv \neg\ \Diamond_P \neg f$ (at all points in the past f was true).

The formula $\forall s.f$ means that for any instance s_0, f is true when you substitute s_0 for s in f, while $\Box_P f$ means f was true at all points in the past.

3.2 Semantics

Next we provide semantics for the logic just presented. These semantics will be given in terms of the formal model presented in Section 2. Again, we begin with the terms of the logic.

—An instance ID S refers to the instance with that ID.

—An instance variable s ranges over all the instances in the model.

—An atomic message M is an atomic message in the model.

—A message variable v varies over messages in the model.

—If s is an instance ID, then $pr(s)$ is the principal executing the instance with ID s.

—The interpretation $\sigma(s.v)$ of $s.v$ in a particular state σ is $B_s(v)$, the value bound to the variable v in instance s in state σ.

—The semantics of message concatenation and encryption are obvious and follow straight from the definitions.

The WFFs of the logic will be interpreted over the traces of a particular model. Recall that a trace consists of a finite, alternating sequence of states and actions $\pi = \sigma_0 \alpha_1 \sigma_1 . . . s_n$. Length of a trace π is denoted by length(π). We give the semantics of WFFs in our model via a recursive definition of the satisfaction relation $\models$. We will write $\langle \pi, i \rangle \models f$ to mean that the ith state in a trace π satisfies the formula f. We begin with atomic propositions.

—$\langle \pi, i \rangle \models m_1 = m_2$ if and only if $\sigma_i(m_1) = \sigma_i(m_2)$. Thus the formula $m_1 = m_2$ is true in a state if the interpretations of m_1 and m_2 are equal. In other words, two message terms are equal in a state if after applying the appropriate substitutions to the variables appearing in the message terms, the resulting messages are equal.

—$\langle \pi, i \rangle \models s$ **Knows** m if and only if $\sigma_i(m) \in \overline{I_j}$ for some instance H_j in σ_i such that $S_j = s$ (the instance ID of H_j is s). In other words, the formula s **Knows** m is true in a state if the instance with ID s can derive message m from its known set of messages in that state. S_Z **Knows** m is true if the adversary Z knows that message m (recall that S_Z is the instance corresponding to the adversary).

—$\langle \pi, i \rangle \models s\ A\ m$ for some user-defined action A if and only if $\alpha_i = s \cdot A \cdot m$. In other words, the formula $s\ A\ m$ is true in a state if the transition taken to enter the current state was one in which instance s took action A with argument m.

The extension of the satisfaction relation to the logical connectives is the same as for standard first-order logic. We use the notation $[f/s \mapsto s_0]$ to denote the result of substituting every free occurrence of the instance variable s in f with the instance ID s_0.

—$\langle \pi, i \rangle \models \neg f$ if and only if $\langle \pi, i \rangle \models f$.

—$\langle \pi, i \rangle \models f_1 \wedge f_2$ if and only if $\langle \pi, i \rangle \models f_1$ and $\langle \pi, i \rangle \models f_2$.

—$\langle \pi, i \rangle \models \exists s.f$ if and only if there exists an honest instance s_0 in the model such that $\langle \pi, i \rangle \models [f/s \mapsto s_0]$.

—$\langle \pi, i \rangle \models \Diamond_P f$ if and only if there exists a $0 \leq j \leq i$ such that $\langle \pi, j \rangle \models f$. In other words, the formula $\Diamond_P f$ is true in a state of a trace π if the formula f is true in any state of the trace up to and including the current state.

A formula f is said to be true in a trace π (denoted as $\pi \models f$) if and only if f is true in *every state* of the trace π. A formula f is true in a model if and only if f is true in every trace of the model.

4. CASE STUDIES

We now turn to three case studies that illustrate how BRUTUS can be used to analyze security protocols. We will first look at the Needham-Schroeder Public Key authentication protocol [Needham and Schroeder 1978]. Next we will look at the 1KP electronic commerce protocol [Bellare et al. 1995]. Finally, we will examine the Wide Mouthed Frog protocol.

4.1 Needham-Schroeder Public Key

The Needham-Schroeder Public Key authentication protocol has received much attention since a new attack was found by Gavin Lowe in 1996 [Lowe 1996]. We present our analysis of this protocol for the sake of comparison with the many other approaches that have also been used to analyze this protocol. The structure of this protocol is given below. We will assume that the initiator is agent A and that it wishes to authenticate with agent B.

(1) First, the initiator, A, generates a nonce N_a (which we can assume is a random number), encrypts the pair N_a, A with B's public key, and constructs the message A, B, $\{N_a, A\}_{K_B}$ which it sends to B.

$$A \rightarrow B : A, B, \{N_a, A\}_{K_B}$$

(2) Upon receiving message number 1, B uses its private key to decrypt $\{N_a, A\}_{K_B}$ and recover the identity of the initiator, A, and its nonce N_a. It then generates its own nonce N_b, encrypts the pair N_a, N_b with A's public key, and constructs the message A, B, $\{N_a, N_b\}_{K_A}$ which it sends to A.

$$B \rightarrow A : A, B, \{N_a, N_b\}_{K_A}$$

(3) Upon receiving message number 2, A uses its private key to decrypt $\{N_a, N_b\}_{K_A}$. It is now convinced of B's identity and that B possesses N_a, a shared secret that A can include in new messages for identification. It now replies to B by encrypting N_b with B's public key and sending the message A, B, $\{N_b\}_{K_B}$.

$$A \rightarrow B : A, B, \{N_b\}_{K_B}$$

(4) Upon receiving message number 3, B can once again use its private key to decrypt $\{N_b\}_{K_B}$. Now B is convinced of A's identity and that A possesses N_b, a shared secret that B can include in new messages for identification.

Let us now look at how we model this protocol in BRUTUS. We must construct a process for each honest principal in the protocol. The role of the initiator is modeled by the following sequence of actions which is parameterized in a, the name of the initiator, and n_a, the value of its random number or nonce. The internal actions "begin-initiate" and "end-initiate" are used to mark the points in the protocol when the initiator has begun, and finished executing the protocol and with whom it is authenticating. These actions have no role in the model, but the specification will refer to them.

```
$INITIATOR$ =
choose $(b)$
internal (“begin-initiate”, $b$)
send $(a, b, \{n_a, a\}_{K_b})$
receive $(a, b, \{n_a, n_b\}_{K_a})$
send $(a, b, \{n_b\}_{K_b})$
internal (“end-initiate”, $b$)
```

There is an analogous role for the responder. This definition is given by the following sequence of actions which is parameterized in b, the name of the responder, and n_b, its random number. Again, the internal actions “begin-respond” and “end-respond” mark the points in the protocol where the responder has begun and finished executing the protocol and with whom it is authenticating.

```
$RESPONDER$ =
receive $(a, b, \{n_a, a\}_{K_b})$
internal (“begin-respond”, $a$)
send $(a, b, \{n_a, n_b\}_{K_a})$
receive $(a, b, \{n_b\}_{K_b})$
internal (“end-respond”, $a$)
```

One of these processes will appear in each instance of an honest agent. We now give a model containing four instances, one initiator instance, and one responder instance for principal “A” and the same for principal “B.” The model is the cross-product of the four instances shown below with an instance for the adversary.

H_1
$N_1 = $ “A”
$S_1 = $ “A1”
$B_1 = \{(a, \text{“A”}), (n_a, \text{“NA1”})\}$
$I_1 = \{A, B, Z, K_A, K_B, K_Z, K_A^{-1}, NA1\}$
$P_1 = INITIATOR$

H_2
$N_2 = $ “A”
$S_2 = $ “A2”
$B_2 = \{(b, \text{“A”}), (n_b, \text{“NA2”})\}$
$I_2 = \{A, B, Z, K_A, K_B, K_Z, K_A^{-1}, NA2\}$
$P_2 = RESPONDER$

H_3
$N_3 = $ “B”
$S_3 = $ “B1”
$B_3 = \{(a, \text{“B”}), (n_b, \text{“NB1”})\}$
$I_3 = \{A, B, Z, K_B, K_B, K_Z, K_B^{-1}, NB1\}$
$P_3 = INITIATOR$

H_4
$N_4 = $ “B”
$S_4 = $ “B2”
$B_4 = \{(b, \text{“B”}), (n_b, \text{“NB2”})\}$
$I_4 = \{A, B, Z, K_B, K_B, K_Z, K_B^{-1}, NB2\}$
$P_4 = RESPONDER$

All that remains is to specify the requirements for the protocol. We will check for three different properties. The first is that if principal A has finished executing a session with B then B must have participated in a session with A. The same should hold when B has finished a session with A. The second property checks that the nonces which are intended to be shared secrets are kept secret from the adversary. The final property is a nonrepudiation property and states that when A finishes a protocol session with B, B knows A's nonce and vice-versa.

—*Authentication:* This property can be used as a generic requirement for authentication protocols. Intuitively, the requirement is that if some principal A has finished executing an authentication protocol with B, then B must have participated in the protocol. This is formalized in our logic with two formulas, one for the initiator and one for the responder.

$$\forall A_0 . A_0 \textbf{ internal } (\texttt{"end-initiate"}, A_0.b) \rightarrow$$

$$\exists B_0 \, . \, (pr(B_0) = A_0.b) \wedge \Diamond_P(B_0 \textbf{ internal } (\texttt{"begin-respond"}, pr(A_0)))$$

This formula states that for all honest instances A_0, if A_0 has performed an "end-initiate" internal action with the principal that it believes is its partner

$$A_0 \textbf{ internal } (\texttt{"end-initiate"}, A_0.b)$$

then there exists an instance B_0 of that partner such that at some point in the past B_0 performed a "begin-respond" internal action with the principal of instance A_0, corresponding to the clause

$$(pr(B_0) = A_0.b) \wedge \Diamond_P(B_0 \textbf{ internal } (\texttt{"begin-respond"}, pr(A_0))).$$

In other words, for all initiators I, if I has finished executing with some principal, then some instance R of that other principal must have at least started executing the protocol with the principal executing I. This requirement assures us of the presence of the other party in an authentication protocol. The protocol satisfies this property.

There is also an analogous property for the responder. Namely, if the responder B_0 has finished executing the protocol with its partner $B_0.a$, then there must be some initiator instance A_0 executing on behalf of that principal $B_0.a$ that has participated in the protocol. This property is violated by the protocol. Figure 1 contains the counterexample trace provided by BRUTUS. Note that at the end of the trace $A2$ has finished responding with B, but there is no instance of B, ($B1$ or $B2$) that has initiated with A. This attack actually occurs with a single initiator and a single responder instance as reported by Lowe [1996]. Figure 2 illustrates this attack. Note that in the attack B believes that it is responding to A, but A is executing an instance with the adversary Z.

—*Secrecy:* The nonces exchanged in the Needham-Schroeder protocol are intended to be shared secrets. As such, the adversary should have no knowledge of them unless an honest agent is trying to authenticate with the adversary.

$A1$ **choose** $Principal\ B$
$A1$ **internal** (“`begin-initiate`”, $Principal\ B$)
$A1$ **send** $Principal\ A, Principal\ B, \{Nonce\ Na1, Principal\ A\}\_Pubkey(Principal\ B)$
$B2$ **choose** $Principal\ Z$
$B2$ **internal** (“`begin-initiate`”, $Principal\ Z$)
$B2$ **send** $Principal\ B, Principal\ Z, \{Nonce\ Nb2, Principal\ B\}\_Pubkey(Principal\ Z)$
$B1$ **receive** $Principal\ A, Principal\ B, \{Nonce\ Nb2, Principal\ A\}\_Pubkey(Principal\ B)$
$B1$ **internal** (“`begin-respond`”, $Principal\ A$)
$B1$ **send** $Principal\ B, Principal\ A, \{Nonce\ Nb2, Nonce\ Nb1\}\_Pubkey(Principal\ A)$
$A2$ **receive** $Principal\ B, Principal\ A, \{Nonce\ Nb2, Principal\ B\}\_Pubkey(Principal\ A)$
$A2$ **internal** (“`begin-respond`”, $Principal\ B$)
$A2$ **send** $Principal\ A, Principal\ B, \{Nonce\ Nb2, Nonce\ Na2\}\_Pubkey(Principal\ B)$
$B2$ **receive** $Principal\ Z, Principal\ B, \{Nonce\ Nb2, Nonce\ Na2\}\_Pubkey(Principal\ B)$
$B2$ **send** $Principal\ B, Principal\ Z, \{Nonce\ Na2\}\_Pubkey(Principal\ Z)$
$B2$ **internal** (“`end-initiate`”, $Principal\ Z$)
$A2$ **receive** $Principal\ B, Principal\ A, \{Nonce\ Na2\}\_Pubkey(Principal\ A)$
$A2$ **internal** (“`end-respond`”, $Principal\ B$)

Fig. 1. Needham-Schroeder counterexample.

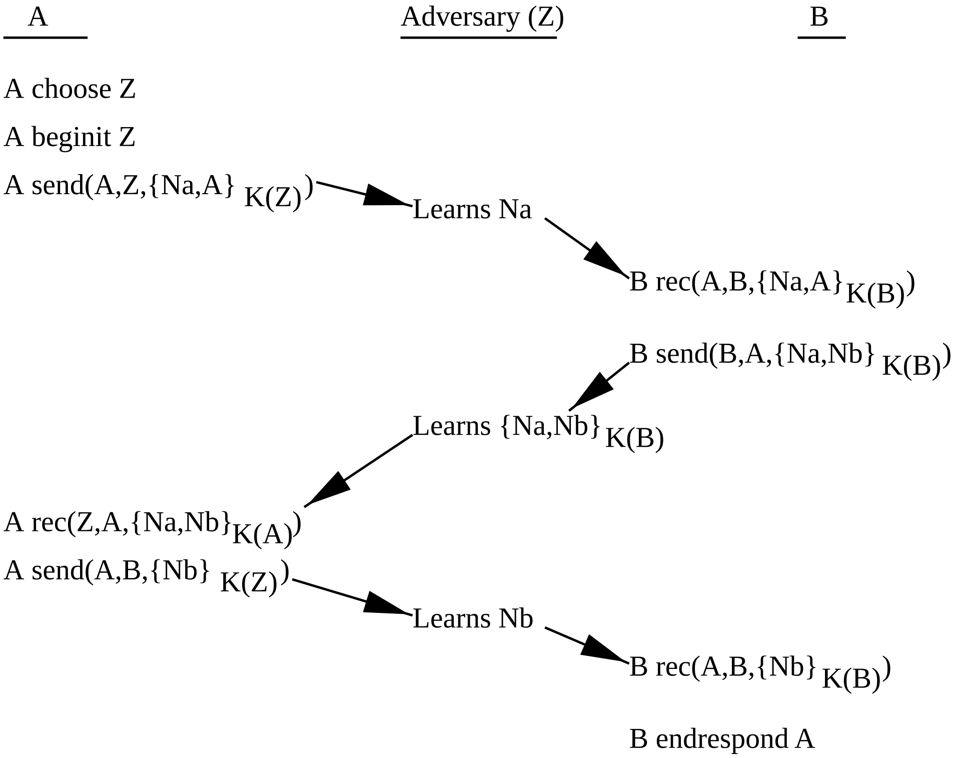

Fig. 2. Needham-Schroeder attack.

$$\forall X.(S_Z\ \mathbf{Knows}\ X.n_a \vee S_Z\ \mathbf{Knows}\ X.n_b) \rightarrow$$
$$\Diamond_P[X\ \mathbf{internal}\ (\text{“}\texttt{begin-initiate}\text{”}, Z)$$
$$\vee X\ \mathbf{internal}\ (\text{“}\texttt{begin-respond}\text{”}, Z)]$$

In other words, if the adversary knows the value of someone else's nonce

$$(S_Z \ \mathbf{Knows}\ X.n_a \vee S_Z \ \mathbf{Knows}\ X.n_b)$$

then that instance must be executing the protocol with the adversary,

$$\Diamond_P[X \ \mathbf{internal}\ (\text{“}\texttt{begin-initiate}\text{''}, N_Z)$$

$$\vee X \ \mathbf{internal}\ (\text{“}\texttt{begin-respond}\text{''}, N_Z)]$$

Recall that S_Z is the instance corresponding to the adversary. This property is also violated by the same trace as before. Four actions from the end of the trace in Figure 1, $B2$ sends $A2$'s nonce encrypted with the key of the adversary Z. At that point the adversary knows $A2$'s nonce, but $A2$ is not trying to authenticate with the adversary.

—*Nonrepudiation:* We would like to be sure that at the end of the protocol both parties actually possess the correct shared secrets. This is a weak notion of nonrepudiation. We are not showing that a principal can prove that someone else possesses the secret. We are simply checking that there is no execution in which some principal A finishes authenticating with B but in which B does not know A's nonce.

$$\forall A_0 \,.\, A_0 \ \mathbf{internal}\ (\text{“}\texttt{end-initiate}\text{''}, A_0.b) \rightarrow$$

$$\exists B_0 \,.\, (pr(B_0) = A_0.b) \wedge (B_0 \ \mathbf{Knows}\ A_0.n_a)$$

The formula given above states that for all instances A_0, if A_0 ends an instance with the partner $A_0.b$

$$A_0 \ \mathbf{internal}\ (\text{“}\texttt{end-initiate}\text{''}, A_0.b)$$

then there should exist an instance B_0 such that the agent or principal executing the session is $A_0.b$ and such that the nonce corresponding to A_0 is known to B_0,

$$\exists B_0 \,.\, (pr(B_0) = A_0.b) \wedge (B_0 \ \mathbf{Knows}\ A_0.n_a).$$

Like the authentication requirement, this requirement has a second formula with the roles of the initiator and responder reversed.

$$\forall B_0 \,.\, B_0 \ \mathbf{internal}\ (\text{“}\texttt{end-respond}\text{''}, B_0.a) \rightarrow$$

$$\exists A_0 \,.\, (pr(A_0) = B_0.a) \wedge (A_0 \ \mathbf{Knows}\ B_0.n_b)$$

Unlike the authentication requirement, however, the protocol satisfies this property. It is the case that if an agent A believes it has finished authenticating with agent B, it has sent out its nonce encrypted with B's key and has received the nonce back again. The only way this is possible

$\underline{SALT_C}$: Random number generated by C used to salt $DESC$
$\underline{PRICE}$: Amount and currency
$\underline{DATE}$: Merchant's date/time stamp
$\underline{NONCE_M}$: Merchant's nonce (random number)
$\underline{ID_M}$: Merchant's ID.
$\underline{TID_M}$: Transaction ID (unique).
$\underline{DESC}$: Description of the goods and delivery information.
$\underline{CAN}$: Customer's account number.
$\underline{R_C}$: Random number chosen by C to form CID.
$\underline{\textbf{Y/N}}$: Yes or no response from credit card authority.

Fig. 3. 1KP atomic messages.

is if B decrypted the nonce so that B must have possession of it, even in the traces where the other two requirements are violated.

Gavin Lowe has suggested a very simple fix for this protocol [Lowe 1996]. The second message, A, B, $\{N_a, N_b\}_{K_A}$, is replaced with A, B, $\{B, N_a, N_b\}_{K_A}$. This slightly changes the protocol so that now at step 2, B generates this new message and so that at step 3, A decrypts the message and checks not only for its own nonce N_a, but also for the identity of B, the principal with whom A is trying to authenticate. When we made this change in our model of the Needham-Schroeder protocol, we could not find a counterexample to the properties given above.

4.2 1KP

We now describe an analysis of 1KP, a member of the *i*KP family of protocols for secure electronic payments over the Internet [Bellare et al. 1995]. The protocol has three participants: a customer, a merchant, and a credit card authority which we will refer to as C, M, and A respectively. Because there are so many atomic messages and because the messages used are quite large we define the atomic messages in Figure 3 and some composite fields in Figure 4. Also, we use $\mathcal{H}(\cdot)$ to denote a one-way hash function. We model $\mathcal{H}(\cdot)$ in BRUTUS by having a special private key called *hash* that has no inverse. This way we can compare results of encrypting with *hash* (applying the hash), but we cannot decrypt the hash (invert the hash function).

We can now turn to the definition of the protocol. It should be noted that there is an assumption that the customer and merchant somehow arrive at the description of the transaction outside of the 1KP protocol. In other words, at the time the protocol is executed, the customer and merchant should already know the values of *DESC* and *PRICE* due to some previous negotiation step.

(1) *Initiate:* The customer generates two random numbers, $SALT_C$ and R_C. It also computes $CID = \mathcal{H}(R_C, CAN)$. It then sends a message to the merchant consisting of the random number $SALT_C$ and the customer

$\underline{CID}$: A customer pseudo-ID formed by $\mathcal{H}(R_C, CAN)$.
$\underline{Common}$: $PRICE, ID_M, TID_M, DATE, NONCE_M, CID, \mathcal{H}(DESC, SALT_C)$
$\underline{Clear}$: $ID_M, TID_M, DATE, NONCE_M, \mathcal{H}(Common)$
$\underline{SLIP}$: $PRICE, \mathcal{H}(Common), CAN, R_C$

Fig. 4. 1KP composite messages.

pseudo-ID *CID*. *CAN* is the customer's account number (see Figure 3). In general, refer to Figures 3 and 4 for the explanation messages.

$$C \rightarrow M : SALT_C, CID$$

(2) *Invoice:* The merchant recovers the values of $SALT_C$ and *CID*. It also generates the values $NONCE_M$ and TID_M. The merchant already knows *PRICE*, *DATE*, and its own identity ID_M so it can create the composite message *Common*. It uses all these components, along with the hash function, to construct the compound message *Clear* as defined in Figure 4 and sends it to the customer.

$$M \rightarrow C : Clear$$

(3) *Payment:* The customer receives *Clear* and retrieves the values ID_M, *DATE*, TID_M, and $NONCE_M$. Since the customer already has *PRICE* and *CID*, it can form *Common*. It computes $\mathcal{H}(Common)$ and checks that this matches what was received in *Clear*. It already has the information necessary to form *SLIP* (Figure 4) and then encrypts this and sends it to the merchant. At this point the customer commits to the transaction.

$$C \rightarrow M : \{SLIP\}_{K_A}$$

(4) *Auth-Request:* The merchant receives the encrypted payment slip and now needs to get authorization from the credit card authority. It forwards the encrypted slip to the authority, along with *Clear* and $\mathcal{H}(DESC, SALT_C)$ so that the authority can check the validity of the *SLIP*. At this point, the merchant is committing to the transaction.

$$M \rightarrow A : Clear, \mathcal{H}(DESC, SALT_C), \{SLIP\}_{K_A}$$

(5) *Auth-Response:* The authority receives the authorization request and performs the following actions.

—The authority extracts the values ID_M, TID_M, *DATE*, and $NONCE_M$ and checks that there is no previous request with these same values. It also extracts the value h_1 which is supposed to be $\mathcal{H}(Common)$.

—The authority decrypts the encrypted *SLIP*. If the decryption is successful, it now has *SLIP* and can extract *PRICE*, *CAN*, R_C, and the value h_2 which is supposed to again be $\mathcal{H}(Common)$.

—The authority verifies that $h_1 = h_2$ which ensures that the customer and merchant agree on the transaction.
—It now also has all the components to construct *Common* and does so. It computes $\mathcal{H}(Common)$ and compares this to the value h_1.
—Assuming everything is in order, it can authorize the payment by signing the pair Y, $\mathcal{H}(Common)$ (Y refers to yes, and N refers to no) and returning this to the merchant.

$$A \rightarrow M : Y, \{Y, \mathcal{H}(Common)\}_{K_A^{-1}}$$

(6) *Confirm:* The merchant receives the authorization response. Assuming the response is Y, it then verifies that it has received a valid signature of Y, $\mathcal{H}(Common)$ from the authority. If so, it forwards this on to the customer who can check it for himself.

$$M \rightarrow C : Y, \{Y, \mathcal{H}(Common)\}_{K_A^{-1}}$$

Like the Needham-Schroeder protocol, the BRUTUS definition of the 1KP protocol consists of a set of instances with at least one instance for each role in the protocol. These roles also consist of mostly send and receive actions with a few internal actions to mark commit points in the protocol as well as when the authority debits and credits accounts. The major difference between the two models is the large size of the 1KP protocol. Not only are there three roles instead of two and six messages instead of three, but the size and complexity of the messages is greatly increased. The messages appearing in the Needham-Schroeder protocol are quite simple. In 1KP, the messages contain many more fields (so many that we had to define composite fields like *Common* and *SLIP* to simplify the presentation). The messages also have multiple levels of nested encryption, signatures, and hashes. Because of this, the number of messages that the adversary can generate in an attempt to subvert the protocol is much greater as is the number of reachable states. Hence, we were forced to look at a smaller model, one in which there was only a single instance of a customer and merchant and two instances of the authority. Using this configuration we were able to perform the analysis, and verify the following properties proposed by the authors in Bellare et al. [1995].

—*Proof of transaction by customer:* When the authority debits a credit card account by a certain amount, it must have unforgeable proof that the credit card owner has authorized the payment. The authors argue that *SLIP* provides this unforgeable proof. While verifying that *SLIP* does provide a proof is outside the capabilities of BRUTUS, we can check that the authority only debits credit cards when it possesses *SLIP*.

$$\forall A_0 \,.\, (pr(A_0) = A) \wedge (A_0 \textbf{ internal } (\texttt{"debit"}, \langle A_0.c, A_0.price\rangle)) \rightarrow$$

$$A_0 \textbf{ Knows } SLIP$$

The formula given above states that for all instances A_0 if the principal executing the instance is an authority $((pr(A_0) = A))$ and a debit action is executed

$$(A_0 \textbf{ internal } (\texttt{"debit"}, \langle A_0.c, A_0.price \rangle))$$

then A_0 must know the slip (A_0 **Knows** *SLIP*). Here *SLIP* is defined as before, but with $A_0.c$ in place of the customer account number *CAN*, and $A_0.price$ instead of the constant *PRICE* which appeared in the definitions. In other words, we are checking that the authority does indeed have a proof with the values that are consistent with this particular debit.

—*Unauthorized payment is impossible:* For this property we require that the authority debit a customer's account, only if the customer has authorized the debit. We are no longer asking if the authority has some kind of a proof, but whether or not the customer did actually authorize the debit.

$$\forall A_0 . (pr(A_0) = A) \wedge (A_0 \textbf{ internal } (\texttt{"debit"}, \langle A_0.c, A_0.price \rangle)) \rightarrow$$

$$\exists C_0 . (pr(C_0) = A_0.c) \wedge \Diamond_P (C_0 \textbf{ internal } (\texttt{"authorize"}, A_0.price))$$

In other words, if some instance A_0 of the authority debits the principal $A_0.c$ (presumably the customer) by the amount $A_0.price$

$$(A_0 \textbf{ internal } (\texttt{"debit"}, \langle A_0.c, A_0.price \rangle),)$$

then there must be some instance executing on behalf of $A_0.c$, who has authorized a debit of the same amount ($A_0.price$)

$$(pr(C_0) = A_0.c) \wedge \Diamond_P (C_0 \textbf{ internal } (\texttt{"authorize"}, A_0.price)).$$

We should note that this by itself does not guarantee the absence of a replay attack, one in which an adversary simply replays a previous message to cause a second transaction to take place. In fact, the first thing the adversary does when it receives an authorization request is to make sure that there is no previous request with the same transaction ID, date, and nonce. Since BRUTUS does not allow for this kind of check, one would expect there to be a replay attack in our model. So we checked the following formula.

$$\neg[A1 \textbf{ internal } (\texttt{"debit"}, \langle C, A1.price \rangle)$$

$$\wedge A2 \textbf{ internal } (\texttt{"debit"}, \langle C, A2.price \rangle)]$$

If there is no replay attack in our model, this formula should hold, since there is a single customer instance and a single merchant instance, so there should be at most one debit. However, there is a counterexample to this formula in which the adversary simply sends the single authorization request from the merchant to both authority instances causing them both to perform debits.

—*Privacy:* Customers want to ensure that the merchant is the only other party that knows the details of the transaction. In addition, we would want to keep the customer's credit card number secret as well. For this we have the following two formulas:

$$\forall S_0 . [S_0 \textbf{ Knows } C1.DESC \rightarrow (pr(S_0) = C \vee pr(S_0) = M)]$$

$$\forall S_0 . [S_0 \textbf{ Knows } C1.CAN \rightarrow (pr(S_0) = C \vee pr(S_0) = A)]$$

In other words, if some instance S_0 knows the customer's description of the transaction

$$(S_0 \textbf{ Knows } C1.DESC)$$

then S_0 must be the customer or the merchant

$$((pr(S_0) = C \vee pr(S_0) = M)).$$

Similarly, the second formula states that if some instance S_0 knows the customer's account number, then S_0 must be either the customer or the authority.

—*Proof of transaction authorization by merchant:* This particular property is not claimed to hold of 1KP. It is meant to hold for 2KP and 3KP, but we tried to verify a variant of it in our model for 1KP. We would like to check if all transactions allowed by the authority are authorized by the *merchant*.

$$\forall A_0 . (pr(A_0) = A) \wedge (A_0 \textbf{ internal } (\text{“}\texttt{credit}\text{”}, \langle A_0.m, A_0.price \rangle)) \rightarrow$$

$$\exists M_0 . (pr(M_0) = A_0.m) \wedge \Diamond_P (M_0 \textbf{ internal } (\text{“}\texttt{Mauthorize}\text{”}, A_0.price))$$

In other words, if the authority credits principal $A_0.m$ (presumably the merchant) by the amount $A_0.price$

$$(A_0 \textbf{ internal } (\text{“}\texttt{credit}\text{”}, \langle A_0.m, A_0.price \rangle))$$

then it must be the case that $A_0.m$ authorized a payment of that amount

$$(pr(M_0) = A_0.m) \wedge \Diamond_P (M_0 \textbf{ internal } (\text{“}\texttt{Mauthorize}\text{”}, A_0.price)).$$

This particular analysis did provide some insight. Although the authors do not claim that 1KP guarantees this property, there was nothing about the protocol that suggested to us that this property could be violated. Indeed BRUTUS finds a counterexample. All the adversary needs is the merchant's ID (ID_M), and some account number to debit. With this information, no one else other than the authority need participate in order to have the authority authorize payments.

4.3 Wide Mouthed Frog

The Wide Mouthed Frog protocol is intended to distribute a freshly generated session key to another party. In the description that follows, A is the initiator wishing to communicate with B, the responder. S is a trusted third party with whom both A and B share a symmetric key. The protocol proceeds as follows:

(1) $A \rightarrow S : A, \{T_a, B, K_{ab}\}_{K_{as}}$

(2) $S \rightarrow B : \{T_s, A, K_{ab}\}_{K_{bs}}$

Implicit in this description is the fact that the parties are checking the timestamps, although how exactly this is done is not specified. One reasonable assumption is that there is some time window for accepting messages. In other words, if a message with timestamp T arrives, it is accepted as long as the current time is less than $T + \delta$ for some specified time window δ. It would then be reasonable to require that if B accepts a key K_{ab} from S with which to communicate with A, then A must have requested that S forward that key within some other time window ϵ. In other words, if B accepts key K_{ab} at time T, then A must have originally sent the key to S at some time T' such that $T' > T - \epsilon$.

Because BRUTUS does not have a notion of time, we do not include timestamps in our model. In our model we simply remove all timestamps from the messages. Our abstracted protocol becomes

(1) $A \rightarrow S : A, \{B, K_{ab}\}_{K_{as}}$

(2) $S \rightarrow B : \{A, K_{ab}\}_{K_{bs}}$.

Since the protocol depends on timestamps to secure against replay, our model exhibits trivial replay attacks. This points to the fact that in the original protocol, a replay attack was possible within the time window allowed by S and B. In other words, if the adversary replayed A's initial message fast enough (before S could start rejecting the message), then S could conceivably send it's message to B in time for B to accept it, thus resulting in a state where B has authenticated twice while A only authenticated once.

However, because we do not model time, we are not able to characterize the time window requirement discussed above. This means that in particular we cannot find the known attack where the adversary uses S as a timestamp oracle. This attack is given below:

$$\begin{array}{ccccccc}
1. & A & \rightarrow & Z(S) & : & A, \{T_a, B, K_{ab}\}_{K_{as}} \\
1^A. & Z(A) & \rightarrow & S & : & A, \{T_a, B, K_{ab}\}_{K_{as}} \\
2^A. & S & \rightarrow & Z(B) & : & \{T_s^A, A, K_{ab}\}_{K_{bs}} \\
1^B. & Z(B) & \rightarrow & S & : & B, \{T_s^A, A, K_{ab}\}_{K_{bs}} \\
2^B. & S & \rightarrow & Z(A) & : & \{T_s^B, B, K_{ab}\}_{K_{as}} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
1^X. & Z(A) & \rightarrow & S & : & A, \{T_s^W, B, K_{ab}\}_{K_{bs}} \\
2^X. & S & \rightarrow & Z(B) & : & \{T_s^X, A, K_{ab}\}_{K_{as}} \\
2. & Z(S) & \rightarrow & B & : & \{T_s^X, A, K_{ab}\}_{K_{as}}
\end{array}$$

In the attack given above, $Z(S)$ represents the adversary masquerading as the server. A similar explanation holds for $Z(B)$ and $Z(A)$. Notice that the final message sent (message labeled 2 with timestamp T_s^X) is sent much later than the original message from A (labeled 1 with timestamp T_a). Presumably enough time has gone by that $T_a < T_s^X$. However, by playing this game, the adversary has kept each individual message within the appropriate time window. In particular, this means that the current time is less than $T_s^X + \delta$, and so B accepts the message from the adversary. This violates the property we discussed earlier. Namely, B does accept a stale message (a message that is older than ϵ time units).

Despite these limitations, the Wide Mouthed Frog protocol is a good protocol to illustrate the advantage of using the symmetry reduction techniques. We have modeled this protocol without timestamps as discussed above. With this model we were able to find the errors discussed above (namely that a replay is possible within the time window allowed). We also demonstrated that the protocol itself provides no guarantees to A about B's participation. A can finish executing the protocol without B ever participating. These errors were all found despite the fact that we were eliminating quite a few traces from consideration because of the symmetry reduction. The symmetry reduction is most effective when we check a property that is correct, such as the property that if B receives a key, then that key did originally come from A. The exhaustive search performed is significantly reduced by eliminating symmetric traces from consideration during the search.

We also check authentication using the same pair of formulas we used to check authentication in the Needham-Schroeder protocol. However, we only check half of the property.

$$\forall B_0 . B_0 \textbf{ internal } (\text{“}\texttt{end-respond}\text{''}, B_0.a) \rightarrow$$

$$\exists A_0 . (pr(A_0) = B_0.a) \wedge \Diamond_P (A_0 \textbf{ internal } (\text{“}\texttt{begin-initiate}\text{''}, pr(B_0)))$$

This property does hold of our model. The other half of the property (if the initiator finishes then the responder must have participated) does not

```
funct $dfs(s)$
  $EN\ (s) = \{\ \alpha \mid en(s, \alpha)\ \}$
  foreach $\alpha \in EN\ (s)$
    do $dfs(\alpha(s))$
```

Fig. 5. Depth-first search algorithm $\mathcal{A}$.

hold in our model. We can see this because the only thing the initiator does in the protocol is send the first message. Hence it could finish executing before/without a corresponding responder executing if the adversary simply prevents the message from reaching the responder.

We can also check the secrecy property. Here we would like to verify that the adversary does not gain knowledge of the session key. Again, this requirement need only hold when the initiator is not trying to execute the protocol with the adversary.

$$\forall X . S_Z\ \textbf{Knows}\ X.k \rightarrow \Diamond_P[X\ \textbf{internal}\ (\texttt{"begin-initiate"}, N_Z)]$$

Again, this property holds of our model.

5. BASIC ALGORITHM AND REDUCTIONS

In this section we first describe the general algorithm used in BRUTUS. The algorithm used in BRUTUS is based on a combination of state space exploration and natural deduction. Most techniques based state space exploration for verifying security protocols suffer from the well-known *state-explosion problem*, i.e., the state space of the system grows exponentially in the number of components. In the domain of model-checking for reactive systems there are numerous techniques for reducing the state space of the system. Two such widely used techniques are *partial-order* and *symmetry* reductions. We have developed variants of these techniques in the context of verification of security protocols. We also describe partial-order and symmetry reductions that are used in BRUTUS. Discussion of these reduction techniques is kept at an informal level, and proof of correctness of these reduction techniques is not provided. However, we mention references that discuss these reductions and their proof of correctness in considerable detail.

5.1 General Algorithm

The algorithm used in BRUTUS explores the state space using depth-first search. During the depth-first search if the algorithm encounters a state where the property being checked is not true, it stops and demonstrates a counterexample. The basic depth-first search algorithm is shown in Figure 5. If an action α is enabled at a state s, then the predicate $en(s,\ \alpha)$ is true. Hence, the set of actions enabled at the state s (denoted by $EN(s)$) is

$$\{\alpha \mid en(s,\ \alpha)\}.$$

$$\frac{m_1 \qquad m_2}{m_1 \cdot m_2}\,\cdot\mathcal{I} \qquad\qquad \frac{m_1 \cdot m_2}{m_1}\,\cdot\mathcal{E}_l \qquad \frac{m_1 \cdot m_2}{m_2}\,\cdot\mathcal{E}_r$$

$$\frac{m \qquad k}{\{m\}_k}\,\{\}_k\mathcal{I} \qquad\qquad \frac{\{m\}_k \qquad k^{-1}}{m}\,\{\}_k\mathcal{E}$$

Fig. 6. Derivation rules for messages.

$$\frac{\dfrac{\dfrac{\{a\}_k \cdot b}{\{a\}_k}\,\cdot\mathcal{E}_l \qquad k^{-1}}{a}\,\{\}_k\mathcal{E} \qquad \dfrac{\{a\}_k \cdot b}{b}\,\cdot\mathcal{E}_r}{a \cdot b}\,\cdot\mathcal{I}$$

Fig. 7. A derivation tree for $\{\{a\}_k \cdot b,\, k^{-1}\} \vdash a \cdot b$.

As can be seen from the description, the algorithm basically explores every action enabled from state s. The state reached from s by executing action α is denoted by $\alpha(s)$. Standard bookkeeping details common to all model-checking algorithms are not shown. However, recall that in our logic at the atomic level we can refer to the knowledge of a principal (e.g., Z **Knows** m), so during the search we often must decide if $m \in \bar{I}$ where I is the set of messages known to a principal or the adversary and where $\bar{I}$ denotes the set of all messages that can be constructed from I using the standard message operations (pairing, projection, encryption, and decryption). If we describe each of these message operations as inference rules, then the question of whether or not $m \in \bar{I}$ is equivalent to whether or not m is derivable from I. The problem now looks very similar to natural deduction, and in fact we will use ideas and terms from natural deduction to describe an efficient algorithm for deciding if $m \in \bar{I}$ and prove its correctness. This combination of state space exploration and natural deduction-style reasoning distinguishes BRUTUS from other approaches that are solely based on state space exploration. Next we describe in detail the algorithm for deciding if a message m is in the closure of a set of messages I.

We will assume that the reader is familiar with derivation trees, so we will not formalize them here. We refer the reader to Prawitz [1965] for a good discussion on natural deduction. We will say that a particular message m is derivable from a set of messages I ($I \vdash m$), if there exists a valid derivation tree using the inference rules shown in Figure 6, such that m appears at the bottom of the tree and such that all messages appearing at the top of the tree are contained in I. An example of such a tree can be found in Figure 7.

```
funct $add(I, m)$
  foreach $i \in I$ do
    if $i = \{x\}_y \wedge m = y^{-1}$
      then $I := add(I, x)$
    fi
  od
  if $m \in \mathcal{A}$
    then return $I \cup \{m\}$
  elsif $m = x \cdot y$
      then return $add(add(I, x), y)$
  elsif $m = \{x\}_y \wedge y^{-1} \in I$
      then if $y \in I$
              then return $add(I, x)$
               else return $add(I \cup \{m\}, x)$
           fi
  else
      return $I \cup \{m\}$
  fi
```

Fig. 8. Augmenting the adversary's knowledge.

Note that each message construction operation (pairing and encryption) is characterized by a pair of inference rules. One is an introduction rule that creates a new message whose principal connective is that operation. For example, the $\{\}_k\mathcal{I}$ rule creates a new encrypted message $\{m\}_k$ from the message m and key k. The second is an elimination rule that removes that particular operation from the compound message. For example the $\cdot\mathcal{E}_l$ rule takes a message $m_1 \cdot m_2$ and returns its left component m_1. In Clarke et al. [1998] we proved Theorem 1.

THEOREM 1. *Let I be a set of messages, and let m be a message derivable from I ($I \vdash m$). Then there exists a derivation tree T for m from assumptions in I such that all elimination rules appear above all introduction rules. We call such derivation trees normalized.*

This theorem is the key to how we maintain each session's knowledge. Whenever a session learns a new message, we add the new message to the set of known messages and close under elimination rules (projection and decryption). Since a session cannot gain any new knowledge by applying elimination rules, any new messages must be constructed using introduction rules. Theorem 1 guarantees that we cannot gain anything by trying to apply elimination rules once we have started applying introduction rules. We formally prove this idea correct below.

Definition 1. $I \vdash_{\mathcal{I}} m$ if and only if there exists a valid derivation tree using only the introduction rules ($\cdot\mathcal{I}$ and $\{\}_k\mathcal{I}$), such that m appears at the bottom of the tree and such that all messages appearing at the top of the tree are contained in I.

THEOREM 2. *Let I be a set of messages, I^* be the closure of I under*

```
funct in(I, m)
  if $m \in I$
    then return true
  elsif $m = x \cdot y$
      then return $in(I, x) \wedge in(I, y)$
  elsif $m = \{x\}_y$
      then return $in(I, x) \wedge in(I, y)$
  else
      return false
  fi
```

Fig. 9. Searching the adversary's knowledge.

elimination rules, and m be a message. $I \vdash m$ *if and only if* $I^* \vdash_{\mathcal{I}} m$ *where* $\vdash_{\mathcal{I}}$ *denotes derivability using only introduction rules.*

Proof of Theorem 2 follows easily from Definition 1 and Theorem 1. Theorem 2 proves the correctness of our algorithm. Recall, whenever an instance learns a new message, we close its knowledge under elimination rules (construct I^*). Whenever we check whether $I \vdash m$, we use only introduction rules, but we are actually using I^* as the set of assumptions. In other words, we are checking $I^* \vdash_{\mathcal{I}} m$ which is equivalent to $I \vdash m$ by Theorem 2.

The pseudocode for these algorithms is given in Figures 8 and 9. Figure 8 implements the closure operation that recomputes I^* whenever a new message is added to I. Notice that whenever *add* generates a new message that should be included in the closure, we recursively call *add* because we will need to compute the closure again once we add this new message. Figure 9 implements the search for a derivation using only introduction rules. If $\{m\}_k \notin I^*$ then $I^* \vdash_{\mathcal{I}} \{m\}_k$ can be true only if there are derivation trees for m and k that can be combined using $\{\}_k\mathcal{I}$ to get a tree for $\{m\}_k$. The same holds for $I^* \vdash_{\mathcal{I}} m_1 \cdot m_2$.

Termination for both algorithms follows from the fact that the recursive calls are performed on submessages. When closing under elimination rules, newly generated messages are smaller than previously existing messages. In the worst case, we will end up adding all submessages of the messages in I, but this is guaranteed to be finite. Similarly, when we are searching for a derivation for m using introduction rules, we recursively search for derivations of submessages of m. Since there are only finitely many submessages, again the algorithm terminates.

5.2 Partial-Order Reductions

Partial-order reductions reduce the search space by ignoring redundant interleavings. The theory of partial-order reductions is well developed in the context of verification of reactive systems [Godefroid et al. 1996; Peled 1996; Valmari 1991]. In this section we describe partial-order reductions as they are employed in BRUTUS. For a detailed description and the proof of correctness the reader is referred to Clarke et al. [2000].

```
funct dfs($s$)
  $EN\ (s) = \{\ \alpha \mid en(s, \alpha)\ \}$
  foreach $\alpha \in ample(EN(s))$
    do dfs($\alpha(s)$)
```

Fig. 10. Modified depth-first search algorithm $\mathcal{A}_{\mathcal{PO}}$.

As already pointed out, the basic algorithm for verifying whether a protocol satisfies a specification works by exploring the state space starting from the initial state using depth-first search. In the ensuing discussion we will focus on the depth-first search algorithm. Recall that algorithm $\mathcal{A}$ given in Figure 5 performs the depth-first search starting from state s. The predicate $en(s,\ \alpha)$ is true if action α is enabled in the state s, i.e., action α can be executed from the state s. The set of enabled actions $EN(s)$ in a state s is $\{\alpha \mid en(s,\ \alpha)\}$. Algorithm $\mathcal{A}_{\mathcal{PO}}$ (shown in Figure 10) is the modified depth-first search procedure with partial-order reductions. The set of actions $ample(EN(s))$ is defined as follows:

—If $EN(s)$ contains an invisible internal action, then $ample(EN(s))$ is an arbitrary invisible action $\{A\}$ picked from $EN(s)$.

—Suppose $EN(s)$ does not contain an invisible internal action, but does contain a **send** action. In this case $ample(EN(s))$ is an arbitrary **send** action picked from the set $EN(s)$.

—If $EN(s)$ does not contain an invisible internal action or a **send** action, $ample(EN(s))$ is equal to the entire set of enabled actions $EN(s)$.

In the original algorithm $\mathcal{A}$ every action that is enabled from a state is explored during the depth-first procedure. In the modified algorithm $\mathcal{A}_{\mathcal{PO}}$ only a subset of the enabled actions from a state s (denoted by $ample(EN(s))$ are explored. As will be clear from the experimental section, this results in considerable savings in the size of the state space.

Note. Partial-order reductions presented here are only sound if the specifications have a restricted form. In the specification, atomic propositions that pertain to the adversary can only appear under an odd number of negation signs. Roughly, this means that one can express that the *adversary does not know something*, but cannot assert that the *adversary knows something*. Fortunately, most specifications we have encountered have this property. The reader can check that all specifications that appear in this article have this property. For a detailed discussion of this question please refer to Clarke et al. [2000].

5.3 Symmetry Reductions

Intuitively, the correctness of the protocol should not depend on the specific assignment of principal names. In other words, by permuting the names of the principals the correctness of the protocol should be preserved. Symmetry reductions have proved very useful in the verification of reactive systems [Clarke et al. 1996; Emerson and Sistla 1996; Ip and Dill 1996; Jensen 1996]. We have also developed the theory of symmetry reductions in the context of verifying security protocols. Intuitively the state space is partitioned into various equivalence classes because of the inherent symmetry present in the system. During the verification process our algorithm only considers one state from each partition. Specific details and the correctness of the symmetry reduction will be discussed in Will Marrero's forthcoming thesis.

Because our models often include replicated components, they often exhibit symmetry. For example, imagine a protocol that has two parties, an initiator who sends the first message (A) and a responder who receives the first message (B). When we model the protocol, we would like to allow for the possibility of multiple executions of the protocol by the same parties. Let us assume then that in our model we create three instances for each participant (A_1, A_2, A_3, B_1, B_2, B_3) so that each A and B can attempt to execute the protocol three times. In the start state, the A_i's are symmetric and the B_i's are also symmetric. They are identical up to their names or instance IDs. At this point in time, anything A_1 can do, A_2 or A_3 can do and similarly for the B_i's.

To see this, imagine some trace involving these six instances. Because A_1 and A_2 were identical in the start state, we could imagine them swapping roles in the trace. Every action A_1 took is now taken by A_2 and vice versa. This should still be a valid trace, since A_1 and A_2 are identical. Now, to take advantage of this symmetry, we note that the specification should not refer to the individual instances. Properties should talk about the principals and not about the individual instances. If this is the case, then the property is insensitive to this symmetry. In other words, we can swap A_1 and A_2 in the requirement specification (usually because they do not even appear in the requirement) without altering the meaning of the formula. In fact, in the usual case, the permuted formula is syntactically identical to the original formula. The result is that the original trace satisfies the original formula if and only if the permuted trace satisfies the permuted formula. Since the original formula and the permuted formula are the same, either both traces satisfy the formula or neither trace satisfies the formula. Therefore, we need only check one of these two cases and we know the result of the other.

In the example given above, we considered a particular trace. Now consider the fact that we can perform these permutations on *any* trace. In any trace, we can swap the roles or behavior of A_1 and A_2. Each trace we consider is symmetric to some other trace we are considering (actually, the roles or behavior of A_1 and A_2 could be identical in a particular trace, and therefore swapping A_1 and A_2 will yield the same trace). So we could restrict ourselves to about half of the set of traces, none of which are symmetric to each other. We would then be assured that if there is an error

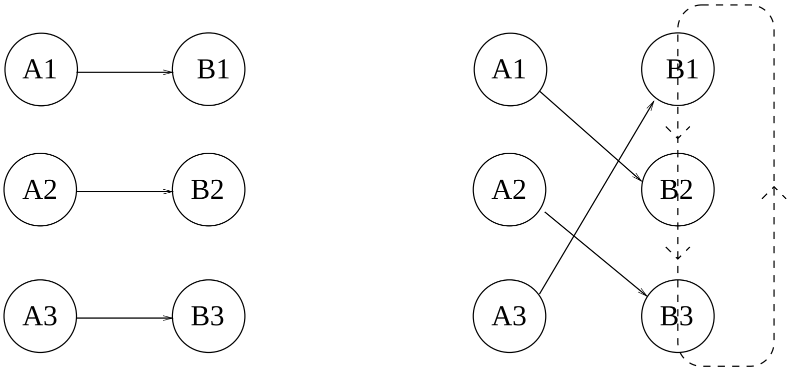

Fig. 11. Exploiting symmetry.

in any of the traces we did not consider, we would still be checking a symmetric trace which would exhibit the same error.

Now consider the fact that we exploited a specific symmetry between A_1 and A_2. In general, all the A_i's are symmetric, and all the B_i's are symmetric. Any permutation on the A_i's and on the B_i's would yield the same results as above, symmetric traces and identical requirement. For example, consider the graphs in Figure 11. The graph on the left is supposed to represent some trace in which B_1 is responding to A_1, B_2 is responding to A_2, and B_3 is responding to A_3. The graph on the right represents the symmetric trace where B_1 is responding to A_3, B_2 is responding to A_1, and B_3 is responding to A_2. In fact, these graphs do not represent specific traces, but any traces that satisfy the sender responder relationships mentioned above. It should be clear that for any trace represented by the graph on the right there is a symmetric trace represented by the graph on the left. This symmetric trace is found by permuting the B_i's in a cycle as indicated on the right. By doing so we get the symmetric trace on the left.

As we have seen, the different symmetries induce an equivalence on the set of traces. All the traces in an equivalence class agree on the requirement, i.e., *they all satisfy the requirement, or none of them satisfies the requirement*. So it is sufficient to check a single representative from each equivalence class. When we do this, how much work do we save? To answer this, we examine how large is each equivalence class. For a model with i initiators and j responders, there are factorial of i (denoted by $i!$) symmetries on the initiators and $j!$ symmetries on the responders for a total of $i!\,j!$ symmetries. So every trace in which the instances exhibit different behavior belongs to an equivalence class of size $i!\,j!$. So in general we can reduce the number of traces that we need to consider by about $i!\,j!$.

The question becomes how do we know when such a symmetry exists. This question is very difficult in general. Our solution is to demonstrate and exploit a number of traces that are guaranteed to be symmetric in *any* model in BRUTUS. We will consider a particular point in the execution of a protocol, namely the point when the responder in the protocol is ready to receive its first message. For the sake of simplicity, we will assume that the model has one initiator A that can participate twice and one responder B that can participate twice. The initiator is simply the principal that sends the first message, and the responder receives the first message. Consider the point when B is ready to receive its first message. There is nothing in the execution so far that distinguishes between the two instances of B. Any trace that results from $B1$ receiving that first message has a symmetric trace in the model where $B2$ behaves as $B1$ and receives that first message. So at this point we arbitrarily choose $B1$ to receive the first message and ignore the case where $B2$ receives the first message. Now let us assume that both $A1$ and $A2$ have sent their first messages. So $B1$ could receive either one. Again we are in a state which is symmetric with respect to $A1$ and $A2$. Any trace that results from $B1$ receiving $A1$'s message has a symmetric trace where $A2$ behaves as $A1$ and $B1$ receives $A2$'s message instead. So at this point, we arbitrarily choose to have $B1$ receive $A1$'s message and ignore the case where $B1$ receives $A2$'s message. Note, however, that now the symmetry is broken. $A1$ can be distinguished from $A2$ because someone has received $A1$'s message while no one has received $A2$'s message. Similarly, $B1$ can be distinguished from $B2$ because $B1$ has received its first message while $B2$ has not.

6. EXPERIMENTAL RESULTS

Table I summarizes the results of applying these reductions to a few protocols. We examined the 1KP secure payment protocol [Bellare et al. 1995], the Needham-Schroeder public key protocol [Needham and Schroeder 1978], and the Wide Mouthed Frog protocol [Burrows et al. 1989; Schneier 1996]. In Table I these protocols are marked as **1KP**, **N-S**, and **WMF** respectively. Columns 2 and 3 give the number of initiator and responder instances used in constructing the model. The other columns give the number of states encountered during state space traversal using exhaustive search or some specific reduction technique. The column marked with **None** refers to results when no reductions were applied. Results corresponding to the partial-order and symmetry reductions are presented in columns marked with **p.o.** and **symm** respectively. Column **p.o.+symm** presents results when both reductions (partial order and symmetry) were applied simultaneously. The entries with an "X" represent computations that were aborted after a day of computation (over 700,000,000 states). Note that there is no reduction when applying symmetry reductions to models that do not have more than one initiator or responder. Notice that for the Needham-Schroeder protocol with one initiator and two responders (third row) we achieve reduction of a factor of around 400. As can be easily seen from the table, the reductions achieved due to the two techniques are significant.

Table I. Table of Results

Protocol	init	resp	None	p.o.	symm	p.o+symm
1KP	1	1	17,905,267	906,307	17,905,267	906,307
N-S	1	1	1,208	146	1,208	146
N-S	1	2	1,227,415	6,503	613,713	3,257
N-S	2	2	X	372,977	X	186,340
N-S	2	3	X	78,917,569	X	12,148,470
WMF	3	3	X	1,286,074	X	7,004
WMF	4	4	X	X	X	455,209
WMF	5	5	X	X	X	47,651,710

7. RELATED WORK

A number of researchers have seen the potential in applying formal methods to analyze security protocols. Some approaches use theorem proving, while others use nonautomated reasoning. Model-checking and rule rewrite systems have also been used. Generally, verification tools based on state-exploration techniques (model-checking being one of them) have the advantage that they are automatic. In addition if the security protocol is incorrect it is easy to demonstrate the flaw by producing a counterexample. The disadvantage of most approaches based on state-exploration over those based on theorem proving is that the process by which the adversary deduces messages from a known set of messages has to be “hard coded” into the model. In theorem-proving-based approaches the rules by which the adversary deduces additional messages are simply additional rules in the logical system. Moreover, techniques based on state space exploration have difficulty handling multiple instances of the same protocol because of the “state-explosion” problem, i.e., the state space grows exponentially in the number of instances. As was already explained before, our tool BRUTUS uses a combination of state space exploration and natural deduction. Hence, the deduction rules corresponding to the adversary can be easily handled. Moreover, because of powerful reduction techniques used in BRUTUS we can model multiple instances for many protocols. Next, we describe each approach in detail and where applicable discuss the specific advantages and disadvantages of each approach over BRUTUS. All except the first approach use the same basic operational model for the adversary, and so the differences result from how one specifies the protocol, how one specifies the properties, how one specifies the adversary, and how the tool goes about trying to perform the analysis.

7.1 Logic of Authentication

One of the earliest successful attempts at formally reasoning about security protocols involved developing logics in which one could express and deduce security properties. The earliest such logic is the Logic of Authentication proposed by Burrows et al. [1989] and is commonly referred to as the BAN logic. Their syntax provided constructs for expressing intuitive properties like

—“A said X” ($A \mid\sim X$)

—“A sees X” ($A \triangleleft X$)

—“A believes X” ($A \models X$)

—“K is a good key for A and B” ($A \overset{K}{\leftrightarrow} B$)

—“X is a fresh message” ($\sharp(X)$)

—“S is an authority on X” ($S \mid\Rightarrow X$).

They also provide a set of proof rules which can then be used to deduce security properties such as “A and B believe K is a good key” ($A \models A \overset{K}{\leftrightarrow} B$ and $B \models A \overset{K}{\leftrightarrow} B$) from a list of explicit assumptions made about the protocol. For example, their inference system provides rules for the following:

—If a message encrypted with a good key K is received by a principal P, then P believes that the other party possessing K said the message.

$$\frac{P \models Q \overset{K}{\leftrightarrow} P, \quad P \triangleleft \{X\}_K}{P \models Q \mid\sim X}$$

—A principal says only things that it believes. Worded differently, if a principal P receives a recent message X from Q, then P believes that Q believes X.

$$\frac{P \models \sharp(X), \quad P \models Q \mid\sim X}{P \models Q \models X}$$

—If a principal P believes that some principal Q has jurisdiction over the statement X, then P trusts Q on the truth of X. If statement X is about an instance key generated by a server, this rule would allow us to infer that participants in a protocol will believe that the instance key is good if it came from a trusted server that has jurisdiction over good keys.

$$\frac{P \models Q \mid\Rightarrow X, \quad P \models Q \models X}{P \models X}$$

—A principal can see the components of compound messages, and a principal can decrypt messages with good keys.

$$\frac{P \triangleleft (XY)}{P \triangleleft X} \qquad \frac{P \models Q \overset{K}{\leftrightarrow} P, \quad P \triangleleft \{X\}_K}{P \triangleleft X}$$

This formalism was successful in uncovering implicit assumptions that had been made in a number of protocols. However, this logic has been criticized for the “protocol idealization” step required when using this formalism. Protocols in the literature are typically given as a sequence of messages. Use of the BAN logic requires that the user transform each message in the protocol into formulas *about* that message, so that the inferences can be made within the logic. For example, if the server sends a message containing the key K_{ab}, then that step might need to be converted into a step where the server sends a message containing $A \overset{K_{ab}}{\leftrightarrow} B$, meaning that the key K_{ab} is a good key for communication between A and B. This simple example is pretty straightforward. In general, however, the idealization step requires that we assign a *meaning* to the messages that appear in the protocol, thus introducing an informal step into the protocol analysis. We should mention, however, that Kindred and Wing [1999] have investigated making this step more formal while also using computer automation to aid with the analysis (we discuss this technique in detail later).

The second objection is that in this kind of analysis all principals are honest, and so it does not allow for a malicious adversary. In other words, we try to argue about how different participants might come to certain beliefs about keys and secrets, but we cannot investigate how a malicious adversary might try to subvert the protocol by possibly modifying and misdirecting messages. For this reason, a number of researchers have looked to analyzing security protocols in a framework that allows for a malicious adversary.

In the tools that follow, researchers have a concrete operational model for how the protocol executes. This operational model includes a description of how the honest participants in the protocol behave (i.e., what it means to execute the protocol) and a description of how an adversary can interfere with the execution of the protocol. The behavior of the adversary has evolved from the model of Dolev and Yao [1989] and allows for the maximum amount of interference from the adversary while maintaining encryption as a black box. The model of the adversary usually allows it to overhear and intercept all messages, misdirect messages, and send fake messages. The adversary can send any message it can generate from previously overheard messages by concatenating and projecting onto components as well as encrypting and decrypting with known keys. The adversary is also allowed to participate in the protocol. In other words it can try to initiate protocol instances, and honest agents are willing to try to initiate instances with the adversary. While the details of *how* this behavior is modeled are different among the different tools, all the tools described below (including BRUTUS) somehow implement this high-level description of an adversary.

7.2 FDR

Gavin Lowe has investigated the use of FDR to analyze CSP [Hoare 1985] models of cryptographic protocols [Lowe 1996; 1997]. CSP seems to be a natural language in which to model asynchronous composition of protocol instances. Each instance of an agent trying to execute the protocol is modeled by a CSP process that alternates between waiting for a message and sending a message. Along with the usual communication channel,

$$
\begin{aligned}
\mathit{INITIATOR}(a,n_a) \equiv\ & \mathit{user.a?b} \rightarrow \mathit{I\_running.a.b} \rightarrow \\
& \mathit{comm!Msg1.a.b.Encrypt.key(b)}.n_a.a \rightarrow \\
& \mathit{comm.Msg2.a.b.Encrypt.key(a)?}n'_a.n_b \rightarrow \\
& \underline{\textbf{if}}\ n_a = n'_a \\
& \quad \underline{\textbf{then}}\ \mathit{comm!Msg3.a.b.Encrypt.key(b)}.n_b \rightarrow \\
& \qquad\qquad \mathit{I\_commit.a.b} \rightarrow \mathit{instance.a.b} \rightarrow \mathit{Skip} \\
& \quad\ \underline{\textbf{else}}\ \mathit{Stop}
\end{aligned}
$$

Fig. 12. FDR example.

channels are used to model possible adversary interference, the external interface, and to keep track of the state of the agents. For example, the event *I_commit.a.b* can be generated by initiator a on channel *I_commit* to represent the fact that it is committing to an instance with responder b.

For the sake of concreteness, a description of the initiator in the Needham-Schroeder protocol is given in Figure 12. The definition is parameterized by a, the name of the initiator, and *n_a*, the nonce used by the initiator in the instance. This definition follows the abstract description of the protocol fairly closely. The initiator waits for a request from the user then begins running the protocol and sends message 1. When it receives message 2, it checks if the nonce in message 2 matches the nonce sent in message 1. If so, it sends message 3, commits to the protocol instance, and begins the instance execution; otherwise it halts. To model the interception of messages by the adversary and the introduction of fake messages by the adversary, a renaming is applied to this process so that actions that occur on the channel *comm* can also occur on the adversary-controlled channels *intercept* or *fake* instead.

Originally, the user also had to provide a description of the adversary. With the development of Casper, the construction of the adversary became automated [Lowe 1997]. The adversary can be thought of as the parallel composition of n processes, one for each of n facts or messages that the adversary may learn during the execution of the protocol. Each process basically has two states, one in which it knows the message and one in which it does not. Each of these processes then can generate a number of events.

—It will synchronize with an agent when overhearing a message that contains the fact.

—It will synchronize with an agent when it sends a message to that agent that contains the fact.

—It will synchronize with other processes representing the adversary when knowledge of those other facts can be used to derive it.

—It will synchronize with another process representing the adversary when it can contribute to the derivation of the fact represented by that other process.

—It can generate a *leak* event to signal that the adversary has acquired knowledge of the fact.

Casper will also construct the specification process for verification. FDR would then check that the protocol in parallel with the adversary is a refinement of the specification process. The specification process for secrecy is simply the CSP process $RUN(\Sigma - L)$ where Σ is the set of all possible events, L is the set of leak events that correspond to the adversary knowing a secret, and $RUN(S)$ is the process that can perform any sequence of events in S. So the specification is the process that can perform any sequence of events that do not include the *leak* events in question. Therefore, authentication specification AS is defined below.

$$AS_0 \equiv R\_running.A.B \rightarrow I\_commit.A.B \rightarrow AS_0$$

$$A \equiv \{R\_running.A.B, I\_commit.A.B\}$$

$$AS \equiv AS_0 \;||\; RUN(\Sigma - A)$$

Intuitively, the specification is the process that can perform all events in any order, except that the occurrence *R_running.A.B* and *I_commit.A.B* events must alternate and must begin with *R_running.A.B*. This means that every time the initiator commits to a protocol run, there must be a separate responder instance that has started responding to the initiator. It is this notion of authentication, which Woo and Lam call *correspondence* [Woo and Lam 1993], that investigators using operational models try to verify.

Advantages over BRUTUS. The advantage of using an existing model checker is that one can obtain results very quickly, and the designer who has experience with that model checker will be already familiar with the input and specification languages.

Advantages of BRUTUS. As seen in the description of BRUTUS, the process model for the honest agents is similar to the CSP model. Each honest agent has a sequence of events that constitute its role in the protocol. In addition, however, we also explicitly maintain the knowledge corresponding to the principals and the adversary. Since this knowledge is represented as a set of "atomic facts" and a set of rewrite rules that can be applied to that set, we can implicitly represent an infinite set of facts. This becomes especially important for the model of the adversary because now we are not forced to artificially limit the set of words that the adversary may learn in order to construct a finite-state model for the adversary. Since in our logic "knowledge" has a first-class status, our logic can express properties about security more naturally. Moreover, we believe that proof of correctness of reduction techniques (such as partial-order and symmetry reductions) is easier and more natural in our framework.

7.3 Murϕ

Mitchell et al. [1997] have investigated using a general-purpose state-enumeration-based model-checking tool, Murϕ, for analyzing cryptographic protocols. In Murϕ, the state of the system is determined by the values of a

```
foreach $i \in 1..num\_initiators$
  foreach $j \in 1..network\_size$
    if ($init[i].state =$ WAITING_FOR_MESSAGE_2 $\wedge$
      $net[j].destination = i$)
    then
          remove $j$ from the network
          if ($net[j].key = i$ $\wedge$
            $net[j].type =$ MESSAGE_2 $\wedge$
            $net[j].nonce1 = i$)
            then
                  set the fields of outgoing message $out$
                  add $out$ to the network
                  $init[i].state :=$ COMMIT
          fi
    fi
```

Fig. 13. Murϕ example.

set of global state variables, including the shared variables that are used to model communication. For example, there is a variable describing which state each of the honest agents is in and a different variable containing the name of its "partner" in the protocol. There is also a set of variables that contain the messages that are sent on the network with one variable describing the type of the message and other variables containing the fields of the message. Transition rules are used to describe how honest agents transition between states and how new messages are inserted into the network.

For example, the structure of the rule describing how the initiator in the Needham-Schroeder protocol responds to message number 2 with message number 3 of the protocol can be found in Figure 13. The rule specifies that if there is some initiator i waiting for message number 2 and if there is some message j on the network whose recipient is i then we remove j from the network, and if j is a message 2 encrypted with i's key and containing i's nonce, then we construct message 3, add it to the network; and then i enters the COMMIT state. Similar rules are written for each of the other messages used in the protocol. The authors mention that rules that capture the behavior of the adversary must also be constructed, and while the rules are not provided, the authors concede that formulating the adversary is complicated [Mitchell et al. 1997]. Presumably, these rules would capture how an adversary can intercept messages, misdirect them, and modify them. However, because the description must necessarily be finite state, it cannot capture the infinite behavior of the adversary. In particular, the description can only keep track of a finite number of words that the adversary may know or learn, and this adversary model would be specific to the particular protocol being analyzed.

The specification for the protocol is given by providing an invariant on the reachable global states of the system. The "usual" correctness property is used, namely that if an initiator i commits to a protocol run with responder r then r must have at least started to respond to i. This is given more formally in Murϕ by an invariant like the one shown in Figure 14.

$$\forall i \in 1..\text{num\_initiators} .$$
$$(\mathit{init[i].state} = \text{ COMMIT } \wedge \mathit{init[i].responder} \in \text{ Responders}) \rightarrow$$
$$(\mathit{resp[init[i].responder].initiator} = i \wedge \mathit{resp[init[i].responder].state} \neq \text{ INITIAL})$$

Fig. 14. Murϕ specification.

There would be an analogous invariant for all responders as well. The authors do not provide a specification for secrecy. Since keeping track of the knowledge of each of the agents is somewhat cumbersome in this approach, we suspect that this would involve a nontrivial extension to the model. Advantages and disadvantages of this approach over BRUTUS are virtually the same as for the FDR case, so we will not repeat it here. However, due to the development of Casper modeling, the adversary is easier in the case of FDR.

7.4 NRL Analyzer

Catherine Meadows has developed the NRL Protocol Analyzer, a special-purpose verification tool for the analysis of cryptographic protocols [Meadows 1994]. Like the approaches based on model-checking, each participant has its own local state, and the global state of the entire system is the composition of these local states with some state information for the environment or adversary. The state of each local participant is maintained by a store of learned facts or *lfacts*. This is represented by a store with four indices which can be thought of as a function of four arguments, $lfact(p, r, n, t) = v$ where

—p is the participant that knows the fact,

—r identifies the run of the protocol (an identifier),

—n describes the nature or name of the fact,

—t is the local time as kept by the participant's counter, and

—v is a list of words or values that make up the content of the fact.

For example,

$$lfact(user(A), N, init\_conv, T) = [user(B)]$$

expresses the fact that A has initiated a conversation with B in run N at local time T. If A has not yet initiated a conversation with B, then the value of the fact would be empty, and so the value of the function would be *nil* or [].

New *lfact* values are computed using the transition rules that describe the behavior of the protocol. For example, consider a fired rule that causes A to perform some action in run C. Also assume that $lfact(A, B, C, X) = Y$. If the rule fires at local time X, it sets A's local counter to $s(X)$. If the rule changes the value of the *lfact* to Z, then $lfact(A, B, C, s(X)) = Z$; otherwise

```
rule(1)
If:
count(user(A,honest)) = [M],
adversaryknows(Z),
lfact(user(A,honest), N, init_nonce, M) = [user(B,W), X].
lfact(user(A,honest), N, init_gotnonce, M) = [],
then:
count(user(A,honest)) = [s(M)],
lfact(user(A,honest), N, init_gotnonce, s(M)) =
  [user(B, W), pke(privkey(user(A,honest))),
  id_check(pke(privkey(user(A,honest)), Z), (X, Y))],
EVENT:
event(user(A,honest), N, init_decrypt, s(M)) =
  [user(B, W), X, Y, Z].

rule(2)
If:
count(user(A, honest)) = [M],
lfact(user(A, honest), N, init_gotnonce, M) =
  [user(B, W), (X, Y), ok],
lfact(user(A, honest), N, init_nonce, M) = [user(B, W), X],
lfact(user(A, honest), N, init_final, M) = [],
then:
count(user(A, honest)) = [s(M)],
adversarylearns(pke(pubkey(user(B, W)), Y)),
lfact(user(A, honest), N, init_final, s(M)) = [Y],
EVENT:
event(user(A, honest), N, init_reply, s(M)) =
[user(B, W), Y].
```

Fig. 15. NRL example.

lfact$(A, B, C, s(X)) = Y$. For the sake of concreteness, an example that parallels the Murϕ example is given in Figure 15. In this example, the first rule checks to see if the initiator has sent message number 1 but still has not received message number 2. If this is the case, it accepts any message Z that the adversary knows and records whether or not it has the correct format for message number 2 in an *lfact* with the name *init_gotnonce*. It checks for the correct format with *id_check* which is simply the identity function. The second rule checks for this *lfact*, and if it exists and the value computed by *id_check* was true, it produces message number 3.

While this description looks somewhat similar to the Murϕ description, this similarity is mostly superficial. Murϕ performs a state space search on explicit state descriptions. The NRL Analyzer uses unification to work on a possibly incomplete state description that would represent a set of states. In addition, the NRL Analyzer works backward. The analysis proceeds by starting with a description of an insecure state. The NRL Analyzer then tries to unify this state description with the right-hand side of a transition rule. If this unification is successful, then applying this substitution to the left-hand side results in a description of the states that could precede the insecure state. We can then continue to search backward from these states in an attempt to reach an initial state. If the initial state is found, then the path to the initial state represents the counterexample trace. Unlike the other finite-state systems, there is also no a priori bound placed on the number of instances of the protocol that can be executed. Therefore, the number of states that could be potentially searched is infinite. The NRL Analyzer provides a way to prove certain sets of states (often these sets are infinite) unreachable in an attempt to prune the search. However, the procedure is still not guaranteed to terminate.

Advantages over BRUTUS. The big advantage of the NRL analyzer over BRUTUS is that the correctness of the protocol is proved without any bounds on the number of instances. In fact, any approach based on theorem proving has this advantage over techniques based on state space exploration. For example, the NRL analyzer has proved the correctness of the Needham-Schroeder protocol without any constraints on the number of instances.

Advantages of BRUTUS. NRL analyzer requires user assistance while it is proving the protocol correct. On the other hand, BRUTUS is completely automatic. Moreover, when the protocol is flawed, BRUTUS outputs a counterexample displaying the flaw. For theorem-proving-based approaches counterexamples are notoriously hard to produce. However, in general it should be noted that any model checker will have these advantages over a theorem prover, such as the NRL analyzer.

7.5 Isabelle

Paulson has investigated the use of Isabelle to prove protocols correct [Paulson 1997b]. Like the models used in Murϕ and the NRL Analyzer, the protocol is encoded with a set of rules that describe how the honest participants in the protocol behave. These rules describe under what circumstances an agent will generate and send a certain message. Murϕ and the NRL Analyzer use these rules to describe the state that results when a particular action is taken or a particular message is sent. Paulson, however, uses these rules to inductively define the set of possible traces. In other words, each of Paulson's rules has the form "**if** the trace *evs* contains certain events, **then** it can be augmented by concatenating the event *ev* to the end of the trace." For example, in the Needham-Schroeder public key protocol, the message the initiator sends in step number 3 is modeled by the rule in Figure 16. If the trace *evs* contains the actions where A sent message 1 containing nonce NA to B and A receives a message 2 containing NA in the first field and NB in the second field, then the trace is augmented with the action where A sends message 3 containing NB.

Since this is a theorem-proving environment, the requirement is given in a syntax identical to that used to model the protocol. Figure 17 gives a possible requirement for the Needham-Schroeder public key protocol. The requirement states that if A sends the nonce NA to B in message 1 and receives a message 2 back that contains NA in the first slot, then B must

```
NS3 [| evs ∈ ns_public;
       Says A B (Crypt (pubK B) | Nonce NA, Agent A |)
            ∈ set_of_list evs;
       Says B' A (Crypt (pubK A) {| Nonce NA, Nonce NB |})
            ∈ set_of_list evs |]
    ⟹
       Says A B (Crypt (pubK B) (Nonce NB))
            # evs ∈ ns_public
```

Fig. 16. Isabelle example.

```
[| Says A B (Crypt (pubK B) {|Nonce NA, Agent A|})
        ∈ set_of_list evs;
   Says B' A (Crypt (pubK A) {|Nonce NA, Nonce NB|})
        ∈ set_of_list evs;
   A ∉ lost; B ∉ lost; evs ∈ ns_public |]
⟹
   Says B A (Crypt (pubK A) {|Nonce NA, Nonce NB|})
        ∈ set_of_list evs
```

Fig. 17. Isabelle requirement example.

have sent this message. The complete report on the analysis of the Needham-Schroeder public key protocol can be found in Paulson [1997a].

Unlike Murϕ, the NRL Analyzer, and BRUTUS, Paulson's technique is based entirely on theorem proving. Because he gives an inductive definition for the set of traces in the protocol, there is no limit placed on the number of instances considered. In other words, Paulson's proof of correctness is for an arbitrary number of instances and not for a specific finite-state model. Like the NRL Analyzer, however, there is no guarantee of termination. In addition, it is not clear how to get feedback about possible errors in the protocol from a failed proof. This suggests that while Paulson's verification technique may be able to prove stronger statements, the model-checking techniques would be more useful to a protocol designer for debugging purposes.

The advantages and disadvantages of the Isabelle approach over BRUTUS are the same as mentioned in the case of the NRL protocol analyzer.

7.6 The Strand Space Model

A new model called the *Strand Space Model* (or SSM) was proposed in Thayer et al. [1998]. Thayer et al. [1998] also show how to use the SSM for protocol representation and manually prove properties related to security about the protocol. A model checker *Athena* which uses SSM as its underlying representation is presented in Song et al. [2001]. Roughly, Athena starts with an initial goal that violates the desired property. After that, the tool works backward using the rules corresponding to the adversary and the protocol and tries to justify the initial goal. As the reader can see, the approach followed in *Athena* is similar to the one employed in the NRL analyzer. The key difference between the two approaches is that *Athena* is fully automatic and uses SSM as its underlying representation.

Advantages over BRUTUS. In some cases Athena is more efficient in terms of space and time than BRUTUS. However, we believe that the underlying algorithm and representation employed in Athena is fundamentally different from the one used in all other approaches described in this article, including BRUTUS. In fact, we are in the process of developing and implementing algorithms and representations similar to the one used in Athena for the logic presented in this article.

Advantages of BRUTUS. We believe that our logic is more expressive and natural for describing properties about security protocols. Moreover, as can be easily inferred from the previous discussion BRUTUS can easily model and reason about protocols where the actions related to security are "mixed" with other actions, such as committing the description of a transaction to a database. On the other hand, the algorithms and representation used in Athena are "fine tuned" for protocols where all the actions are related to security. However, as already pointed out we believe that the underlying algorithms and representations used in Athena can be employed in our framework as well.

7.7 Theory Generation

Kindred and Wing [1999] introduced a technique called *theory generation*. Theory generation takes as its input a set of initial assumptions and rules of inference of a logical system and then constructs a set of facts that can be derived from the initial assumptions and the rules of inference. As an example consider the *BAN* logic presented earlier. Theory generation will take as its input the initial assumptions about all the principals and the rules of inference in the *BAN* logic. After that, theory generation produces all the facts or beliefs about various principals that can be derived from the initial beliefs and the rules of inference of the *BAN* logic. Once the process of "theory generation" has been performed, the set of facts produced can be "queried" to gain insight into the protocol. Kindred and Wing also describe a tool called REVERE based on the techniques presented in Kindred and Wing [1999]. We believe that theory generation has a fundamentally different goal than BRUTUS and can be used by verification tools (like BRUTUS). For example, when we check whether m is in the closure of a set of messages I known by the adversary, we are essentially performing theory generation to compute the closure of I using the messages in I as the set of initial assumptions and the rules of inference corresponding to the adversary. Perhaps theory generation can be used to incorporate other adversary model into BRUTUS. Sanjit Seshia, Nick Hopper, and Jeannette Wing have used BRUTUS and REVERE in a novel way which demonstrates the "synergy" between the two tools [Hopper et al. 2000]. They analyzed an authentication protocol using BRUTUS which produced a counterexample. Then they used REVERE to find the "reason" for the counterexample, i.e., the beliefs that caused the counterexample. Moreover, a flawed assumption discovered by REVERE can be used to point out configurations in BRUTUS that might lead to a counterexample. This leads us to believe that a combination of BRUTUS and REVERE can be used to provide a user-friendly tool, where BRUTUS is used as a verification engine, and REVERE provides a user-friendly explanation for the counterexamples.

7.8 Reductions

A reduction similar to the partial-order reduction presented in this article appears in Shmatikov and Stern [1998]. Shmatikov and Stern [1998] use Murϕ to verify cryptographic protocols. The connection to partial-order reductions was not made in Shmatikov and Stern [1998], and the set of reductions considered in Shmatikov and Stern [1998] are more restrictive than the ones considered here. Moreover, the arguments presented in Shmatikov and Stern [1998] only apply to a restrictive logic. The reduction techniques presented in this article are much more precise and apply to a much richer logic.

Symmetry reductions for model-checking reactive systems are discussed in Clarke et al. [1996], Emerson and Sistla [1996], and Ip and Dill [1996]. Symmetry reductions in the context of checking relational specifications are described in Jackson et al. [1998]. However, we were unable to locate related research on use of symmetry for verification of security or cryptographic protocols.

8. CONCLUSION

In this article we presented the design of a tool BRUTUS for verifying security protocols. We argued how by automating the job of verifying a security protocol BRUTUS eases the burden on a software architect. Reduction techniques that make BRUTUS more efficient were also presented. In the future, we plan to look for techniques that will make BRUTUS even more efficient. We are in the process of investigating the Strand Space Model in this context. We feel that there is still room for improvement in this area. In particular, we want to look at abstraction techniques in the context of verifying security protocols. Another area of future work is to provide semantics to internal actions. Recall that in the logic presented in this article internal actions (e.g., **debit**) do not have any semantics associated with them (i.e., they are treated as purely symbolic). In the future, we want to explore the possibility of having semantics for internal actions, e.g., the **debit** action actually debits a customer's account. Notice that this will entail combining traditional correctness techniques (based on pre/post conditions) with techniques used for verifying security properties. We also want to compile a database of various properties relevant to security protocols in our logic. This database can serve as a set of *patterns* for the system architect and will make the task of writing requirements easier.

ACKNOWLEDGMENTS

We would like to thank Helmut Veith, Sanjit Seshia, and the anonymous referees of TOSEM for their helpful comments.

REFERENCES

BELLARE, M., GARAY, J., HAUSER, R., HERBERG, A., KRAWCZYK, H., STEINER, M., TSUDIK, G., AND WAIDNER, M. 1995. *i*KP—A family of secure electronic payment protocols. In *Proceedings of the 1st USENIX Workshop on Electronic Commerce* (July). USENIX Assoc., Berkeley, CA.

BURROWS, M., ABADI, M., AND NEEDHAM, R. 1989. A logic of authentication. Tech. Rep. 39 (February). DEC Systems Research Center.

CLARKE, E. M., ENDERS, R., FILKORN, T., AND JHA, S. 1996. Exploiting symmetry in temporal logic model checking. *Formal Methods Syst. Des. 9*, 1/2, 77–104.

CLARKE, E. M., JHA, S., AND MARRERO, W. 1998. Using state space exploration and a natural deduction style message derivation engine to verify security protocols. In *Proceedings of the IFIP Working Conference on Programming Concepts and Methods* (PROCOMET).

CLARKE, E. M., JHA, S., AND MARRERO, W. 2000. Partial order reductions for security protocol verification. In *Proceedings on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS).

DOLEV, D. AND YAO, A. 1983. On the security of public key protocols. *IEEE Trans. Inf. Theor. 29*, 2, 198–208.

EMERSON, E. A. AND SISTLA, A. P. 1996. Symmetry and model checking. *Formal Methods Syst. Des. 9*, 1/2, 105–130.

GODEFROID, P., PELED, D., AND STASKAUSKAS, M. 1996. Using partial order methods in the formal validation of industrial concurrent programs. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis* (ISSTA '96, San Diego, CA, Jan. 8–10), S. J. Zeil and W. Tracz, Eds. ACM Press, New York, NY, 261–269.

HOARE, C. A. R. 1985. *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science. Prentice-Hall, Inc., Upper Saddle River, NJ.

HOPPER, N., SESHIA, S., AND WING, J. 2000. Combining theory generation and model checking for security protocol analysis. CMU–CS-00-107.

IP, C. N. AND DILL, D. L. 1996. Better verification through symmetry. *Formal Methods Syst. Des. 9*, 1/2, 41–75.

JACKSON, D., JHA, S., AND DAMON, C. 1998. Isomorph-free model enumeration: A new method for checking relational specifications. *ACM Trans. Program. Lang. Syst. 20*, 6, 727–751.

JENSEN, K. 1996. Condensed state spaces for symmetrical coloured Petri nets. *Formal Methods Syst. Des. 9*, 1/2, 7–40.

KINDRED, D. AND WING, J. 1999. Theory generation for security protocols. Submitted.

LOWE, G. 1996. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS '96, Passau, Germany, Mar.). 146–166.

LOWE, G. 1997. Casper: A compiler for the analysis of security protocols. In *Proceedings of the 1997 IEEE Computer Society Symposium on Research in Security and Privacy* (Oakland, CA, May). IEEE Computer Society Press, Los Alamitos, CA, 18–30.

MEADOWS, C. 1994. The NRL protocol analyzer: An overview. In *Proceedings of the Second International Conference on Practical Applications of Prolog*.

MITCHELL, J., MITCHELL, M., AND STERN, U. 1997. Automated analysis of cryptographic protocols using Mur ϕ. In *Proceedings of the 1997 IEEE Symposium on Research in Security and Privacy*. IEEE Computer Society Press, Los Alamitos, CA, 141–153.

NEEDHAM, R. AND SCHROEDER, M. 1978. Using encryption for authentication in large networks of computers. *Commun. ACM 21*, 12 (Dec.), 993–999.

PAULSON, L. 1997a. Mechanized proofs of security protocols: Needham-schroeder with public keys. Tech. Rep. 413. Cambridge Univ., Cambridge, MA.

PAULSON, L. 1997b. Proving properties of security protocols by induction. In *Proceedings of the 1997 IEEE Computer Society Symposium on Research in Security and Privacy* (Oakland, CA, May). IEEE Computer Society Press, Los Alamitos, CA, 70–83.

PELED, D. 1996. Combining partial order reductions with on-the-fly model-checking. *Formal Methods Syst. Des. 8*, 1, 39–64.

PRAWITZ, D. 1965. *Natural Deduction: A Proof-Theoretical Study*. Almquist and Wiskell International, Stockholm, Sweden.

SCHNEIER, B. 1996. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. 2nd ed. John Wiley and Sons, Inc., New York, NY.

SHMATIKOV, V. AND STERN, U. 1998. Efficient finite-state analysis for large security protocols. In *Proceedings on 11th IEEE Computer Security Foundation Workshop* (CSFW'98, Rockport, MA, June). IEEE Press, Piscataway, NJ.

SONG, D., BEREZIN, S., AND PERRIG, A. 2001. Athena: a novel approach to efficient automatic security protocol analysis. *J. Comput. Secur. 7*. To be published.

THAYER, F. J., HERZOG, J. C., AND GUTTMAN, J. D. 1998. Strand spaces: Why is a security protocol correct? In *Proceedings of the 1998 IEEE Symposium on Security and Privacy* (Oakland, CA, May). IEEE Computer Society Press, Los Alamitos, CA, 160–171.

VALMARI, A. 1991. Stubborn sets of colored petri nets. In *Proceedings of the 12th International Conference on Application and Theory of Petri Nets* (Gjern, Denmark). 102–121.

WOO, T. AND LAM, S. 1993. A semantic model for authentication protocols. In *Proceedings of the 1993 IEEE Symposium on Security and Privacy*. IEEE Press, Piscataway, NJ, 178–194.

Accepted: July 2000

This research was sponsored in part by National Science Foundation (NSF) grant no. CCR-0122581.