Fast, Automatic Checking of Security Protocols

Darrell Kindred Jeannette M. Wing
Computer Science Department
Carnegie Mellon University
$\{$*dkindred,wing*$\}$*@cs.cmu.edu*

Abstract

Protocols in electronic commerce and other security-sensitive applications require careful reasoning to demonstrate their robustness against attacks. Several logics have been developed for doing this reasoning formally, but protocol designers usually do the proofs by hand, a process which is time-consuming and error-prone.

We present a new approach, *theory checking*, to analyzing and verifying properties of security protocols. In this approach we generate the entire finite theory, Th, of a logic for reasoning about a security protocol; determining whether it satisfies a property, ϕ, is thus a simple membership test: $\phi \in Th$. Our approach relies on (1) modeling a finite instance of a protocol in the way that the security community naturally, though informally, presents a security protocol, and (2) placing restrictions on a logic's rules of inference to guarantee that our algorithm terminates, generating a finite theory. A novel benefit to our approach is that because of these restrictions we can provide an automatic *theory-checker generator*. We applied our approach and our theory-checker generator to three different logics for reasoning about authentication and electronic commerce protocols: the Burrows-Abadi-Needham logic of authentication, AUTLOG, and Kailar's accountability logic [4, 8, 6]. For each we verified the desired properties using specialized theory checkers; most checks took less than two minutes, and all less than fifteen.

1 Motivation for our Approach

Past approaches to reasoning about security protocols, e.g., authentication protocols, have relied on either pencil-and-paper proof or machine-assisted proof through the use of interactive theorem provers, e.g., the Boyer-Moore Prover [3], Gypsy [5], HDM [9], Ina Jo [10], and UNISEX [7]. Proofs of properties of these protocols relied on either specialized logics, e.g., Burrows-Abadi-Needham's Logic of Authentication, or an encoding of a specialized logic in the theorem prover's general-purpose logic. These proofs are tedious to do. If done by hand, they are prone to error; and they can be applied only to small examples. If done by machine, they require tremendous user patience since the machine usually insists on treating the critically creative and the boring bookkeeping steps of the proof all with equal importance; they often take hours to complete, even discounting human user time; and they are also prone to error since they still rely on human intervention and ingenuity.

In this paper we present a completely different approach to verifying properties about security protocols. It relies on the observation that informal reasoning about these protocols by the security community is invariably done in terms of a *finite* number of entities, e.g., parties communicating, messages exchanged, types of messages, encryption and decryption keys. Thus, we reason about a finite model of a protocol rather than its generalization. Our approach further relies on the observation that logics used to reason about these protocols have a finite number of rules of inference that "grow in a controlled manner" (discussed in Section 2.1). It is this controlled growth that enables us to build an automatic checker for a given logic and class of protocols we wish to verify. It is the small size in the number of entities we need to model and the small size in the number of rules in any given logic that enables us to do this checking fast.

Stated simply, our method is to build the entire theory, *Th*, given a logic and a model of the protocol we want to check. Since the model is finite and since the logic's rules always shrink or grow in a controlled manner, the theory we generate is finite. Then checking whether a property, ϕ, holds of a protocol boils down to a simple membership test: $\phi \in Th$? In all the examples we have looked at, generating Th takes no more than a few minutes; membership is a trivial test.

In fact, we do even better. We provide more than a

This research is sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330. Views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing official policies or endorsements, either expressed or implied, of Wright Laboratory or the United States Government.

single fast, automatic checker for verifying properites of a class of security protocols, but better yet, we provide a tool that for any given logic *generates* one of these fast, automatic checkers. In other words our tool is a *theory-checker generator* (and each checker it generates implements the method described above). And best of all, the tool we built generates these checkers completely automatically as well!

In this paper, we first describe our method and argue informally its correctness and termination properties. We illustrate in Section 3 how we applied this method to three different logics, each proposed as the basis for reasoning about security protocols:

Logic	**Class of Protocols**	**Properties to Check**
BAN	authentication	authentication
AUTLOG	authentication	authentication, privacy
Kailar	elec. commerce	accountability

We start with the BAN logic since AUTLOG is a variation of it and Kailar's logic is similar in flavor to BAN. We discuss some of the implementation details in Section 4 and explain how we are able to build a tool that can generate checkers automatically. Section 5 discusses related work. We close in Section 6 by summarizing our contributions and discussing two important future directions of this work: why logics for reasoning about security protocols are particularly amenable to our method and the general applicability of our method to domains outside of security.

2 Our Method

The technique for checking whether some desired property, ϕ, holds for a given protocol, P, consists of these basic steps:

- Given a logic, L (consisting of a finite set of axioms and controlled-growth rules), our tool automatically generates a checker, C, specialized to that logic.
- Given the protocol, P, for which we want to check property ϕ, encode the initial assumptions and messages of P as formulas of L; call this set T^0.
- Using C, exhaustively enumerate the theory, T^*, that is, the set of facts (formulas) derivable from the formulas in T^0.
- Determine whether ϕ is in T^* by a simple membership test.

In the following sections, we explain the class of logics to which this method can be applied, give a more detailed description of the algorithm, and present informal arguments for its correctness and termination.

2.1 Class of Logics

The method we use requires several restrictions on the logic, L. First, formulas of the logic are the smallest set such that

- V is a formula for any variable V, and
- if $F_1, \ldots, F_n$ are formulas and S is a function symbol, then $S(F_1, \ldots, F_n)$ is a formula (for any $n \geq 0$).

A constant can be represented as a function symbol with no arguments (we will omit the parentheses in this case). In particular, no conjunction, disjunction, negation, or quantifiers are permitted. Here are some legal formulas:

$$believes(A, shared\_key(K_{ab}, A, B))$$
$$sees(B, encrypt(K_{BS}, shared\_key(K_{ab}, A, B)))$$
$$controls(S, shared\_key(K_{ab}, A, B))$$

We must be able to separate the rules of inference of the logic L into three classes:

- *S-rules*: An S-rule (shrinking rule) consists of a set of premises, $\{P_1, \ldots, P_n\}$, and a conclusion C, such that if $P_1, \ldots, P_n$ unify simultaneously with a set of valid[1] formulas, then C (appropriately instantiated) is a valid formula. The conclusion must be the same size as or smaller than one of the premises, by some well-founded measure. Each variable occurring in the conclusion must also occur in one or more premises.
- *G-rules*: A G-rule (growing rule) has the same form as an S-rule, but the conclusion must be strictly larger than each of the premises, by the same measure, and each variable occurring in the premises must also occur in the conclusion.
- *Rewrites*: A rewrite is a pair of formulas, (f_1, f_2), such that any occurrence of f_1 in a valid formula may be replaced by f_2, yielding another valid formula. The formula f_2 must be the same size as f_1 and contain the same variables.

The classification of a set of rules as S-rules and G-rules can be done automatically if an appropriate measure

[1] We mean *valid* in the technical sense; that is, derivable from the assumptions.

is supplied. The measure must be compositional; that is, replacing a subformula by a formula of the same measure must not change the measure of the whole formula. Typically, a simple measure such as the number of function symbols will suffice.

Finally, we require that each S-rule must still meet the S-rule criteria when all its premises that could match G-rule conclusions are removed. We will refer to this as the *S/G restriction* since it constrains the manner in which S- and G-rules can interact with each other. Whereas the other restrictions are local properties of individual rules and thus can be checked for each rule in isolation, this global restriction involves the whole set of rules.

2.2 The Algorithm

The encoding of a protocol in the target logic will normally take the form of a set of formulas that represent initial assumptions held by the involved parties and the effects of the messages sent during a run of the protocol. The core of the verification method is an algorithm for computing the transitive closure, under the rules of inference, of this initial set of valid formulas.

First, we consider the case in which there are S-rules, but no G-rules or rewrites. The basic strategy we use is a breadth-first traversal in which each node of the traversed dag is a valid formula. The roots of the dag, i.e., the initial fringe, consist of all the formulas representing assumptions and messages. At each step in the traversal, we consider all possible applications of the S-rules that *use* a formula from the fringe. By *use* we mean that at least one of the premises unifies with such a formula. The new fringe consists of all the new conclusions reached by those S-rule applications. By the argument given in Section 2.3, this process will eventually halt. The resulting set of formulas is the complete set of valid formulas.

Next, we introduce the G-rules. The G-rules must be applied more judiciously than the S-rules, since applying them eagerly can produce infinitely many valid formulas. We apply them only as necessary to enable further application of S-rules. For each S-rule, we separate its premises into those that unify with some G-rule conclusion and those that do not. When applying that S-rule, we first unify the latter group of premises with known valid formulas. We can then proceed to search for a match for the remaining premises by applying the G-rules "in reverse" to the premises until we either reach a set of formulas all of which are known valid, or we cannot apply the G-rules further. (This process is guaranteed to terminate: see Section 2.3.)

Finally, we introduce the rewrites. Since they neither produce larger formulas nor introduce new variables, we could apply them eagerly like the S-rules. This can lead to exponential blowup in the set of valid formulas, though, so it is better to use them only as needed. Since rewrites can be applied to any subformula, we augment a simple unification algorithm by trying rewrites at each recursive step of the unification [15]. Plotkin described a similar technique for building equational axioms into the unification process [16].

Since the G-rules and rewrites are applied lazily, the full algorithm does *not* generate the complete set of valid formulas. Any formula whose formal proof has a rewrite or G-rule as its last step may not appear. The generated set of formulas will have the property that any valid formula will be reachable from this set by a proof involving only G-rules and rewrites. In practice, we have found that the G-rules are often used solely for establishing side-conditions, and that all the "interesting" formulas will in fact appear in the generated set (possibly after applying rewrites). Nonetheless, if we are interested in a specific formula we can easily test whether it holds by performing a simple preprocessing step prior to testing for membership in the generated set.

2.3 Correctness and Termination

The soundness of the algorithm is easy to establish. For any formula that is added to the fringe, we can easily construct a proof of it by tracing back up the dag to formulas in the initial valid set.

To show completeness (i.e., that any valid formula is reachable from the generated set using only G-rules and rewrites), we use induction on the number of S-rule applications in the proof of a formula, F. If there exists a proof using no S-rule applications, then the initial valid set of formulas is sufficient, since we only need to capture a set of formulas from which F can be reached by G-rules and rewrites. If there exists a proof of F using n S-rule applications, we know by the induction hypothesis that all lines in the proof before the nth S-rule application are either in the valid set or can be reached from an element of the valid set by applying G-rules and rewrites. Since we apply as many G-rules and rewrites as necessary to enable S-rule applications, the result of the nth S-rule application will be in the valid set, and thus F is reachable from the valid set using only G-rules and rewrites.

The algorithm is guaranteed to terminate because of the restrictions we impose on the rules. Consider a modified logic in which any S-rule premise that could match a G-rule conclusion is removed, and the G-rules are eliminated (since they cannot now make any contribution). The modified S-rules still meet the S-rule criteria by the S/G restriction.

If we run the algorithm on this modified logic with some finite set of initial assumptiones, each new formula is added as the result of an S-rule and thus is smaller or the same size as one of its premises. The rewrites can only be applied finitely many times since they preserve the formula's size and set of variables. Thus we will never produce a formula that is larger than the largest initial assumption, and we will introduce no new variables, so the set of formulas is finite and the algorithm must terminate.

If we then reintroduce the G-rules and the missing premises and run the algorithm, every S-rule application will correspond to exactly one S-rule application from the first run. Furthermore, each reverse-application of the G-rules will terminate since G-rules are strictly growing, so the algorithm terminates.

3 Examples

We used our theory-checker generator to build three theory checkers, encoding three different logics: the well-known BAN logic of authentication [4]; AUTLOG [8], an extension of the BAN logic; and Kailar's logic for reasoning about accountability in electronic commerce protocols [6]. We describe each of these encodings below.

3.1 BAN

The BAN logic is a natural case to consider; it motivated the original development of our method and tool. This logic is normally applied to authentication protocols. It allows certain notions of belief and trust to be expressed in a simple manner, and it provides rules for interpreting encrypted messages exchanged among the parties (principals) involved in a protocol.

3.1.1 The Logic

In encoding the BAN logic and its accompanying sample protocols, we had to make several adjustments and additions to the logic as originally presented [4], to account for rules and assumptions that were missing or implicit.

Figure 1 shows the functions used in the encoding. The first eleven correspond directly to constructs in the original logic, and have clear interpretations. The last two were added: *inv* makes explicit the relationship implied between K and K^{-1}, and *distinct* expresses that two principals are different.

The BAN logic consists of eleven basic rules of inference:

- three message-meaning rules that allow one principal to deduce that a given message was once uttered by some other principal;
- one nonce-verification rule whereby a principal can determine that some message was sent recently, by examining nonces;
- one jurisdiction rule, expressing one principal's trust of another;
- five rules for extracting components of messages, some requiring knowledge of the appropriate keys; and
- one freshness rule, which states that a conjunction is fresh if any part of it is fresh.

We encode each of these rules directly as a single S-rule, with the exception of the freshness rule, which is a G-rule since its conclusion is larger than its premise:

$$\frac{believes(P, fresh(X))}{believes(P, fresh(comma(X, Y)))}$$

The message-meaning and extraction rules involving encryption carry a side-condition that the principal who encrypted the message is different from the one interpreting it. We encode this explicitly using a three-place *encrypt* function and the extra *distinct* function, by adding an extra premise to each of these rules.

We add a few rules to this original set to make it more complete. There are two additional component-extraction S-rules:

$$\frac{believes(P, said(Q, comma(X, Y)))}{believes(P, said(Q, X))}$$

$$\frac{believes(P, believes(Q, comma(X, Y)))}{believes(P, believes(Q, X))}$$

The BAN Kerberos protocol analysis indirectly refers to (and depends on) these rules; AUTLOG includes them explicitly.

We also add two S-rules that do the work of message-meaning and nonce-verification simultaneously. The shared-key version of this rule is

$$\frac{\begin{array}{c} believes(P, fresh(K)) \\ believes(P, shared\_key(K, Q, P)) \\ sees(P, encrypt(X, K, R)) \\ distinct(P, R) \end{array}}{believes(P, believes(Q, X))}.$$

One of these rules is implicitly required by the BAN Andrew secure RPC analysis. AUTLOG handles this case somewhat more elegantly by introducing a "recently said" notion, adding extra conclusions to its message-meaning ("authentication") rules, and creating new "contents" rules.

Function	BAN interpretation
believes(P, X)	P **believes** X
sees(P, X)	P **sees** X
said(P, X)	P **said** X
controls(P, X)	P **controls** X
fresh(X)	**fresh**(X)
shared_key(K, P, Q)	$P \overset{K}{\leftrightarrow} Q$
public_key(K, P)	$\overset{K}{\mapsto} P$
secret(Y, P, Q)	$P \overset{K}{\rightleftharpoons} Q$
encrypt(X, K, P)	$\{X\}_K$ *from* P
combine(X, Y)	$\langle X \rangle_Y$
comma(X, Y)	X, Y (conjunction)
inv(K_1, K_2)	K_1 and K_2 are a public/private key pair
distinct(P, Q)	principals P and Q are not the same

Figure 1: BAN functions

We add seven freshness G-rules: four to reflect the fact that an encrypted (or combined) message is fresh if either the body or the key is, and three that extend the freshness of a key to freshness of statements about that key. The example protocol verifications in the BAN paper require some of these extra freshness rules, and we include the rest for completeness. AUTLOG includes the first four of them.

Finally, since we represent message composition explicitly (via *comma*), we include three rewrites that express the commutativity and associativity of the *comma* function; three more rewrites provide commutativity for *shared_key*, *secret*, and *distinct*. The *shared_key* rewrite looks like this:

$$\frac{shared\_key(K, P, Q)}{shared\_key(K, Q, P)}$$

Appendix A contains the complete set of rules and rewrites.

The logic restrictions from Section 2 do not permit the use of universal quantifiers, as BAN specifications sometimes do in delegation statements [4]:

$$A \textbf{ believes } \forall K.(S \textbf{ controls } (A \overset{K}{\longleftrightarrow} B))$$

However, since none of the BAN rules introduce new keys, we can get the effect of this universal quantification in assumptions by instantiating the statement with each of the keys mentioned in the other assumptions. We could do this automatically as a preprocessing step. It may be possible to extend our method slightly to allow universal quantification at the outermost level.

After encoding the rules, we entered each of the four protocols examined in the BAN paper and checked all the properties claimed there [4]. (We added most of the extensions above after some verification attempt failed.)

3.1.2 Kerberos

Through a sequence of four messages, the Kerberos protocol establishes a shared key for communication between two principals, using a trusted server [12]. The BAN analysis of this protocol starts by constructing a three-message idealized protocol; the idealized protocol ignores message 1 since it is unencrypted. The BAN analysis then goes on to list ten initial assumptions regarding client/server shared keys, trust of the server, and freshness of the timestamps used [4]. We express each of these three messages and ten assumptions directly (the conversion is purely syntactic), and add four more assumptions (see Figures 2 and 3).

The first extra assumption—that A must believe its own timestamp to be fresh—is missing in [4], and the last three are required to satisfy the distinctness side-conditions. After making these adjustments, we can run the 14 initial assumptions and 3 messages through our automatically generated BAN-checker, and it will generate an additional 50 true formulas in 90 seconds[2].

[2]All timings were done on an IBM RS/6000 model 25T with an 80MHz PowerPC 601 CPU.

Message 2. $S \rightarrow A : \{T_s, A \stackrel{K_{ab}}{\longleftrightarrow} B, \{T_s, A \stackrel{K_{ab}}{\longleftrightarrow} B\}_{K_{bs}}\}_{K_{as}}$

$$\begin{aligned}&sees(A, encrypt(comma(comma(T_s, shared\_key(K_{ab}, A, B)),\\ &\qquad encrypt(comma(T_s, shared\_key(K_{ab}, A, B)),\\ &\qquad\quad K_{bs}, S))),\\ &\quad K_{as}, S))\end{aligned}$$

Message 3. $A \rightarrow B : \{T_s, A \stackrel{K_{ab}}{\longleftrightarrow} B\}_{K_{bs}}, \{T_a, A \stackrel{K_{ab}}{\longleftrightarrow} B\}_{K_{ab}}$ *from A*

$$\begin{aligned}&sees(B, comma(encrypt(comma(T_s, shared\_key(K_{ab}, A, B)), K_{bs}, S),\\ &\qquad encrypt(comma(T_a, shared\_key(K_{ab}, A, B)), K_{ab}, A)))\end{aligned}$$

Message 4. $B \rightarrow A : \{T_a, A \stackrel{K_{ab}}{\longleftrightarrow} B\}_{K_{ab}}$ *from B*

$$sees(A, encrypt(comma(T_a, shared\_key(K_{ab}, A, B)), K_{ab}, B))$$

Figure 2: Kerberos protocol messages, in BAN idealized form and converted to our syntax.

$$\begin{aligned}&believes(A, shared\_key(K_{as}, S, A))\\ &believes(B, shared\_key(K_{bs}, S, B))\\ &believes(S, shared\_key(K_{as}, A, S))\\ &believes(S, shared\_key(K_{bs}, B, S))\\ &believes(S, shared\_key(K_{ab}, A, B))\\ &believes(A, controls(S, shared\_key(K_{ab}, A, B)))\\ &believes(B, controls(S, shared\_key(K_{ab}, A, B)))\\ &believes(A, fresh(T_s))\\ &believes(B, fresh(T_s))\\ &believes(B, fresh(T_a))\end{aligned}$$

$$\begin{aligned}&believes(A, fresh(T_a))\\ &distinct(A, S)\\ &distinct(A, B)\\ &distinct(B, S)\end{aligned}$$

Figure 3: Encoding of the Kerberos initial assumptions. All but the last four assumptions appear in the BAN analysis [4].

By running a simple membership test, we verified that these four desired results are among them:

$$\begin{gathered} believes(A, shared\_key(K_{ab}, A, B)) \\ believes(B, shared\_key(K_{ab}, A, B)) \\ believes(B, believes(A, shared\_key(K_{ab}, A, B))) \\ believes(A, believes(B, shared\_key(K_{ab}, A, B))) \end{gathered}$$

These results agree with the original BAN analysis. They indicate that each of the two parties believes it shares a key with the other, and that each believes that the other believes the same thing. Appendix B illustrates one step in the algorithm: applying an S-rule with the help of a G-rule and rewrites.

If we remove the optional final message from the protocol and run the checker again, it will generate 41 valid formulas. By computing the difference between this set and the first set of 50, we can determine exactly what the final message contributes. Among the 9 formulas in this difference is

$$believes(A, believes(B, shared\_key(K_{ab}, A, B))),$$

(the last of the four results above). This confirms the claim in the original analysis that "the three-message protocol does not convince A of B's existence" [4]. This technique of examining the set difference between the deduced properties of two versions of a protocol is a simple but powerful benefit of our approach; it helps in understanding differences between protocol variants and it supports "rapid" protocol design.

3.1.3 Andrew RPC, Needham-Schroeder, and CCITT X.509

We encoded the assumptions and messages of the three variants of the Andrew secure RPC protocol given in the BAN paper, and got the expected results. The last of these verifications required an extra freshness assumption not mentioned in the BAN analysis, as well as some of our added freshness rules and the simultaneous message-meaning/nonce-verification rule.

We duplicated the BAN results for two variants of the Needham-Schroeder public-key secret-exchange protocol. Finally, we ran the checker on two variants of the CCITT X.509 protocol explored in the BAN paper. One of these checks failed to produce the expected results, and this led us to discover an oversight in the BAN analysis: they observe a weakness in the original X.509 protocol and claim, "The simplest fix is to sign the secret data Y_a and Y_b before it is encrypted for privacy." In fact we must sign the secret data together with a nonce to ensure freshness. After we corrected this, the verifications proceeded as expected.

3.2 AUTLOG

AUTLOG is an extension of the BAN logic, proposed by Kessler and Wedel [8]. It introduces several concepts, including a simulated eavesdropper for detecting information leaks, and the idea of principals "recognizing" decrypted messages.

Our encoding of AUTLOG uses all the BAN functions, and a few extras: *recognizable*, *mac* (for "message authentication codes"), *hash*, and *recently_said*. The original rules of inference from AUTLOG can be entered almost verbatim. There are 23 S-rules and 19 G-rules; the rules governing freshness and recognition are the only G-rules.

To check a protocol for leaks using AUTLOG, one finds the closure over the "seeing" rules of the transmitted messages. The resulting list will include everything an eavesdropper could see. Our tool is well-suited to computing this list; the seeing rules are all S-rules, so the checker will generate exactly the desired list.

Kessler and Wedel present two simple challenge-response protocols: one in which only the challenge is encrypted and another in which only the response is encrypted. We encoded both of these protocols and verified the properties Kessler and Wedel claim: that both achieve the authentication goal

$$believes(B, recently\_said(A, R_B))$$

where R_B is the secret A provides to prove its identity. Furthermore, through the eavesdropper analysis mentioned above, we can show that in the encrypted-challenge version, the secret is revealed and thus the protocol is insecure. (The BAN logic cannot express this.)

We also checked that the Kerberos protocol, expressed in AUTLOG, satisfied properties similar to those described in Section 3.1. Since AUTLOG has a large set of rules, this verification took roughly 13 minutes; this was significantly longer than any other verification. In Section 4 we mention some optimizations we expect to reduce this time.

3.3 Kailar's Accountability Logic

More recently, Kailar has proposed a simple logic for reasoning about accountability in electronic commerce protocols [6]. The central construct in this logic is

$$P \ \mathbf{CanProve} \ X$$

which means that principal P can convince anyone in an intended audience sharing a set of assumptions that X holds, without revealing any "secrets" other than X.

We encoded the version of this logic using "strong proof" and "global trust", but the weak-proof and nonglobal-trust versions would be equally simple. We used these functions: *CanProve*, *IsTrustedOn*, *Implies*, *Authenticates*, *Says*, *Receives*, *SignedWith*, *comma*, and *inv*.

We encoded the four main rules of the logic as follows:

$$\mathbf{Conj}: \frac{CanProve(P, X),\ CanProve(P, Y)}{CanProve(P, comma(X, Y))}$$

$$\mathbf{Inf}: \frac{CanProve(P, X),\ Implies(X, Y)}{CanProve(P, Y)}$$

$$\mathbf{Sign}: \frac{\begin{array}{c} Receives(P, SignedWith(M, K^{-1})) \\ CanProve(P, Authenticates(K, Q)) \\ Inv(K, K^{-1}) \end{array}}{CanProve(P, Says(Q, M))}$$

$$\mathbf{Trust}: \frac{\begin{array}{c} CanProve(P, Says(Q, X)) \\ CanProve(P, IsTrustedOn(Q, X)) \end{array}}{CanProve(P, X)}$$

The **Conj** and **Inf** rules allow building conjunctions and using initially-assumed implications. The **Sign** and **Trust** rules correspond roughly to the BAN logic's public-key message-meaning and jurisdiction rules. We replace the construct $X\ in\ M$ (representing interpretation of part of a message) with three explicit rules for extracting components of a message. We add rewrites expressing the commutativity and associativity of *comma*, as in the other logics. This encoding does not require any G-rules.

We verified the variants of the IBS (NetBill) electronic payment protocol that Kailar analyzes [6]. Figure 4 contains an encoding of part of the "service provision" phase of the asymmetric-key version of this protocol. If we run the Kailar-logic checker on these messages and assumptions, it will apply the **Sign** rule to produce these two formulas:

$$\begin{array}{l} CanProve(S, Says(E, comma(SignedWith(Price, \\ \qquad\qquad\qquad\qquad\qquad\qquad K_s^{-1}), \\ \qquad\qquad\qquad\qquad Price))) \\ CanProve(S, Says(E, ServiceAck)). \end{array}$$

It will then apply the component-extracting rule to produce

$$\begin{array}{l} CanProve(S, Says(E, SignedWith(Price, K_s^{-1}))) \\ CanProve(S, Says(E, Price)). \end{array}$$

Finally, it will apply **Inf** to derive these results, which Kailar gives [6].

$$\begin{array}{l} CanProve(S, Says(E, ReceivedOneServiceItem(E))) \\ CanProve(S, Says(E, AgreesToPrice(E, pr))) \end{array}$$

It will stop at this point, since no further rule applications can produce new formulas.

We verified the rest of Kailar's results for two variants of the IBS protocol and for the SPX Authentication Exchange protocol. Each check with this logic took less than ten seconds.

4 Implementation

Implemented in the Standard ML programming language, our tool makes heavy use of SML's *modules* support. SML modules can be *signatures* (interface descriptions), *structures* (implementations), or *functors* (generic or parameterized implementations). With our tool, each logic is represented by a structure containing its function names (e.g., **believes** and **sees**), S-rules, G-rules, rewrites, and a measure function on its formulas. The checker-generator is a functor that takes as an argument a logic module, and generates a new module which is a checker specialized to that logic. For instance, to create the three checkers we used, we invoke the functor, `TheoryChecker`, with three different `Logic` structures:

```
structure BanChecker =
  TheoryChecker(structure Logic = BAN
               ...)
structure AutlogChecker =
  TheoryChecker(structure Logic = AUTLOG
               ...)
structure KailarChecker =
  TheoryChecker(structure Logic = Kailar
               ...)
```

Then, for a particular protocol such as Kerberos, we first compute the closure of the list of initial assumptions appended (`@`) with the list of messages, and then we run a simple check for any desired property:

```
val formulas = BanChecker.closure
                 (Kerberos.assumptions
                  @ Kerberos.messages);
check(formulas, Kerberos.A_knows_Kab);
```

The current implementation is a rough first cut. We plan to make it significantly more efficient by representing sets of formulas with more sophisticated data structures that support fast lookups, union, and fast unification against the whole set of formulas; we also know of some possible optimizations to the search procedure which should help. Nonetheless, all but three of our verification attempts ran in less than two minutes each, and the remaining three took less than fifteen minutes each.

One way in which the current implementation could be improved is to allow type declarations for the functions in each logic. For instance, the BAN logic might

IBS protocol messages:

Message 3. $E \rightarrow S : \{\{Price\}_{K_s^{-1}}, Price\}_{K_e^{-1}}$

$$Receives(S, SignedWith(comma(SignedWith(Price, K_s^{-1}), Price), K_e^{-1}))$$

Message 5. $S \rightarrow E : \{Service\}_{K_s^{-1}}$

$$Receives(E, SignedWith(Service, K_s^{-1}))$$

Message 6. $E \rightarrow S : \{ServiceAck\}_{K_e^{-1}}$

$$Receives(S, SignedWith(ServiceAck, K_e^{-1}))$$

Initial assumptions:

$$CanProve(S, Authenticates(K_e, E))$$
$$Implies(Says(E, Price), AgreesToPrice(E, pr))$$
$$Implies(Says(E, ServiceAck), ReceivedOneServiceItem(E))$$

Figure 4: Excerpt from IBS protocol and initial assumptions.

have types representing principals, keys, and formulas, and the *encrypt* function would expect a formula, a key, and a principal as its arguments. This would help eliminate errors in encoding both protocols and rules, and the search mechanism can then take advantage of the type information as well.

5 Related Work

Model checking is a technique whereby a finite state machine is checked for a property through exhaustive case analysis. It has been used successfully in hardware verification and protocol analysis. Most recently, the FDR model checker [17] has been used by Lowe [11] to debug and fix the Needham-Schroeder public key protocol and by Roscoe [18] to check noninterference of a simple security hierarchy (high/low). Like model checking, our method relies on the finiteness of the entity being verified; unlike model checking, however, we generate a finite theory, not a finite model.

Theorem proving, usually machine-assisted, is the more traditional approach to verifying security properties, including a long history of work based on the Bell-LaPadula model [1]. As in theorem proving, we manipulate the syntactic representation, i.e., the logic, of the entity we are verifying; by restricting the nature of the logic, however, unlike machine-assisted theorem proving, we enumerate the entire theory rather than (with human assistance) develop lemmas and theorems as needed. We also express the messages exchanged in a protocol as a set of initial non-logical axioms (and thus express them in the same language as the logic), avoiding the need for the user to learn more than one input language. Moreover, our method is completely automatic and fast.

Before building our theory-checker generator, we implemented the BAN logic within the PVS verification system [14], and reproduced the Kerberos protocol proofs with it. The encoding of BAN in PVS was quite natural, but the proofs were tedious, primarily because for each rule application, we had to enter the variable instantiations manually since the prover could not guess the right ones. This would be a smaller problem in a verification system with support for unification.

Our approach is the closest in nature to logic programming, embodied in programming languages like Prolog. Indeed, AUTLOG has been implemented in Prolog. Our approach, however, is more powerful because we can produce results without any dependence on rule ordering, and because we use a modified unification algorithm that can handle simple rewrite rules. At the same time, we are more specific; by tailoring our reasoning to security protocols, we exploit the nature of the domain in ways that make exhaustive enumeration a computationally feasible and tractable approach to automatic verification.

The idea of computing the "closure" (or "completion") of a theory is used in the theorem proving system SATURATE [13]. Our restrictions on the input logic allow us to generate a saturated set of formulas that we find easier to interpret than the sets generated by these more general systems.

Finally, our approach makes it easy to generate specialized checkers automatically. Just as Jon Bentley has argued the need for "little languages" [2], our tool provides a way to construct "little tools" for "little logics."

6 Summary and Future Directions

Our approach was motivated by the need to debug protocols in the security domain. When someone presents a security protocol, there is always an uneasiness on our part. These are typical questions that we pose ourselves when simply trying to understand a security protocol:

- Why is that message needed?
- Is the ordering of these messages required or incidental?
- Why is that message or part of the message encrypted? Is it necessary?

To argue the correctness, or better yet to find performance optimization, for example by deleting a message, unordering messages (and thus allowing concurrency), or avoiding encryption, necessitates at least a systematic way of reasoning about these protocols. Using specialized logics helps; using tools that automate these logics helps even more.

The restrictions we impose on the input logic offer the substantial advantage that we can guarantee termination, and furthermore a logic can be automatically checked for compliance with these restrictions (given a measure).

A further advantage of our approach, as illustrated by the "diff" example of Section 3.1.2, is that we can study closely-related theories by examining the formulas that appear in the generated set of one but not the other.

With fast, automatic verification, through the use of our checkers for "little logics," protocol designers can invent and debug their protocols quickly, with the same ease as compiling a program. Thus as a compiler gives programmers assurance that their programs are type correct, our checkers can give protocol designers additional assurance that their protocols are "correct" (i.e., satisfy certain desired properties).

With the explosion of the Internet and the wide range of electronic commerce protocols proposed and in commercial use, the problem of verifying and debugging these protocols is going to get worse. Even well-understood authentication protocols are subject to attacks by the environment that do not satisfy their original assumptions.

Our method has two promising future directions. First, we readily acknowledge that we greatly exploit the nature of the domain: security protocols are most often explained informally in terms of a small number of parties (Alice, Bob, a server, and perhaps an eavesdropper), a small number of message exchanges (usually not more than ten), a small number of keys (public and private keys for each party involved), a small number of nested encryptions (usually under two), and so on. We also exploit the smallness of the logics involved: the BAN logic (as we encode it) has only twenty rules of inference; moreover, it has only eight G-rules. It is possible (though we have only an intuition at this point) that there is something inherent to security protocols and logics for reasoning about them that makes our theory-generation technique especially appropriate.

Second, of course, our technique is not only applicable to protocols in security. While we do impose significant restrictions on the logics, these restrictions are expressed in general terms and may well be satisfied by useful logics from other domains.

A BAN logic encoding

This appendix contains the complete encoding of the BAN logic, from which the BAN checker is automatically generated. See Section 3.1 for further explanation.

The three message-meaning S-rules:

$$\frac{\begin{array}{c} believes(P, shared\_key(K, Q, P)) \\ sees(P, encrypt(X, K, R)) \\ distinct(P, R) \end{array}}{believes(P, said(Q, X))}$$

$$\frac{\begin{array}{c} believes(P, public\_key(K_1, Q)) \\ sees(P, encrypt(X, K_2, R)) \\ inv(K_1, K_2) \\ distinct(P, R) \end{array}}{believes(P, said(Q, X))}$$

$$\frac{believes(P, secret(Y, Q, P)), \quad sees(P, combine(X, Y))}{believes(P, said(Q, X))}$$

The nonce-verification S-rule:

$$\frac{\begin{array}{c} believes(P, said(Q, X)) \\ believes(P, fresh(X)) \end{array}}{believes(P, believes(Q, X))}$$

The jurisdiction S-rule:

$$\frac{\begin{array}{c} believes(P, controls(Q, X)) \\ believes(P, believes(Q, X)) \end{array}}{believes(P, X)}$$

The seven S-rules for extracting components of messages:

$$\frac{\begin{array}{c} believes(P, shared\_key(K, Q, P)) \\ sees(P, encrypt(X, K, R)) \\ distinct(P, R) \end{array}}{sees(P, X)}$$

$$\frac{\begin{array}{c} believes(P, public\_key(K, P)) \\ sees(P, encrypt(X, K, R)) \end{array}}{sees(P, X)}$$

$$\frac{\begin{array}{c} believes(P, public\_key(K_1, Q)) \\ sees(P, encrypt(X, K_2, R)) \\ inv(K_1, K_2) \\ distinct(P, R) \end{array}}{sees(P, X)}$$

$$\frac{sees(P, combine(X, Y))}{sees(P, X)}$$

$$\frac{sees(P, comma(X, Y))}{sees(P, X)}$$

$$\frac{believes(P, said(Q, comma(X, Y)))}{believes(P, said(Q, X))}$$

$$\frac{believes(P, believes(Q, comma(X, Y)))}{believes(P, believes(Q, X))}$$

The two combined message-meaning and nonce-verification S-rules:

$$\frac{\begin{array}{c} believes(P, fresh(K)) \\ sees(P, encrypt(X, K, R)) \\ distinct(P, R) \\ believes(P, shared\_key(K, Q, P)) \end{array}}{believes(P, believes(Q, X))}$$

$$\frac{\begin{array}{c} believes(P, fresh(Y)) \\ sees(P, combine(X, Y)) \\ believes(P, secret(Y, Q, P)) \end{array}}{believes(P, believes(Q, X))}$$

The eight G-rules dealing with freshness:

$$\frac{believes(P, fresh(X))}{believes(P, fresh(comma(X, Y)))}$$

$$\frac{believes(P, fresh(K))}{believes(P, fresh(shared\_key(K, Q, R)))}$$

$$\frac{believes(P, fresh(K))}{believes(P, fresh(public\_key(K, Q)))}$$

$$\frac{believes(P, fresh(Y))}{believes(P, fresh(secret(Y, Q, R)))}$$

$$\frac{believes(P, fresh(Y))}{believes(P, fresh(combine(X, Y)))}$$

$$\frac{believes(P, fresh(K))}{believes(P, fresh(encrypt(X, K, R)))}$$

$$\frac{believes(P, fresh(X))}{believes(P, fresh(encrypt(X, K, R)))}$$

$$\frac{believes(P, fresh(X))}{believes(P, fresh(combine(X, Y)))}$$

Finally, the various commutativity and associativity rewrites:

$$\frac{comma(X, Y)}{comma(Y, X)}$$

$$\frac{comma(comma(X, Y), Z)}{comma(X, comma(Y, Z))}$$

$$\frac{comma(X, comma(Y, Z))}{comma(comma(X, Y), Z)}$$

$$\frac{believes(P, shared\_key(K, Q, R))}{believes(P, shared\_key(K, R, Q))}$$

$$\frac{believes(P, secret(Y, Q, R))}{believes(P, secret(Y, R, Q))}$$

$$\frac{distinct(P, Q)}{distinct(Q, P)}$$

B BAN checker sample step

Here we illustrate one step in the algorithm as applied to the BAN/Kerberos example from Section 3.1.2, to show how a new formula gets added to the fringe. After two levels of the breadth-first traversal are completed, there are 37 formulas in the known-valid set. Of these, 16 are in the fringe, including this one:

$$believes(B, said(S, comma(T_S, shared\_key(K_{ab}, A, B))))$$

This formula unifies with the first premise of the "nonce-verification" S-rule (see Appendix A), so we apply its unifier to the second premise of that rule, yielding

$$believes(B, fresh(comma(T_S, shared\_key(K_{ab}, A, B)))) \ .$$

None of the 37 formulas unifies with this additional premise, so we attempt to work backwards from it, using G-rules and rewrites. If we reverse-apply the first freshness G-rule, we get

$$believes(B, fresh(T_S)),$$

which is one of the initial assumptions of the protocol (and thus one of the 37 known formulas). Since all premises for the nonce-verification rule have now been matched, we insert its (instantiated) conclusion into the new fringe:

$$believes(B, believes(S, comma(T_S, shared\_key(K_{ab}, A, B)))).$$

References

[1] D. E. Bell and L. J. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, The MITRE Corporation, Bedford, MA, March 1976.

[2] J. Bentley. Little languages. *Communications of the ACM*, 29(8):711–721, 1986.

[3] R. S. Boyer and J. S. Moore. *A Computational Logic*. ACM monograph series. Academic Press, New York, 1979.

[4] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

[5] D. I. Good, R. L. London, and W. W. Bledsoe. An interactive program verification system. *IEEE Transactions on Software Engineering*, 1(1):59–67, 1979.

[6] Rajashekar Kailar. Accountability in electronic commerce protocols. *IEEE Transactions on Software Engineering*, 22(5):313–328, May 1996.

[7] R. A. Kemmerer and S. T. Eckmann. *A User's Manual for the UNISEX System*. Dept. of Computer Science, UCSB, 1983.

[8] Volker Kessler and Gabriele Wedel. AUTLOG—an advanced logic of authentication. In *Proc. the Computer Security Foundations Workshop VII*, pages 90–99. IEEE Comput. Soc., June 1994.

[9] K. N. Levitt, L. Robinson, and B. A. Silverberg. The HDM handbook, vols. 1–3. Technical report, SRI International, Menlo Park, California, 1979.

[10] R. Locasso, J. Scheid, D. V. Schorre, and P. R. Eggert. The Ina Jo reference manual. Technical Report TM-(L)-6021/001/000, System Development Corporation, Santa Monica, California, 1980.

[11] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055. Springer-Verlag, March 1996. Lecture Notes in Computer Science.

[12] S. P. Miller, C. Neuman, J. I. Schiller, and J. H. Saltzer. *Kerberos authentication and authorization system*, chapter Sect. E.2.1. MIT, Cambridge, Massachusetts, July 1987.

[13] P. Nivela and R. Nieuwenhuis. Saturation of first-order (constrained) clauses with the Saturate system. In *Proc. of the Fifth International Conference on Rewriting Techniques and Applications*, pages 436–440, June 1993.

[14] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[15] Laurence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, Cambridge, 1991.

[16] G. D. Plotkin. Building-in equational theories. *Machine Intelligence*, 7:73–90, 1972.

[17] A. W. Roscoe. Model-checking CSP. In A. W. Roscoe, editor, *A Classical Mind: Essays in Honour of C. A. R. Hoare*. Prentice-Hall, 1994.

[18] A. W. Roscoe. CSP and determinism in security modelling. In *Proc. of the 1995 IEEE Symp. on Security and Privacy*, pages 114–127, 1995.