Theory Generation for Security Protocols

Darrell Kindred

1999

CMU-CS-99-130

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee

Jeannette M. Wing, Chair
Daniel Jackson
Doug Tygar
John Rushby, SRI International

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

This research was sponsored by the National Science Foundation (NSF) under Grant Nos. CCR-9357792 and CCR-9523972, and by the Defense Advanced Research Projects Agency (DARPA) under Rome Laboratory, Air Force Materiel Command, USAF, Agreement No. F30602-97-2-0031 and Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, Grant No. F33615-93-1-1330. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring agencies or the U.S. government.

Keywords: formal verification, security, protocols, theorem proving, BAN, theory generation, RV

Abstract

This thesis introduces *theory generation*, a new general-purpose technique for performing automated verification. Theory generation draws inspiration from, and complements, both automated theorem proving and symbolic model checking, the two approaches that currently dominate mechanical reasoning. At the core of this approach is the notion of producing a finite representation of a theory—all the facts derivable from a set of assumptions. An algorithm is presented for producing compact theory representations for an expressive class of simple logics.

Security-sensitive protocols are widely used today, and the growing popularity of electronic commerce is leading to increasing reliance on them. Though simple in structure, these protocols are notoriously difficult to design properly. Since specifications of these protocols typically involve a small number of principals, keys, nonces, and messages, and since many properties of interest can be expressed in "little logics" such as the Burrows-Abadi-Needham (BAN) logic of authentication, this domain is amenable to theory generation.

Theory generation enables fast, automated analysis of these security protocols. Given the theory representation generated from a protocol specification, one can quickly test for specific desired properties, as well as directly manipulate the representation to perform other kinds of analysis, such as protocol comparison. This thesis describes applications of theory generation to more than a dozen security protocols using three existing logics of belief; these examples confirm, or in some cases expose flaws in earlier analyses.

This thesis introduces a new logic, RV, for security protocol analysis. While drawing on the BAN heritage, RV addresses a common criticism of BAN-like logics: that the *idealization* step can mask vulnerabilities present in the concrete protocol. By formalizing message interpretation, RV allows the verification of *honesty* and *secrecy* properties, in addition to the traditional belief properties. The final contribution of this thesis, the REVERE protocol analysis tool, has a theory generation core with plug-in modules for RV and other logics. Its performance is suitable for interactive use; verification times are under a minute for all examples.

Contents

Acknowledgements

I suspected for a long time that completing a dissertation would be harder for me than anything else I had ever done. I was right about that much, but I did not predict how heavily I would depend on other people to keep it from being truly impossible.

First, I want to thank Jeannette Wing, who has been a far better advisor than I could have hoped for. Her relentless enthusiasm and confidence pushed me through one technical or mental impasse after another. She combined eagerness with patience very effectively, and her knack for presenting an idea has, I hope, rubbed off on me to some degree.

The Carnegie Mellon Computer Science Department is a great place to be a grad student. To a large extent, this environment is the product of the herculean administrative efforts of Sharon Burks, Catherine Copetas, Karen Olack, and many others. The faculty and students deserve credit for continuing to resist the influences that threaten the unique flavor of this program, and the computing facilities staff has spoiled me.

The graduate student community in this department will be hard to leave. I have enjoyed the stimulation, commiseration, amusement, and education available from the Zephyr crowd. The CS volleyball group has been key to preserving my mental health; being part of a team for a couple of hours helps relieve the isolation that comes with dissertation-writing. I have been especially lucky to meet good friends Arup Mukherjee, Eka Ginting, Corey Kosak, Rob Driskill, and Marius Minea.

I am fortunate to have other friends from before my CMU days who still remember me and remind me of life outside computer science. Of these, Karen Watson deserves special mention for actually visiting us in Pittsburgh, and Drs. Valerie Sims and Matt Chin provided invaluable inspiration and support.

My mother and father are most directly responsible for who I am and whatever I can do today. They, with Josh, Lloyd, Erin, and Jonathan, have made all my visits home warm, joyful, and restorative.

Finally, I thank my wife, Sally. I have no way to express how much her love and support have meant to me throughout this experience, except through trying to return them over the rest of our life together.

Chapter 1

Introduction

1.1 Motivation

Security-sensitive protocols are widely used today, and we will rely on them even more heavily as electronic commerce continues to expand. This class of protocols includes well-known authentication protocols such as Kerberos [MNSS87] and Needham-Schroeder [NS78], newer protocols for electronic commerce such as NetBill [ST95] and Secure Electronic Transactions (SET) [VM96], and "security-enhanced" versions of existing network protocols, such as Netscape's Secure Sockets Layer (SSL) [FKK96], Secure HTTP [RS96], and Secure Shell (SSH) [Gro99].

These protocols are notoriously difficult to design properly. Researchers have uncovered critical but subtle flaws in protocols that had been scrutinized for years or even decades. Protocols that were secure in the environments for which they were designed have been used in new environments where their assumptions fail to hold, with dire consequences. These assumptions are often implicit and easily overlooked. Furthermore, security protocols by their nature demand a higher level of assurance than many systems and programs, since the use of these protocols implies that the user perceives a threat from malicious parties. The weakest-link argument requires that every component of a system be secure; since almost every modern distributed system makes use of some of these protocols, their security is crucial.

Given these considerations, we must apply careful reasoning to gain confidence in security protocols. In current practice, these protocols are analyzed sometimes using formal methods based on security-related logics such as the Burrows, Abadi, and Needham (BAN) logic of authentication, and sometimes using informal arguments and public review. While informal approaches play an important role, formal methods offer the best hope for producing convincing evidence that a protocol meets its requirements. It is for critical system properties like security that the cost of applying formal methods can most easily be justified. The formal protocol analyses, when they exist, normally take the form of pencil-and-paper specifications and proofs, sometimes checked by mechanized verification systems such as PVS [ORSvH95]. The process of encoding protocols and security properties in a general-purpose verification system is often cumbersome and error-prone, and it sometimes requires that the protocol be expressed in an unnatural way. As a result, the cost of applying formal reasoning may be seen as prohibitive and the benefits uncertain; thus, protocols are often used with only informal or possibly-flawed formal arguments for their soundness. If we can develop formal methods that demand less from the user while still providing strong assurances, the result should be more dependable protocols and systems.

1.2 Overview of Approach and Thesis Claim

This dissertation introduces a new technique, *theory generation*, which can be used to analyze these protocols and facilitate their development. This approach provides fully automated verification of the properties of interest, and feedback on the effects of refinements and modifications to protocols.

At the core of this approach is the notion of producing a finite representation of all the facts derivable from a protocol specification. The common protocols and logics in this domain have some special properties that make this approach appealing. First, the protocols can usually be expressed in terms of a small, finite number of participants, keys, messages, nonces, and so forth. Second, the logics with which we reason about them often comprise a finite number of rules of inference that cause "growth" in a controlled manner. The BAN logic of authentication, along with some other logics of belief and knowledge, meets this criterion. Together, these features of the domain make it practical to produce such a finite representation quickly and automatically.

The finite representation takes the form of a set of formulas T, which is essentially the transitive closure of the formulas constituting the protocol specification, over the rules of inference in a logic. This set is called the *theory*, or *consequence closure*. Given such a representation, verifying a specific property of interest, ϕ, requires a simple membership test: $\phi \in T$. In practice, the representation is not the entire transitive closure, and the test is slightly more involved than simple membership, but it is similar in spirit. Beyond this traditional property-testing form of verification, we can make further uses of the set T, for instance in comparing different versions of a protocol. We can capture some of the significant differences between protocols P and Q by examining the formulas that lie in the set difference $T_P \setminus T_Q$ and those that lie in $T_Q \setminus T_P$.

Using this new approach, we can provide protocol designers with a powerful and automatic tool for analyzing protocols while allowing them to express the protocols in a natural way. In addition, the tool can be instantiated with a simple representation of a logic, enabling the development of logics tailored to the verification task at hand without sacrificing push-button operation.

Beyond the domain of cryptographic protocols, theory generation could be applied to reasoning with any logic that exhibits the sort of controlled growth mentioned above (and explained formally in Chapter 2). For instance, researchers in artificial intelligence often use such logics to represent planning tasks, as in the Prodigy system [VCP+95].

The new approach makes it easy to generate automatically a checker specialized to a given logic. Just as Jon Bentley has argued the need for "little languages" [Ben86], this generator provides a way to construct "little checkers" for "little logics." The checkers are lightweight and quick, just as the logics are little in the sense of having limited connectives and restricted rules, but the results can be illuminating. As part of this thesis research, we implement a system that generates these little checkers, and we apply four such checkers to a variety of protocols.

The thesis of this work is that theory generation offers an effective form of automated reasoning, and furthermore that theory generation can be applied in the domain of authentication and electronic commerce protocols to analyze a wide range of critical security properties.

1.3 Contributions

In demonstrating this thesis, we make several significant contributions to the fields of formal methods and computer security.

First, theory generation is a new approach for formal verification. The method of producing and directly manipulating finite theory representations enables new kinds of analysis, such as the difference approach alluded to above for comparing two specifications. The TG_ℓ algorithm for theory generation provides a simple means of producing useful but compact theory representations for an expressive class of logics. As well as the algorithm itself, we provide proofs of its correctness and termination, a practical implementation, and suggestions for further enhancements.

Second, the application of theory generation to the analysis of security protocols is a new development. The utility of this approach is demonstrated through many examples, in which theory generation is applied both to existing belief logics and to a new logic, RV, for verifying protocol properties. Future protocol analysis tools could benefit from this technique.

The RV logic itself is a contribution in reasoning about security protocols. This logic provides a formal, explicit representation of message interpretations, allowing us to bridge the "idealization gap" that has been a weakness of BAN and related logics. In addition to the traditional belief properties of the sort BAN deals with, RV can express *honesty*, *secrecy*, and *feasibility* properties.

Finally, the REVERE protocol analysis tool offers protocol designers an environment in which they can quickly check properties of new protocols under development, and it can serve as a model for future implementations of theory generation and a base for development of more sophisticated protocol analyzers.

1.4 Related Work

There is a rich history of research on computer-assisted formal verification and on reasoning about security. This section contains a brief survey of the work most relevant to the thesis, focusing on the important differences between existing approaches and theory generation as applied to security protocols.

1.4.1 Theorem Proving

General-purpose automated theorem proving is the more traditional approach to verifying security properties. Early work on automated reasoning about security made use of the Affirm [GMT$^+$80], HDM [LRS79], Boyer-Moore [BM79], and Ina Jo [LSSE80] theorem-proving methodologies. This line of work was largely based on the Bell-LaPadula security model [BL76]. In proving the theorems that expressed security properties of a system or protocol, an expert user would carefully guide the prover, producing lemmas and narrowly directing the proof search to yield results.

More recent theorem-proving efforts have used the HOL [GM93], PVS [ORSvH95], and Isabelle [Pau94] verification systems to express and reason about security properties. These sophisticated verification systems support specifications in higher-order logic and allow the user to create custom proof strategies and tactics with which the systems can do more effective automated proof search. Though simple lemmas can be proved completely automatically, human guidance is still necessary for most interesting proofs.

We have done limited experiments in applying PVS to the BAN logic as an alternative to theory generation. The encoding of the logic is quite natural, but the proofs are tedious because PVS is often unable to find the right quantified-variable instantiations to apply the BAN logic's rules of inference.

Paulson uses the Isabelle theorem prover to demonstrate a range of security properties in an "inductive approach" [Pau96, BP97]. In this work, he models a protocol as a set of event traces, defined inductively by the protocol specification. He defines rules for deriving several standard message sets from a trace, such as the set of messages (and message fragments) that can be derived from a trace H using only the keys contained in H. Given these definitions, he proposes various classes of properties that can be verified: possibility properties, forwarding lemmas, regularity lemmas, authenticity theorems, and secrecy theorems. Paulson's approach has the advantage of being based on a small set of simple principles, in contrast to the sometimes complex and subtle sets of rules assumed by the BAN logic and related belief logics. It does not, however, provide the same high-level intuition into why a protocol works that the belief logics can. Paulson demonstrates proof tactics that can be applied to prove some lemmas automatically, but significant human interaction still appears to be required.

Brackin has recently developed a system within HOL for converting protocol specifications in an extended version of the GNY logic [GNY90] to HOL theories [Bra96]. His system then attempts to prove user-specified properties and certain default properties automatically. This work looks promising; one drawback is that it is tied to a specific logic. Modifying that logic or applying the technique to a new logic would require substantial effort and HOL expertise. In theory generation, the logic can be expressed straightforwardly, and the proving mechanism is independent of the logic.

Like general-purpose theorem proving, theory generation involves manipulation of the syntactic representation of the entity we are verifying. However, by restricting the nature of the logic, unlike machine-assisted theorem proving, we can enumerate the entire theory rather than (with human assistance) develop lemmas and theorems as needed. Moreover, the new method is fast and completely automatic, and thus more suitable for integration into the protocol development process.

1.4.2 Belief Logics

For reasoning about protocols in the security domain, the BAN logic and its kin have attracted significant attention in recent years [BAN90, GNY90, KW94, Kai96, SvO94]. This work diverges from the algebraic approach to cryptographic protocol modeling introduced earlier by Dolev and Yao [DY81]. In the Dolev-Yao approach, encryption and decryption are modeled as algebraic transformations on words, and reasoning about a protocol consists of proving certain words do not belong to the language corresponding to a given protocol. The BAN family emphasizes the evolution of beliefs by participants in a protocol.

Burrows, Abadi, and Needham developed their logic of authentication (BAN) around the notion of belief. Each message in a protocol is represented by a set of beliefs it is meant to convey, and principals acquire new beliefs when they receive messages, according to a small set of rules. The BAN logic allows reasoning not just about the authenticity of a message (the identity of its sender), but also about *freshness*, a quality attributed to messages that are believed to have been sent recently. This allowed BAN reasoning to uncover certain replay attacks like the well-known flaw in the Needham-Schroeder shared-key protocol [DS81].

Several other logics were developed to improve upon BAN by providing support for different cryptographic operations such as secure hashes, simplifying the set of rules, introducing the concept of a recognizable message, and other such enhancements: GNY [GNY90], SVO [SvO94], AUTLOG [KW94], and Kailar's accountability logic [Kai96] are some of the more prominent. In Chapters 3 and 4 we discuss the advantages and disadvantages of these logics further, and show how theory generation can be applied to several of them. The REVERE system described in Chapter 6 can generate "little checkers" for these logics within a unified framework.

There has been some other work on modeling security protocols with more expressive logics that are somewhat more difficult to reason with, such as temporal logics and the nonmonotonic logic of knowledge and belief introduced by Moser [IM95, Mos89]. Theory generation restricts itself to a simpler class of logics that is adequate to express many of the interesting properties of these protocols without sacrificing fully automated reasoning.

1.4.3 Model Checking

Model checking is a verification technique wherein the system to be verified is represented as a finite state machine, and properties to be checked are typically expressed as formulas in some temporal logic. The system is searched exhaustively for a path or state satisfying the given formula (or its complement). The process can be accelerated greatly through symbolic execution and the use of compact representations such as Binary Decision Diagrams (BDDs) [Bry86], which efficiently encode transition relations [BCM$^+$90, McM92]. Symbolic model checking has been used successfully to verify many concurrent hardware systems, and it has attracted significant interest in the wider verification community due to its high degree of automation and its ability to produce counterexamples when verification fails.

Jonathan Millen's Interrogator tool could be considered the first model checker for cryptographic protocol analysis [Mil84, KMM94]. It is a Prolog [CM81] system in which the user specifies a protocol as a set of state transition rules, and further specifies a scenario corresponding to some undesirable outcome (e.g., an intruder learns a private key). The system then searches, with Prolog backtracking, for a message history matching the given scenario. If such a history is found, the Interrogator has demonstrated a flaw in the protocol; however, if no matching message history is found, little can be concluded. The construction of the scenario and protocol specification constrains the search and thus may cause valid attacks to be ignored.

Recently, advanced general-purpose model checkers have been applied to protocol analysis with some encouraging results. Lowe used the FDR model checker [Ros94] to demonstrate a flaw in, and then fix, the Needham-Schroeder public key protocol [Low96] and (with Roscoe) the TMN protocol [LR97], and Roscoe used FDR to check noninterference of a simple security hierarchy (high security/low security) [Ros95]. Heintze, Tygar, Wing, and Wong used FDR to check some atomicity properties of NetBill [ST95] and Digicash [CFN88] protocols [HTWW96]. Mitchell, Mitchell, and Stern developed a technique for analyzing cryptographic protocols using Murϕ, a model-checker that uses explicit state representation, and, with Shmatikov, have applied it to the complex SSL protocol [MMS97, MSS98]. Finally, Marrero, Clarke, and Jha have produced a specialized model checker for reasoning about security protocols, which takes a simple protocol specification as input and does not require a protocol-specific adversary process [CJM98].

Some of these modern model checkers borrow from the semantic model introduced by Woo and Lam [WL93], in which authentication protocols are specified as collections of processes that perform sequences of actions such as sending and receiving messages, initiating and terminating a session, and creating nonces. Correctness properties in the Woo-Lam model are classified as *secrecy* or *correspondence* properties. The correspondence properties link actions taken by two different principals in a protocol run; for instance, if principal Q completes a protocol run with an $\mathit{EndInit}(P)$ action, then P must have taken a $\mathit{BeginRespond}(P)$ action earlier.

All these model-checking approaches share the limitation that they can consider only a limited number of runs of a protocol—typically one or two—before the number of states of the finite state machine becomes unmanageable. This limitation results from the well-known state explosion problem exhibited by concurrent systems. In some cases it can be worked around by proving that any possible attack must correspond to an attack using at most n protocol runs.

The theory generation technique takes significant inspiration from the desirable features of model checking. Like model checking, theory generation allows "push-button" verification with no lemmas to postulate and no invariants to infer. Whereas model checking achieves this automation by requiring a finite model, theory generation achieves it by requiring a simple logic. Also like model checking, theory generation seeks to provide more interesting feedback than a simple "yes, this property holds" or "I cannot prove this property." In model checking, counterexamples give concrete illustrations of failures, while theory generation offers the opportunity to directly examine and compare theories corresponding to various protocols. By taking advantage of the intuitive belief logics, theory generation can better provide the user with a sense of why a protocol works or how it might be improved; model checking is more of a "black box" in this respect. The two approaches can complement each other, as model checking provides answers without specifying belief interpretations for the protocol, while theory generation presents the user with a higher-level view of the protocol's effects.

1.4.4 Hybrid Approaches

Meadows' NRL Protocol Analyzer is perhaps the best known tool for computer-assisted security protocol analysis [Mea94]. It is a semi-automated tool that takes a protocol description and a specification of some bad state, and produces the set of states that could immediately precede it. In a sense the Analyzer represents a hybrid of model checking and theorem proving approaches: it interleaves brute-force state exploration with the user-guided derivation of lemmas to prune the search space. It has the notable advantages that it can reason about parallel protocol run attacks and that it produces sample attacks, but it sometimes suffers from the state explosion problem and it requires significant manual guidance. It has been applied to a number of security protocols, and continuing work has increased the level of automation [Mea98]. Theory generation, however, offers greater automation, the benefits of intuitive belief logics, and protocol comparison abilities not available in the Analyzer.

While not a protocol analysis approach in its own right, the Common Authentication Protocol Specification Language (CAPSL) [Mil97] plays an important role in this field. Developed by Millen in cooperation with other protocol analysis researchers, CAPSL is intended to provide a unified notation for describing protocols and their accompanying assumptions and goals. It captures and standardizes notions shared by most modern protocol analysis techniques: principals, encryption, nonces, freshness, concatenation, and so forth, but it also allows modular extensions for expressing user-specified abstractions. As more analysis tools are constructed to CAPSL, it should become easier to share protocol specifications among different tools, and more practical to develop and use specialized tools for an integrated analysis.

1.4.5 Logic Programming and Saturation

Theory generation shares some elements with logic programming, as embodied in programming languages like Prolog. Through careful use of Prolog backtracking, one can enumerate derivable formulas to produce results similar to those of theory generation. Automated reasoning with AUTLOG has been implemented in Prolog, and similar techniques could probably be applied to automate a variety of belief logics. We encoded the BAN logic in Prolog, but found that the encoding was very sensitive to the ordering of rules and required hand-tuning to ensure termination. Reasoning via theory generation is more general in that we can produce results and guarantee termination without any dependence on rule ordering, and because we use a modified unification algorithm that can handle simple rewrite rules. At the same time, it is more specific; by tailoring our reasoning to a specific domain—these "little" belief logics—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 a finite "closure" (or "completion") of a theory is used in the theorem proving system SATURATE [NN93]. SATURATE takes a set of first-order logic formulas and attempts to compute a finite set of formulas that represents the *saturated theory* induced by those axioms. This saturated theory is a set of deduced clauses without redundancy, where a clause is redundant if it is entailed by smaller clauses in the set. If it succeeds, the theory is guaranteed to be consistent, and the saturated set can be used to create a decision procedure for the theory. 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 this more general system. The primary purposes of the completion computed by SATURATE are to provide a demonstration that the input theory is consistent, and to allow a fast decision procedure to operate on the saturated theory. We choose to emphasize different uses for the finite sets—in particular, direct analysis and comparison of these sets. In addition, the restricted framework we adopt frees the user from worrying about termination; SATURATE accepts a broader range of inputs and thus cannot ensure that the completion operation will halt. In limited experiments applying SATURATE to the BAN logic, we were unable to find a configuration in which saturation would terminate on most trials.

1.5 Road Map

In the remainder of this dissertation, we describe the theory and practice of theory generation for security protocol verification. In Chapter 2 we describe theory generation formally and present an algorithm for performing theory generation. Chapter 3 presents theory generation as applied with existing belief logics to analyze various protocols. In Chapter 4 we develop a new belief logic, RV, whose explicit notion of interpretation yields significant advantages over existing belief logics. In Chapter 5 we lay out a step-by-step method for doing more comprehensive protocol verification with RV, with sample applications to several protocols. Discussion of practical issues arising in the implementation of the theory generation algorithm appears in Chapter 6, along with a description of the REVERE protocol verification tool and some performance benchmarks. Finally, Chapter 7 contains reflections on the contributions of this work and directions for future research.

Chapter 2

Theory Generation

When working with a logic, whether for program verification, planning, diagnosis, or any other purpose, a natural question to ask is, "What conclusions can we derive from our assumptions?" Given a logic, and some set of assumptions, T^0, we consider the complete theory, T^*, induced by T^0. Also known as the *consequence closure*, T^* is simply the (possibly infinite) set of formulas that can be derived from T^0, using the rules and axioms of the logic. We typically explore T^* by probing it at specific points: "Can we derive formula F? What about G?" Sometimes we test whether T^* contains every formula; that is, whether the theory is inconsistent. It is interesting, however, to consider whether we can characterize T^* more directly, and if so what benefits that might bring. In the following sections, we explore for a special class of logics a general way of representing T^* and a technique, called *theory generation*, for mechanically producing that representation.

2.1 Logics

Theory generation may in principle be applied to any logic, but in this chapter we consider only those falling within a simple fragment of first-order logic. We start by defining briefly the standard components of first-order logic. More complete treatments may be found in introductory logic texts [Bar77].

Definition 2.1 *A* first-order language*, L, is a possibly infinite set of variables, constants, function symbols, and predicate symbols.*

In this chapter, we will use X and Y for variables; S and T for terms; F, G, P, C, and ϕ for formulas; and f, g, and p for function and predicate symbols.

Definition 2.2 *The set of* terms *of L is the smallest set such that every variable is a term, and every expression $f(T_1, \ldots, T_n)$ is a term, where f is a function symbol, $T_1, \ldots, T_n$ are terms, and $n \geq 0$.*

Definition 2.3 *The* well-formed formulas *(*wff*s) are built from terms as follows:*

- *$p(T_1, \ldots, T_n)$ is a wff if $T_1, \ldots, T_n$ are terms and p is a predicate symbol (for any $n \geq 0$);*
- *$\forall X.F$ is a wff if X is a variable and F is a wff; and*
- *$(\neg F)$ and $(F \Rightarrow G)$ are wffs if F and G are wffs.*

The other propositional connectives ($\vee$, $\wedge$, $\iff$) and the existential quantifier ($\exists$) may be used as abbreviations for their equivalents in terms of $\forall$, $\neg$, and $\Rightarrow$.

In first-order logic with equality, the set of wffs above is extended to include all formulas of the form

$$S = T$$

where S and T are terms.

We restrict our attention to the fragment of first-order logic we call $\mathcal{L}$, defined here. (We use $\overline{X}$ as a shorthand for $X_1, \ldots, X_k$.)

Definition 2.4 *The logic, $\mathcal{L}$, is that fragment of first-order logic whose well-formed formulas are all of the following form:*

$$\forall \overline{X}.((P_1 \wedge P_2 \wedge \cdots \wedge P_m) \Rightarrow C)$$

where $m \geq 0$, $n \geq 0$, the P_i and C are formulas containing no connectives or quantifiers, and $\overline{X}$ are all the free variables occurring in those formulas.

For formulas of $\mathcal{L}$ in which $m = 0$, the standard form reduces to

$$\forall \overline{X}.C \tag{2.1}$$

We will sometimes interpret formulas of $\mathcal{L}$ (in which $m > 0$) as rules of inference, writing them in this form:

$$\frac{P_1 \quad P_2 \quad \cdots \quad P_m}{C}, \tag{2.2}$$

Note that this class of formulas is equivalent to Horn clauses. We will normally refer to formulas in the forms of (2.1) and (2.2) as *assumptions* and *rules* respectively. (In Prolog terminology, these are facts and rules.) We reserve the term *rules of inference* to refer specifically to the inference rules for $\mathcal{L}$ itself (such as *modus ponens*, below).

We will also consider an extension of $\mathcal{L}$ that includes a limited form of equality:

Definition 2.5 *The logic, $\mathcal{L}^{=}$, is the fragment of first-order logic with equality whose well-formed formulas include all the wffs of $\mathcal{L}$, as well as formulas of this form:*

$$\forall \overline{X}.(S = T) \tag{2.3}$$

where S and T are terms, and the X_i are all the free variables occurring in S and T.

We call these universally-quantified equalities *rewrites*.

We will use Hilbert-style axioms and rules of inference for first-order logic. The axioms include

- all propositional tautologies,
- all formulas of the form

  $$(\forall x.\phi(x)) \Rightarrow \phi(t)$$

  where x is any variable and t any term, and
- for first-order logic with equality, the standard equality axioms:
  - $(T = T)$ (reflexivity),
  - $(S = T) \Rightarrow (T = S)$ (commutativity),
  - $((T_1 = T_2) \wedge (T_2 = T_3)) \Rightarrow (T_1 = T_3)$ (transitivity),
  - $(S_1 = T_1 \wedge \cdots \wedge S_n = T_n) \Rightarrow (p(S_1, \ldots, S_n) \Rightarrow p(T_1, \ldots, T_n))$, and
  - $(S_1 = T_1 \wedge \cdots \wedge S_n = T_n) \Rightarrow$
    $([S_1 \backslash X_1, \ldots, S_n \backslash X_n]t = [T_1 \backslash X_1, \ldots, T_n \backslash X_n]t)$

  (The notation $[S_1 \backslash X_1, \ldots, S_n \backslash X_n]t$ denotes the term obtained by substituting S_1 for free occurrences of X_1 in t, S_2 for X_2, and so on.)

The rules of inference are *modus ponens*:

$$\frac{(F \Rightarrow G) \quad F}{G}$$

and the generalization rule:

$$\frac{F \Rightarrow G}{F \Rightarrow \forall y.[y \backslash x]G}$$

(where x is not free in F).

Given these axioms and rules of inference, we can define the formal notion of proof:

Definition 2.6 *A* proof *of some formula F from a set of assumptions Γ is a finite sequence, $\{F_1, \ldots, F_n\}$, of formulas, where F_n is F, and each element F_i is either an axiom instance, a member of Γ, or the result of an application of one of the rules of inference using formulas in $F_1, \ldots, F_{i-1}$.*

We say a formula F is *derivable from* Γ (written $\Gamma \vdash F$) if a proof of F from Γ exists.

We now introduce a class of "little logics" that are, in a way, equivalent to $\mathcal{L}^=$, parameterized by a fixed set of rules and rewrites:

Definition 2.7 *The logic, ℓ_{RW}, where R is a set of rules in the form of (2.2), and W is a set of rewrites in the form of (2.3), has as its formulas all connective-free formulas of $\mathcal{L}$. The rules of inference of ℓ_{RW} are all the rules in R, as well as instantiation, and substitution of equal terms using the rewrites in W.*

Proofs in ℓ_{RW} are thus finite sequences of formulas in which each formula is either an assumption, an instance of an earlier formula, the result of applying one of the rules in R, or the result of a replacement using a rewrite in W.

The following theorem shows the correspondence between ℓ_{RW} and $\mathcal{L}^=$:

Theorem 2.1 *If ϕ is a formula of ℓ_{RW}, and Γ is a set of formulas of ℓ_{RW}, then*

$$\Gamma \vdash_{\ell_{RW}} \phi$$

if and only if

$$\Gamma \cup R \cup W \vdash_{\mathcal{L}^=} \phi$$

That is, proof in ℓ_{RW} is equivalent to proof in $\mathcal{L}^=$ using the same rules and rewrites (R and W). The proof of this theorem is long but fairly mechanical; it appears in Appendix A.

Finally, we define the notion of *theory* (also known as *consequence closure*):

Definition 2.8 *Let L be a logic, and T^0 a set of wffs in that logic. T^*, the* theory induced by T^0, *is the possibly infinite set of wffs containing exactly those wffs that can be derived from T^0 and the axioms of L, using the rules of L.*

T^* is closed with respect to inference, in that every formula that can be derived from T^* is a member of T^*. In the following sections, we consider theories in the context of ℓ_{RW} logics and show how to generate representations of those theories.

2.2 Theory Representation

The goal of theory generation is to produce a finite representation of the theory induced by some set of assumptions; in this section we consider the forms this representation might take, and the factors that may weigh in favor of one representation or another. These factors will be determined in part by the purposes for which we plan to use the representation. There are three primary uses for the generated theory representation:

- It may be used in a decision procedure for determining the derivability of specific formulas of interest (see Section 2.4).
- It may be manipulated and examined by a human through assorted filters, in order to gain insight into the nature of the complete theory.
- It may be directly compared with the representation of some other theory, to reveal differences between the two theories.

These applications are discussed in greater detail in Chapter 5.

The full theory is a possibly infinite set of formulas entailed by some initial assumptions, so the clearest requirement of the representation we generate is that it be finite. A natural way to achieve this is to select a finite set of "representative formulas" that belong to the theory, and let this set represent the full theory. Other approaches are conceivable; we could construct a notation for expressing certain infinite sets of formulas, perhaps analogous to regular expressions or context-free grammars. However, the logic itself is already quite expressive; indeed, the set of initial assumptions and rules "expresses" the full theory in some sense. There is no clear benefit of creating a separate language for theory description.

Given that we choose to represent a theory by selecting some subset of its formulas, it remains to decide what subset is best. Here are a few informal criteria that may influence our choice:

C1. The set of formulas must be finite, and should be small enough to make direct manipulation practical. An enormous, though finite, set would probably be not only inefficient to generate, but unsuitable for examination by a human except through very fine filters.

C2. There should be an algorithm that generates the set with reasonable efficiency.

C3. Given an already-generated set, there should be an efficient decision procedure for the full theory, using that set. Since the simplest way to characterize a theory is to test specific formulas for membership, a quick decision procedure is important.

C4. The set should be canonical. For a given set of initial assumptions, the generated theory representation should be uniquely determined. This makes direct comparison of the sets more useful.

C5. The set should include as many of the "interesting" formulas in the theory as possible, and as few of the "uninteresting" ones as possible. For instance, when a large class of formulas exists in the theory, but all represent essentially the same fact, it might be best for the theory representation to include only the simplest formula from this class. This will enable humans to glean useful information from the generated set without sifting through too much chaff.

Ganzinger, Nivela, and Niewenhuis's SATURATE prover takes one approach to this problem [NN93]. SATURATE is designed to work with a very general logic: full first-order logic over transitive relations. It can, under some circumstances, produce a finite "saturated theory" that enables an efficient decision procedure. The saturation process can also be used to check the consistency of a set of formulas, since **false** will appear in the saturated set if the original set is inconsistent. The price SATURATE pays for its generality is that saturation is not guaranteed to terminate in all cases; there are a number of user-tunable parameters that control the saturation process and make it more or less thorough and more or less likely to terminate. This flexibility is sometimes necessary, but we choose to focus on more limited logics in which we can make stronger guarantees regarding termination and require less assistance from the user.

For ℓ_{RW} logics, we select a class of theory representations we refer to as (R, R') *representations*:

Definition 2.9 *From the set of rules, R, choose some subset, R'. An (R, R')* representation *of the theory induced by T^0 contains a set of formulas derivable from T^0, such that any proof from the assumptions, T^0, in which the last rule application (if any) is an application of some rule in R', the conclusion of that proof is equivalent to some formula in the set. Furthermore, every formula in the set is equivalent to the conclusion of some such proof. Finally, no two formulas in the set are equivalent.*

In this formulation, the rules in R' are "preferred" in that they are applied as far as possible. The equivalence used in this definition may be strict identity or some looser equivalence relation. We can test a formula for membership in the full theory by using the theory representation and just the rules in $(R - R')$ (this is proved in Section 2.4). This can be significantly more efficient than the general decision problem using all of R, so criterion C3 can be satisfied. If we select the representation to include only one canonical formula from any equivalence class, then the representation is completely determined by T^0, R, and R', so criterion C4 is satisfied.

In order to satisfy the remaining criteria (C1, C2, and C5), we must choose R' carefully. We could choose $R' = \{\}$, but the resulting theory representation would just be T^0, the initial assumptions: unlikely to enable an efficient decision procedure and certainly not very "interesting" (in the sense of C5). For some sets of rules, we could let $R' = R$, but in many cases this would yield an infinite representative set, violating C1. In the next section we describe a method for selecting R' that is guaranteed to produce a finite representative set, and that satisfies the remaining criteria in practice.

2.3 The Theory Generation Algorithm, TG_ℓ

In an ℓ_{RW} logic, as defined in Section 2.1, we can automatically generate a finite representation of the theory induced by some set of formulas, Γ, provided the logic and the formulas in Γ satisfy some additional restrictions. In the following section, we describe these preconditions and explain how they can be checked. Section 2.3.2 contains a description the algorithm itself, which is followed by proofs of its correctness and termination in Sections 2.3.3 and 2.3.4.

2.3.1 Preconditions

In order to apply our theory generation algorithm, TG_ℓ, to a logic, ℓ_{RW}, and a set of assumptions, Γ, some preconditions on the rules and rewrites (R and W) of ℓ_{RW}, and on Γ, must hold. These preconditions require that the set of rules (R) can be partitioned into *S-rules* and *G-rules*, that the rewrites (W) be *size-preserving*, and that the formulas in Γ be *mostly-ground*. Informally, the S-rules are "shrinking rules," since they tend to produce conclusions no larger than their premises, and the G-rules are "growing rules" since they have the opposite effect. The S-rules are the principal rules of inference; in the generated theory representation, they will be treated as the preferred rules (the R' set in an (R, R') representation). We define these terms and formally present the preconditions in this section.

The TG_ℓ algorithm repeatedly applies rules, starting from the assumptions, to produce an expanding set of derivable formulas. The basic intent of these preconditions is to limit the ways in which formulas can "grow" through the application of rules. As long as the process of applying rules cannot produce formulas larger than the ones we started with, we can hope to reach a fixed point, where no further application of the rules can yield a new formula. The algorithm eagerly applies the S-rules to the assumptions as far as possible, using the G-rules and rewrites only as necessary. The restrictions below ensure first that the algorithm can find a new S-rule application in a finite number of steps, and second that each new formula derived is smaller than some already-known formula, so the whole process will terminate.

The preconditions below are defined with respect to a pre-order (a reflexive and transitive relation) on terms and formulas, $\preceq$. This pre-order may be defined differently for each application of the algorithm, but it must always satisfy these conditions:

P1. The pre-order must be monotonic; that is,

$$(T_1 \preceq T_2) \Rightarrow ([T_1 \backslash X]F \preceq [T_2 \backslash X]F)$$

P2. The pre-order must be preserved under substitution:

$$(F \preceq G) \Rightarrow ([T \backslash X]F \preceq [T \backslash X]G)$$

P3. The set $\{F \mid F \preceq G\}$ must be finite (modulo variable renaming) for all formulas G. This is a form of well-foundedness.

Intuitively, $F \preceq G$ means that the formula F is no larger than G; with this interpretation, condition P3 means that there are only finitely many formulas no larger than G. In Chapter 3, we construct a specific $\preceq$ relation satisfying these conditions, which can be used with only minor variations in a variety of situations.

If there exists some such $\preceq$ under which the preconditions below are met, then the TG_ℓ algorithm can be applied to ℓ_{RW} and Γ.

We can weaken the condition P3 above slightly, to allow broader application of the algorithm. We introduce a syntactic constraint on formulas, represented as a set of formulas, $\mathcal{C}$. Every assumption in Γ must be in $\mathcal{C}$, and $\mathcal{C}$ must be closed over all the rules and rewrites (that is, when applied to formulas in $\mathcal{C}$, rules and rewrites must yield conclusions in $\mathcal{C}$). Furthermore, applying a substitution to a formula in $\mathcal{C}$ must always yield another formula in $\mathcal{C}$. We can then allow a pre-order $\preceq$ for which P3 above may not hold but P3$'$ does:

P3$'$. The set $\{F \in \mathcal{C} \mid F \preceq G\}$ must be finite (modulo variable renaming) for all formulas G.

In describing the preconditions, we use a notion of *mostly-ground* formulas:

Definition 2.10 *A formula, F, is* mostly-ground *with respect to a pre-order, $\preceq$, and some syntactic constraint, $\mathcal{C}$, if*

$$\forall \sigma.(\sigma F \in \mathcal{C} \Rightarrow \sigma F \preceq F)\ ,$$

where σ ranges over substitutions. That is, every instance of F that meets the syntactic constraint is no larger than F itself.

Any ground (variable-free) formula is trivially mostly-ground. For some definitions of $\preceq$ and some constraints on formulas, however, certain formulas containing variables may also be mostly-ground. Chapter 3 contains such an example, in which $\mathcal{C}$ limits the possible values for some function arguments to a finite set. Note that, as a consequence of P3$'$, there exist only a finite number of instances (modulo variable renaming) of any mostly-ground formula. The preconditions require that every formula in Γ be mostly-ground, in order to limit the number of new formulas that can be derived through simple instantiation.

Before proceeding with the preconditions, we need to define *unification modulo rewrites* briefly:

Definition 2.11 *Formulas F_1 and F_2 can be* unified modulo rewrites *if and only if there exists a substitution, σ, of terms for variables such that*

$$W \Rightarrow (\sigma F_1 = \sigma F_2)$$

where W is the set of all rewrites. The substitution σ is called a unifier *for F_1 and F_2.*

With this definition, if F_1 and F_2 can be unified modulo rewrites, then we can prove F_2 from F_1 by applying instances of zero or more rewrites. Except where explicitly noted, unification is always assumed to be modulo rewrites.

Now we define the allowed classes of rules in R: S-rules and G-rules.

Definition 2.12 *An* S-rule *(shrinking rule) is a rule in which some subset, pri, of the P_i (premises) are designated "primary premises," and for which the conclusion, C, satisfies*

$$\exists P \in pri.(C \preceq P) \ .$$

That is, for an S-rule, the conclusion is no larger than some primary premise. We call the non-primary premises *side conditions*. The premises of an S-rule may be partitioned into primary premises and side conditions in any way such that this condition holds, and the S/G restriction (described later) is also satisfied.

The *G-rules* are applied only as necessary, so it is safe for them to yield formulas larger than their premises. In fact, it is required that the G-rules "grow" in this way, so that backward chaining with them will terminate.

Definition 2.13 *A* G-rule *(growing rule) is a rule for which*

$$\forall k.(P_k \preceq C)$$

In a G-rule, the conclusion is at least as large as any premise.

The *rewrites* are intended to provide simple transformations that have no effect on the size of a formula. They are often useful for expressing associativity or commutativity of certain functions in the logic. The preconditions require that all rewrites (the set W) be *size-preserving*:

Definition 2.14 *A rewrite,*

$$\forall \overline{X}.(S = T) \ ,$$

is size-preserving *if $S \preceq T$ and $T \preceq S$.*

From this definition and pre-order conditions P1 and P2, it follows immediately that if we apply a rewrite to a formula, F, producing F', then

$$\forall G.(G \preceq F) \iff (G \preceq F')$$

and

$$\forall G.(F \preceq G) \iff (F' \preceq G) \ .$$

That is, the rewrite has not affected the "size" of F.

Finally, to guarantee termination, the algorithm requires that the *S/G restriction* hold for each S-rule:

Definition 2.15 S/G restriction: *Given an S-rule with primary premises P_i, side conditions S_i, and conclusion C,*

- *each primary premise P_i must not unify with any G-rule conclusion, and*
- *each side-condition, S_i, must satisfy $S_i \preceq C$.*

Note that this restriction constrains the manner in which S- and G-rules can interact with each other. Whereas the other restrictions are local properties and thus can be checked for each rule, rewrite, and assumption in isolation, this global restriction involves all rules and rewrites. It can, however, be checked quickly and automatically. Along with the S-rule definition, this restriction defines the role of the side conditions: unlike the primary premises, whose instantiations largely determine the form of the S-rule's result, the side conditions serve mainly as qualifications that can prevent or enable a particular application of the S-rule.

We can now define the full precondition that ensures the TG_ℓ algorithm will succeed with a given set of inputs.

Definition 2.16 *The TG_ℓ precondition holds for some finite sets of assumptions (Γ), rules (R), and rewrites (W); some syntactic constraint ($\mathcal{C}$); and some pre-order ($\preceq$), if and only if all of the following are true:*

- *The pre-order, $\preceq$, satisfies conditions P1, P2, and P3$'$ (given $\mathcal{C}$).*
- *Every formula in Γ is mostly-ground, with respect to $\preceq$.*
- *Every rule in R is either a G-rule or an S-rule, with respect to $\preceq$.*
- *Every rewrite in W is size-preserving, with respect to $\preceq$.*
- *The S/G restriction holds for each S-rule in R.*

Given a pre-order, $\preceq$, that is computable and satisfies the P1-P3$'$ conditions, and a test for mostly-groundness corresponding to $\preceq$, it is possible to check the last four components of this precondition automatically. The partitioning of the rules into S- and G-rules may not be completely determined by the definitions above. If a rule has no premise larger than its conclusion, and the conclusion no larger than any premise, it could go into either category. In some cases, the S/G restriction may determine the choice, but in others it must be made arbitrarily or at the user's suggestion. (A rule whose conclusions are rarely interesting in their own right should probably be designated a G-rule.) The S-rules' primary premises can be identified automatically as those premises that match no G-rule conclusions.

There are ℓ_{RW} logics for which regardless of the $\preceq$ pre-order chosen, the TG_ℓ preconditions cannot hold. Here is one such logic:

$$\mathbf{R1} : \frac{g(f(X))}{g(X)} \qquad \mathbf{R2} : \frac{g(X)}{g(f(X))}$$

(There are no rewrites or syntactic constraints.) To see why this logic can never meet the TG_ℓ preconditions, consider first the case that $\mathbf{R1}$ is an S-rule. Since $\mathbf{R1}$'s premise matches the conclusion of $\mathbf{R2}$, it follows that $\mathbf{R2}$ must also be an S-rule or the S/G restriction would fail. Since $\mathbf{R2}$ is an S-rule, that implies that its conclusion is no larger than its premise:

$$g(f(X)) \preceq g(X)$$

By pre-order condition P2 and reflexivity of pre-orders, all formulas of the form, $g(f(f(...f(X))))$, are $\preceq g(X)$. Since there are infinitely many such formulas, pre-order condition P3 is violated, so we have a contradiction, and $\mathbf{R1}$ must not be an S-rule. The only other possibility is that $\mathbf{R1}$ is a G-rule. From the G-rule definition, though, we have

$$g(f(X)) \preceq g(X)$$

which, again, is impossible, so this pair of rules is unusable with TG_ℓ.

2.3.2 The Algorithm

The theory generation algorithm, TG_ℓ, essentially consists of performing *forward chaining* with the S-rules starting from the assumptions, with *backward chaining* at each step to satisfy S-rule premises using G-rules and rewrites. Forward chaining is the repeated application of rules to a set of known formulas, adding new known formulas to this set until a fixed point is reached. Backward chaining is the process of searching for a derivation of some desired formula by applying rules “in reverse,” until all premises can be satisfied from known formulas. The basic components of the algorithm—forward chaining, backward chaining, and unification—are widely used methods in theorem proving and planning. We assemble them in a way that takes advantage of the S-rule/G-rule distinction, and that applies rewrites transparently through a modified unification procedure. In this section, we describe this combined forward/backward chaining approach in detail.

The skeleton of the TG_ℓ algorithm is the exploration of a directed graph which has a node for each formula that will be in the generated theory representation. The roots of this graph are the assumptions, and an edge from F to G indicates that the formula F is used (perhaps via G-rules and rewrites) to satisfy a premise of an S-rule which yields G. The algorithm enumerates the nodes in this graph through a breadth-first traversal. At each step of the traversal, we consider all possible applications of the S-rules using the formulas derived so far—the visited nodes of the graph. The new fringe consists of all the new conclusions reached by those S-rule applications. When no new conclusions can be reached, the exploration is complete and the formulas already enumerated constitute the theory representation.

Before describing the algorithm in detail, we present a simple example application of TG_ℓ. The logic we use has one S-rule:

$$\mathbf{S1} : \frac{\mathit{wearing}(P, X) \qquad \mathit{thick}(X)}{\mathit{warm}(P)}$$

one G-rule:

$$\mathbf{G1} : \frac{\mathit{thick}(X)}{\mathit{thick}(\mathit{layered}(X, Y))}$$

and one rewrite:

$$\mathbf{W1} : \mathit{layered}(X, Y) = \mathit{layered}(Y, X) \ .$$

We apply TG_ℓ to these initial assumptions:

$$\begin{array}{l} \mathit{wearing}(\mathit{alice}, \mathit{layered}(\mathit{blouse}, \mathit{sweater})) \\ \mathit{thick}(\mathit{sweater}) \end{array}$$

First, the algorithm tries to apply rule $\mathbf{S1}$; unifying its primary premise with one of the known formulas gives this substitution:

$$P = \mathit{alice}, X = \mathit{layered}(\mathit{blouse}, \mathit{sweater})$$

To satisfy the other premise, we do backward chaining with the G-rule, starting with

$$thick(layered(blouse, sweater))$$

This matches no known-valid formula directly, so we try reverse-applying the G-rule ($\mathbf{G1}$). Unifying the desired formula with $\mathbf{G1}$'s conclusion (modulo the rewrite, $\mathbf{W1}$), we get the two substitutions,

$$X = blouse, Y = sweater$$

and

$$X = sweater, Y = blouse\ .$$

We instantiate $\mathbf{G1}$'s premise with the first substitution, and get

$$thick(blouse)$$

which fails to match any known formula or G-rule conclusion. We then instantiate $\mathbf{G1}$'s premise with the second substitution, and reach

$$thick(sweater)$$

which matches one of the initial assumptions. Since all $\mathbf{S1}$'s premises have been satisfied, we proceed with the application, and we add $warm(alice)$ to the set of known-valid formulas. No further applications of S1 are possible, so TG_ℓ terminates, producing this theory representation:

$$\begin{array}{l} wearing(alice, layered(blouse, sweater)) \\ thick(sweater) \\ warm(alice) \end{array}$$

A pseudocode sketch of the algorithm appears in Figures 2.1–2.2. The $theory\_gen$ function verifies the precondition and invokes $closure$ to generate the theory. The $closure$ function performs the basic breadth-first traversal; it builds up a set of formulas in T which will eventually be the theory representation, while keeping track of a "fringe" of newly added formulas. At each step, $closure$ finds all formulas derivable from $(T \cup fringe)$ with a single S-rule application by calling $apply\_srule$ for each S-rule. It then takes the canonical forms of these derivable formulas, and puts any new ones (not already derived) in the new fringe.

The formulas added to T are always canonical representatives of their equivalence classes, under a renaming/rewriting equivalence relation. To be precise,

$\big[$ Generate the theory representation induced by the given assumptions, S-rules, G-rules, and rewrites, under the pre-order, $\preceq$.
The arguments to $\mathit{theory\_gen}$ are assumed to be in scope in all other functions.

function $\mathit{theory\_gen}(\mathit{Assumptions}, \mathit{S\_rules}, \mathit{G\_rules}, \mathit{Rewrites}, \preceq) =$
  if $\neg \mathit{SG\_restriction\_ok}(\mathit{S\_rules}, \mathit{G\_rules}, \mathit{Rewrites}, \preceq)$ **then**
    raise `BadRules`
  else
    return $\mathit{closure}(\mathit{make\_canonical}(\mathit{Assumptions}), \{\})$

$\big[$ Given a partially-generated theory representation (T) and some new formulas (fringe), return the theory representation of $T \cup \mathit{fringe}$.

function $\mathit{closure}(\mathit{fringe}, T) =$
  if $\mathit{fringe} = \{\}$ **then**
    return T
  else
    $T' \leftarrow T \cup \mathit{fringe}$
    $\mathit{fringe}' \leftarrow \bigcup_{R \in \mathit{S\_rules}} \mathit{make\_canonical}(\mathit{apply\_srule}(R, T')) \setminus T'$
    return $\mathit{closure}(\mathit{fringe}', T')$

$\big[$ Apply the given S-rule in every possible way using the formulas in known, with help from the G-rules and rewrites.

function $\mathit{apply\_srule}(R, \mathit{known}) =$
  return $\{\mathit{apply\_subst}(\sigma, \mathit{conclusion}(R))$
      $\mid \sigma \in \mathit{backward\_chain}(\mathit{premises}(R), \mathit{known}, \{\})\}$

Figure 2.1: Sketch of the TG_ℓ algorithm (part 1 of 2).

this relation equates formulas F and G if there exists a renaming σ such that the rewrites imply $\sigma F = \sigma G$. (A renaming is a substitution that only substitutes variables for variables.) By selecting these canonical representatives, we prevent some redundancy in the theory representation. The renaming equivalence is actually necessary to ensure termination, since variables can be renamed arbitrarily by the lower-level functions. Requiring that formulas be canonical modulo rewrites, while not strictly necessary for termination, does make the algorithm faster, and perhaps more importantly, makes the theory representation canonical in the sense of criterion C4 (from Section 2.2). The canonical representatives may be chosen efficiently using a simple lexicographical order.

The $apply\_srule$ function simply calls $backward\_chain$ in order to find all ways to satisfy the given S-rule's premises. Note that, as represented here, $closure$ finds all possible S-rule applications and simply ignores the ones that do not produce new formulas. If the $apply\_srule$ function is told which formulas are in the fringe, and can pass this information along to $backward\_chain$, it can avoid many of these redundant S-rule applications, and return only formulas whose derivations make use of some formula from the fringe. This optimization, which yields substantial speedups, is discussed further in Section 6.1.1.

Figure 2.2 contains the three mutually recursive backward-chaining functions: $backward\_chain$, $backward\_chain\_one$, and $reverse\_apply\_grule$. The purpose of $backward\_chain$ is to satisfy a set of goals in all possible ways, with the help of G-rules and rewrites. It calls $choose\_goal$ to select the first goal to work on (g). Goals which match some G-rule conclusion, and thus may require a deeper search, are postponed until all other goals have been met. (Note that these goals can only arise from G-rule premises or S-rule side-conditions.) The $choose\_goal$ function may apply further heuristics to help narrow the search early; see Section 6.1.1 for some such approaches.

The $backward\_chain\_one$ function searches for a derivation of a single formula (ϕ) with G-rules and rewrites. It first checks whether a canonical equivalent of ϕ occurs in the $visited$ set, and fails if so, since those formulas are assumed to be unprovable. This occurrence corresponds to a derivation search which has hit a cycle. If ϕ is not in this set, the function renames variables occurring in ϕ uniquely, to avoid conflicts with variables in the G-rules, rewrites, and $known$. It then collects all substitutions that unify ϕ (modulo rewrites) with some formula in $known$, and for each G-rule, calls $reverse\_apply\_grule$ to see whether that G-rule could be used to prove ϕ. Each substitution that satisfies ϕ either directly from $known$ or indirectly via G-rules is returned, composed with the variable-renaming substitution.

In $reverse\_apply\_grule$, we simply find substitutions under which the conclusion of the given G-rule's conclusion matches ϕ, and for each of those substitutions, call $backward\_chain$ recursively to search for derivations of each of the G-rule's (instantiated) premises.

The TG_ℓ algorithm relies on several low-level utility functions, listed in Figure 2.3. Most of these are simple and require no further discussion, but the $unify$ function is somewhat unusual. Since rewrites can be applied to any subformula, we can most efficiently handle them by taking them into account during unification. We augment a simple unification algorithm by trying rewrites at each recursive step of the unification. In this way, we avoid applying rewrites until

[Find the set of substitutions under which the given goals can be derived from $known$ using G-rules and rewrites, assuming formulas in $visited$ to be unprovable.

function $backward\_chain(goals, known, visited)$ =
  if $goals = \{\}$ **then**
    return $\{[\,]\}$
  else
    $(g, gs) \leftarrow choose\_goal(goals)$
    return $\{compose(\sigma_2, \sigma_1)$
      $\mid \sigma_1 \in backward\_chain\_one(g, known, visited),$
      $\sigma_2 \in backward\_chain(apply\_subst(\sigma_1, gs), known, visited)\}$

[Find the set of substitutions under which ϕ can be derived from $known$ using G-rules and rewrites, assuming formulas in $visited$ to be unprovable.

function $backward\_chain\_one(\phi, known, visited)$ =
  $\hat{\phi} \leftarrow make\_canonical(\phi)$
  if $\hat{\phi} \in visited$ **then**
    return $\{\}$
  else
    $\sigma_r \leftarrow unique\_renaming(\phi)$
    $\phi_r \leftarrow apply\_subst(\sigma_r, \phi)$
    $regular\_substs \leftarrow \bigcup_{F \in known} unify(\phi_r, F)$
    $grule\_substs \leftarrow \bigcup_{R \in G\_rules} reverse\_apply\_grule(R, \phi_r, known, visited \cup \{\hat{\phi}\})$
    return $\{compose(\sigma, \sigma_r) \mid \sigma \in regular\_substs \cup grule\_substs\}$

[Find the set of substitutions under which ϕ can be derived from $known$ by a proof using G-rules and rewrites, and ending with the given G-rule (R), assuming formulas in $visited$ to be unprovable.

function $reverse\_apply\_grule(R, \phi, known, visited)$ =
  return $\{compose(\sigma_4, \sigma_3)$
    $\mid \sigma_3 \in unify(\phi, conclusion(R)),$
    $\sigma_4 \in backward\_chain(apply\_subst(\sigma_3, premises(R)),$
      $known, visited)$

Figure 2.2: Sketch of the TG_ℓ algorithm, continued (part 2 of 2).

$SG\_restriction\_ok(S, G, R, \preceq)$	Check that the S/G restriction holds for the given rules, rewrites, and pre-order.
$choose\_goal(goals)$	Select a goal to satisfy first, and return that goal and the remaining goals as a pair; prefer goals that match no G-rule conclusions.
$apply\_subst(\sigma, \Phi)$	Replace variables in Φ (a formula or set of formulas), according to the substitution, σ.
$make\_canonical(F)$	Return a canonical representative of the set of formulas equivalent to F modulo rewrites and variable renaming (can also be applied to sets of formulas).
$unify(F, G)$	Return a set containing each most-general substitution, σ, under which the rewrites imply $\sigma F = \sigma G$.
$unique\_renaming(F)$	Return a substitution that replaces each variable occurring in F with a variable that occurs in no other formulas.
$compose(\sigma_1, \sigma_2)$	Return the composition of substitution σ_1 with σ_2.

Figure 2.3: Auxiliary functions used by the TG_ℓ algorithm.

and unless they actually have some effect on the unification process. Plotkin described a similar technique for building equational axioms into the unification process [Plo72]. Because of the rewrites, the unification procedure produces not just zero or one, but potentially many "most general unifiers."

The TG_ℓ algorithm described here can be implemented quite straightforwardly, but various optimizations and specialized data structures can be employed to provide greater efficiency if desired. In Section 6.1, we discuss one implementation and a set of useful optimizations it uses. The remainder of this chapter is devoted to discussion of the correctness and termination properties of the TG_ℓ algorithm, and its use in a decision procedure for ℓ_{RW}.

2.3.3 Correctness

The TG_ℓ algorithm is intended to produce a theory representation in the (R, R') form (see Section 2.2), where the "preferred" rules R' are exactly the S-rules. To prove that it does this, we need a few lemmas describing the behavior of certain components of the TG_ℓ algorithm. These lemmas assume that the various functions always terminate; the termination proofs appear in the next section.

All formulas and proofs referred to below are assumed to be in ℓ_{RW} unless explicitly stated otherwise. In the correctness and termination proofs, we will make use of two restricted forms of proof within ℓ_{RW}. We write,

$$\Gamma \vdash_{\mathrm{W}} \phi$$

if there exists a proof of ϕ from Γ using only rewrites (and instantiation). If there exists a proof of ϕ from Γ using rewrites and G-rules, we write,

$$\Gamma \vdash_{\mathrm{GW}} \phi \ .$$

We present the following claims regarding the unify function without proof, since it is a standard building block and we have not described its implementation in detail.

Claim 2.2 (*unify* soundness) *If ϕ_1 and ϕ_2 are formulas of ℓ_{RW}, and σ is a substitution such that*

$$\sigma \in \mathit{unify}(\phi_1, \phi_2) \ ,$$

then

$$\{\phi_1\} \vdash_W \sigma\phi_2$$

and

$$\{\phi_2\} \vdash_W \sigma\phi_1 \ .$$

Claim 2.3 (*unify* completeness) *Let ϕ_1 and ϕ_2 be formulas of ℓ_{RW} that share no variables, and let σ be a substitution, such that*

$$\{\phi_1\} \vdash_W \sigma\phi_2$$

or

$$\{\phi_2\} \vdash_W \sigma\phi_1 \ .$$

There exists a substitution, σ', such that

$$\sigma' \in \mathit{unify}(\phi_1, \phi_2)$$

and σ is an extension of σ'. (That is, there exists σ'' such that $\sigma = \sigma'' \circ \sigma'$).

This lemma demonstrates the soundness of $backward\_chain$: that each substitution it returns can be applied to the given goals to yield provable formulas.

Lemma 2.4 (*backward_chain* soundness) *Let Φ and V be sets of formulas of ℓ_{RW}, let Γ be a set of formulas of ℓ_{RW}, and let σ be a substitution, such that*

$$\sigma \in backward\_chain(\Phi, \Gamma, V) \ .$$

Then, for every $\phi \in \Phi$,

$$\Gamma \vdash_{\mathit{GW}} \sigma\phi \ .$$

Proof: This proof is by induction on the total number of recursive invocations of $backward\_chain$. We assume that any invocation of $backward\_chain$ that causes fewer than n recursive calls satisfies the lemma, and show that the result holds for n recursive calls as well.

In order for $backward\_chain$ to return σ, it must be the case that either Γ is empty and σ is the identity (in which case the result follows trivially), or $\sigma = \sigma_2 \circ \sigma_1$, where

$$\begin{array}{l}
\Phi = \{g\} \cup gs \\
\sigma_1 \in backward_chain_one(g, known, visited) \\
\sigma_2 \in backward_chain(apply_subst(\sigma_1, gs), known, visited) \ .
\end{array}$$

Examining $backward\_chain\_one$, we see that σ_1 can arise in two ways: from $regular\_substs$ or from $grule\_substs$. We will show that in each case,

$$\Gamma \vdash_{\mathrm{GW}} \sigma_1 g \ .$$

Case 1: σ_1 comes from $regular\_substs$. There must exist a formula $F \in \Gamma$, and a substitution σ_1', such that

$$\begin{array}{l}
\sigma_1 = \sigma_1' \circ \sigma_r \\
\sigma_1' \in unify(\sigma_r g, F)
\end{array}$$

By Claim 2.2 ($unify$ soundness), we have

$$\Gamma \vdash_{\mathrm{GW}} \sigma_1'(\sigma_r g) \ ,$$

so

$$\Gamma \vdash_{\mathrm{GW}} \sigma_1 g$$

and this case is done.

Case 2: σ_1 comes from $grule\_substs$. There must exist a G-rule, R, and a substitution σ_1', such that

$$\begin{aligned}&\sigma_1 = \sigma_1' \circ \sigma_r\\&\sigma_1' \in reverse\_apply\_grule(R, \sigma_r g, \Gamma, \ldots)\end{aligned}$$

(We ignore the $visited$ argument, since it is irrelevant to soundness.) Following into $reverse\_apply\_grule$, we find that

$$\begin{aligned}&\sigma_1' = \sigma_4 \circ \sigma_3\\&\sigma_3 \in unify(\sigma_r g, conclusion(R))\\&\sigma_4 \in backward\_chain(apply\_subst(\sigma_3, premises(R)), \Gamma, \ldots)\end{aligned}$$

Claim 2.2 ($unify$ soundness) implies that

$$\{conclusion(R)\} \vdash_{\mathrm{W}} \sigma_3(\sigma_r g) \ .$$

By the induction assumption, for every P in $premises(R)$,

$$\Gamma \vdash_{\mathrm{GW}} \sigma_4(\sigma_3 P) \ .$$

We can rename variables in these premise-proofs using σ_r, then combine them and add an application of the G-rule, R, with the substitution $\sigma_1' \circ \sigma_r$, to get

$$\Gamma \vdash_{\mathrm{GW}} \sigma_1' \sigma_r g \ .$$

This simplifies to

$$\Gamma \vdash_{\mathrm{GW}} \sigma_1 g \ ,$$

so Case 2 is done.

We now return to $backward\_chain$, armed with the knowledge that $\sigma_1 g$ has a proof from Γ. By adding instantiation steps, we can convert this proof to a proof of $\sigma_2 \sigma_1 g$, so we have

$$\Gamma \vdash_{\mathrm{GW}} \sigma_2(\sigma_1 g)$$

We can apply the induction assumption to the recursive $backward\_chain$ invocation, yielding

$$\Gamma \vdash_{\mathrm{GW}} \sigma_2(\sigma_1 G)$$

for every $G \in gs$. Since $\Phi = \{g\} \cup gs$, and $\sigma = \sigma_2 \circ \sigma_1$, we have shown that for every $\phi \in \Phi$, there exists a proof of

$$\Gamma \vdash_{\mathrm{GW}} \sigma\phi \ ,$$

using no S-rules. ■

Now we show the dual of Lemma 2.4, the completeness of $backward\_chain$; that is, that it returns every most-general substitution under which the goals are provable.

Lemma 2.5 (*$backward\_chain$* completeness) *Let Γ and V be sets of formulas, let Φ be a set of formulas ($\{\phi_1, \ldots, \phi_n\}$), let σ be a substitution, and let $\mathcal{P}_1 \ldots \mathcal{P}_n$ be proofs (using no S-rules), such that for $1 \leq i \leq n$,*

$$\Gamma \overset{\mathcal{P}_i}{\vdash}_{GW} \sigma\phi_i$$

and the proofs, $\mathcal{P}_i$, contain no rewrites of formulas in V. Then, if $backward\_chain$ terminates, there exists a substitution, σ', such that

$$\sigma' \in backward\_chain(\Phi, \Gamma, V)$$

and σ is an extension of σ'.

Proof: We prove this lemma by induction on the total number of G-rule applications in $\mathcal{P}_1 \ldots \mathcal{P}_n$.

Without loss of generality, we assume that the proofs, $\mathcal{P}_1 \ldots \mathcal{P}_n$, have no "sharing." That is, for any proof line that is used as a premise more than once, we duplicate the proof prefix ending with that line to eliminate the sharing. To carry out the induction, we now assume that the lemma holds when $\mathcal{P}_1 \ldots \mathcal{P}_n$ contain a total of fewer than n G-rule applications, and show that it holds when there are n G-rule applications.

In the case where Γ is empty, the lemma holds trivially, so assume Γ is non-empty. Let ϕ_i be the first goal selected by $choose\_goal$. There are two cases to consider, depending on whether $\mathcal{P}_i$ contains any G-rule applications.

Case 1: $\mathcal{P}_i$ contains no G-rule applications.

Looking at the call to $backward\_chain\_one$, we can see that the $\hat{\phi} \in visited$ check will not be triggered since ϕ_i must be in the proof $\mathcal{P}_i$, and thus no rewrite of it can appear in V. Since σ_r is just a renaming, there exists some σ' such that

$$\sigma = \sigma' \circ \sigma_r \ .$$

Thus,

$$\Gamma \vdash_{\mathrm{GW}} \sigma'\sigma_r\phi_i$$

and Claim 2.3 implies there exists some σ'' for which

$$\sigma'' \in \bigcup_{F \in known} unify(\phi_r, F)$$

and σ' is an extension of σ''. The set returned by $backward\_chain\_one$ will thus include $\sigma'' \circ \sigma_r$, of which $\sigma' \circ \sigma_r$ (that is, σ) is an extension. Therefore, $backward\_chain$ will return as one of its results, $\sigma_2 \circ \sigma_1$, where σ is an extension of σ_1, and

$$\sigma_2 \in backward\_chain(apply\_subst(\sigma_1, gs), \Gamma, V) \ .$$

We have thus reduced this case to a case with the same total number of G-rule applications and one fewer formula in Φ, so without loss of generality we can assume the second case.

Case 2: $\mathcal{P}_i$ contains at least one G-rule application.

The recursive call to $backward\_chain$ passes a (partially instantiated) subset of Φ (all but ϕ_i); since $\mathcal{P}_i$ contains some G-rule applications, the proofs for this subset must contain fewer G-rule applications than $\mathcal{P}_1 \ldots \mathcal{P}_n$, so we can apply the induction hypothesis. This implies that, as long as σ is an extension of some σ_1, the lemma holds. It remains only to demonstrate this.

In $backward\_chain\_one$, again, the $\hat{\phi} \in visited$ check will not be triggered for the same reason as in Case 1. As in Case 1, there exists some σ' such that

$$\sigma = \sigma' \circ \sigma_r \ .$$

Let R be the last G-rule applied in the proof, $\mathcal{P}_i$. If we can prove that

$$reverse\_apply\_grule(R, \phi_r, \Gamma, V \cup \{\hat{\phi}\})$$

returns some σ'' of which σ' is an extension, then by the argument in Case 1, $backward\_chain\_one$ will return a substitution of which σ is an instance, so the lemma will hold.

Since σ' is just a variable-renaming followed by σ, we can transform the proof of $\sigma\phi_i$, $\mathcal{P}_i$, into a proof of $\sigma'\phi_i$, called $\mathcal{P}'_i$, by simple renaming. From $\mathcal{P}'_i$, we can extract a proof of each premise of its last G-rule application, and also a proof of $\sigma'\phi_i$ from the G-rule conclusion. By Claim 2.3, σ' is an extension of some σ_3 that will be returned by $unify(\sigma_r\phi_i, conclusion(R))$. The proofs of R's premises will not contain any rewrites of V, since these proofs come from $\mathcal{P}'_i$, and we will further assume they contain no rewrites of ϕ_i. (If they did, the application of R could be eliminated from $\mathcal{P}_i$, so there is no loss of generality from this assumption.) Since we have proofs of R's premises (instantiated by σ'), which have fewer total G-rule applications than the original $\mathcal{P}_i$, and since these proofs contain no rewrites of formulas in the (expanded) $\mathit{visited}$ set, we can apply the induction hypothesis and find that for some σ_4 returned by the $\mathit{backward\_chain}$ call, σ' is an extension of $\sigma_4 \circ \sigma_3$, which will be returned by $\mathit{reverse\_apply\_grule}$. This is the final result we required to complete the proof of the lemma. ■

Now we can prove the soundness and completeness of the $\mathit{closure}$ function, the heart of the TG_ℓ algorithm.

Lemma 2.6 (*closure* soundness) *Let Γ and Γ' be sets of formulas of ℓ_{RW}. For any formula, F, where*

$$F \in \mathit{closure}(\Gamma', \Gamma) \ ,$$

there exists a proof, $\mathcal{P}$, of $\Gamma \cup \Gamma' \vdash F$, in which the last rule application (if any) is of an S-rule.

Proof: The proof is by induction on the number of recursive calls to $\mathit{closure}$. If there are no such calls, Γ' (the fringe) must be empty, so Γ is returned, and the lemma is trivially satisfied. Otherwise, $\mathit{closure}$ is called recursively with the formulas in fringe added to Γ and fringe' becomes the new fringe. If we can show that all of the fringe' formulas have proofs from Γ of the appropriate form, then we can apply the induction assumption and the lemma is proved.

For every S-rule, R, $\mathit{closure}$ calls $\mathit{apply\_srule}(R, \Gamma \cup \Gamma')$, which will return R's conclusion, instantiated by σ, where σ comes from the $\mathit{backward\_chain}$ result. By Lemma 2.4 ($\mathit{backward\_chain}$ soundness), for each premise, P, of R, there exists a proof of

$$\Gamma \cup \Gamma' \vdash \sigma P$$

We can concatenate these proofs, followed by an application of the S-rule, R, to yield a proof of σC (where C is R's conclusion). Furthermore, this proof has no rule applications following the application of R, so the proof is of the appropriate form. It is therefore safe to add

$$\mathit{apply\_subst}(\sigma, \mathit{conclusion}(R))$$

to the fringe. ■

To express the next lemma, we introduce a notion of *partial theory representations*:

Definition 2.17 *If, for any formula, ϕ, and proof, $\mathcal{P}$, such that $\mathcal{P}$ has only one S-rule application and no other rule applications following it, and where*

$$\Gamma \overset{\mathcal{P}}{\vdash} \phi \ ,$$

it is also the case that

$$(\Gamma' \cup \Gamma) \vdash_w \phi \ ,$$

then we call the ordered pair, $\langle \Gamma', \Gamma \rangle$, a partial theory representation.

Note that if T is a theory representation, then $\langle \{\}, T \rangle$ is a partial theory representation. The *closure* function takes a partial theory representation and produces a theory representation:

Lemma 2.7 (*closure* completeness) *Let T and fringe be sets of formulas (of ℓ_{RW}), such that $\langle \mathit{fringe}, T \rangle$ is a partial theory representation. For any formula, ϕ, and proof, $\mathcal{P}$, whose last rule application is an S-rule application, where*

$$(T \cup \mathit{fringe}) \overset{\mathcal{P}}{\vdash} \phi \ ,$$

there exists some ϕ', where (assuming $\mathit{closure}$ terminates)

$$\phi' \in \mathit{closure}(\mathit{fringe}, T) \ ,$$

such that

$$\phi' \vdash_w \phi \ .$$

<u>Proof</u>: The proof is by induction on the number of recursive calls to $\mathit{closure}$, which is guaranteed to be finite since we assume $\mathit{closure}$ terminates.

If there are no recursive calls to $\mathit{closure}$, then fringe is empty, and $\mathit{closure}$ returns just T, which by Definition 2.17 must satisfy

$$T \vdash_{\mathrm{w}} \phi \ ,$$

so this case is done.

If fringe is non-empty, then the function returns the result of

$$\mathit{closure}(\mathit{fringe}', T') \ .$$

Since

$$T' \supset (T \cup \mathit{fringe}) \ ,$$

the induction hypothesis will yield the desired result if we can prove that

$$\langle fringe', T' \rangle$$

is a partial theory representation.

Let $\phi^\star$ be a formula as given in Definition 2.17, where

$$T' \overset{\mathcal{P}^\star}{\vdash} \phi^\star \ ,$$

and $\mathcal{P}^\star$ has exactly one S-rule application, which is its last rule application. Let R be the last S-rule applied in $\mathcal{P}^\star$, let C be R's conclusion, and let σ be the substitution under which R was applied. From Lemma 2.5 ($backward\_chain$ completeness), it follows that σ is an extension of some substitution returned by

$$backward\_chain(premises(R), T', \{\}) \ ,$$

and thus

$$apply\_srule(R, T')$$

will return some formula of which σC is an instance. Since $make\_canonical$ only transforms one formula into another that is equivalent modulo rewrites, we get

$$make\_canonical(apply\_srule(R, T')) \vdash_{\mathrm{w}} \sigma C$$

and finally, this implies that

$$fringe' \cup T' \vdash_{\mathrm{w}} \sigma C \ . \tag{2.4}$$

Now, returning to the proof, $\mathcal{P}^\star$, since no more rules are applied after R, it follows that

$$\sigma C \vdash_{\mathrm{w}} \phi \ .$$

This, combined with (2.4), gives the result we need:

$$fringe' \cup T' \vdash_{\mathrm{w}} \phi \ .$$

This completes the induction. ■

We can now prove the claim made at the beginning of this section, which corresponds to the following theorem:

Theorem 2.8 (TG_ℓ **Correctness)** *If* Γ *is a set of mostly-ground formulas of* ℓ_{RW}*, and* S_rules*,* G_rules*,* $Rewrites$*, and* $\preceq$ *meet the* TG_ℓ *algorithm preconditions given in Definition 2.16, then*

$$theory\_gen(\Gamma, S\_rules, G\_rules, Rewrites, \preceq)$$

returns an (R, R') *representation of the theory induced by* Γ*, where* R' *is the set of S-rules, and the equivalence used is equivalence modulo rewrites and variable renaming.*

Proof: First we prove that every formula returned by $theory\_gen$ is in the (R, R') representation. Let F be a formula returned by $theory\_gen$. It must be the case that

$$F \in closure(make\_canonical(\Gamma), \{\}) ,$$

and so by Lemma 2.6 ($closure$ soundness), there exists a proof, $\mathcal{P}$, such that

$$make\_canonical(\Gamma) \overset{\mathcal{P}}{\vdash} F ,$$

and where the last rule application in $\mathcal{P}$ is of an S-rule. Since $make\_canonical$ only transforms by rewrites, we know that

$$make\_canonical(\Gamma) \overset{\mathcal{P}}{\vdash} F \implies \Gamma \overset{\mathcal{P}'}{\vdash} F$$

and furthermore, that since the last rule applied in $\mathcal{P}$ is an S-rule, the same is true of $\mathcal{P}'$. Lastly, since every formula returned by $closure$ has been canonicalized, no two formulas returned by $theory\_gen$ are equivalent modulo rewrites and variable renamings. Therefore, every formula returned by $theory\_gen$ is in the (R, R') representation.

It remains to prove that any formula in the (R, R') representation is returned by $theory\_gen$. Let F be a formula and $\mathcal{P}$ a proof whose last rule applied is an S-rule, such that

$$\Gamma \overset{\mathcal{P}}{\vdash} F .$$

By the same argument made above, it follows that

$$make\_canonical(\Gamma) \overset{\mathcal{P}'}{\vdash} F ,$$

where $\mathcal{P}'$ has the same property. By Lemma 2.7 ($closure$ completeness), there exists some F',

$$\phi' \in closure(make\_canonical(\Gamma), \{\}) ,$$

such that

$$F' \vdash_{\mathrm{w}} F \ .$$

Therefore, F is equivalent (modulo rewrites and variable renaming) to some formula in the set returned by $\mathit{theory\_gen}$. This completes the correctness proof for the TG_ℓ algorithm. ■

2.3.4 Termination

The completeness proofs in Section 2.3.3 assumed that the functions making up the TG_ℓ algorithm always terminated. In this section, we show how the TG_ℓ preconditions ensure this. The proofs below assume the existence of a fixed set of S-rules, G-rules, and rewrites, and a pre-order, $\preceq$, all of which satisfy the TG_ℓ preconditions in Definition 2.16.

Roughly speaking, the proof goes as follows. The $\mathit{backward\_chain}$ function first satisfies the primary premises, and then applies G-rules in reverse to satisfy the partially instantiated side-conditions. Since the G-rules "grow" when applied in the forward direction, they "shrink" when applied in reverse (if the goal formula is sufficiently ground), so this backward chaining must terminate. The $\mathit{closure}$ function finds each way of applying the S-rules with help from $\mathit{backward\_chain}$, and repeats until it reaches a fixed point. Since the S-rules "shrink," they can never produce a formula larger than all the initial assumptions, and so this process will halt as well.

A more formal proof follows.

Definition 2.18 *A formula, F, is* size-bounded *by a finite set of formulas, Γ, when, for any substitution, σ, there exists $G \in \Gamma$ such that*

$$\sigma F \preceq G \ .$$

Note that if F is mostly-ground then F is size-bounded by $\{F\}$.

Lemma 2.9 *If ϕ_1 is size-bounded by Γ, and $\phi_2 \preceq \phi_1$, then ϕ_2 is size-bounded by Γ.*

Proof: Since $\phi_2 \preceq \phi_1$, we can use pre-order condition P2 ($\preceq$ preserved under substitution) to show that, for any σ,

$$\sigma\phi_2 \preceq \sigma\phi_1 \ .$$

Since ϕ_1 is size-bounded by Γ, there exists some $G \in \Gamma$ such that

$$\sigma\phi_1 \preceq G \ .$$

Applying the transitive property of pre-orders, we get

$$\sigma\phi_2 \preceq G \ ,$$

so ϕ_2 is size-bounded by Γ. ■

Lemma 2.10 *If the formula, F, is size-bounded by Γ, then for any formula, F', such that*

$$\{F\} \vdash_w F' \ ,$$

F' is also size-bounded by Γ.

Proof: We show that size-boundedness is preserved by instantiation and by rewriting, the only transformations possible in the proof of

$$\{F\} \vdash_{\mathrm{w}} F' \ .$$

Let σ be some substitution. Since F is size-bounded by Γ, there exists some $G \in \Gamma$ such that for any substitution, σ',

$$\sigma'\sigma F \preceq G \ ,$$

so σF is size-bounded by Γ. Let $S = T$ be a rewrite, and let F' be the result of applying that rewrite to F. Rewrites are required to be size-preserving, so $S \preceq T$ and $T \preceq S$. By pre-order condition P1, this implies $F' \preceq F$, and we can apply Lemma 2.9 to see that F' is size-bounded by Γ. ■

Lemma 2.11 *If F is size-bounded by Γ, then $make\_canonical(F)$ is size-bounded by Γ.*

Proof: Since $make\_canonical$ only transforms by rewrites and variable renaming, it follows directly from Lemma 2.10 that $make\_canonical(F)$ is size-bounded by Γ if F is. ■

Lemma 2.12 *For any finite set of formulas, Γ, there are finitely many formulas that are both size-bounded by Γ and canonical with respect to variable-renaming.*

Proof: By Definition 2.18, any formula, ϕ, that is size-bounded by Γ must satisfy $\phi \preceq G$ for some $G \in \Gamma$. By pre-order condition P3, since Γ is finite, there are finitely many such formulas, ϕ, modulo variable renaming.

Lemma 2.13 *If ϕ is size-bounded by the set of formulas, Γ', and R is a G-rule, then*

$$\mathit{reverse\_apply\_grule}(R, \phi, \Gamma, V)$$

will always pass a set of formulas, Φ, to $\mathit{backward\_chain}$, where all formulas in Φ are size-bounded by Γ'. (Γ need have no relation to Γ'; in particular Γ' may contain larger formulas than Γ.)

Proof: Let C be R's conclusion. For any σ_3, such that

$$\sigma_3 \in \mathit{unify}(\phi, C) ,$$

Claim 2.2 (unify soundness) implies that

$$\{\phi\} \vdash_{\mathrm{w}} \sigma_3 C .$$

Since ϕ is size-bounded by Γ', it follows that $\sigma_3 C$ is also size-bounded by Γ'. From the G-rule definition, for all premises, P, of R,

$$P \preceq C ,$$

and so

$$\sigma_3 P \preceq \sigma_3 C .$$

By Lemma 2.9, $\sigma_3 P$ must be size-bounded by Γ'. The call to $\mathit{backward\_chain}$ sends only formulas of this form. ∎

Lemma 2.14 *If all formulas in Γ are size-bounded by Γ', and ϕ matches no G-rule conclusion, then for every σ such that*

$$\sigma \in \mathit{backward\_chain\_one}(\phi, \Gamma, V) ,$$

$\sigma\phi$ is size-bounded by Γ'.

Proof: Since ϕ matches no G-rule conclusion, neither will the renamed version, ϕ_r, and so $\mathit{grule\_substs}$ will be empty. By Claim 2.2 (unify soundness), for every σ' such that

$$\sigma' \in \mathit{unify}(\phi_r, F)$$

where $F \in \Gamma$, we know that
$$\{F\} \vdash_{\mathrm{w}} \sigma'\phi_r \ .$$

Therefore, by Lemma 2.10, since Γ is size-bounded by Γ', so is $\sigma'\phi_r$. Finally, the substitutions returned by $\mathit{backward\_chain\_one}$ are $\sigma' \circ \sigma_r$, and

$$\sigma'\phi_r = \sigma'\sigma_r\phi \ ,$$

so $(\sigma' \circ \sigma_r)\phi$ is size-bounded by Γ'. ■

Lemma 2.15 *If* Γ *is size-bounded by* Γ'*, and there exists some* $\phi \in \Phi$ *such that* ϕ *matches no G-rule conclusion, then for every* σ *such that*

$$\sigma \in \mathit{backward\_chain}(\Phi, \Gamma, V) \ ,$$

$\sigma\phi$ *is size-bounded by* Γ'.

Proof: Each recursive call to $\mathit{backward\_chain}$ applies another substitution to the remaining goals, and substitution cannot cause a formula to match a G-rule conclusion if it did not already, and it also preserves size-boundedness. Therefore, in some call to $\mathit{backward\_chain}$, the chosen goal, g, will match no G-rule conclusions and will be size-bounded by Γ'. By Lemma 2.14, every σ_1 substitution such that
$$\sigma_1 \in \mathit{backward\_chain\_one}(g, \Gamma, V)$$
will give $\sigma_1 g$ size-bounded by Γ'. The substitutions returned by the original $\mathit{backward\_chain}$ invocation are extensions of these σ_1 substitutions, so $\sigma\phi$ will be size-bounded by Γ'. ■

Lemma 2.16 *If* ϕ *is size-bounded by* Γ'*, then* $\mathit{backward\_chain\_one}(\phi, \Gamma, V)$ *will always pass the formula* ϕ_r *to* $\mathit{reverse\_apply\_grule}$*, where* ϕ_r *is also size-bounded by* Γ'.

Proof: This follows directly from the definition of size-bounded, since ϕ_r is the result of a substitution applied to ϕ. ■

Lemma 2.17 *If* Γ *is size-bounded by* Γ'*, and* Φ *is the set of premises of some S-rule, then* $\mathit{backward\_chain}(\Phi, \Gamma, V)$ *will terminate.*

Proof: Since $choose\_goal$ always selects goals that match no G-rule conclusions first, $backward\_chain$ will satisfy all the primary premises in Φ before examining side-conditions. For each primary premise, $backward\_chain\_one$ will clearly terminate since $reverse\_apply\_grule$ will not call $backward\_chain$.

Let σ_P be the accumulated substitution once the primary premises have been satisfied. Since σC is size-bounded by Γ', and each side-condition, P, satisfies

$$P \preceq C ,$$

it follows from Lemma 2.9 that for each side-condition, σP is size-bounded by Γ'. A simple induction shows that the recursive call to $backward\_chain$ in $backward\_chain$ itself will preserve this property, and reduces the number of goals by one, so the only possibly non-terminating call is the one to $backward\_chain\_one$.

The call to $backward\_chain\_one$ passes a formula, ϕ, that is size-bounded by Γ'. By Lemmas 2.13 and 2.16, any recursive call made to $backward\_chain$ in $reverse\_apply\_grule$ will use goals also size-bounded by Γ'. Since $backward\_chain\_one$ adds the canonical form of ϕ to the visited set, and terminates if ϕ is already in that set, the recursive nesting depth is bounded by the number of such formulas ϕ that are distinct modulo renaming. Since each such ϕ is size-bounded by Γ, pre-order condition P3 implies that there are a finite number of possible ϕ's. Therefore, this recursion must halt, so $backward\_chain$ will terminate. ■

Lemma 2.18 *If the formulas in Γ are size-bounded by Γ', and R is an S-rule, then the formulas returned by $apply\_srule(R, \Gamma)$ are size-bounded by Γ'.*

Proof: Let C be the conclusion of the S-rule, R, and let P be a primary premise of R that satisfies

$$C \preceq P .$$

By the S-rule definition, some such P must exist, and by the S/G restriction, P must match no G-rule conclusions. Lemma 2.15 thus implies that for every substitution, σ, such that

$$\sigma \in backward\_chain(premises(R), \Gamma, \{\}) ,$$

σP is size-bounded by Γ'. By the S-rule definition,

$$C \preceq P ,$$

so by pre-order condition P2,

$$\sigma C \preceq \sigma P \ ,$$

and since σP is size-bounded by Γ', Lemma 2.9 implies that σC is size-bounded by Γ'. ■

Lemma 2.19 *If formulas in* Γ *are size-bounded by* Γ'*, and* R *is an S-rule, then* $apply\_srule(R, \Gamma)$ *will terminate.*

Proof: The call to $backward\_chain$ passes the premises of R and the set Γ, so the antecedent of Lemma 2.17 is satisfied, and $backward\_chain$ (and thus $apply\_srule$) will terminate. ■

Lemma 2.20 *In* $closure$*, if the formulas in* $fringe$ *and* T *are size-bounded by* Γ*, then formulas in* $fringe'$ *and* T' *are also size-bounded by* Γ.

Proof: First, T' is clearly size-bounded by Γ since T' is just $fringe \bigcup T$. By Lemma 2.11 and Lemma 2.18, for each S-rule, R,

$$make\_canonical(apply\_srule(R, T'))$$

is size-bounded by Γ, and so $fringe'$ is also size-bounded by Γ. ■

Lemma 2.21 *If formulas in* $fringe$ *and* T *are size-bounded by some finite set,* Γ*, and canonical with respect to rewrites and variable renaming, then* $closure(fringe, T)$ *will terminate.*

Proof: In each recursive call, the set $fringe \bigcup T$ must grow monotonically, until $fringe$ is empty and $closure$ terminates. By Lemma 2.19, $apply\_srule(R, T')$ must terminate

By Lemma 2.20, and the definition of $make\_canonical$, these invariants are preserved:

- Formulas in $fringe \bigcup T$ are size-bounded by Γ.
- Formulas in $fringe \bigcup T$ are canonical with respect to rewrites

Therefore, by Lemma 2.12, there are finitely many formulas that can ever be added to $fringe \bigcup T$, and so the recursion must terminate. ■

We can now prove the termination theorem for the TG_ℓ algorithm:

Theorem 2.22 *If the* TG_ℓ *preconditions hold, then* $theory\_gen$ *will terminate.*

Proof: Assumptions are mostly-ground, so they are size-bounded by themselves. By Lemma 2.11, the formulas passed to $closure$ are size-bounded by Γ, and they are also canonical with respect to rewrites and variable renaming, so Lemma 2.21 implies that $closure$, and thus $theory\_gen$, will terminate. ■

2.4 The Decision Problem

One of the goals (Section 2.2) for the theory representation was that it would enable an efficient decision procedure for the full theory. In other words, we should be able to make use of the theory representation generated from Γ to answer the question,

$$\Gamma \overset{?}{\vdash} \phi \tag{2.5}$$

when Γ is a set of mostly-ground formulas, and ℓ_{RW} satisfies the TG_ℓ preconditions.

In this section, we first discuss the decidability of the ℓ_{RW} logics in general, and then give a simple and fast decision procedure that makes use of the generated theory representation.

2.4.1 Decidability of ℓ_{RW}

Since logical implication for full first-order logic is undecidable, we might well ask whether (2.5) can be decided in ℓ_{RW}.

The $\Gamma \overset{?}{\vdash} \phi$ question is equivalent to deciding the logical validity of this (general first-order) formula:

$$(A_1 \wedge \ldots \wedge A_n) \wedge (R_1 \wedge \ldots \wedge R_n) \wedge (W_1 \wedge \ldots \wedge R_n) \Rightarrow \phi \tag{2.6}$$

where $\Gamma = \{A_1, \ldots, A_n\}$, and the R_i and W_i are the rules and rewrites of ℓ_{RW}. Each of these formulas (A_i, R_i, W_i, ϕ) is a well-formed formula of $\mathcal{L}^=$, so recalling the restriction on formulas in $\mathcal{L}^=$, we know that they each have the form $\forall X_1, \ldots, X_m.F$, where F contains no quantifiers. We can pull the quantifiers in (2.6) outward, renaming bound variables to avoid clashes, to produce this equivalent formula:

$$(\forall \overline{X}.(F_1 \wedge F_2 \wedge \ldots \wedge F_n)) \Rightarrow (\forall \overline{Y}.G) \tag{2.7}$$

This formula, in turn, can be rewritten as

$$\forall \overline{Y}.\exists \overline{X}.((F_1 \wedge F_2 \wedge \ldots \wedge F_n) \Rightarrow G)$$

Since the F_i and G are quantifier-free, this formula has the form

$$\forall Y_1. \ldots \forall Y_n.\exists X_1. \ldots \exists X_m.S$$

Validity is known to be decidable for first-order formulas in this form [DG79], and thus the logical implication question stated above is decidable.

2.4.2 Decision Procedure

Given a generated theory representation for Γ, and a mostly-ground formula ϕ, the procedure for deciding $\Gamma \overset{?}{\vdash} \phi$ is simply this:

function $derivable(\phi, theory\_rep) =$
  return $backward\_chain(\{\phi\}, theory\_rep, \{\}) \neq \{\}$

The correctness and termination of this decision procedure follow directly from the correctness and termination of the $backward\_chain$ function (Lemmas 2.4, 2.5, and 2.17). It is efficient in practice.

2.5 Theory Generation: Summary

We have defined the general theory generation approach: build a representation of the full theory induced by some set of assumptions, and use that representation to directly and indirectly explore the consequences of those assumptions. We then considered theory generation in the context of a particular fragment of first-order logic, ℓ_{RW}. We specified the (R, R') class of theory representations, based on selecting a set (R') of "preferred" rules, to be applied aggressively. For assumptions, rules, and rewrites satisfying the specified preconditions, we described a theory generation algorithm that produces a representation in this class. Finally, we presented a decision procedure for mostly-ground formulas in ℓ_{RW} which makes use of the theory representation. The decision procedure (*derivable*) satisfies the following property, where $\mathrm{preconds}$ refers to the TG_ℓ preconditions given in Definition 2.16:

$$\begin{aligned}&\mathrm{preconds}(T^0, S\_rules, G\_rules, Rewrites, \preceq)\\ &\qquad\qquad \Longrightarrow (\phi \in T^* \Longleftrightarrow derivable(\phi, theory\_gen(T^0)))\end{aligned}$$

That is, for a logic and initial set of formulas, T^0, that satisfy the TG_ℓ preconditions, a formula, ϕ, is in the theory induced by T^0 if and only if the decision procedure returns true, given ϕ and the results of the TG_ℓ algorithm.

In the following chapters, we apply theory generation in the domain of cryptographic protocol verification, and see what practical benefits it offers.

Chapter 3

Theory Generation for Belief Logics

"Little logics" have been used successfully to describe, analyze, and find flaws in cryptographic protocols. The BAN logic is the best-known member of a family of logics that seem to capture some important features of these protocols while maintaining a manageable level of abstraction. There has been little in the way of tools for automated reasoning with these logics, however. Some of the BAN analyses were mechanically verified, and the designers of AUTLOG produced a prover for their logic, but prominent automated tools, such as the NRL Protocol Analyzer and Paulson's Isabelle work, have used very different approaches. The lack of emphasis on automation for these logics results in part from their apparent simplicity; it can be argued that proofs are easily carried out by hand. Indeed, the proofs rarely require significant ingenuity once appropriate premises have been established, but manual proofs even in published work often miss significant details and assume preconditions or rules of inference that are not made explicit; automated verification keeps us honest. Furthermore, with fast, automated reasoning we can perform some analyses that would otherwise be impractical or cumbersome, such as enumerating beliefs held as the protocol progresses.

The development of theory generation was partially motivated by the need for automated reasoning with this family of logics. Using theory generation, and the TG_ℓ algorithm in particular, we can do automated reasoning for all these logics with a single, simple tool.

In this chapter, we examine three belief logics in the BAN family: the BAN logic of authentication, AUTLOG, and Kailar's logic of accountability, and we show how theory generation can be applied to each of them. For each logic, we present its representation as a set of functions and predicates, its axioms (the "rules" fed to the TG_ℓ algorithm), and an appropriate pre-order ($\preceq$).

3.1 BAN

As the progenitor of this family, the BAN logic of authentication is a natural case to consider. 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. In Section 3.1.1 we enumerate the fundamental concepts expressible in the BAN logic: belief, trust, message freshness, and message receipt and transmission; Section 3.1.2 contains the rules of inference; Sections 3.1.3–3.1.5 give examples of their application.

3.1.1 Components of the Logic

In encoding the BAN logic and its accompanying sample protocols, we must make several adjustments and additions to the logic as originally presented [BAN90], to account for rules, assumptions, and relationships that are missing or implicit.

Figure 3.1 shows the functions used in the encoding, and their intuitive meanings. The first twelve correspond directly to constructs in the original logic, and have clear interpretations. The last two are new: inv makes explicit the relationship implied between the keys in a key pair (K, K^{-1}) under public-key (asymmetric) cryptography, and $distinct$ expresses that two principals are not the same.

As a technical convenience, we always assume that for every function (e.g., $believes$), there is a corresponding predicate by the same name and with the same arity, which is used when the operator occurs at the outermost level. For instance, in the BAN formula

$$A \textbf{ believes } B \textbf{ said } A \textbf{ believes } A \stackrel{K}{\longleftrightarrow} B$$

the first **believes** is represented by the $believes$ predicate, while the second is represented by the $believes$ function. The function and predicate have the same name merely for convenience; there is formally no special relationship between the two. The duplication is required since we have chosen a first-order logic setting; if instead we used modal logic, we could replace the function/predicate pair with a single modal operator. This is a technical distinction with no significant implications in practice.

We provide a finite but unspecified set of uninterpreted 0-ary functions (constants), which can be used to represent principals, keys, timestamps, and so forth in a specific protocol description.

Function	BAN notation	Meaning
$believes(P, X)$	P **believes** X	P believes statement X
$sees(P, X)$	P **sees** X	P sees message X
$said(P, X)$	P **said** X	P said message X
$controls(P, X)$	P **controls** X	if P claims X, X can be believed
$fresh(X)$	**fresh**(X)	X has not been uttered before this protocol run
$shared\_key(K, P, Q)$	$P \overset{K}{\leftrightarrow} Q$	K is a symmetric key shared by P and Q
$public\_key(K, P)$	$\overset{K}{\mapsto} P$	K is P's public key
$secret(Y, P, Q)$	$P \overset{Y}{\rightleftharpoons} Q$	Y is a secret shared by P and Q
$encrypt(X, K, P)$	$\{X\}_K$ *from* P	message X, encrypted under key K by P
$combine(X, Y)$	$\langle X \rangle_Y$	message X combined with secret Y
$comma(X, Y)$	X, Y	concatenation
$A, B, S, T, \ldots$	$A, B, S, T, \ldots$	0-ary functions (constants)
$inv(K_1, K_2)$		K_1 and K_2 are a public/private key pair
$distinct(P, Q)$		principals P and Q are not the same

Figure 3.1: BAN functions

In order to apply the TG_ℓ algorithm to reason with the BAN logic, we must define a pre-order, $\preceq_{BAN}$, on terms and formulas. Before we can do that, however, we must place some constraints on the terms and formulas in the BAN encoding. Specifically, we require that function and predicate arguments corresponding to principal names must be simple variables or constants.

Definition 3.1 *The* atomic arguments *of BAN functions and predicates are those in positions indicated by P and Q in Figure 3.1.*

Definition 3.2 *The terms and formulas of BAN are all those terms and formulas* *in ℓ_{RW} built from the functions and predicates in Figure 3.1, and in which all atomic arguments are either variables or 0-ary functions.*

With this constraint, we can still express all BAN formulas. This constraint, combined with the following pre-order, enables us to apply the TG_ℓ algorithm.

Definition 3.3 *The pre-order $\preceq_{BAN}$ is defined over BAN terms and formulas as follows:*

$$\begin{array}{rcl}
F \preceq_{BAN} G & \equiv & (nsyms(F) \leq nsyms(G)) \\
& & \wedge\ (\forall v.occ(v, F) \leq occ(v, G)) \\
nsyms(F) & \equiv & \textit{the number of functions, predicates, and variables} \\
& & \textit{in } F\textit{, excluding those in atomic arguments} \\
occ(v, F) & \equiv & \textit{the number of occurrences of variable } v \textit{ in } F\textit{, ex-} \\
& & \textit{cluding occurrences in atomic arguments}
\end{array}$$

For this relation to be acceptable for use with the TG_ℓ algorithm, it must be a pre-order and also satisfy conditions P1–P3 (Section 2.3).

Claim 3.1 *The relation $\preceq_{BAN}$ is a pre-order and satisfies conditions P1–P3.*

Proof: The relation is clearly reflexive and transitive, so it is a pre-order.

When we substitute a term, T, for all occurrences of a variable, X, in F, we can compute $nsyms$ and occ exactly for the new formula as follows:

$$nsyms([T\backslash X]F) = nsyms(F) + occ(X, F) \cdot nsyms(T) \tag{3.1}$$

$$occ(v, [T\backslash X]F) = \begin{cases} occ(X, F) \cdot occ(v, T) & \text{if } v = X \\ occ(v, F) + occ(X, F) \cdot occ(v, T) & \text{otherwise} \end{cases} \tag{3.2}$$

It then follows that, for any terms T_1 and T_2,

$$nsyms(T_1) \leq nsyms(T_2) \Rightarrow (nsyms([T_1\backslash X]F) \leq nsyms([T_2\backslash X]F))$$

and

$$occ(v, T_1) \leq occ(v, T_2) \Rightarrow (occ(v, [T_1\backslash X]F) \leq occ(v, [T_2\backslash X]F))$$

so condition P1 is satisfied by $\preceq_{BAN}$:

$$(T_1 \preceq_{BAN} T_2) \Rightarrow ([T_1\backslash X]F \preceq_{BAN} [T_2\backslash X]F)$$

From (3.1) we can further derive that for any formulas, F and G, and any term, T,

$$\begin{gathered}((nsyms(F) \leq nsyms(G)) \wedge (occ(X,F) \leq occ(X,G))) \\ \Rightarrow (nsyms([T\backslash X]F) \leq nsyms([T\backslash X]G))\end{gathered}$$

and from (3.2),

$$\begin{gathered}((occ(v,F) \leq occ(v,G)) \wedge (occ(X,F) \leq occ(X,G))) \\ \Rightarrow (occ(v,[T\backslash X]F) \leq occ(v,[T\backslash X]G))\end{gathered}$$

so condition P2 is satisfied by $\preceq_{BAN}$:

$$(F \preceq_{BAN} G) \Rightarrow ([T\backslash X]F \preceq_{BAN} [T\backslash X]G)$$

Finally, we must show that for any G, the set $\{F \mid F \preceq_{BAN} G\}$ is finite modulo variable renaming (P3). The non-atomic arguments in each such F must contain only variables appearing in G (since $\forall v. occ(v,F) \leq occ(v,G)$), and there are finitely many functions and predicates that can be used to construct F, each having a fixed arity. The atomic arguments in F must each be either a single variable or a member of the finite collection of constants. The single variables are either one of the finite set of variables in G, or do not appear outside atomic arguments and can thus be canonically renamed. Therefore both the length of F and the alphabet it is built from are bounded, so the set of all such Fs is finite. ■

3.1.2 Rules of Inference

The BAN logic contains eleven basic rules of inference, each of which can be expressed as an ℓ_{RW} rule, written in the form

$$\frac{P_1, P_2, \ldots, P_m}{C}$$

Each of these rules is either an S-rule or a G-rule under the pre-order $\preceq_{BAN}$, and each preserves the atomic-arguments constraint on BAN formulas.

There are three message-meaning rules that allow one principal to deduce that a given message was once uttered by some other principal:

$$\textbf{BAN1}: \frac{\begin{gathered}believes(P, shared\_key(K,Q,P)) \\ sees(P, encrypt(X,K,R)) \\ distinct(P,R)\end{gathered}}{believes(P, said(Q,X))}$$

$$\textbf{BAN2}: \frac{\begin{array}{c} \mathit{believes}(P, \mathit{public\_key}(K_1, Q)) \\ \mathit{sees}(P, \mathit{encrypt}(X, K_2, R)) \\ \mathit{inv}(K_1, K_2) \end{array}}{\mathit{believes}(P, \mathit{said}(Q, X))}$$

$$\textbf{BAN3}: \frac{\begin{array}{c} \mathit{believes}(P, \mathit{secret}(Y, Q, P)) \\ \mathit{sees}(P, \mathit{combine}(X, Y)) \end{array}}{\mathit{believes}(P, \mathit{said}(Q, X))}$$

In each of these rules, the conclusion precedes the second premise in $\preceq_{BAN}$, so they are valid S-rules. Like the other rules below, they also preserve the constraint on BAN formulas: each atomic argument in the conclusion comes directly from an atomic argument in a premise. The original message-meaning 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 $\mathit{encrypt}$ function and the extra $\mathit{distinct}$ function, by adding an extra premise to each of these rules.

There is one nonce-verification S-rule, whereby a principal can determine that some message was sent recently by examining nonces:

$$\textbf{BAN4}: \frac{\begin{array}{c} \mathit{believes}(P, \mathit{said}(Q, X)) \\ \mathit{believes}(P, \mathit{fresh}(X)) \end{array}}{\mathit{believes}(P, \mathit{believes}(Q, X))}$$

This jurisdiction S-rule expresses one principal's trust of another:

$$\textbf{BAN5}: \frac{\begin{array}{c} \mathit{believes}(P, \mathit{controls}(Q, X)) \\ \mathit{believes}(P, \mathit{believes}(Q, X)) \end{array}}{\mathit{believes}(P, X)}$$

These seven S-rules are for extracting components of messages, and require knowledge of the appropriate keys in the case of encrypted messages. The last two are not given explicitly in the BAN paper [BAN90], but they are necessary and do appear in a technical report by the same authors [BAN89].

$$\textbf{BAN6}: \frac{\begin{array}{c} \mathit{believes}(P, \mathit{shared\_key}(K, Q, P)) \\ \mathit{sees}(P, \mathit{encrypt}(X, K, R)) \end{array}}{\mathit{sees}(P, X)}$$

$$\textbf{BAN7}: \frac{\begin{array}{c} \mathit{believes}(P, \mathit{public\_key}(K, P)) \\ \mathit{sees}(P, \mathit{encrypt}(X, K, R)) \end{array}}{\mathit{sees}(P, X)}$$

$$\textbf{BAN8}: \frac{\begin{array}{c} \mathit{believes}(P, \mathit{public\_key}(K_1, Q)) \\ \mathit{sees}(P, \mathit{encrypt}(X, K_2, R)) \\ \mathit{inv}(K_1, K_2) \end{array}}{\mathit{sees}(P, X)}$$

$$\textbf{BAN9}: \frac{\mathit{sees}(P, \mathit{combine}(X, Y))}{\mathit{sees}(P, X)}$$

$$\textbf{BAN10}: \frac{\mathit{sees}(P, \mathit{comma}(X, Y))}{\mathit{sees}(P, X)}$$

$$\textbf{BAN11}: \frac{\mathit{believes}(P, \mathit{said}(Q, \mathit{comma}(X, Y)))}{\mathit{believes}(P, \mathit{said}(Q, X))}$$

$$\textbf{BAN12}: \frac{\mathit{believes}(P, \mathit{believes}(Q, \mathit{comma}(X, Y)))}{\mathit{believes}(P, \mathit{believes}(Q, X))}$$

There is one freshness G-rule; its premise precedes its conclusion under $\preceq_{BAN}$. It states that a conjunction is fresh if any part of it is fresh:

$$\textbf{BAN13}: \frac{\mathit{believes}(P, \mathit{fresh}(X))}{\mathit{believes}(P, \mathit{fresh}(\mathit{comma}(X, Y)))}$$

We add seven related 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 they are alluded to in the technical report. We include the rest for completeness.

$$\textbf{BAN14}: \frac{\mathit{believes}(P, \mathit{fresh}(K))}{\mathit{believes}(P, \mathit{fresh}(\mathit{shared\_key}(K, Q, R)))}$$

$$\textbf{BAN15}: \frac{\mathit{believes}(P, \mathit{fresh}(K))}{\mathit{believes}(P, \mathit{fresh}(\mathit{public\_key}(K, Q)))}$$

$$\textbf{BAN16}: \frac{\mathit{believes}(P, \mathit{fresh}(Y))}{\mathit{believes}(P, \mathit{fresh}(\mathit{secret}(Y, Q, R)))}$$

$$\textbf{BAN17}: \frac{\mathit{believes}(P, \mathit{fresh}(Y))}{\mathit{believes}(P, \mathit{fresh}(\mathit{combine}(X, Y)))}$$

$$\textbf{BAN18}: \frac{\mathit{believes}(P, \mathit{fresh}(K))}{\mathit{believes}(P, \mathit{fresh}(\mathit{encrypt}(X, K, R)))}$$

$$\textbf{BAN19}: \frac{\mathit{believes}(P, \mathit{fresh}(X))}{\mathit{believes}(P, \mathit{fresh}(\mathit{encrypt}(X, K, R)))}$$

$$\textbf{BAN20}: \frac{\mathit{believes}(P, \mathit{fresh}(X))}{\mathit{believes}(P, \mathit{fresh}(\mathit{combine}(X, Y)))}$$

We add two S-rules that do the work of message-meaning and nonce-verification simultaneously:

$$\textbf{BAN21}: \frac{\begin{array}{c}\mathit{believes}(P, \mathit{fresh}(K))\\ \mathit{sees}(P, \mathit{encrypt}(X, K, R))\\ \mathit{distinct}(P, R)\\ \mathit{believes}(P, \mathit{shared\_key}(K, Q, P))\end{array}}{\mathit{believes}(P, \mathit{believes}(Q, X))}$$

$$\textbf{BAN22}: \frac{\begin{array}{c}\mathit{believes}(P, \mathit{fresh}(Y))\\ \mathit{sees}(P, \mathit{combine}(X, Y))\\ \mathit{believes}(P, \mathit{secret}(Y, Q, P))\end{array}}{\mathit{believes}(P, \mathit{believes}(Q, X))}$$

One of these rules is implicitly required by the published BAN analysis of the Andrew Secure RPC protocol.

Finally, since we represent message composition explicitly (via $comma$), we include two rewrites that express the commutativity and associativity of the $comma$ function; three more rewrites provide commutativity for $shared\_key$, $secret$, and $distinct$:

$$\begin{aligned}
comma(X, Y) &= comma(Y, X)\\
comma(comma(X, Y), Z) &= comma(X, comma(Y, Z))\\
\mathit{believes}(P, \mathit{shared_key}(K, Q, R)) &= \mathit{believes}(P, \mathit{shared_key}(K, R, Q))\\
\mathit{believes}(P, \mathit{secret}(Y, Q, R)) &= \mathit{believes}(P, \mathit{secret}(Y, R, Q))\\
\mathit{distinct}(P, Q) &= \mathit{distinct}(Q, P)
\end{aligned}$$

We have shown that $\preceq_{BAN}$ satisfies conditions P1, P2, and P3′, and that each of the rules above is an S-rule or a G-rule. All the rewrites are clearly size-preserving, so it remains only to show that the S/G restriction holds. Note that the only G-rules produce conclusions regarding freshness. Every freshness premise will therefore be a side-condition, and we can easily check that each of these is no larger (in $\preceq_{BAN}$) than the corresponding conclusion, so the S/G restriction is satisfied, and we can safely apply the TG_ℓ algorithm.

Having encoded the rules, we can analyze each of the four protocols examined in the BAN paper and check all the properties claimed there [BAN90].

3.1.3 A Simple Example

The following example illustrates how the BAN rules can be used to derive properties of a simple single-message "protocol." This protocol requires the following initial assumptions:

$$\begin{gathered} \mathit{believes}(B, \mathit{public\_key}(K_a, A)) \\ \mathit{believes}(B, \mathit{fresh}(T_a)) \\ \mathit{believes}(B, \mathit{controls}(A, \mathit{secret}(Y, A, B))) \\ \mathit{inv}(K_a, K_a^{-1}) \end{gathered}$$

Here, Y is a variable, and A, B, K_a, K_a^{-1}, and T_a are constants. The single message is

Message 1. $A \rightarrow B : \{A, B, T_a, N_{ab}\}_{K_b}$

(We use the notation $\{M\}_K$ to indicate encryption of the message, M, using the key, K.) We "idealize" this message by converting it to a BAN formula including the beliefs it is meant to convey:

$$\mathit{sees}(B, \mathit{encrypt}(\mathit{comma}(T_a, \mathit{secret}(N_{ab}, A, B)), K_a^{-1}, A))$$

First, we want to prove that B believes this message originated from A. Applying Rule 2, with the idealized message and the first and fourth assumptions, we reach this conclusion:

$$\mathit{believes}(B, \mathit{said}(A, \mathit{comma}(T_a, \mathit{secret}(N_{ab}, A, B))))$$

In order to derive any further meaningful beliefs from this message, we must show that B believes that A currently believes it—that it is not a replayed message from the distant past. We apply one of the freshness rules (13), with B's initial assumption that the timestamp, T_a, is fresh, to show that the whole message is fresh:

$$\mathit{believes}(B, \mathit{fresh}(\mathit{comma}(T_a, \mathit{secret}(N_{ab}, A, B))))$$

The nonce-verification rule (4) now yields that B believes A believes the message:

$$\mathit{believes}(B, \mathit{believes}(A, \mathit{comma}(T_a, \mathit{secret}(N_{ab}, A, B))))$$

Finally, by applying the first rewrite (commutativity of comma), an extraction rule (12), and the jurisdiction rule (5), we conclude that, after receiving this message, B believes he shares the secret, N_{ab}, with A:

$$\mathit{believes}(B, \mathit{secret}(N_{ab}, A, B))$$

For more complex examples, see the original BAN papers [BAN90, BAN89].

3.1.4 Kerberos

Through a sequence of four messages, the Kerberos protocol establishes a shared key for communication between two principals, using a trusted server [MNSS87]. The simplified concrete protocol assumed in the original BAN analysis is the following:

Message 1. $A \rightarrow S : A, B$
Message 2. $S \rightarrow A : \{T_s, L, K_{ab}, B, \{T_s, L, K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$
Message 3. $A \rightarrow B : \{T_s, L, K_{ab}, A\}_{K_{bs}}, \{A, T_a\}_{K_{ab}}$
Message 4. $B \rightarrow A : \{T_a + 1\}_{K_{ab}}$

Initially, A wants to establish a session key for secure communication with B. A sends Message 1 to the trusted server, S, as a hint that she wants a new key to be shared with B. The server responds with Message 2, which is encrypted with K_{as}, a key shared by A and S. In this message, S provides the new shared key, K_{ab}, along with a timestamp (T_s), the key's lifetime (L), B's name, and an encrypted message intended for B. In Message 3, A forwards this encrypted message along to B, who decrypts the message to find K_{ab} and its associated information. In addition, A sends a timestamp (T_a) and A's name, encrypted under the new session key, to demonstrate to B that A has the key. Finally, B responds with Message 4, which is simply $T_a + 1$ encrypted under the session key, to show A that B has the key as well.

The BAN analysis of this protocol starts by constructing a three-message idealized protocol; the idealized protocol ignores Message 1, since it is unencrypted and thus cannot safely convey any beliefs. 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 [BAN90]. We express each of these three messages and ten assumptions directly (the conversion is purely syntactic), and add four more assumptions (see Figures 3.2 and 3.3).

The first extra assumption—that A must believe its own timestamp to be fresh—is missing in the original paper, 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 the TG_ℓ algorithm, and it produces an additional 50 true formulas.

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

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

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

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

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

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

Figure 3.2: Kerberos protocol messages, in BAN idealized form and converted to the syntax of our encoding.

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

$\mathit{believes}(A, \mathit{fresh}(T_a))$
$\mathit{distinct}(A, S)$
$\mathit{distinct}(A, B)$
$\mathit{distinct}(B, S)$

Figure 3.3: Encoding of the Kerberos initial assumptions. All but the last four assumptions appear in the BAN analysis [BAN90].

By running the simple decision procedure described in Chapter 2, we can verify that these four desired properties hold:

$$\begin{gathered}
\mathit{believes}(A, \mathit{shared_key}(K_{ab}, A, B)) \\
\mathit{believes}(B, \mathit{shared_key}(K_{ab}, A, B)) \\
\mathit{believes}(B, \mathit{believes}(A, \mathit{shared_key}(K_{ab}, A, B))) \\
\mathit{believes}(A, \mathit{believes}(B, \mathit{shared_key}(K_{ab}, A, B)))
\end{gathered}$$

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.

If we remove the optional final message from the protocol and run the algorithm again, it generates 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

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

(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" [BAN90]. This technique of examining the set difference between the deduced properties of two versions of a protocol is a simple but powerful benefit of the theory generation approach; it helps in understanding differences between protocol variants and it supports rapid prototyping during protocol design.

In the context of the Kerberos protocol and the BAN logic, we illustrate here a single step in the TG_ℓ algorithm, showing how a new formula gets added to the fringe. After several iterations of the closure function are completed, there are 37 formulas in the known-valid set. Of these, 16 are in the fringe, including this one:

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

This formula unifies with the first premise of the "nonce-verification" S-rule (4), so we apply its unifier to the second premise of that rule, yielding

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

None of the 37 formulas unifies directly with this additional premise, so we attempt to work backwards from it, using G-rules and rewrites. If we apply the first freshness G-rule (13) in reverse, we get

$$\mathit{believes}(B, \mathit{fresh}(T_S)),$$

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:

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

This newly derived formula represents the fact that B now believes that S currently believes in this message B has received.

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

We encode the assumptions and messages of the three variants of the Andrew secure RPC handshake protocol given in the BAN paper, and the TG_ℓ algorithm produces the expected results. The last of these verifications requires an extra freshness assumption not mentioned in the BAN analysis:

$$believes(B, fresh(Kab'))$$

It also requires one of the added freshness rules (14) and one of the simultaneous message-meaning/nonce-verification rules (21).

We can also duplicate the BAN results for two variants of the Needham-Schroeder public-key secret-exchange protocol. Finally, we have run the algorithm 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 to the discovery of 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. We replace the occurrence of Y_a in the original protocol by

$$encrypt(comma(Y_a, T_a), K'_a, A)$$

and the occurrence of Y_b by

$$encrypt(comma(Y_b, N_a), K'_b, B)\ .$$

After correcting this, the verifications proceed as expected.

The full encodings of these protocols appear in Appendix B.

3.2 AUTLOG

AUTLOG is an extension of the BAN logic, proposed by Kessler and Wedel [KW94]. It incorporates several new concepts, some of which appear in other BAN variants, such as the GNY logic developed by Gong, Needham, and Yahalom [GNY90]. It allows analysis of a simulated eavesdropper for detecting some information leaks, uses the notion of principals "recognizing" decrypted messages, and introduces a "recently said" notion which is more precise than BAN's beliefs about beliefs.

The encoding of AUTLOG uses all the BAN functions, and a few extras, listed in Figure 3.4. 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. The full set of rules appears in Appendix B.

Function
$\mathit{recognizable}(X)$
$\mathit{mac}(K, X)$
$\mathit{hash}(X)$
$\mathit{recently\_said}(P, X)$

Figure 3.4: Extra AUTLOG functions

In applying the TG_ℓ algorithm, we can use a similar pre-order for AUTLOG to that used for the BAN logic ($\preceq_{BAN}$). AUTLOG has one wrinkle, however, that requires a modification to this pre-order. In AUTLOG, there are rules like the following:

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

Under $\preceq_{BAN}$, the conclusion of this rule is larger than any of its premises. We can see from examining the rules, though, that this rule will not lead to formulas of unbounded size, so we try changing the pre-order. Essentially, we want to collapse the $\mathit{believes}$–said–$\mathit{encrypt}$ sequence and treat it as equivalent to sees–$\mathit{encrypt}$. In order to do this, we define a partial order on function names ($\sqsubset$), and define a new nsyms function recursively:

function $\mathit{nsyms}(F, g) =$

if F is a variable or 0-ary function (constant) **then**
  return 1
else (F must have the form $f(F_1, \ldots, F_n)$)
  $args \leftarrow \{F_i \mid F_i$is not in an atomic argument$\}$
  $sum = \sum_{F' \in args} nsyms(F', f)$
  if $f \sqsubset g$ **then**
    return sum
  else
    return $1 + sum$

We can then define the pre-order as follows:

$$\begin{array}{rcl} F \preceq_{AUTLOG} G & \equiv & (\forall v. occ(v, F) \leq occ(v, G)) \\ & & \wedge\ (\forall f. \forall \sigma. nsyms(\sigma F, f) \leq nsyms(\sigma G, f)) \end{array}$$

The proof that this pre-order satisfies conditions P1–P3 is similar to that in the BAN case.

To check a protocol for leaks using AUTLOG, one finds the consequence closure over the "seeing" rules of the transmitted messages. The resulting list will include everything an eavesdropper could see. The TG_ℓ algorithm is well-suited to computing this list; the seeing rules are all S-rules, so the algorithm 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 have 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 have also checked that the Kerberos protocol, expressed in AUTLOG, satisfies properties similar to those described in Section 3.1.

3.3 Kailar's Accountability Logic

More recently, Kailar has proposed a simple logic for reasoning about accountability in electronic commerce protocols [Kai96]. The central construct in this logic is

$$P \textbf{ CanProve } X$$

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 itself.

Kailar provides different versions of this logic, for "strong" and "weak" proof, and for "global" and "nonglobal" trust. These parameters determine what evidence will constitute an acceptable proof of some claim. The logic we choose uses strong proof and global trust, but the other versions would be equally easy to encode. The encoding uses these functions: $\mathit{CanProve}$, $\mathit{IsTrustedOn}$, $\mathit{Implies}$, $\mathit{Authenticates}$, Says, $\mathit{Receives}$, $\mathit{SignedWith}$, comma, and inv.

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

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

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

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

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

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 can again use $\preceq_{BAN}$ as the pre-order. This makes **Conj** a G-rule; the other three are S-rules. There are a total of six S-rules, one G-rule, and three rewrites in our encoding of this logic; the extra S-rules and rewrites do simple comma-manipulation.

We can replace the construct

$$X \ \mathit{in} \ M$$

(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.

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 3.5: Excerpt from IBS protocol and initial assumptions.

We have verified the variants of the IBS (NetBill) electronic payment protocol that Kailar analyzes [Kai96]. Figure 3.5 contains an encoding of part of the "service provision" phase of the asymmetric-key version of this protocol. The customer, E, first sends the merchant, S, a message containing a price quote, signed by the merchant; this message is itself signed by the customer to indicate his acceptance of the quoted price. The merchant responds by providing the service itself (some piece of data), signed with her private key. The last message of this phase is an acknowledgement by the customer that he received the service, signed with the customer's private key.

When we run the TG_ℓ algorithm on these messages and assumptions, it applies the **Sign** rule to produce these two formulas:

$$CanProve(S, Says(E, comma(SignedWith(Price, K_s^{-1}), Price)))$$

$CanProve(S, Says(E, ServiceAck))$.

These conclusions are not particularly noteworthy in their own right; they reflect the fact that S (the seller) can prove that E (the customer) has presented two specific messages. With these formulas, the TG_ℓ next applies a comma-extracting rule to produce

$$CanProve(S, Says(E, SignedWith(Price, K_s^{-1})))$$
$$CanProve(S, Says(E, Price)).$$

This shows that S can prove E sent individual components of the earlier messages. Finally, TG_ℓ applies **Inf** to derive these results, which agree with Kailar's [Kai96]:

$$CanProve(S, Says(E, ReceivedOneServiceItem(E)))$$
$$CanProve(S, Says(E, AgreesToPrice(E, pr)))$$

These represent two desired goals of the protocol: that the seller can prove the customer received the service, and that the seller can prove what price the customer agreed to. The TG_ℓ algorithm stops at this point, since no further rule applications can produce new formulas.

We have verified the rest of Kailar's results for two variants of the IBS protocol and for the SPX Authentication Exchange protocol. The full encodings of these protocols appear in Appendix B.

3.4 Summary

In this chapter, we have shown how theory generation can be applied to several existing logics for protocol analysis, and successfully reproduced manual verification results for assorted protocols. The table in Figure 3.6 contains some results of these applications of theory generation. Each line in the table shows, for a given protocol, the number of initial assumptions and messages fed to the TG_ℓ algorithm, and the number of formulas in the theory representation it generated. In each case, we were able to use theory generation to prove that the protocols satisfied (or failed to satisfy) various desired belief properties. Note that the generated theory representations typically contained on the order of several dozens of formulas.

The mere fact that certain desired properties of a protocol are derivable in one of these belief logics does not imply that no attacks exist. The rules of each logic incorporate various assumptions, such as "perfect encryption" and constraints on

Logic	Protocol	Assumps.	Msgs.	Th. Rep.
BAN	Kerberos	14, 13	3	61, 52
	Andrew RPC	8, 8, 7	4	32, 39, 24
	Needham-Schroeder	19, 19	5	41, 41
	CCITT X.509	13, 12	3	69, 74
	Wide-Mouth Frog	12, 12	2	34, 34
	Yahalom	9, 17, 17	5	40, 60, 62
AUTLOG	challenge-response 1	2	2	10
	challenge-response 2	4	2	13
	Kerberos	18	3	79
	SKID	8	2	12
Kailar's Accountability	IBS variant 1	14	7	44, 39
	IBS variant 2	20	7	46, 52
	SPX Auth. Exchange	18	3	36

Figure 3.6: Protocol analyses performed with existing belief logics, with the number of formulas in the initial assumptions, messages transmitted, and generated theory representation. (Some analyses involved several variations on the same protocol.)

how messages can be broken up and reconstituted. Beyond this limitation, the logics are not designed to address every form of attack. BAN, for instance, cannot express the property that secrets are not leaked. In Chapter 5, we explore further uses of theory generation for protocol analysis. In particular, we work with a new logic (introduced in Chapter 4) that allows more comprehensive protocol analyses, allowing more confidence in protocols that pass its checks. We also examine applications of theory generation beyond simple property-checking. Details regarding the implementation and performance of the system used to produce these results appear in Chapter 6.

Chapter 4

Interpretation and Responsibility

Using theory generation with existing logics as described in Chapter 3, we can quickly check the standard belief properties of protocols described in the conventional "idealized" form. There are, however, many other questions about a protocol that we might want to answer, so it is reasonable to ask whether the same technique could be applied to a wider class of properties, or to different forms of protocol description. Ideally, we would endow a protocol-verification system with a larger toolkit of analyses without requiring from the user undue amounts of extra information or patience.

In this chapter, we consider several new kinds of checks such a system can perform at relatively low cost. Taken individually, each of these checks can provide more confidence in a protocol, and when applied together and in concert with traditional belief checks, they complement one another to form a unified and more comprehensive analysis.

We first discuss a way to more fully formalize the process of interpreting messages (Section 4.1); this will allow us to reason about protocols at a more concrete level. We present the formalization in the context of a new belief logic, RV, which borrows concepts and rules from the BAN, GNY, and AUTLOG logics. Next we introduce two classes of properties, *honesty* and *secrecy*, which rely on the formalized interpretations and can expose flaws not addressed by the belief logics considered earlier. The honesty properties restrict the messages that a principal may safely "sign," to prevent the messages' recipients from reaching faulty conclusions. The secrecy properties regulate the disclosure of information to other parties, and also depend critically on explicit interpretation. A participant in a protocol who satisfies both honesty and secrecy properties is said to be "responsible." We claim that demonstrating the responsibility of participants in a protocol is a necessary counterpart to proving traditional belief properties.

4.1 Interpretation

The BAN-style belief logics allow protocol designers to think about a protocol at a convenient level of abstraction; however, the gap between the "idealized" protocol and a typical concrete protocol implementation is substantial. The idealization step is widely recognized as a liability of these logics [Syv91, NS93, MB94]. It is an informal process; we typically write the concrete protocol side-by-side with a proposed idealized version, and then attempt to derive desired properties. When these derivations fail, we augment the idealized messages with extra statements or introduce additional initial assumptions. In each case the burden is on the (human) verifier to ensure that the additions are "safe." Finally, with no formal description of concrete messages, the implicit assumptions about their form, such as whether keys and nonces are distinguishable, are easily forgotten or left unspecified.

The original BAN analysis of the Needham/Schroeder public key protocol used a bad idealization that went undetected until Lowe found a flaw in the protocol and demonstrated it using model checking methods [Low95, Low96]. The flaw escaped detection for more reasons than the bad idealization—we will look at these reasons closely in Section 4.2—but the idealization was certainly flawed. The fact that the fix Lowe proposed cannot even be expressed in the idealized level indicates the need for more concrete grounding of BAN-style reasoning. We discuss this flaw and its connection to idealization in Sections 4.2.3 and 4.3.2.

4.1.1 Possible Approaches

Several approaches to bridging this idealization gap have been proposed or deserve consideration.

Abadi and Needham provide a set of practical guidelines for constructing good idealizations (or conversely, constructing good concrete protocols) [AN96]. Though they are instructive and merit careful consideration, these guidelines are not amenable to formal verification. Furthermore, as the authors acknowledge, they are neither necessary nor sufficient. Since the principles are often legitimately violated in practice, it is hard to identify the truly improper idealizations that lead to problems.

We could take a somewhat more drastic approach and abandon the idealized level of protocol description completely. By working solely at the concrete level, we avoid having to assign meanings to messages. This is what recent verification techniques based on model checking do [Low96, MMS97, MCJ97]. While this approach has the appeal of producing counterexamples and not requiring the construction of idealizations, it does have some disadvantages. With no formal notion of the "meaning" of a message, we can no longer reason about belief, which is a natural notion in the context of authentication. We can analyze data flow: secrets successfully communicated, information leaked, and so on, and we can check the *correspondence* properties defined by Woo and Lam [WL93], but the richer and perhaps more intuitive belief properties are out of reach. We cannot factor out the abstract core of a protocol from its various possible implementations, and there is little indication of *why* a protocol works.

Mao has proposed a method of developing idealizations in a principled way [Mao95], by breaking the idealization process down into a sequence of incremental steps, each of which can be justified individually. The approach described below is more suitable for automation via theory generation, and can more directly express the desired qualities of idealizations.

Many of the problems with idealizations result from idealized messages containing more information than their concrete counterparts, which leads to ambiguous mappings from concrete to idealized messages. We could solve this problem by establishing a single one-to-one mapping for all protocols, for instance using an ASN.1 notation [ASN94]. This leads to larger messages; in most environments today, public-key encryption is expensive relative to other costs of communication, so increased message size may be unacceptable. (This problem is smaller with elliptic-curve cryptography, since it can be implemented more efficiently than traditional public-key methods.) More importantly, this technique imposes significant constraints on the form of the concrete protocol, so few existing protocols would be accepted, even if the concrete-to-abstract mapping were protocol-dependent. Finally, this approach only solves the ambiguous-interpretation problem; the protocol designer must still devise an idealized protocol.

4.1.2 The RV Logic core

Before describing the formalization of explicit interpretations, we present here the core of a new belief logic, RV (for *Responsibility Verification*), on which those interpretations will be built. The core rules and operators of this logic are very similar to those of BAN, AUTLOG, and GNY; the significant new features are introduced in the sections that follow.

Figure 4.1 shows the functions and predicates available in the RV core. The

Function	Notation	Meaning
$believes(P, X)$	$P \mid\equiv X$	P believes X
$sees(P, X)$	$P \triangleleft X$	P has seen message X
$said(P, X)$	$P \mid\sim X$	P uttered X at some time
$says(P, X)$	$P \mid\approx X$	P uttered X "recently"
$controls(P, X)$	P **controls** X	P is an authority on X
$fresh(X)$	$\#(X)$	X was first used in this run
$shared\_key(K, P, Q)$	$P \overset{K}{\longleftrightarrow} Q$	P and Q share key K
$public\_key(K, P)$	$\overset{K}{\mapsto} P$	P's public key is K
$secret(Y, P, Q)$	$P \overset{Y}{\rightleftharpoons} Q$	P and Q share secret Y
$encrypt(X, K)$	$\{X\}_K$	X encrypted under key K
$inv(K_1, K_2)$	$K_1 = K_2^{-1}$	public/private key pair
$in(Y, X)$	Y **in** X	Y occurs in X
$comma(X, Y)$	(X, Y)	conjunction
$A, B, S, T, \ldots$	$A, B, S, T, \ldots$	0-ary functions (constants)

Figure 4.1: Core functions of RV

full function names are in the left column, but we use the traditional concise notation in the center column when describing the rules. All of these functions appear in the BAN logic encoding (Section 3.1), with the exceptions of $says$ and in. The $says$ function, also found in AUTLOG, expresses that a principal has *recently* sent some message, whereas $said$ only ensures that the principal uttered it at some time. In BAN the $says$ notion is subsumed by $believes$, in that $P \mid\equiv Q \mid\approx X$ ("P believes Q says X") is written as $P \mid\equiv Q \mid\equiv X$ ("P believes Q believes X"). The in function, similar to one in GNY, indicates that one message is part of another message. Note that the $combine$ operator has been removed; this is discussed below.

RV has the following three message-meaning S-rules. As in BAN, they allow one principal to determine who sent a given message. Shared-secret authentication is achieved without an explicit secret-combining operator ($\langle X \rangle_Y$); the presence of a shared secret anywhere in a message is sufficient to authenticate it. This change, also made in GNY, simplifies the honesty properties described later.

$$\mathbf{RV1}: \frac{P \mid\equiv Q \overset{K}{\longleftrightarrow} P \qquad P \triangleleft \{X\}_K}{P \mid\equiv Q \mid\sim (X, K)}$$

$$\mathbf{RV2}: \frac{P \models \overset{K_1}{\mapsto} Q \qquad P \lhd \{X\}_{K_2} \qquad K_1 = K_2^{-1}}{P \models Q \mid\sim X}$$

$$\mathbf{RV3}: \frac{P \models Q \overset{Y}{\rightleftharpoons} P \qquad P \lhd X \qquad Y \textbf{ in } X}{P \models Q \mid\sim X}$$

The nonce-verification S-rule allows a principal to determine that a message was sent recently if it believes it is fresh:

$$\mathbf{RV4}: \frac{P \models Q \mid\sim X \qquad P \models \#(X)}{P \models Q \mid\approx X}$$

Using the jurisdiction S-rule, a principal gains a new belief based on a recent statement by a party it considers an authority on the matter in question:

$$\mathbf{RV5}: \frac{P \models Q \textbf{ controls } X \qquad P \models Q \mid\approx X}{P \models X}$$

In RV, as in AUTLOG, a principal "sees" every statement it believes. This G-rule makes the decryption rules simpler:

$$\mathbf{RV6}: \frac{P \models X}{P \lhd X}$$

We add this *introspection* rule, similar to one appearing in the SVO logic.

$$\mathbf{RV7}: \frac{P \lhd X}{P \models P \lhd X}$$

A principal can decrypt any message whose key it has seen using these S-rules:

$$\mathbf{RV8}: \frac{P \lhd Q \overset{K}{\longleftrightarrow} P \qquad P \lhd \{X\}_K}{P \lhd X} \qquad \mathbf{RV9}: \frac{P \lhd \overset{K}{\mapsto} P \qquad P \lhd \{X\}_K}{P \lhd X}$$

$$\mathbf{RV10}: \frac{P \lhd \overset{K_1}{\mapsto} Q \qquad P \lhd \{X\}_{K_2} \qquad K_1 = K_2^{-1}}{P \lhd X}$$

These S-rules provide decomposition of compound messages built with $comma$ (the **in** function is introduced below in $\mathbf{RV22}$).

$$\mathbf{RV11}: \frac{P \lhd M \qquad X \textbf{ in } M}{P \lhd X}$$

$$\mathbf{RV12}: \frac{P \models Q \mid\!\sim M \qquad X \textbf{ in } M}{P \models Q \mid\!\sim X} \qquad \mathbf{RV13}: \frac{P \models Q \mid\!\approx M \qquad X \textbf{ in } M}{P \models Q \mid\!\approx X}$$

We include a complete set of G-rules for determining the freshness of a message given the freshness of one if its components:

$$\mathbf{RV14}: \frac{P \models \#(X)}{P \models \#((X,Y))} \qquad \mathbf{RV15}: \frac{P \models \#(K)}{P \models \#\left(Q \xleftrightarrow{K} R\right)}$$

$$\mathbf{RV16}: \frac{P \models \#(K)}{P \models \#\left(\overset{K}{\mapsto} Q\right)} \qquad \mathbf{RV17}: \frac{P \models \#(Y)}{P \models \#\left(Q \overset{Y}{\rightleftharpoons} R\right)}$$

$$\mathbf{RV18}: \frac{P \models \#(K)}{P \models \#(\{X\}_K)} \qquad \mathbf{RV19}: \frac{P \models \#(X) \qquad P \lhd \overset{K}{\mapsto} Q}{P \models \#(\{X\}_K)}$$

$$\mathbf{RV20}: \frac{P \models \#(X) \qquad P \lhd \overset{K_2}{\mapsto} Q \qquad P \lhd K_2 = K_1^{-1}}{P \models \#\left(\{X\}_{K_1}\right)}$$

$$\mathbf{RV21}: \frac{P \models \#(X) \qquad P \lhd Q \xleftrightarrow{K} R}{P \models \#(\{X\}_K)}$$

Finally, this G-rule, whose premise is empty, allows us to determine when one message is part of a larger message. Note that in only “looks inside” the $comma$ function, and not $encrypt$ or any others. The reasons for this will become clear in the following sections.

$$\mathbf{RV22}: \frac{\cdot}{X \textbf{ in } (X,Y)}$$

The following equivalences express the usual associative and commutative properties of $comma$ and other operators.

$$(X,Y) = (Y,X)$$

$$((X,Y),Z) = (X,(Y,Z))$$

$$(Q \xleftrightarrow{K} R) = (R \xleftrightarrow{K} Q)$$

$$(Q \overset{Y}{\rightleftharpoons} R) = (R \overset{Y}{\rightleftharpoons} Q)$$

4.1.3 Concrete Message Syntax

To properly formalize the mapping of concrete messages to abstract messages, we must of course carefully specify the form concrete messages take. This requires striking a balance between allowing flexibility in implementations and permitting tractable verification. We will assume messages are composed of atoms (such as nonces, keys, and principal names), combined by concatenation and encryption. We introduce a new RV concatenation operation, $[X.Y]$ (see Figure 4.2). The concatenation operator is ordered: $[X.Y]$ and $[Y.X]$ are distinguishable, as they would be in typical implementations. We will make the usual "perfect cryptography" assumptions: $\{X\}_K$ can only be produced by a principal who knows $\{X\}_K$ or both X and K, and seeing $\{X\}_K$ reveals no information about X or K to a principal who does not know K. We assume that the implementation can identify where each atom begins and ends, so a principal expecting message $[X.Y]$ will reject a message $[N1]$ or $[N1.N2.N3]$, and if it receives $[N1.N2]$, will always match $N1$ to X and $N2$ to Y rather than splitting the message elsewhere. We also assume that decrypting a message with the wrong key will yield results distinguishable from any normal message. We could eliminate this last assumption, if necessary, by introducing a *recognizable* operator as in AUTLOG.

Function	Notation	Meaning
$dot(X,Y)$	$[X.Y]$	concatenation
nil	$[]$	concatenated list terminator

Figure 4.2: Extra functions of RV for supporting concrete messages.

To give concrete messages the properties described above, we need a canonical representation for messages containing sequences of atoms, so we introduce a nil atom and using the Lisp list-building convention. For convenience, we write $[X.\,[Y.\,[Z.nil]]]$ as $[X.Y.Z]$. We require every concrete message in a protocol description to be constructed as follows:

- The special constant nil is a valid concrete message.
- If X is a constant and M is a valid concrete message, then $[X.M]$ is a valid concrete message.
- If M is a valid concrete message and K is a constant, then $\{M\}_K$ is a valid concrete message.

We do not assume that keys, nonces, and principal names are always distinguishable, that principals always recognize their own messages, or that components of message j are always distinguishable from components of message k ($k \neq j$). In environments where some or all of these assumptions are safe, we can easily apply them by having the verification tool automatically tag message components with their types. Finally, we must extend the rules for deducing freshness and determining (X **in** Y) to accommodate the new concatenation operator:

$$\mathbf{RV23}: \frac{P \models \#(X)}{P \models \#([X.Y])} \qquad \mathbf{RV24}: \frac{P \models \#(Y)}{P \models \#([X.Y])}$$

$$\mathbf{RV25}: \frac{\cdot}{X \textbf{ in } [X.Y]} \qquad \mathbf{RV26}: \frac{\cdot}{Y \textbf{ in } [X.Y]}$$

4.1.4 Explicit Interpretations

The following approach to the idealization problem maintains the spirit of the BAN family of belief logics, and fits nicely with the theory generation verification technique. We replace the informal idealization step by making concrete-message interpretation explicit within the logic. The form of the BAN-like logics suggests a natural expression of idealizations: as extra rules added to the logic.[1] Each protocol will have a set of interpretation rules that can be used by all principals to assign abstract meanings to the concrete messages they receive. For instance, if Alice sees the signed message `[A.B.K]`, she might be allowed to assume that the meaning of that message is $A \stackrel{K}{\longleftrightarrow} B$. To allow this sort of interpretation, the protocol could include this rule:

$$\frac{P \models Q \mid\approx [P.Q.K]}{P \models Q \mid\approx P \stackrel{K}{\longleftrightarrow} Q}$$

Kailar uses a similar technique to assign complex meanings such as "P agrees to sell N items at price X" to messages [Kai96]. With these extra interpretation rules, we start from the usual initial assumptions, and derive all properties of the protocol directly from the concrete protocol messages, rather than starting off with idealized messages.

[1] If we took the approach of Abadi and Tuttle [AT91] (and others), allowing principals to use *modus ponens*, these interpretation rules could be expressed instead as formulas held as initial assumptions. This is equivalent to the approach described here, but does not work as well with theory generation.

We could allow each protocol to use any custom interpretation rules its designer (or specifier) chooses, but this flexibility would make it impossible to prove any meaningful claims about the results of RV protocol analyses in general. Instead we restrict the allowed interpretation rules to those conforming to some pre-established rule schemata.

To start with, we will allow interpretation rules in any of the following forms:

$$\mathbf{S1}: \frac{P \models Q \mid\sim \mathsf{M}}{P \models Q \mid\sim \mathsf{M}'} \qquad \mathbf{S2}: \frac{P \models Q \mid\approx \mathsf{M}}{P \models Q \mid\approx \mathsf{M}'} \qquad \mathbf{S3}: \frac{P \lhd \mathsf{M}}{P \lhd \mathsf{M}'}$$

Here, M is a pattern matching concrete messages, and M' is a formula representing an idealized message, which may include variables from the premise. The first two forms will be used to interpret authenticated messages; since the recipient is certain who sent the concrete message, it can safely use that information for interpretation. For instance, the variable Q may occur in M', in which case the idealized meaning is dependent on the sender's identity. The third form will be used for messages that have useful meanings despite having undetermined origin. This situation does not often arise in BAN reasoning, but it does in RV, in particular for the honesty and secrecy checks introduced later in this chapter.

We impose some additional restrictions on the form of these interpretation rules:

I1. In M, the pattern matching the concrete message, the only function that may occur is dot.

I2. In M', the idealized meaning, the $encrypt$ function must not occur.

I3. In both M and M', there must be no constants (0-ary functions such as A and S), unless the value of those constants are fixed across all protocol runs. For instance, a server name S is permitted in these arguments only if S is the same principal in every run of the protocol.

I4. For a given protocol, any given concrete message must be able to match at most one of the interpretation rules. Along with the honesty condition described later, this prevents ambiguous interpretations of concrete messages.

These restrictions on the form of interpretation rules help ensure that the usual BAN-style inferences remain reasonable regardless of the interpretations used. For instance, the message-meaning rules such as

$$\frac{P \models Q \stackrel{K}{\longleftrightarrow} P \qquad P \lhd \{X\}_K}{P \models Q \mid\sim (X, K)}$$

would make no sense if interpretations could arbitrarily map $\{X\}_{K_1}$ to $\{X\}_{K_2}$ (in violation of condition I2), since K_2 could be a shared key that the sender had no knowledge of, leading the recipient to draw a dangerous conclusion about the origin of the message. We discuss further the reasons for these restrictions in Section 4.2.2, where we examine honesty properties in the context of these explicit interpretations.

We need to extend the RV notation and interpretation schemata further to express interpretations required by some common protocols. Abadi and Needham, Mao, and others have observed that nonces are used in cryptographic protocols not just to establish freshness, but as abbreviations for sets of principal names, keys, and other nonces [Mao95, AN96]. The bindings established for these nonces are specific to a run of the protocol, and we will see later that it is important to handle them explicitly in the interpretation process. To support this practice, we introduce a new operator: $bind$ (see Figure 4.3). The statement, $N \leadsto X$, represents

Function	Notation	Meaning
$bind(N, X)$	$N \leadsto X$	N stands for the formula X

Figure 4.3: Functions of RV for supporting explicit interpretations.

the assumption that nonce N "stands for" the formula X. We allow the interpretation rule forms described above to be extended to make use of bound nonces in two ways:

- Interpretation rules of the forms $\mathbf{S1}$ and $\mathbf{S2}$ (interpretations of authenticated messages) may be extended with the following set of extra premises:

  $$P \models N_i \leadsto \mathsf{F} \qquad P \models P \overset{N_i}{\rightleftharpoons} Q \qquad N_i \textbf{ in } M$$

  This represents the "expansion" of a bound nonce by the principal who produced that nonce. Note that it requires the nonce to be a shared secret; otherwise the recipient cannot safely assume that, for instance, the sender has seen the correct $bind$ expression.

- Interpretation rules of the forms $\mathbf{S1}$ and $\mathbf{S2}$ may also be extended with the following set of extra premises:

  $$P \models Q \mid\!\sim N_i \leadsto \mathsf{F} \qquad N_i \textbf{ in } M$$

  This extension allows a principal to expand a nonce whose binding has been previously declared by the sender.

These extensions can be applied repeatedly, so a single interpretation rule may handle several nonce bindings simultaneously. In practice this is rarely necessary. Note that the restrictions I1–I4 still apply to the extended interpretation rules; in particular the N_i must be variables.

We require that bound nonces be believed fresh by their creators:

$$\mathbf{RV27} : \frac{P \models N \rightsquigarrow X}{P \models \#(N)}$$

These bindings are thus limited in scope to a single protocol run.

4.1.5 Example

The following example will illustrate the use of explicit interpretations for the Otway-Rees key-exchange protocol [OR87]. The protocol allows a server to distribute a session key, K_{ab}, to the participating principals using just four messages. These are the messages in their concrete form:

$$\begin{array}{ll}
\text{Message 1.} & A \rightarrow B : \left[M.A.B.\,[N_a.M.A.B]_{K_{as}}\right] \\
\text{Message 2.} & B \rightarrow S : \left[M.A.B.\,[N_a.M.A.B]_{K_{as}}.\,[N_b.M.A.B]_{K_{bs}}\right] \\
\text{Message 3.} & S \rightarrow B : \left[M.\,[N_a.K_{ab}]_{K_{as}}.\,[N_b.K_{ab}]_{K_{bs}}\right] \\
\text{Message 4.} & B \rightarrow A : \left[M.\,[N_a.K_{ab}]_{K_{as}}\right]
\end{array}$$

This protocol is similar to the Kerberos protocol, but it uses nonces (N_a, N_b, and M), rather than timestamps, to ensure freshness. As a result, its security does not depend on synchronized clocks.

The original BAN idealization of this protocol makes use of a new nonce, N_c, which "corresponds to" (M, A, B). We will use a more straightforward idealization without this potentially hazardous informal abbreviation:

$$\begin{array}{ll}
\text{Message 1.} & A \rightarrow B : \{N_a \rightsquigarrow [M.A.B]\}_{K_{as}} \\
\text{Message 2.} & B \rightarrow S : \{N_a \rightsquigarrow [M.A.B]\}_{K_{as}}, \{N_b \rightsquigarrow [M.A.B]\}_{K_{bs}} \\
\text{Message 3.} & S \rightarrow B : \{N_a, A \xleftrightarrow{K_{ab}} B\}_{K_{as}}, \{N_b, A \xleftrightarrow{K_{ab}} B\}_{K_{bs}} \\
\text{Message 4.} & B \rightarrow A : \{N_a, A \xleftrightarrow{K_{ab}} B\}_{K_{as}}
\end{array}$$

We have added $bind$ expressions to the idealizations of the first two messages. Because they have told the server these bindings, A and B can safely (and unambiguously) interpret the later messages from S as containing shared keys for

A and B. In the next section, we will discuss the precise restrictions on message sending and interpretation that make these bindings necessary.

Now that we have a desired idealized protocol, it remains to define a set of interpretations that produce these idealized messages from the corresponding concrete ones. The following interpretations suffice:

$$\frac{P \models Q \mid\!\sim [N_1.N_2.R_1.R_2]}{P \models Q \mid\!\sim N_1 \leadsto [N_2.R_1.R_2]}$$

$$\frac{P \models Q \mid\!\sim [N_1.K] \qquad P \models N_1 \leadsto [N_2.R_1.R_2] \qquad P \models P \overset{N_1}{\rightleftharpoons} Q \qquad N_1 \textbf{ in } [N_1.K]}{P \models Q \mid\!\sim \left(N_1, R_1 \overset{K}{\longleftrightarrow} R_2\right)}$$

The first interpretation is intended to apply to messages 1 and 2, and the second to messages 3 and 4. To analyze this protocol, we start with initial assumptions similar to those used by BAN, with the following additions:

$$\begin{array}{ll} A \models N_a \leadsto [M.A.B] & \qquad A \models A \overset{N_a}{\rightleftharpoons} S \\ B \models N_b \leadsto [M.A.B] & \qquad B \models B \overset{N_b}{\rightleftharpoons} S \end{array}$$

Note that the nonces N_a and N_b are required to be secret in this formalization of the protocol, whereas they were not in the BAN formalization. In the original BAN analysis, in fact, "optimizations" to the protocol were suggested, which involved sending these nonces in the clear. The authors later realized that this was unsafe, but the BAN logic itself provides no way to detect this problem.

The effect of sending each message is represented by the receiver seeing the concrete message:

$$\begin{array}{l} B \lhd \left[M.A.B.\,[N_a.M.A.B]_{K_{as}}\right] \\ S \lhd \left[M.A.B.\,[N_a.M.A.B]_{K_{as}}.\,[N_b.M.A.B]_{K_{bs}}\right] \\ B \lhd \left[M.\,[N_a.K_{ab}]_{K_{as}}.\,[N_b.K_{ab}]_{K_{bs}}\right] \\ A \lhd \left[M.\,[N_a.K_{ab}]_{K_{as}}\right] \end{array}$$

Using the standard message-meaning and nonce-verification rules, we can derive from message 4 that

$$A \models S \mid\!\approx [N_a.K_{ab}]$$

Then, applying the second interpretation rule with A's initial binding and secrecy assumptions yields the desired idealized message:

$$A \models S \mid\!\approx \left(N_a, A \xleftrightarrow{K_{ab}} B\right)$$

This satisfies one of the authentication goals of the protocol.

The remainder of the belief-property derivation for this protocol proceeds similarly to the original BAN analysis and produces similar conclusions.

4.2 Honesty

Burrows, Abadi, and Needham recommend, "for the sake of soundness, we always want to guarantee that each principal believes the formulas that he generates as messages" [BAN90]. This requirement has intuitive appeal: the protocol should not require principals to "lie." If legitimate participants were free to send arbitrary messages, recipients could derive faulty conclusions about the senders' beliefs from those messages. We will refer to this kind of restriction as an *honesty property*. Without honesty there is little hope of demonstrating that derived beliefs are always sound with respect to some reasonable model. In the next two sections, we discuss honesty first for idealized protocols, and then in the context of explicit interpretations.

4.2.1 Honesty in Idealized Protocols

We need a more precise statement of the honesty property. The primary goal is to prevent message recipients from deducing beliefs of other principals that are invalid.[2] Later we will want to prevent other sorts of misinterpretations as well. To achieve this, we must consider the circumstances under which one principal can decide another principal holds some belief. This typically happens through the application of a message-meaning rule, such as the rule for interpreting messages signed with a shared key, followed by the application of a nonce-verification rule. We focus on the message-meaning step: any message (formula) that is encrypted under some key (e.g., $\{A \xleftrightarrow{K_{ab}} B\}_{K_s}$), or combined with some secret

[2]Strictly speaking, principals do not deduce one another's beliefs in the RV logic; they only deduce that another principal recently made some statement ($P \models Q \mid\!\approx X$). However, the effect is the same as in BAN, where $P \models Q \models X$ can be derived.

(e.g., $\langle A \overset{N_b}{\rightleftharpoons} B\rangle_{N_a}$) could potentially be interpreted as a belief of the principal doing the encrypting or combining. Therefore, we will state the honesty property in terms of messages *signed*, where a principal signs a message whenever it applies encryption or secret-combination to that message. (Note that this notion is broader than that of public-key digital signatures.) A principal must take responsibility for any statement it signs, since those are the statements that might later be used by other principals to derive beliefs of the signer.

We can now state the honesty property:

> **Honesty property** (for idealized protocols): For every message component M that a principal P *signs*, it must be true that $P\ \mathit{believes}\ M$ at the point at which P sent the message containing M.

The requirement that the belief hold at the time the message is sent prevents circular situations in which two (or more) principals send each other statements that neither believes initially, but that both come to believe after receiving the other's message.

Most of the idealized protocols in the published BAN analyses will not pass this test, since they typically contain messages like $\{T_a, A \overset{K}{\longleftrightarrow} B\}_{K_{ab}}$, where the signer does not actually "believe" T_a. We can fix this by replacing each troublesome message fragment, X, with some formula the sender actually believes, such as $A\ \mathit{sees}\ X$ or $\mathit{fresh}(X)$. This approach requires introducing some extra "seeing" rules, such as, $P\ \mathit{sees}\ Q\ \mathit{said}\ X\ \vdash P\ \mathit{sees}\ X$.

4.2.2 Honesty with Explicit Interpretations

The honesty property is fairly simple in the context of reasoning about idealized protocols, but when we introduce interpretation rules, it becomes more interesting. Since concrete messages have no direct meaning, it no longer makes sense to talk about believing the contents of a message; we can only discuss believing some interpretation of a message. Roughly, we want to extend the honesty property to say that, for every message component a principal signs, that principal must believe all possible interpretations of that message component. This allows honesty to proscribe both blatant lies and statements open to misinterpretation. We must, however, refine the notion of interpretation for this definition to be useful.

By making interpretations explicit, we expose the possibility that a single concrete message may be interpreted differently in different runs of the protocol. This would make the honesty property difficult to verify; we would prefer to confine our reasoning to a single run, but have the results hold in the presence of multiple runs. By imposing restrictions I1–I4 (Section 4.1.4) on the interpretations, we can ensure that no additional interpretations are possible in other runs of the protocol. For instance, restriction I3 requires that interpretation rules make no mention of constants whose values are specific to a run of the protocol. This guarantees that the result of applying an interpretation to a given concrete message will not depend on the protocol run in which the interpretation is applied. This restriction, which corresponds loosely to Abadi and Needham's "explicitness" principle, is actually stronger than necessary, but in combination with the $bind$ operator introduced in Section 4.1, it seems to be flexible enough to handle most protocols, and it substantially simplifies verification.

We model encryption explicitly at the concrete level, so there is no need for interpretations to introduce encryption. To preserve orthogonality with the message-meaning rules, interpretations should also not perform decryption. This is enforced through the simple syntactic restrictions I1 and I2, which disallow encryption operators on either side of an interpretation. Now, since secret-combination is only useful within encrypted messages and encryption is unaffected by interpretation, we can express honesty as a (conservative) restriction on concrete messages that a principal *encrypts*.

The honesty property for the explicit interpretations case is as follows:

> **Honesty property** (with explicit interpretations): For every message M that a principal P encrypts, and for every *possible interpretation*, M', of M, it must be the case that P $believes$ M' at the point at which P sent M.

It remains to define what a *possible interpretation* is in this context. The honesty property should be a function of the sender's beliefs and the global interpretations, so an interpretation M^* of M will be considered possible if P cannot determine it is impossible. This means that, in principle, P must consider all possible recipients of the message at all times, with any conceivable set of beliefs. It is difficult in a belief-logic setting to reason about such a broad set of possibilities, but fortunately we get some help from the notions of shared and public keys, secrets, and freshness. For instance, if a principal A believes $A \overset{K}{\longleftrightarrow} B$ then she implicitly believes that there is no C who currently believes $C \overset{K}{\longleftrightarrow} D$ where $C, D \neq A, B$, and that any C who holds such a belief in the future will not consider this message fresh.

Together with the syntactic restrictions on interpretation rules, the honesty property provides many of the guarantees we would like to hold for idealizations.

The I4 restriction provides a further guarantee that simplifies honesty checking via theory generation: at most one of the interpretation rules for a protocol will match any given concrete message.

Function	Notation	Meaning
$legit(M)$	$\textbf{legit}(M)$	M is a legitimate message to send, according to the honesty property
$signed(M, M_s, P, Q)$	$\textbf{signed}(M, M_s, P, Q)$	M_s is a signed message for P from Q

Figure 4.4: Functions of RV for supporting honesty.

To reason about honesty within RV, we introduce two new functions (Figure 4.4): $legit$, indicating that a message is safe to transmit from an honesty perspective, and $signed$, describing situations where a principal may be held responsible for a transmitted message.

To derive $legit$, we introduce rules that are complementary to the interpretation rules for the protocol. An interpretation rule specifies what meaning a receiver may safely assign to a concrete message; a $legit$ rule specifies a concrete message whose meaning is certain to be believed by the sender. For each interpretation rule of the form $\mathbf{S1}$ or $\mathbf{S2}$, we add a rule of the following form, with the same M and M':

$$\mathbf{S1}'/\mathbf{S2}' : \frac{Q \models M' \qquad Q \models \textbf{signed}(M, M_s, P, Q)}{Q \models \textbf{legit}(M_s)}$$

This means that if the message has an interpretation that the sender believes, then a signed version of that message is safe to transmit. Similarly, if a message can be given an "anonymous interpretation" that the sender believes, it can be sent. We therefore add a rule of the following form for each $\mathbf{S3}$ rule:

$$\mathbf{S3}' : \frac{Q \models M'}{Q \models \textbf{legit}(M)}$$

To account for bound nonces, we must extend $\mathbf{S1}'/\mathbf{S2}'$ rules in ways analogous to the two allowed extensions of $\mathbf{S1}/\mathbf{S2}$ interpretations. For bound-nonce extensions of the first kind, we add the following premises:

$$Q \lhd N \leadsto \mathsf{F} \qquad Q \lhd P \stackrel{N}{\rightleftharpoons} Q \qquad N \textbf{ in } \mathsf{M}$$

For bound-nonce extensions of the second kind, we add these premises:

$$Q \models N \rightsquigarrow \mathsf{F} \qquad N \textbf{ in } \mathsf{M}$$

We then need rules for determining when a message is believed to be signed:

$$\mathbf{RV28}: \frac{Q \models P \models P \stackrel{K}{\longleftrightarrow} Q}{Q \models \textbf{signed}(M, \{M\}_K, P, Q)}$$

$$\mathbf{RV29}: \frac{Q \lhd \stackrel{K_1}{\mapsto} Q \qquad K_1 = K_2^{-1}}{Q \models \textbf{signed}(M, \{M\}_{K_2}, P, Q)}$$

$$\mathbf{RV30}: \frac{Q \models P \models P \stackrel{Y}{\rightleftharpoons} Q \qquad Y \textbf{ in } M}{Q \models \textbf{signed}(M, \{M\}_K, P, Q)}$$

Finally, we introduce two message transformations that preserve "legitimacy," for the purpose of producing compound messages:

$$\mathbf{RV31}: \frac{Q \models \textbf{legit}(X) \qquad Q \models \textbf{legit}(Y)}{Q \models \textbf{legit}([X.Y])}$$

$$\mathbf{RV32}: \frac{Q \models \textbf{legit}(X) \qquad Q \models \stackrel{K}{\mapsto} P}{Q \models \textbf{legit}(\{X\}_K)}$$

4.2.3 Example

At several steps in this development, we have imposed restrictions that were stronger than necessary for soundness. We will demonstrate here that a practical protocol can still pass these tests. The Needham-Schroeder public-key protocol illustrates the trouble principals can get into if they are not bound by honesty. The full protocol involves seven messages, but four of these (messages 1, 2, 4, and 5) are concerned only with requesting and issuing public-key certificates for A and B. We focus on the three messages exchanged between A and B here:

Message 3. $A \rightarrow B : \{[N_a.A]\}_{K_b}$
Message 6. $B \rightarrow A : \{[N_a.N_b]\}_{K_a}$
Message 7. $A \rightarrow B : \{N_b\}_{K_b}$

In the omitted messages of this protocol (1, 2, 4, and 5), A and B get each other's public keys from a trusted server. The three messages shown above are a simple exchange of the secrets N_a and N_b, in which each message is encrypted with the recipient's public key. The presence of N_a in Message 6 and N_b in Message 7 is intended to acknowledge the successful receipt of those secrets.

The idealized form of these messages given in the BAN analysis is equivalent to the following:

$$
\begin{array}{ll}
\text{Message 3.} & A \rightarrow B : \{N_a\}_{K_b} \\
\text{Message 6.} & B \rightarrow A : \left\{ \left(A \stackrel{N_b}{\rightleftharpoons} B, N_a \right) \right\}_{K_a} \\
\text{Message 7.} & A \rightarrow B : \left\{ \left(\left(A \stackrel{N_a}{\rightleftharpoons} B, B \models A \stackrel{N_b}{\rightleftharpoons} B \right), N_b \right) \right\}_{K_b}
\end{array}
$$

A and B hold the following initial assumptions:

$$
\begin{array}{ll}
A \models \#(N_a) & B \models \#(N_b) \\
A \models A \stackrel{N_a}{\rightleftharpoons} B & B \models A \stackrel{N_b}{\rightleftharpoons} B \\
A \models \stackrel{K_a}{\mapsto} A & B \models \stackrel{K_b}{\mapsto} B \\
A \models \stackrel{K_b}{\mapsto} B & B \models \stackrel{K_a}{\mapsto} A
\end{array}
$$

In constructing interpretation rules to match the idealization above, we would first run into trouble because Messages 3 and 6 have the same structure (two fields). We can handle this if we assume principals and nonces are distinguishable, by applying the automatic tagging approach mentioned above.

A more interesting problem arises, however, when we look at the interpretation required for Message 6. The following rule appears to produce the desired idealization:

$$
\frac{A \models B \mid\approx [N_1.N_2]}{A \models B \mid\approx \left(A \stackrel{N_2}{\rightleftharpoons} B, N_1 \right)}
$$

Certainly A can determine that the concrete Message 6 came from B, since it contains the secret N_a, and thus the interpretation succeeds. However, when we apply the honesty check to B, we find that B cannot be certain that N_a is a secret shared by A and B, since it has no authenticated message from A to that effect. Thus, B fails the honesty check for Message 6.

This honesty failure corresponds to the attack discovered by Lowe [Low95]. In this man-in-the-middle attack, an intruder participates in two concurrent runs of the protocol. In one run, the intruder conducts a legitimate transaction with

A, using his true identity, while in the other run, he successfully masquerades as A to another principal, B. The honesty property and interpretation restrictions allow us to protect against such multiple-run attacks without explicitly representing separate protocol runs. Multiple-run attacks typically rely on messages being misinterpreted when they are received in a different context (run) than the one in which they were produced. Millen has proven that some protocols are vulnerable to parallel attacks (those involving concurrent runs in which the same principal plays the same role in different runs), despite being secure against all non-parallel attacks [Mil99]. The RV interpretation restrictions ensure that each message is interpreted consistently regardless of which run of the protocol it appears in, and thus the honesty properties derived apply across all protocol runs.

4.3 Secrecy

Critics of belief logics often note that the logics do not address secrecy properties. It was observed very early that BAN reports no problems with a protocol in which secret keys are transmitted in the clear [Nes90]. Indeed, traditional BAN analysis omits plaintext parts of messages from consideration since they "contribute nothing to the meaning of the protocol."

BAN adherents sometimes assume that this shortcoming is easily overcome since one can determine by inspection whether sensitive data is sent in the clear or to untrusted parties. However, this argument ignores the subtle part of secrecy properties: determining exactly what things must be kept secret, and from whom. The Needham-Schroeder public key flaw Lowe discovered is fundamentally a secrecy flaw: A reveals a secret, N_B, to B without being certain that B is allowed to know it. This flaw remained undiscovered for over fifteen years, suggesting that secrecy can be just as subtle as authentication.

4.3.1 Formalization

Function	Notation	Meaning
$maysee(P, X)$	P **maysee** X	P is allowed to see X
I	I	an intruder

Figure 4.5: Functions of RV for supporting secrecy checks.

The class *responsibility* properties includes both the honesty properties discussed earlier and secrecy properties. These are properties that require principals to avoid revealing information that may be secret.

Like honesty, secrecy can be formalized within the belief-logic context in a way that is amenable to theory generation. Central to this formalization of secrecy is the notion that a principal P *may see* (is allowed to see) some message X. We introduce a new operator to reflect this: $P\ maysee\ X$. For instance, if A and B share a key K_{ab}, they will both normally believe $A\ maysee\ K_{ab}$ and $B\ maysee\ K_{ab}$. We will also introduce a special principal, I, corresponding to an intruder, and consider $maysee(I, X)$ as equivalent to $\forall P.\ P\ maysee\ X$. Principals may derive $maysee$ by applying any of several new rules:

$$\mathbf{RV33}: \frac{P \triangleleft Q \overset{K}{\longleftrightarrow} R}{P \models Q \textbf{ maysee } K} \qquad \mathbf{RV34}: \frac{P \triangleleft Q \overset{Y}{\rightleftharpoons} R}{P \models Q \textbf{ maysee } Y}$$

$$\mathbf{RV35}: \frac{P \triangleleft \overset{K_1}{\mapsto} Q \qquad K_1 = K_2^{-1}}{P \models Q \textbf{ maysee } K_2}$$

$$\mathbf{RV36}: \frac{P \triangleleft (Q \textbf{ maysee } Y)}{P \models Q \textbf{ maysee } Y} \qquad \mathbf{RV37}: \frac{P \triangleleft (Q \triangleleft Y)}{P \models Q \textbf{ maysee } Y}$$

$$\mathbf{RV38}: \frac{P \models Q \textbf{ maysee } X}{P \models Q \textbf{ maysee } \{X\}_K} \qquad \mathbf{RV39}: \frac{P \models Q \textbf{ maysee } (X, Y)}{P \models Q \textbf{ maysee } X}$$

$$\mathbf{RV40}: \frac{P \models Q \textbf{ maysee } X \qquad P \models Q \textbf{ maysee } Y}{P \models Q \textbf{ maysee } [X.Y]}$$

$$\mathbf{RV41}: \frac{P \models I \textbf{ maysee } X}{P \models Q \textbf{ maysee } X}$$

We further introduce four rules for determining what messages may safely be seen by an intruder:

$$\mathbf{RV42}: \frac{P \triangleleft \overset{K}{\mapsto} Q}{P \models I \textbf{ maysee } K}$$

$$\mathbf{RV43}: \frac{P \models Q \textbf{ maysee } X \qquad P \models \overset{K}{\mapsto} Q}{P \models I \textbf{ maysee } \{X\}_K}$$

$$\mathbf{RV44}: \frac{P \models Q \textbf{ maysee } X \qquad P \models R \textbf{ maysee } X \qquad P \models Q \overset{K}{\longleftrightarrow} R}{P \models I \textbf{ maysee } \{X\}_K}$$

Given these definitions, the secrecy property can be expressed quite simply:

> **Secrecy property**: For every message M that a principal P sends, P believes $maysee(I, M)$ at the point at which P sent M.

The rules above for deriving $maysee$ are conservative. For instance, it would be safe for a principal to derive $maysee(I, X)$ if it sees X unencrypted. This would require introducing a new operator similar to $sees$, and some corresponding rules. In practice, it seems that these additions are not necessary.

4.3.2 Example

We can illustrate secrecy too in the context of the Needham-Schroeder public key protocol:

Message 3. $A \to B : \{[N_a.A]\}_{K_b}$
Message 6. $B \to A : \{[N_a.N_b]\}_{K_a}$
Message 7. $A \to B : \{N_b\}_{K_b}$

Suppose that, in an attempt to repair the honesty failure observed in Section 4.2.3, we give Message 6 a more conservative interpretation:

$$\frac{A \lhd [N_1.N_2]}{A \lhd (N_2, N_1)}$$

With this interpretation, A does not need to believe that B was the sender in order to interpret the message, but interpretation restriction I3 (from Section 4.1.4) also prevents A from concluding anything about B. In particular, A cannot decide from this message that N_2 is a secret between A and B.

As a result of this lack of information, A will run afoul of the secrecy check in sending Message 7. In order for A to safely send the message $\{N_b\}_{K_b}$, secrecy requires that we can derive

$$A \models B \textbf{ maysee } N_b \ .$$

Since A does not know the origin of Message 6, it cannot be sure that it is safe to reveal N_b to B, so secrecy is violated.

In Chapter 5, we present a variety of examples in which the RV logic is applied to both correct and incorrect protocols.

4.4 Feasibility

Since the RV logic represents protocols at a concrete level, it is well suited to expressing what we call *feasibility properties*. This is a class of simple properties that represent the ability of the parties in a protocol to carry out their assigned roles. In order for a protocol to be feasible, it must be guaranteed at each step that the party who is to send the next message knows all the information required to produce that message. This may involve decrypting and disassembling earlier messages it has received, and constructing and encrypting the new message. Checking these properties does not tend to reveal subtle protocol flaws, but is rather a simple sanity check that is useful for early detection of mistakes in the protocol specification.

Function	Notation	Meaning
$can\_produce(P, X)$	P **canProduce** X	P can generate the message X

Figure 4.6: Function of RV for supporting feasibility checks.

The rules for deriving $can\_produce$ are straightforward. A principal can produce any message (or message fragment) it has seen:

$$\mathbf{RV45}: \frac{P \lhd M}{P \ \textbf{canProduce} \ M}$$

A principal can concatenate messages it can produce:

$$\mathbf{RV46}: \frac{P \ \textbf{canProduce} \ X \qquad P \ \textbf{canProduce} \ Y}{P \ \textbf{canProduce} \ X.Y}$$

Finally, a principal can encrypt a message using a key it can produce:

$$\mathbf{RV47}: \frac{P \ \textbf{canProduce} \ X \qquad P \ \textbf{canProduce} \ K}{P \ \textbf{canProduce} \ \{X\}_K}$$

To check feasibility for a protocol, we verify for each message M_i to be sent by P_i, that the protocol prefix $\{M_0, \ldots, M_{i-1}\}$ implies

$$P_i \ \textbf{canProduce} \ M_i \ .$$

Chapter 5 contains an example feasibility check.

4.5 Summary

We have now presented the full RV logic, which provides the following significant features:

- It provides *explicit interpretations*, through which a protocol can be specified at a concrete level and reasoned about using abstract belief constructs. A set of interpretation restrictions serves to prevent various kinds of flawed or hazardous interpretations.

- It expresses traditional *belief properties*, with added confidence due to the formal interpretation step.

- It expresses *honesty properties*, which assert that principals participating in a protocol are not required to send any messages whose interpretations they do not believe. In conjunction with the interpretation restrictions, this catches some protocol flaws resulting from insufficiently precise messages.

- It expresses *secrecy properties*, which show that a protocol does not require its participants to transmit any information that might be a secret. This check is performed through the positive, monotonic derivation of *maysee* properties, rather than by proving the absence of secret leaks.

- It expresses *feasibility properties*, which ensure that each participant in a protocol is capable of constructing the messages it is require to transmit.

In the next chapter, we explore the practical use of RV in protocol verification with theory generation.

Chapter 5

Theory Generation for RV

In this chapter we describe the practical application of the theory generation approach to the RV logic described in Chapter 4. We explain a general step-by-step procedure for analyzing a protocol in RV using theory generation, and then work through several protocol analyses in the context of RV. The focus here is on honesty and secrecy properties, since the examples in Chapter 3 addressed traditional belief properties for BAN, AUTLOG, and Kailar's logic of accountability.

5.1 Technique

To apply theory generation with the RV logic, we must first express the protocol under examination in a form suitable as input to TG_ℓ. A protocol specification in RV has five parts:

- The sequence of concrete messages, $M_1 \ldots M_n$, and their respective senders and receivers, $s_1 \ldots s_n$ and $r_1 \ldots r_n$. In contrast to the BAN approach, it is important to specify the sender as well as the receiver of each message, so that the responsibility of senders and the protocol's feasibility can be checked.

- The initial assumptions held by each principal. As in BAN, these typically include assumptions of freshness, jurisdiction, shared secrets, and keys and their functions. In addition, RV specifications may contain $maysee$, $public$, and $bind$ assumptions.

- The set of added interpretation rules required by the protocol.

- The belief goals of the protocol, that is, the beliefs each principal is expected to hold after a successful run. A whole class of protocols may share similar belief goals.
- The set of constants whose values are fixed across all runs of the protocol. (Often this set is empty, but in some cases the identity of a trusted server, for instance, may be fixed.)

When analyzing an existing protocol, one can usually determine the concrete messages sent and received easily. The appropriate belief goals may be somewhat less obvious, but they are normally shared by similar protocols. The most challenging parts of the specification for the human to produce are the initial assumptions and the added interpretation rules. The process of generating these corresponds to constructing a BAN-like idealization, so it is not surprising that it requires some ingenuity. With RV, however, mistakes at this step will normally lead to verification failures (and revision of the interpretations), rather than to a silently faulty analysis.

Given an RV specification in the form above, we must first check that the provided interpretation rules meet the conditions described in Section 4.1.4; they must each be of the form S1, S2, or S3, and satisfy restrictions I1–I4. These conditions can be verified mechanically.

Next, we can use theory generation, with the RV rules and the additional interpretation rules, to produce representations of the protocol state at each step. First, we run the TG_ℓ algorithm with only the initial assumptions as input, producing a theory representation, P_0. Then, we add a formula representing the receipt of the first message:

$$r_1 \lhd M_1$$

Running TG_ℓ on the theory representation, P_0, and this additional formula will produce P_1, representing the state of the protocol after the first message is sent. We then generate $P_2 \ldots P_n$ in the same way. In Chapter 6, we discuss an optimization for doing such incremental theory generation efficiently.

With these P_i protocol states in hand, we can use the decision procedure from Chapter 2 to verify the honesty properties quickly. For each $i < n$, we verify that P_i entails the corresponding honesty property:

$$s_{i+1} \models \textbf{legit}(M_{i+1})$$

When an honesty test fails, the first thing to check is whether

$$s_{i+1} \models M'_{i+1}$$

holds, where M'_{i+1} is the intended idealized meaning of M_{i+1}. If this belief does not hold, the real problem likely lies earlier in the protocol; if it does hold, the interpretation used for this message may be faulty.

After checking honesty, we can proceed to secrecy. We use the decision procedure again to check that for each $i < n$, P_i entails the corresponding secrecy property:

$$s_{i+1} \models \textbf{public}(M_{i+1})$$

If a secrecy check fails, examining the set of formulas in P_i of the forms

$$s_{i+1} \models \textbf{public}(X)$$
$$s_{i+1} \models P \textbf{ maysee } X$$

may suggest the problem.

Next, we can check feasibility properties for the protocol, in a similar manner. For each $i < n$, we use the decision procedure to check that P_i entails

$$s_{i+1} \textbf{ canProduce } M_{i+1}$$

When a feasibility check fails, the problem may be that part of the concrete protocol has been omitted in the specification. Checking P_i for formulas of the form

$$s_{i+1} \lhd X$$

should indicate which key, nonce, or name the sender cannot produce.

Finally, we can check P_n for the stated belief goals of the protocol, as in the examples in Chapter 3, and perform any further desired analyses, such as tracing the development of beliefs held by a participant as the protocol progresses.

5.2 Applications

To illustrate the practical use of both the theory generation technique and the RV logic, we now explore three published authentication protocols using these tools. The highlights of these analyses are presented below.

5.2.1 Denning-Sacco Public-key Protocol

The Denning-Sacco protocol allows A to generate a shared key for communication with B, and to securely hand that key to B, where a server, S, provides trusted public-key certificates for A and B. The concrete messages are exchanged as follows:

$$
\begin{array}{ll}
M_1. & A \rightarrow S : [A.B] \\
M_2. & S \rightarrow A : \left[\{[B.K_b.T_s]\}_{K'_s} . \{[A.K_a.T_s]\}_{K'_s}\right] \\
M_3. & A \rightarrow B : \left[\{[K_{ab}.T_a]\}_{K'_a} . \{[B.K_b.T_s]\}_{K'_s} . \{[A.K_a.T_s]\}_{K'_s}\right]
\end{array}
$$

In the first message, A gives the server (S) a hint as to what public-key certificates it needs. S responds with timestamped, signed certificates giving A's and B's public keys. In the third message, A sends both certificates to B, along with a signed and timestamped message containing a shared key, K_{ab}, which A has generated. The following initial assumptions are fairly routine:

$$
\begin{array}{ll}
A \models \stackrel{K_a}{\mapsto} A & A \models \stackrel{K_s}{\mapsto} S \\
B \models \stackrel{K_b}{\mapsto} B & B \models \stackrel{K_s}{\mapsto} S \\
S \models \stackrel{K_a}{\mapsto} A & S \models \stackrel{K_b}{\mapsto} B \\
S \models \stackrel{K_s}{\mapsto} S & A \models A \stackrel{K_{ab}}{\longleftrightarrow} B \\
S \models \#(T_s) & A \models \#(T_a) \\
A \models S \textbf{ controls } \stackrel{K}{\mapsto} B & B \models S \textbf{ controls } \stackrel{K}{\mapsto} A \\
K_a = K_a'^{-1} & K_b = K_b'^{-1} \\
K_s = K_s'^{-1} &
\end{array}
$$

The following, however, are somewhat unusual:

$$
\begin{array}{ll}
B \models A \textbf{ controls } A \stackrel{K}{\longleftrightarrow} B & A \models \#(T_s) \\
B \models \#(T_s) & B \models \#(T_a)
\end{array}
$$

These assumptions assert that B trusts A to generate shared keys for the two of them, and that A and B believe that A and S's timestamps are fresh (and thus that their clocks are reasonably synchronized).

The participants in this protocol must be able to interpret signed public-key certificates, and B must be able to interpret the signed message in which A sends their shared key. To accomplish this, we propose the following two interpretation rules:

$$
\frac{P \models Q \mid\!\approx [R.K.T]}{P \models Q \mid\!\approx \stackrel{K}{\mapsto} R} \qquad \frac{P \models Q \mid\!\approx [K.T]}{P \models Q \mid\!\approx P \stackrel{K}{\longleftrightarrow} Q}
$$

As in most key exchange protocols, at the end of the protocol, we want A and B to both believe they have a good shared key, K_{ab}, and if possible, we want them each to believe that the other claims this key to be good. This gives us the following standard belief goals:

$$\begin{array}{ll} A \models A \stackrel{K_{ab}}{\longleftrightarrow} B & \qquad A \models B \mid\!\approx A \stackrel{K_{ab}}{\longleftrightarrow} B \\ B \models A \stackrel{K_{ab}}{\longleftrightarrow} B & \qquad B \models A \mid\!\approx A \stackrel{K_{ab}}{\longleftrightarrow} B \end{array}$$

To complete the RV specification of this protocol, we will assume no constants need be fixed across protocol runs.

We now proceed with the protocol analysis. First, we note that the interpretation rules are both of the form S2, and both satisfy conditions I1–I4, so they are acceptable, and we can generate the incremental protocol-state theories P_0, P_1, P_2, P_3. We are now ready to attempt the honesty check.

For message M_1, the honesty check is trivial, since unencrypted messages are always believed legitimate. For message M_2, we use the decision procedure to verify that

$$S \models \textbf{legit}(\Big[\{[B.K_b.T_s]\}_{K_s'} \cdot \{[A.K_a.T_s]\}_{K_s'}\Big])$$

follows from P_1. The derivation of this result follows from the fact that S believes K_s is its public key, and that the interpretation of that message is independent of its recipient.

For message M_3, the honesty check fails. A quick examination of the generated theory P_2 reveals that

$$A \models \textbf{legit}(\{[K_{ab}.T_a]\}_{K_a'})$$

does not hold. The reason is that A believes this message is signed for any recipient (not just B), yet the (only) available interpretation for the message depends on the recipient. This failure corresponds to a genuine flaw in the protocol, noted by Abadi and Needham [AN96]. Since A cannot be certain who will interpret this message, B could replay it in another run of the protocol, and thus impersonate A. The fix suggested by Abadi and Needham is to replace message M_3 with the more explicit

$$M_3. \quad A \to B : \Big[\{[A.B.K_{ab}.T_a]\}_{K_a'} \cdot \{[B.K_b.T_s]\}_{K_s'} \cdot \{[A.K_a.T_s]\}_{K_s'}\Big]$$

With this version of the protocol, we can replace the two interpretations used above with the following receiver-independent interpretation, which allows the honesty check to succeed:

$$\frac{P \models Q \mid\!\approx [Q.R.K.T]}{P \models Q \mid\!\approx Q \stackrel{K}{\longleftrightarrow} R}$$

With the corrected protocol, we can run the secrecy checks, and find that they all pass:

$$P_0 \vdash A \models \textbf{public}([A.B])$$
$$P_1 \vdash S \models \textbf{public}(\left[\{[B.K_b.T_s]\}_{K'_s}.\{[A.K_a.T_s]\}_{K'_s}\right])$$
$$P_2 \vdash A \models \textbf{public}(\left[\{[A.B.K_{ab}.T_a]\}_{K'_a}.\{[B.K_b.T_s]\}_{K'_s}.\{[A.K_a.T_s]\}_{K'_s}\right])$$

The only nontrivial secrecy check is for M_3, in which A transmits K_{ab}. Since it is encrypted under K_b, and

$$P_2 \vdash A \models B \textbf{ maysee } K_{ab}$$
$$P_2 \vdash A \models \overset{K_b}{\mapsto} B$$

it follows that A believes M_3 to be public.

The feasibility property and belief goals follow straightforwardly, with the exception of the third belief goal,

$$A \models B \mid\approx A \overset{K_{ab}}{\longleftrightarrow} B$$

which cannot hold since A never receives a message from B. Despite the fact that A cannot be sure B knows or believes in the shared key, the other three standard belief goals have been met, so the protocol has accomplished something. Second-order beliefs of this sort are unnecessary in some contexts, and by sacrificing them we can often reduce the number of messages that must be sent. This limitation should always be kept in mind, however, particularly when the protocol's context changes.

5.2.2 Neuman-Stubblebine Shared-key Protocol

The Neuman-Stubblebine protocol allows mutual authentication of two parties (A and B) in four messages, using shared keys and without relying on synchronized clocks [NS93]. Each participant shares a long-term key with the trusted server, which generates and distributes a session key for each run of the protocol.

The protocol consists of the following four concrete messages:

$$M_1.\quad A \to B : [A.N_a]$$
$$M_2.\quad B \to S : \left[B.N_b.\{[A.N_a.T_b]\}_{K_{bs}}\right]$$
$$M_3.\quad S \to A : \left[\{[B.N_a.K_{ab}.T_b]\}_{K_{as}}.\{[A.K_{ab}.T_b]\}_{K_{bs}}.N_b\right]$$

$$M_4.\quad A \to B : \left[\{[A.K_{ab}.T_b]\}_{K_{bs}} . \{N_b\}_{K_{ab}}\right]$$

In message M_1, A sends B an unauthenticated hint that it would like to initiate a session, where A identifies that session by the nonce N_a. B then sends the server (S) a message containing B's name, the nonce, N_b, by which B identifies this session, and a final encrypted fragment containing A's name and nonce as well as a timestamp (T_b). In the third message, S sends three fragments directly to A: an encrypted message for A containing the session key K_{ab} and identifying information, a similar encrypted message for B, and in the clear, B's nonce, N_b. A forwards the second part verbatim to B, and attaches N_b encrypted under the session key, to demonstrate that A now has that key.

We use the following initial assumptions, which are typical of shared-key protocols:

$$\begin{array}{ll}
A \models A \xleftrightarrow{K_{as}} S & B \models B \xleftrightarrow{K_{bs}} S \\
S \models A \xleftrightarrow{K_{as}} S & S \models B \xleftrightarrow{K_{bs}} S \\
S \models A \xleftrightarrow{K_{ab}} B & \\
A \models \#(N_a) & B \models \#(N_b) \\
 & B \models \#(T_b) \\
A \models S \textbf{ controls } A \xleftrightarrow{K} B & B \models S \textbf{ controls } A \xleftrightarrow{K} B
\end{array}$$

Since there are no secrets in this protocol, other than the assorted keys, we can declare all principal names, nonces, and timestamps public. This produces the following additional initial assumptions:

$$\begin{array}{lll}
A \models \textbf{public}(A) & B \models \textbf{public}(A) & S \models \textbf{public}(A) \\
A \models \textbf{public}(B) & B \models \textbf{public}(B) & S \models \textbf{public}(B) \\
A \models \textbf{public}(N_a) & B \models \textbf{public}(N_a) & S \models \textbf{public}(N_a) \\
A \models \textbf{public}(N_b) & B \models \textbf{public}(N_b) & S \models \textbf{public}(N_b) \\
A \models \textbf{public}(T_b) & B \models \textbf{public}(T_b) & S \models \textbf{public}(T_b)
\end{array}$$

We could instead add these assumptions on-demand, as the verification requires them. This would yield a somewhat smaller set of assumptions, which is in general a desirable goal. We could further reduce this set of assumptions if the RV logic were extended to allow principals to infer, for instance, that data received unencrypted must be public (as suggested in Section 4.3.1).

Now we can construct the necessary interpretation rules for this protocol. We start with the most obvious message fragments: the parts of M_3 and M_4 encrypted under K_{as} and K_{bs} must (at least) convey the belief $A \xleftrightarrow{K_{ab}} B$, so we produce the corresponding interpretation rules:

$$\mathbf{NSt1}: \frac{P \models Q \mid\approx [R.N.K.T]}{P \models Q \mid\approx P \xleftrightarrow{K} R} \qquad \mathbf{NSt2}: \frac{P \models Q \mid\approx [R.K.T]}{P \models Q \mid\approx P \xleftrightarrow{K} R}$$

We turn now to the second part of message M_4: $\{N_b\}_{K_{ab}}$. Intuitively, this message confirms for B that A has the session key, as A has encrypted the fresh nonce N_b using it. If we were doing a simple BAN analysis of the protocol, we might be tempted to idealize this message fragment as

$$\left\{A \xleftrightarrow{K_{ab}} B\right\}_{K_{ab}}$$

so that we achieve this final result:

$$B \models A \models A \xleftrightarrow{K_{ab}} B$$

In RV, though, we are forced to be more careful. As the protocol description stands, there is not enough information in the concrete message $\{N_b\}_{K_{ab}}$ to produce the idealized version above: B's name is not given explicitly. We could try binding N_b to B, but there is no message that will allow A to trust that binding. Thus when B sees this message, he knows that A has the key K_{ab}, but he cannot be certain that A believes she shares that key with B. We can reach this weaker conclusion without needing an extra interpretation rule:

$$B \models A \mid\approx \{N_b\}_{K_ab}$$

Will the two interpretation rules above ($\mathbf{Nst1}$ and $\mathbf{Nst2}$) suffice, or are more necessary? The encrypted component of message M_2 is not yet interpreted. Where encryption occurs in a protocol, it normally serves to convey beliefs (in which case an interpretation is normally needed), or to conceal secrets, or both. M_2 contains no secrets, so we expect that it is intended to convey beliefs. Since it is unclear what beliefs might be conveyed by this message or required by the verification, we will leave the message uninterpreted for now.

Given the interpretations above, we can check the honesty property for this protocol as described in Section 5.1. Messages M_3 and M_4 present no problems. The server believes $A \xleftrightarrow{K_{ab}} B$, and believes further that K_{as} and K_{bs} are keys shared with A and B, respectively, so we can derive

$$S \models \textbf{legit}(\{[B.N_a.K_{ab}.T_b]\}_{K_{as}})$$
$$S \models \textbf{legit}(\{[A.K_{ab}.T_b]\}_{K_{bs}})$$

We run into trouble when checking honesty for the encrypted part of M_2. This message,

$$\{A.N_a.T_b\}_{K_{bs}}$$

can be confused with the first part of M_4:

$$\{A.K_{ab}.T_b\}_{K_{bs}}$$

The honesty check on M_2 reveals this because it applies the M_4 interpretation and requires the following belief, which does not hold:

$$B \models B \overset{N_a}{\longleftrightarrow} S$$

This could correspond to a real vulnerability in some implementations. An eavesdropper could record M_2 and replay the encrypted part in M_4, thus convincing B that N_a is the session key. Since the eavesdropper can read the value of N_a from M_1, it is now able to masquerade as A to B. The simplest solution to this problem is to tag the encrypted message in M_2 to distinguish it from the messages the server sends. After this fix is applied, the honesty check succeeds.

Having checked honesty, we can proceed to secrecy. The secrecy properties for the Neuman-Stubblebine protocol are straightforward to check and not particularly interesting since the only secret ever transmitted is the session key, so we will move on to feasibility.

The most interesting message for the feasibility check is M_4. We must verify that A can produce

$$\left[\{[A.K_{ab}.T_b]\}_{K_{bs}} . \{N_b\}_{K_{ab}}\right]$$

We can derive this from the information A receives in M_3. First, we derive that A sees the first component of M_4 by extracting the second part of M_3:

$$\frac{A \triangleleft \left[\{[B.N_a.K_{ab}.T_b]\}_{K_{as}} . \{[A.K_{ab}.T_b]\}_{K_{bs}} .N_b\right]}{A \triangleleft \{[A.K_{ab}.T_b]\}_{K_{bs}}}$$

Second, we derive that A sees K_{ab} by decrypting the first part of M_3:

$$\frac{\dfrac{A \triangleleft \{[B.N_a.K_{ab}.T_b]\}_{K_{as}} \qquad A \triangleleft K_{as}}{A \triangleleft [B.N_a.K_{ab}.T_b]}}{A \triangleleft K_{ab}}$$

Finally, we use similar steps to show that A can produce $\{N_b\}_{K_{ab}}$, and can concatenate that with the forwarded part of M_3 to produce the full message M_4.

These belief goals for Neuman-Stubblebine follow quickly from the interpretations and concrete protocol given above:

$$A \models A \overset{K_{ab}}{\longleftrightarrow} B$$
$$B \models A \overset{K_{ab}}{\longleftrightarrow} B$$
$$B \models A \mid\approx \{N_b\}_{K_{a}b}$$

Ideally, we would like A to have some assurance that B has at least participated in this run of the protocol. (It is unrealistic to expect A to believe much more than that about B since B sends no messages after receiving the session key.) Examining the concrete protocol, we see that A receives some information indirectly from B: both the nonce, N_b, and the timestamp, T_b. Can A be sure that either of these was recently uttered by B? She can if we interpret the message S sends her a bit differently. Consider the following interpretation, which replaces the existing interpretation with the same premise:

$$\frac{P \models Q \mid\approx [R.N.K.T]}{P \models Q \mid\approx \left(P \overset{K}{\longleftrightarrow} R, R \mid\sim N\right)}$$

Here the server is affirming that B uttered N_a. We add the initial assumption,

$$A \models S \textbf{ controls } B \mid\sim X \ ,$$

indicating that A trusts S to tell her what B said. We can then use A's belief that N_a is fresh to reach a new conclusion:

$$A \models B \mid\approx N_a$$

Now both A and B have some assurance of each other's presence in the protocol run. Note that we have finally made use of the encryption in M_2; without it, S would not believe $B \mid\sim N_a$, and could not pass the honesty test with this newly added interpretation rule.

5.2.3 Woo-Lam Public-key Protocol

In 1992, Woo and Lam proposed the following "peer-peer authentication protocol," which uses public-key cryptography to establish a shared session key for communication between A and B, who trust S to generate the key and issue signed public-key certificates [WL92a, WL92b]:

$$
\begin{aligned}
&M_1. \quad A \rightarrow S : [A.B] \\
&M_2. \quad S \rightarrow A : \{[B.K_b]\}_{K'_s} \\
&M_3. \quad A \rightarrow B : \{[N_a.A]\}_{K_b} \\
&M_4. \quad B \rightarrow S : \left[B.A.\{N_a\}_{K_s}\right] \\
&M_5. \quad S \rightarrow B : \left[\{[A.K_a]\}_{K'_s}.\left\{\{[N_a.K_{ab}.A.B]\}_{K'_s}\right\}_{K_b}\right] \\
&M_6. \quad B \rightarrow A : \left\{\left[\{[N_a.K_{ab}.A.B]\}_{K'_s}.N_b\right]\right\}_{K_a} \\
&M_7. \quad A \rightarrow B : \{N_b\}_{K_{ab}}
\end{aligned}
$$

Here, A notifies S that she wishes to communicate with B. S provides a certificate of B's public key (K_b), then A sends a nonce (N_a) to B, using K_b. In M_4, B sends A's nonce, encrypted, to S, who responds with a certificate of A's public key and a four-part message giving the session key (K_{ab}). B decrypts the signed session key message, adds a challenge (N_b), re-encrypts it for A, and sends it along. Finally, A responds to B's challenge by encrypting it under the session key.

The participants will hold the expected initial assumptions regarding keys:

$$
\begin{array}{ll}
A \models \stackrel{K_a}{\mapsto} A & A \models \stackrel{K_s}{\mapsto} S \\
B \models \stackrel{K_b}{\mapsto} B & B \models \stackrel{K_s}{\mapsto} S \\
S \models \stackrel{K_a}{\mapsto} A & S \models \stackrel{K_b}{\mapsto} B \\
S \models \stackrel{K_s}{\mapsto} S & S \models A \stackrel{K_{ab}}{\longleftrightarrow} B \\
A \models S \textbf{ controls } \stackrel{K}{\mapsto} B & B \models S \textbf{ controls } \stackrel{K}{\mapsto} A \\
A \models S \textbf{ controls } A \stackrel{K}{\longleftrightarrow} B & B \models S \textbf{ controls } A \stackrel{K}{\longleftrightarrow} B \\
K_a = K_a'^{-1} & K_b = K_b'^{-1} \\
K_s = K_s'^{-1} &
\end{array}
$$

Both A and B will generate secret nonces satisfying these assumptions:

$$
\begin{array}{ll}
A \models \#(N_a) & A \models A \stackrel{N_a}{\rightleftharpoons} B \\
B \models \#(N_b) & B \models A \stackrel{N_b}{\rightleftharpoons} B
\end{array}
$$

A will also trust the server not to disclose its nonce:

$$A \models S \textbf{ maysee } N_a$$

Before going further with this protocol, we should make a correction. The signed public-key certificates the server sends in M_2 and the first part of M_5 contain no timestamps, nonces, or other indications of their freshness. This will cause the belief-goal verification, as well as other tests, to fail. In practical terms, this means that once a principal's public key is compromised, an adversary who has stashed away the appropriate certificate can pose as that principal forever. This error would be caught in a standard BAN analysis, so we will not explore it further. We propose the following corrected protocol:

$$
\begin{array}{ll}
M_1. & A \rightarrow S : [A.B.T_a] \\
M_2. & S \rightarrow A : \{[B.K_b.T_a]\}_{K_s'} \\
M_3. & A \rightarrow B : \{[N_a.A]\}_{K_b} \\
M_4. & B \rightarrow S : \left[B.A.\{[N_a.T_b]\}_{K_s}\right] \\
M_5. & S \rightarrow B : \left[\{[A.K_a.T_b]\}_{K_s'}.\left\{\{[N_a.K_{ab}.A.B]\}_{K_s'}\right\}_{K_b}\right] \\
M_6. & B \rightarrow A : \left\{\left[\{[N_a.K_{ab}.A.B]\}_{K_s'}.N_b\right]\right\}_{K_a} \\
M_7. & A \rightarrow B : \{N_b\}_{K_{ab}}
\end{array}
$$

This protocol differs from the original in that timestamps T_a and T_b have been added to messages M_1, M_2, M_4, and M_5. A single server-assigned timestamp could be used instead, but that would require synchronized clocks. In this version, A and B only check timestamps they issued themselves. We require two extra initial assumptions:

$$A \models \#(T_a) \qquad\qquad B \models \#(T_b)$$

Again, we can take the approach of constructing interpretations lazily, as the verification requires additional beliefs. The most obvious interpretation is for the message containing the session key (the second part of M_5):

$$\frac{P \models S \mid\!\approx [N.K.Q.R]}{P \models S \mid\!\approx Q \xleftrightarrow{K} R}$$

The interpretation of the server-generated public key certificates is also clear:

$$\frac{P \models S \mid\!\approx [Q.K.T]}{P \models S \mid\!\approx \overset{K}{\mapsto} Q}$$

The honesty and secrecy analyses for this protocol as stated are somewhat involved. We again let every principal name and timestamp be public. We can derive that all public keys are public from the **public** RV rules.

$$
\begin{array}{lll}
A \models \textbf{public}(A) & B \models \textbf{public}(A) & S \models \textbf{public}(A) \\
A \models \textbf{public}(B) & B \models \textbf{public}(B) & S \models \textbf{public}(B) \\
A \models \textbf{public}(T_a) & B \models \textbf{public}(T_a) & S \models \textbf{public}(T_a) \\
A \models \textbf{public}(T_b) & B \models \textbf{public}(T_b) & S \models \textbf{public}(T_b)
\end{array}
$$

We will look at each message briefly:

- M_1: No honesty or secrecy issues since there is no encryption and A and B are public.

- M_2: B and K_b are public, so secrecy is maintained. S believes the interpretation of the message ($\stackrel{K_b}{\mapsto} B$), so honesty is satisfied.

- M_3: From
$$A \models A \stackrel{N_a}{\rightleftharpoons} B$$
and
$$A \models \stackrel{K_b}{\mapsto} B$$
we can derive
$$A \models \textbf{public}(\{[N_a.A]\}_{K_b})$$
Honesty is satisfied trivially since there is no interpretation.

- M_4: Here we hit a secrecy snag: B does not believe S **maysee** N_a. We can safely repair this by making the interpretation of M_4 include the statement S **maysee** N_a in its conclusion. Honesty is satisfied trivially.

- M_5: Reasoning for the first part is the same as for M_2. For the second part, S can derive B **maysee** K_{ab} and B **maysee** N_a if we interpret M_4 as stating $A \stackrel{N_a}{\rightleftharpoons} B$. Honesty follows from S's initial assumptions.

- M_6: The first part passes secrecy if we add the interpretation to M_5 that A **maysee** $[N_a.K_{ab}.A.B]$. For the second part, B can derive A **maysee** N_b since B believes $A \stackrel{N_b}{\rightleftharpoons} B$. Honesty is trivial.

- M_7: If we add the interpretation to M_6 that $A \stackrel{N_b}{\rightleftharpoons} B$ then we satisfy the secrecy property. Honesty is trivial.

The iterative process illustrated above, in which interpretations are refined and assumptions added as necessary to push the verification through allows us to more easily identify the assumptions and interpretations that are crucial and those that are redundant or useless. Backing up and retrying the analysis after each of these changes would be cumbersome if done by hand, but it is easy when assisted by a theory-generation-based verifier.

If we go through this process without the initial assumptions that A and B believe their nonces to be secret, the analysis becomes dramatically simpler, and we find that we can achieve the same set of belief goals without relying on the secrecy of those nonces. The concrete protocol is clearly designed to keep the nonces secret, so we might well ask what the consequences would be of removing the extra encryption. If we compare the generated theories for the original protocol with a modified protocol where nonces are passed in the clear, we find that the belief goals are unaffected by the change.

In fact Woo and Lam arrived at the same conclusion in a followup article two months after the original [WL92b]. They suggest a simplified, five-step protocol in which the nonces are not kept secret. This protocol meets the same belief goals as the original.

Chapter 6

Implementation

In this chapter, we discuss the REVERE protocol verification system and its implementation of the TG_ℓ algorithm, show how it supports standard belief logics as well as extended analysis with RV, and present some performance measurements for the system.

The REVERE tool contains a general-purpose theory generation core, but its interface is tailored to the protocol analysis domain. It uses RV as well as other belief logics, and the core could be reused to perform theory generation with any logic meeting the restrictions laid out in Chapter 2.

REVERE is written primarily in the Standard ML (SML) programming language [MTH90, MTHM97]. SML is a mostly functional, strongly typed language with exceptions and a sophisticated module system. It supports user-defined datatypes with a pattern matching syntax that allows very natural manipulation of formulas and expressions. A formal semantics for SML has been published, making it especially appropriate as a basis for building verification systems. REVERE makes significant uses of SML's module system, some of which are described in the following sections. We first address issues relating to the implementation of TG_ℓ itself.

6.1 TG_ℓ Algorithm Implementation

The TG_ℓ implementation within REVERE takes as input a module containing the rules and rewrites for a logic, and produces a module which is specialized to performing theory generation for that logic. This generated module includes a function (`closure`) which performs theory generation given an initial set of formulas

```
signature TGL =
  sig
    structure Logic : LOGIC
    type formula = Logic.RuleTypes.Fol.term
    type s_rule = Logic.RuleTypes.s_rule
    type theory_rep
    val closure     : s_rule list -> formula list
                          -> theory_rep
    val closure_add : theory_rep -> formula list
                          -> theory_rep
    val derivable   : theory_rep -> formula -> bool
    val all_formulas : theory_rep -> formula list
  end
```

Figure 6.1: Signature for the theory generation module

and an optional set of extra S-rules to use. The module matches the `TGL` signature given in Figure 6.1, and contains three functions in addition to `closure`.

The `closure_add` function takes an already-generated theory representation and a set of new formulas, and generates a representation of the theory induced by the old and new formulas together. This function is particularly useful for RV verifications, in which we must do theory generation for the initial assumptions alone, then the initial assumptions plus the first message, and so on. The implementation of `closure_add` is described in Section 6.1.1.

The `derivable` function takes a generated theory representation and a formula, and determines whether the formula is in the theory. This function is exactly the decision procedure presented in Section 2.4.2.

The purpose of `all_formulas` is simply to provide access to the set of formulas in a theory representation, so that other modules can compare theories, present them to the user, and store them.

Logics (such as BAN and RV) are represented by modules matching the `LOGIC` signature in Figure 6.2, which refers to the datatypes corresponding to S-rules, G-rules, and rewrites, laid out in the `RULETYPES` signature. A `LOGIC` module specifies the S-rules, G-rules, and rewrites, as well as the $\preceq$ pre-order (`no_larger`).

To create a theory generation module for a particular logic such as BAN, we apply the general-purpose `TGL` functor.

```
signature RULETYPES =
  sig
    structure Fol : FOL
    type formula = Fol.term
    datatype rewrite =
      Rewrite of  pair: formula * formula,
                   name: string
    datatype s_rule =
      S_Rule of  premises: formula list,
                  conclusions: formula list,
                  name: string
    datatype g_rule =
      G_Rule of  premises: formula list,
                  conclusion: formula,
                  name : string
  end
signature LOGIC =
  sig
    structure RuleTypes : RULETYPES
    type formula = RuleTypes.formula
    val S_rules  : RuleTypes.s_rule list
    val G_rules  : RuleTypes.g_rule list
    val rewrites : RuleTypes.rewrite list
    val no_larger : formula * formula -> bool
  end
```

Figure 6.2: Signature for describing ℓ_{RW} logics

```
  structure BanTGL : TGL =
      MakeTGL(structure Logic = Ban ...)
```

A *functor* is an SML construct that creates a module (structure) given other modules as parameters. In this case, the `Ban` structure is passed, along with some additional structure arguments omitted here. MakeTGL performs assorted checks and does some pre-computation to facilitate the theory generation process.

6.1.1 Data Structures and Optimizations

REVERE makes use of a few specialized data structures for representing formulas, rules, rewrites, and the relationships among them, and uses some optimizations and heuristics to speed up theory generation. We describe a few here.

Many operations in TG_ℓ manipulate sets of formulas. In particular, the set of formulas constituting the partially generated theory is added to by $closure$ and iterated over by $backward\_chain\_one$ in searching for formulas matching a given pattern (see Section 2.3). We accelerate the process of searching for matches by observing that the number of patterns ever matched against is limited: all patterns are instances of S- or G-rule premises, and many patterns are exactly those premises. We keep along with the set of formulas a table mapping each premise to all formulas in the set that match it. Then, when we need to find all matches for a pattern, P, we can accelerate the search by restricting it to those formulas known to match the premise of which P is an instance. Each time a formula is added to the set, it is matched against each of the premises and added to the appropriate sets. In principle, we could do further refinement of this table to include some patterns that are partially instantiated premises, but for the typical verifications we attempted, the overhead would not be justified.

To further accelerate theory generation, we make the implementation of the $apply\_srule$ function (Section 2.3.2) aware of which formulas are in the fringe. By propagating this information, we enable the function to select only those S-rule applications that make use of formulas from the fringe, since any other S-rule applications must already have been done. The representation described above requires us to match the new formulas against all premises already, so it costs little to add this optimization, and it eliminates a considerable number of redundant rule applications.

Since RV verification requires theory generation for each step of the protocol, we provide the `closure_add` function (Figure 6.1), for adding new formulas to a generated theory. A trivial implementation of `closure_add` could just invoke TG_ℓ on the set containing the old theory representation and the new formulas. To accelerate the search, however, we take advantage of the knowledge that the old theory representation is closed. We treat the new formulas as the fringe, and the old theory representation as a partial theory representation. This gives the $closure$ function a "head start" when combined with the `apply_srule` optimization above.

Finally, the process of unifying modulo rewrites involves frequent invocations of a function that applies rewrites to a term at the outermost level, in all possible ways. With many rewrites, this process can become quite expensive, so we use limited memoization to save some computation.

```
datatype protocol =
  REVEREPROTOCOL of
    { name : string,
      assumptions : Logic.formula list,
      messages : { sender : Logic.formula,
                   receiver : Logic.formula,
                   message  : Logic.formula } list,
      goals : Logic.formula list,
      interpretations : Logic.RuleTypes.s_rule list,
      ... }
```

Figure 6.3: SML datatype representing a protocol in REVERE

6.2 REVERE System Structure

Beyond the theory generation core, the REVERE system provides a simple verification environment tailored for protocols. The user interface is a menu-driven XEmacs package that interacts behind the scenes with the SML theory generation engine. Logics and protocols are represented internally as SML modules (structures); logics match the `LOGIC` signature given earlier, and protocols are values of the `protocol` type defined in Figure 6.3. The initial assumptions, messages, and desired belief goals are all given as formulas in the appropriate logic. For RV verifications, interpretations can be provided as a set of extra rules that will be used in theory generation.

A front-end parser converts user-written protocol specifications to these SML values. The protocol specification language is a variant of CAPSL, an emerging standard for cryptographic protocol description [Mil97].

The user interface provides different ways to sort, filter, and display generated theories, and allows comparing the generated theories for two protocols. There is a special RV support module providing interpretation validation as well as honesty, secrecy, and feasibility checks.

6.3 Performance

The REVERE theory generation implementation is simple—the core is roughly 1000 lines of code—and it could be further optimized in various ways as described in Chapter 7. Nonetheless, its performance on the examples we have used is

Logic	**Protocol**	**Elapsed TG Time**
BAN	Kerberos	4.7s
	Andrew RPC	3.2s
	Needham-Schroeder (Shared)	1.5s
	CCITT X.509	23.8s
	Wide-Mouth Frog	19.3s
	Yahalom	23.0s
AUTLOG	challenge-response 1	0.3s
	challenge-response 2	0.3s
	Kerberos	11.3s
Kailar's Accountability	IBS variant 1	0.3s
	IBS variant 2	0.3s
	SPX Auth. Exchange	0.2s
RV	Needham-Schroeder (Pub)	23.0
	Otway-Rees	34.4
	Denning-Sacco	38.1
	Neuman-Stubblebine	20.1s
	Woo-Lam	50.8s

Figure 6.4: Elapsed theory generation times in REVERE for various protocols. All timings were done on an Digital AlphaStation 500, with 500MHz Alpha 21164 CPU.

suitable for interactive use. The table in Figure 6.4 shows that the elapsed theory generation time for all examples was less than one minute. All other operations take insignificant time. It is certainly conceivable that logics and specifications in another domain could cause this theory generation implementation to exhibit unacceptable performance. The presence of very long sequences in messages or rules can cause slowdowns, as canonicalization and unification modulo rewrites become expensive. In Chapter 7 we discuss specialized support for associative-commutative operators that would significantly alleviate this problem. Logics containing a very large number of rules, and especially those in which G-rules play a dominant role in reasoning, could prove inefficient to manage as well; better rule-selection optimizations would be necessary to reduce this cost. For the cases we have studied, however, the theory generation speed was satisfactory, making these optimizations unwarranted.

Chapter 7

Conclusions and Future Directions

In this chapter, we conclude by summarizing the results of the thesis work, which support the two parts of the thesis claim from Section 1.2, and by suggesting avenues for further research, which range from minor algorithm tuning to broad areas for exploration.

7.1 Summary of Results

First, we conclude that theory generation is an effective method of automated reasoning. In support of this claim, we have produced several artifacts and made observations of them. We have described a simple algorithm (TG_ℓ) for producing finite representations of theories. This algorithm can be applied to any logic in the ℓ_{RW} class that also meets the preconditions in Definition 2.16. The theory representations produced by TG_ℓ are well suited to direct comparison since they are nearly canonical, and they can also be used in an efficient decision procedure. We have proved that the TG_ℓ algorithm and this decision procedure terminate and are correct.

As further evidence of the effectiveness of theory generation, we have produced a practical implementation including several optimizations that improve on the basic TG_ℓ algorithm. We have successfully incorporated this implementation into a new protocol analysis system, REVERE, where theory generation serves several purposes. REVERE verifies specific properties of protocols using the theory-generation-based decision procedure, compares protocols by displaying the difference between their respective theory representations, and does message-by-message analysis using incremental theory generation. Through many protocol analyses using four different logics, we have observed that, in practice, the theory generation completes quickly (in seconds or minutes), and that the theory representations generated are consistently of manageable size.

We support the second part of the thesis claim, that theory generation can be applied to security protocols to analyze a wide range of critical security properties, by various example applications of the REVERE tool. We used REVERE (and thus theory generation) to verify the properties that can be expressed by the three existing belief logics described in Chapter 3: the BAN logic of authentication, AUTLOG, and Kailar's accountability logic. These properties include, for idealized authentication and key exchange protocols, beliefs regarding public and shared keys, secrets, freshness, and statements made by other principals. For idealized electronic commerce protocols, we can also check non-repudiation properties ("A can prove X to B"). Using REVERE, we reproduced published protocol analyses using these belief logics, and in some cases we exposed errors in the earlier analyses.

Beyond these existing logics, we have developed a new logic, RV, which allows more grounded reasoning about protocols, in that the mapping from concrete messages to abstract meanings is made explicit. Using theory generation, we have applied this logic to several existing protocols, checking honesty, secrecy, feasibility, and interpretation validity properties. These properties are not fully addressed by other belief logics, and they are critical in that failure to check them can lead (and has led) to vulnerabilities. Like other belief logics, RV takes a constructive approach to protocol verification, in that it focuses on deriving positive protocol properties, rather than searching for attacks. Through extending this approach to honesty and secrecy properties, RV can expose flaws that correspond to concurrent-run attacks without explicitly modeling either an intruder or some set of runs. In the model checking approaches, one must typically establish bounds on the number of protocol runs, and sometimes also specify an intruder endowed with a specific set of capabilities. In return, however, when a property fails to hold, the model checkers can supply a counterexample, which in this domain usually correspond to attacks.

Through this work we have learned (or re-learned) other valuable lessons. Perhaps the most glaring is that formal arguments, no matter how trivial, should not be trusted until they have been subjected to mechanical verification. Again and again we find manual proofs that appear completely sound and yet rely on unstated assumptions or intuitively obvious steps that are not formally warranted. The development of a new logic, a particularly perilous undertaking, should always be done with the assistance of automated verification to ensure that the logic is truly powerful enough to prove what it is intended to prove. Of course, mechanical verification is not in itself sufficient to provide full confidence. It serves to complement human understanding of the formalized assumptions and conclusions at the "edges" of the verification process.

Finally, the theory generation approach has its limitations. The TG_ℓ preconditions can be somewhat cumbersome to establish, although in most cases they can be handled relatively easily by using standard pre-orders. Theory generation, and the TG_ℓ algorithm in particular, are not well-suited to all logics; for instance, logics encoding arithmetic and temporal logics are probably poor candidates. For some sets of rewrites, the unification modulo rewrites can be expensive; one typical example is a set of rewrites encoding associative and commutative properties, applied to assumptions or rules that involve long sequences. The enhancements to theory generation described in the next section could address some of these shortcomings.

7.2 Future Work

We can divide directions for future work into those relating to theory generation in general, and those applicable to the security domain in particular.

7.2.1 Theory Generation Refinements and Applications

The TG_ℓ algorithm itself could be enhanced, or its preconditions relaxed, in a variety of ways. We look at the termination guarantees first.

The purpose for most of the preconditions is to ensure that TG_ℓ will always terminate, but there is a tradeoff between making the preconditions easy to check and allowing as many logics as possible. The preconditions given in Definition 2.16 are sufficient but not necessary to ensure termination. We could replace them with the simple condition that each S-rule application must produce a formula no larger than any of the formulas used (perhaps through G-rules) to satisfy its premises. This imposes a considerable burden of proof, but it is a more permissive precondition than the one we use. Furthermore, while it is sufficient to ensure that a finite theory representation exists, it is not sufficient to ensure termination, as we must also prove that each S-rule application attempt must terminate. In some cases, this extra effort may be justified by the increased flexibility.

It can be tricky to construct the $\preceq$ pre-order, as Section 3.1.1 makes clear. However, in practice we may sometimes be fairly confident that a suitable pre-order exists but not want to go through the trouble of producing it. We can skip specifying the pre-order if we are willing to accept the risk of non-termination. Correctness will not be sacrificed, so if the algorithm terminates, we can be sure it has generated the right result.

As an alternative approach to ensuring termination, we could draw on existing research in automatic termination analysis for Prolog—for instance the approach proposed by Lindenstrauss and Sagiv [LS97]—to check whether a set of rules will halt. This would require either adjusting these methods to take account of our use of rewrites, or perhaps encoding the TG_ℓ algorithm itself as a Prolog program whose termination could be analyzed in the context of a fixed set of rules and rewrites.

To improve the performance of TG_ℓ, we could introduce a special case to handle associative-commutative rewrites efficiently, using known techniques for associative-commutative unification. The general-purpose unification modulo equalities implemented for REVERE has acceptable performance for the examples we ran, but it could become expensive when formulas or rules include long sequences. We might also get better performance in searching for formulas matching a given pattern by adapting the Rete algorithm used in some AI systems [For82].

The TG_ℓ algorithm could be modified quite easily to keep track of the proof of each derived formula. This information could prove useful in providing feedback to the user when the verification of some property fails; we could, for instance, automatically fill in "missing" side conditions in an attempt to push the proof through, and then display a proof tree with the trouble spots highlighted. The proofs could also be fed to an independent verifier to double-check the TG_ℓ results.

Thinking further afield, we might consider extensions such as providing theory representations other than the (R, R') representations described in Definition 2.9, or even representations other than sets of formulas in the target logic. These alternative theory representations might prove necessary in applying theory generation to domains other than security protocols. In looking for other such domains, we should keep in mind the features of the security protocol domain that make it amenable to theory generation: subtle properties of the domain can be captured by very simple rules, the verification problems involve small numbers of distinct entities (keys, messages, principals, nonces, etc.), and the properties of interest can be succinctly expressed by relatively small formulas.

7.2.2 Enhancements to the RV Logic and REVERE

Beyond improvements to the theory generation method, we can suggest several possible ways to strengthen the RV logic and the REVERE tool.

Support for secure hash functions would be a useful incremental RV extension; AUTLOG and other belief logics provide such support already, so the main task would be to determine what the legitimate interpretations of a secure hash are.

RV could express more protocols if it provided a notion of *skolem principals*. The purpose of such a principal is to act as a place-holder when a principal's identity is not (yet) known. This concept could prove useful in cases like the following. Suppose A initiates a protocol run with B by sending the message,

$$\{[A.B.K_a.N_a]\}_{K_b} \ ,$$

in which A is introducing herself to B, and providing her public key (K_a) and a secret the two will share (N_a). At this point, B does not yet know with confidence who A is, but B should be able to conclude that he can safely transmit the secret, N_a, encrypted under the key K_a. To represent this belief, we can say that B knows there exists some principal, P, such that $\overset{K_a}{\mapsto} P$ and $P \overset{N_a}{\longleftrightarrow} B$. By introducing a skolem principal to represent P, we can demonstrate that B does not violate secrecy later in the protocol run.

The feasibility check in RV could be extended to include demonstrating that each participant knows what decryption keys to use, which nonce values to check received messages against, and other such knowledge. This would require modelling protocols more like little programs, as some model-checking approaches do [DDHY92, CJM98].

Perhaps the most challenging aspect of specifying a protocol in RV or other belief logics is constructing the message interpretations (idealizations). Providing some automated support for this process could greatly improve the usability of a tool like REVERE. One approach that seems promising is to assume that each message is intended to convey all the beliefs the sender has at the time he sent it. The resulting interpretation will almost certainly be invalid, but we can remove beliefs from the interpretation until it becomes valid. The result would be, in some sense, a maximal valid interpretation, and might suffice for at least some protocols.

The REVERE tool could benefit from a strongly typed model for protocols and logics, in which keys, nonces, principal names, and so forth are members of distinct types. This should help catch some simple errors in specifications and rules, such as function arguments out of order. More importantly, though, the information could be used to automatically tag concrete message components (and their corresponding interpretations) with the appropriate type. Such a model would match an implementation in which it is always possible to distinguish, e.g., keys from principal names. Where the implementation does not warrant this assumption, the user could specify type equivalences to indicate which types can be confused.

Finally, in this work, we focus on analyzing one protocol at a time. However, in practice, suites of related protocols must often coexist. This creates the possibility that some attack may make use of runs of two different protocols. If, however, we use the same set of interpretations and initial assumptions in analyzing each protocol in a suite, we should be able to detect vulnerabilities to such attacks.

7.3 Closing Remarks

In this thesis we introduced theory generation, a new general-purpose technique for performing automated verification. Theory generation borrows from, and complements, both automated theorem proving and symbolic model checking, the two major approaches that currently dominate the field of mechanical reasoning. Broadly speaking, theory generation provides more complete automation than theorem proving, but with less generality. Likewise, theory generation has the advantage over model checking of producing conclusive proofs of some correctness properties without arbitrary bounds, while it is less well suited than model checking to proving temporal properties and producing useful counterexamples. This thesis has demonstrated the utility of theory generation for analyzing security protocols, but this is only a start; further investigation will tell whether it can yield similar benefits in other domains.

We do not yet have a complete solution to security protocol verification; perhaps that is an unattainable goal. However, today we can provide substantial support for the design and analysis of these protocols, and potential sharing of information among different tools via CAPSL. A thorough protocol design process should start with adherence to principles and guidelines such as Abadi and Needham's [AN96]. The designers could apply theory generation with RV or other belief logics to prove that the protocol meets its goals and make explicit the assumptions on which the protocol depends. They could use symbolic model checkers to generate attack scenarios. With interactive, automated theorem proving systems, they could demonstrate that the underlying cryptography meets its requirements, and make the connection between the protocol's behavior and that of the system in which it is used. Finally, the proposed protocol could be presented for public review, so that others might independently apply their favorite formal and informal methods. Each step in this process focuses on some level of abstraction and emphasizes some set of properties, in order to build confidence in the protocol and the system.

Appendix A

Equivalence of ℓ_{RW} and $\mathcal{L}^=$

This appendix contains the proof of Theorem 2.1, which shows the correspondence between ℓ_{RW} and $\mathcal{L}^=$:

> *If ϕ is a formula of ℓ_{RW}, and Γ is a set of formulas of ℓ_{RW}, then*
>
> $$\Gamma \vdash_{\ell_{RW}} \phi$$
>
> *if and only if*
>
> $$\Gamma \cup R \cup W \vdash_{\mathcal{L}^=} \phi$$

Proof: We start with the forward direction: that any formula, ϕ, which can be proved in ℓ_{RW} given assumptions, Γ, can also be proved in $\mathcal{L}^=$ using those same assumptions plus the rules and rewrites of ℓ_{RW}, R and W. We proceed by induction on the length of the proof of ϕ in ℓ_{RW}.

In the base case, ϕ has a single-step proof, and must therefore be some formula in Γ or the instantiated conclusion of a rule in R with no premises. In either case ϕ can be proved in $\mathcal{L}^=$ by introducing the formula from Γ or the rule from R, possibly followed by instantiation (introduction of an axiom of the form, $(\forall x.\phi(x)) \Rightarrow \phi(t)$, followed by *modus ponens*).

For the induction step, we have three cases, depending on the rule of inference used in the last step of the n-step proof (in ℓ_{RW}). We can ignore the case where the last line of the proof is an axiom or assumption, since the reasoning from the base case applies. The three cases follow:

- Case: *The last proof step is application of a rule in R.* Let the rule applied be

$$P_1, \ldots, P_m \vdash C \; .$$

There exists a substitution, σ, such that σC is the new conclusion (ϕ), and there must exist proofs in ℓ_{RW} of fewer than n steps (from Γ) of each of the σP_i. By the induction hypothesis, there exist proofs in $\mathcal{L}^=$ (from $\Gamma \cup R \cup W$) of the σP_i. We can concatenate these proofs, then add the rule from R,

$$\forall \overline{P} \forall C.(P_i \wedge \cdots \wedge P_m \Rightarrow C)$$

then instantiate this rule with σ and apply *modus ponens* to produce the following:

$$\sigma P_1 \wedge \cdots \wedge \sigma P_m \Rightarrow \sigma C)$$

Applying *modus ponens* again with propositional tautologies, we arrive at

$$\sigma C$$

so the proof in $\mathcal{L}^=$ is complete.

- Case: *The last proof step is instantiation.* Instantiation in ℓ_{RW} can be simulated easily in $\mathcal{L}^=$ by introducing an instantiation axiom:

$$(\forall x.\phi(x)) \Rightarrow \phi(t)$$

and applying modus ponens.

- Case: *The last proof step is a substitution of equal terms using an equality in W.* Let the equality be $T_1 = T_2$. There exists a substitution, σ, and a formula, $p(S_1, \ldots, S_m)$, such that $\Gamma \vdash_{\ell_{RW}} p(S_1, \ldots, S_m)$ has a proof of fewer than n steps, and ϕ is $[T_2 \backslash T_1]p(S_1, \ldots, S_m)$. By the induction hypothesis, $\Gamma \cup R \cup W \vdash_{\mathcal{L}^=} G$. We can extend this $\mathcal{L}^=$ proof as follows. First, introduce the rewrite from W:

$$T_1 = T_2$$

Next, introduce equality axioms for each of the S_i:

$$(T_2 = T_1 \wedge T_2 = T_2) \Rightarrow [T_2 \backslash T_2]S_i = [T_2 \backslash T_1]S_i$$

Apply propositional tautologies to yield, for each S_i,

$$[T_2 \backslash T_2]S_i = [T_2 \backslash T_1]S_i$$

Finally, we can use the fourth equality axiom with propositional tautologies to get

$$p([T_2 \backslash T_1]S_1, \ldots, [T_2 \backslash T_1]S_m)$$

which completes the proof of ϕ in $\mathcal{L}^=$.

This completes the proof that if ϕ is derivable in ℓ_{RW} from Γ, ϕ must also be derivable in $\mathcal{L}^=$ from the corresponding assumptions.

We now turn to the reverse direction: any formula, ϕ, which can be proved in $\mathcal{L}^=$ from $\Gamma \cup R \cup W$ must also have a proof from Γ in ℓ_{RW}.

We appeal to the well-known result that the *resolution* principle is complete for clauses in first-order logic [CL73]. Resolution is an inference rule that consists of applying substitutions to two clauses (disjunctions of terms), then generating a new clause by combining the disjuncts from both clauses and eliminating pairs of clauses, $(t, \neg t)$. All rules in R are Horn clauses (clauses with at most one positive disjunct). For such formulas, resolution corresponds to satisfying and removing a premise of the rule, and any sequence of resolution steps that produces a formula of ℓ_{RW} from a rule in R and some set of facts (formulas of ℓ_{RW}) can be simulated in ℓ_{RW} by applying an instance of that rule and using those facts to satisfy its premises. It follows that any formula of ℓ_{RW} that can be proved in $\mathcal{L}$ from $\Gamma \cup R \cup W$ can also be proved from Γ in ℓ_{RW}, but we are not quite done, since we need this result for $\mathcal{L}^=$, which includes the equality axioms:

EQ1. $(T = T)$ (reflexivity),

EQ2. $(S = T) \Rightarrow (T = S)$ (commutativity),

EQ3. $((T_1 = T_2) \wedge (T_2 = T_3)) \Rightarrow (T_1 = T_3)$ (transitivity),

EQ4. $(S_1 = T_1 \wedge \cdots \wedge S_n = T_n) \Rightarrow (p(S_1, \ldots, S_n) \Rightarrow p(T_1, \ldots, T_n))$, and

EQ5. $(S_1 = T_1 \wedge \cdots \wedge S_n = T_n) \Rightarrow$
$([S_1 \backslash X_1, \ldots, S_n \backslash X_n]t = [T_1 \backslash X_1, \ldots, T_n \backslash X_n]t)$

Note that the axiom schemata EQ1, EQ2, EQ3, and EQ5, when applied with the resolution principle, can only produce more equalities as their results. Since equalities are not formulas of ℓ_{RW}, we can focus on axiom schema EQ4, which can produce ℓ_{RW} formulas. We must show that any application of an EQ4 axiom, in the context of a proof in $\mathcal{L}^=$ using assumptions $\Gamma \cup R \cup W$, can be simulated in ℓ_{RW} by the substitution of equal terms using an equality in W.

We first assume that only one (S_i, T_i) pair in the axiom is not identical; that is, we consider only EQ4 axioms of this form (EQ4′):

$$\begin{aligned}(S_1 = S_1 \wedge \cdots S_k = T_k \wedge \cdots \wedge S_n = S_n)\\ \Rightarrow (p(S_1, \ldots, S_k, \ldots, S_n) \Rightarrow p(S_1, \ldots, T_k, \ldots, S_n))\end{aligned}$$

Any other EQ4 axiom application can be simulated by repeated application of this form and use of EQ1, so there is no loss of generality. Note that in order for this axiom to be applied fully and produce an ℓ_{RW} formula, there must be a proof of $S_k = T_k$. Note further that this proof can use only the equality axioms and equalities in W, since all "premises" of equality axioms are equalities. We propose the following induction hypothesis:

> If either $S_k = T_k$ or $T_k = S_k$ has a proof of fewer than n steps, then the application of EQ4$'$ can be simulated in ℓ_{RW}.

In the base case, a single step proof, either S_k and T_k are identical (an application of EQ1), in which case EQ4$'$ produces a trivial result, or the formula $S_k = T_k$ (or $T_k = S_k$) is in W, in which case the result of EQ4$'$ is the same as that of a simple ℓ_{RW} substitution using that equality.

For the induction step, we assume that either $S_k = T_k$ or $T_k = S_k$ has a proof of n steps, and we have three cases, depending on whether that proof ends with an application of EQ2, EQ3, or EQ5.

- Case: *The last proof step is an application of EQ2 (commutativity).* In this case, either $S_k = T_k$ or $T_k = S_k$ has a proof shorter than n steps, so the induction hypothesis gives the required result.

- Case: *The last proof step is an application of EQ3 (transitivity).* There must be proofs shorter than n of $S_k = U$ and $U = T_k$ (for some U). We can apply EQ4$'$ using each of these two equalities in sequence to get the same result, so by the induction hypothesis, this result is provable in ℓ_{RW}.

- Case: *The last proof step is an application of EQ5.* Without loss of generality, assume EQ5 is only used with $n = 1$; that is, in the following form:

$$(S_1 = T_1) \Rightarrow ([S_1 \backslash X_1]t = [T_1 \backslash X_1]t)$$

  (We can simulate the general case by a sequence of applications of EQ5 and EQ4$'$.) If either $S_1 = T_1$ or $T_1 = S_1$ is an equality in W, then this application of EQ5 and EQ4$'$ yields the same result as a simple ℓ_{RW} substitution using that equality. If $S_1 = T_1$ is the result of EQ1 (so S_1 is identical to T_1), then EQ5 and EQ4$'$ produce a trivial result. If $S_1 = T_1$ is the result of EQ3, then just as in the EQ3 case we can simulate the result of EQ5 and EQ4$'$ by a series of two applications of EQ5 and EQ4$'$. Therefore, the application of EQ4$'$ following EQ5 can always be simulated in ℓ_{RW}.

The induction is complete, so it follows that the equality axioms can be safely added without compromising the completeness of ℓ_{RW} with respect to $\mathcal{L}^{=}$. This concludes the proof of Theorem 2.1. ■

Appendix B

Logics and Protocol Analyses

This Appendix contains the main REVERE specifications for BAN, AUTLOG, Kailar's accountability logic, and RV, as well as sample protocol specifications and analyses.

B.1 BAN

First, the BAN logic, protocol specifications, and analysis results.

```
LOGIC BAN;

REWRITES
  comma_commutative:
    comma(?X, ?Y) = comma(?Y, ?X)
  comma_associative_1:
    comma(comma(?X, ?Y), ?Z) = comma(?X, comma(?Y, ?Z))
  comma_associative_2:
    comma(?X, comma(?Y, ?Z)) = comma(comma(?X, ?Y), ?Z)
  shared_key_commutative:
    shared_key(?K, ?Q, ?R) = shared_key(?K, ?R, ?Q)
  secret_commutative:
    secret(?Y, ?Q, ?R) = secret(?Y, ?R, ?Q)
  distinct_commutative:
    distinct(?P, ?Q) = distinct(?Q, ?P)

S-RULES
  message_meaning_shared:
     believes(?P, shared_key(?K, ?Q, ?P))
     sees(?P, encrypt(?X, ?K, ?R))
```

```
   distinct(?P, ?R)
  --------------------------------------
   believes(?P, said(?Q, ?X))

message_meaning_public:
   believes(?P, public_key(?K1, ?Q))
   sees(?P, encrypt(?X, ?K2, ?R))
   inv(?K1, ?K2)
   distinct(?P, ?R)
  -----------------------------------
   believes(?P, said(?Q, ?X))

message_meaning_secret:
   believes(?P, secret(?Y, ?Q, ?P))
   sees(?P, combine(?X, ?Y))
  ----------------------------------
   believes(?P, said(?Q, ?X))

nonce_verification_1:
   believes(?P, said(?Q, ?X))
   believes(?P, fresh(?X))
  --------------------------------
   believes(?P, believes(?Q, ?X))

jurisdiction:
   believes(?P, controls(?Q, ?X))
   believes(?P, believes(?Q, ?X))
  --------------------------------
   believes(?P, ?X)

extract_shared:
   believes(?P, shared_key(?K, ?Q, ?P))
   sees(?P, encrypt(?X, ?K, ?R))
   distinct(?P, ?R)
  --------------------------------------
   sees(?P, ?X)

extract_public_1:
   believes(?P, public_key(?K, ?P))
   sees(?P, encrypt(?X, ?K, ?R))
  ----------------------------------
   sees(?P, ?X)

extract_public_2:
   believes(?P, public_key(?K1, ?Q))
```

```
    sees(?P, encrypt(?X, ?K2, ?R))
    inv(?K1, ?K2)
    distinct(?P, ?R)
   -----------------------------------
    sees(?P, ?X)

 extract_combine:
    sees(?P, combine(?X, ?Y))
   ---------------------------
    sees(?P, ?X)

 extract_comma_2:
    sees(?P, comma(?X, ?Y))
   -------------------------
    sees(?P, ?X)

 extract_comma_3:
    believes(?P, said(?Q, comma(?X, ?Y)))
   ---------------------------------------
    believes(?P, said(?Q, ?X))

 extract_comma_4:
    believes(?P, believes(?Q, comma(?X, ?Y)))
   -------------------------------------------
    believes(?P, believes(?Q, ?X))

 mm_nv_1:
    believes(?P, fresh(?K))
    sees(?P, encrypt(?X, ?K, ?R))
    distinct(?P, ?R)
    believes(?P, shared_key(?K, ?Q, ?P))
   --------------------------------------
    believes(?P, believes(?Q, ?X))

 mm_nv_2:
    believes(?P, fresh(?Y))
    sees(?P, combine(?X, ?Y))
    believes(?P, secret(?Y, ?Q, ?P))
   ----------------------------------
    believes(?P, believes(?Q, ?X))

G-RULES
 fresh_extends_1:
    believes(?P, fresh(?X))
   ------------------------------------
```

```
     believes(?P, fresh(comma(?X, ?Y)))

  fresh_extends_2:
     believes(?P, fresh(?K))
    ---------------------------------------------
     believes(?P, fresh(shared_key(?K, ?Q, ?R)))

  fresh_extends_3:
     believes(?P, fresh(?K))
    -----------------------------------------
     believes(?P, fresh(public_key(?K, ?Q)))

  fresh_extends_4:
     believes(?P, fresh(?Y))
    -----------------------------------------
     believes(?P, fresh(secret(?Y, ?Q, ?R)))

  fresh_extends_5:
     believes(?P, fresh(?Y))
    --------------------------------------
     believes(?P, fresh(combine(?X, ?Y)))

  fresh_extends_6:
     believes(?P, fresh(?K))
    ------------------------------------------
     believes(?P, fresh(encrypt(?X, ?K, ?R)))

  fresh_extends_7:
     believes(?P, fresh(?X))
    ------------------------------------------
     believes(?P, fresh(encrypt(?X, ?K, ?R)))

  fresh_extends_8:
     believes(?P, fresh(?X))
    --------------------------------------
     believes(?P, fresh(combine(?X, ?Y)))

END;

// ------------------------------------------------
PROTOCOL Kerberos_1;  // Logic: BAN

VARIABLES
    A, B, S: Principal;
    Kab, Kas, Kbs: SKey;
    Ta, Ts: Field;
```

```
ASSUMPTIONS
    believes(A, shared_key(Kas, S, A));
    believes(B, shared_key(Kbs, S, B));
    believes(S, shared_key(Kas, A, S));
    believes(S, shared_key(Kbs, B, S));
    believes(S, shared_key(Kab, A, B));
    believes(A, controls(S, shared_key(?K, A, B)));
    believes(B, controls(S, shared_key(?K, A, B)));
    believes(A, fresh(Ts));
    believes(B, fresh(Ts));
    believes(A, fresh(Ta));
    believes(B, fresh(Ta));
    distinct(A, S);
    distinct(A, B);
    distinct(B, S);

MESSAGES
    1. S -> A: encrypt([Ts, shared_key(Kab, A, B), encrypt([Ts,
         shared_key(Kab, A, B)], Kbs, S)], Kas, S);
    2. A -> B: [encrypt([Ts, shared_key(Kab, A, B)], Kbs, S),
         encrypt([Ta, shared_key(Kab, A, B)], Kab, A)];
    3. B -> A: encrypt([Ta, shared_key(Kab, A, B)], Kab, B);

GOALS
    believes(A, shared_key(Kab, A, B));
    believes(B, shared_key(Kab, A, B));
    believes(B, believes(A, shared_key(Kab, A, B)));
    believes(A, believes(B, shared_key(Kab, A, B)));

END;

Final theory representation (size 61):
    believes(A, believes(B, Ta))
    believes(A, believes(B, [Ta, shared_key(Kab, A, B)]))
    believes(A, believes(B, shared_key(Kab, A, B)))
    believes(A, believes(S, Ts))
    believes(A, believes(S, [Ts, encrypt([Ts, shared_key(Kab, A,
         B)], Kbs, S), shared_key(Kab, A, B)]))
    believes(A, believes(S, [encrypt([Ts, shared_key(Kab, A,
         B)], Kbs, S), shared_key(Kab, A, B)]))
    believes(A, believes(S, encrypt([Ts, shared_key(Kab, A, B)],
         Kbs, S)))
    believes(A, believes(S, shared_key(Kab, A, B)))
    believes(A, controls(S, shared_key(?CAN0, A, B)))
```

```
believes(A, fresh(Ta))
believes(A, fresh(Ts))
believes(A, said(B, Ta))
believes(A, said(B, [Ta, shared_key(Kab, A, B)]))
believes(A, said(B, shared_key(Kab, A, B)))
believes(A, said(S, Ts))
believes(A, said(S, [Ts, encrypt([Ts, shared_key(Kab, A,
     B)], Kbs, S), shared_key(Kab, A, B)]))
believes(A, said(S, [encrypt([Ts, shared_key(Kab, A, B)],
     Kbs, S), shared_key(Kab, A, B)]))
believes(A, said(S, encrypt([Ts, shared_key(Kab, A, B)],
     Kbs, S)))
believes(A, said(S, shared_key(Kab, A, B)))
believes(A, shared_key(Kab, A, B))
believes(A, shared_key(Kas, A, S))
believes(B, believes(A, Ta))
believes(B, believes(A, [Ta, shared_key(Kab, A, B)]))
believes(B, believes(A, shared_key(Kab, A, B)))
believes(B, believes(S, Ts))
believes(B, believes(S, [Ts, shared_key(Kab, A, B)]))
believes(B, believes(S, shared_key(Kab, A, B)))
believes(B, controls(S, shared_key(?CAN0, A, B)))
believes(B, fresh(Ta))
believes(B, fresh(Ts))
believes(B, said(A, Ta))
believes(B, said(A, [Ta, shared_key(Kab, A, B)]))
believes(B, said(A, shared_key(Kab, A, B)))
believes(B, said(S, Ts))
believes(B, said(S, [Ts, shared_key(Kab, A, B)]))
believes(B, said(S, shared_key(Kab, A, B)))
believes(B, shared_key(Kab, A, B))
believes(B, shared_key(Kbs, B, S))
believes(S, shared_key(Kab, A, B))
believes(S, shared_key(Kas, A, S))
believes(S, shared_key(Kbs, B, S))
distinct(A, B)
distinct(A, S)
distinct(B, S)
sees(A, Ta)
sees(A, Ts)
sees(A, [Ta, shared_key(Kab, A, B)])
sees(A, [Ts, encrypt([Ts, shared_key(Kab, A, B)], Kbs, S),
     shared_key(Kab, A, B)])
sees(A, [encrypt([Ts, shared_key(Kab, A, B)], Kbs, S),
     shared_key(Kab, A, B)])
```

```
    sees(A, encrypt([Ta, shared_key(Kab, A, B)], Kab, B))
    sees(A, encrypt([Ts, encrypt([Ts, shared_key(Kab, A, B)],
         Kbs, S), shared_key(Kab, A, B)], Kas, S))
    sees(A, encrypt([Ts, shared_key(Kab, A, B)], Kbs, S))
    sees(A, shared_key(Kab, A, B))
    sees(B, Ta)
    sees(B, Ts)
    sees(B, [Ta, shared_key(Kab, A, B)])
    sees(B, [Ts, shared_key(Kab, A, B)])
    sees(B, [encrypt([Ta, shared_key(Kab, A, B)], Kab, A),
         encrypt([Ts, shared_key(Kab, A, B)], Kbs, S)])
    sees(B, encrypt([Ta, shared_key(Kab, A, B)], Kab, A))
    sees(B, encrypt([Ts, shared_key(Kab, A, B)], Kbs, S))
    sees(B, shared_key(Kab, A, B))
critical properties for this theory:
    believes(A, believes(B, Ta))
    believes(A, believes(B, shared_key(Kab, A, B)))
    believes(A, believes(S, Ts))
    believes(A, believes(S, encrypt([Ts, shared_key(Kab, A, B)],
         Kbs, S)))
    believes(A, believes(S, shared_key(Kab, A, B)))
    believes(A, said(B, Ta))
    believes(A, said(S, Ts))
    believes(A, shared_key(Kab, A, B))
    believes(A, shared_key(Kas, A, S))
    believes(B, believes(A, Ta))
    believes(B, believes(A, shared_key(Kab, A, B)))
    believes(B, believes(S, Ts))
    believes(B, believes(S, shared_key(Kab, A, B)))
    believes(B, said(A, Ta))
    believes(B, said(S, Ts))
    believes(B, shared_key(Kab, A, B))
    believes(B, shared_key(Kbs, B, S))

desired property: believes(A, shared_key(Kab, A, B))
   is TRUE
desired property: believes(B, shared_key(Kab, A, B))
   is TRUE
desired property: believes(B, believes(A, shared_key(Kab, A,
         B)))
   is TRUE
desired property: believes(A, believes(B, shared_key(Kab, A,
         B)))
   is TRUE

// ---------------------------------------------
```

```
PROTOCOL Kerberos_2;  // Logic: BAN

VARIABLES
    A, B, S: Principal;
    Kab, Kas, Kbs: SKey;
    Ta, Ts: Field;

ASSUMPTIONS
    believes(A, shared_key(Kas, S, A));
    believes(B, shared_key(Kbs, S, B));
    believes(S, shared_key(Kas, A, S));
    believes(S, shared_key(Kbs, B, S));
    believes(S, shared_key(Kab, A, B));
    believes(A, controls(S, shared_key(?K, A, B)));
    believes(B, controls(S, shared_key(?K, A, B)));
    believes(A, fresh(Ts));
    believes(B, fresh(Ts));
    believes(A, fresh(Ta));
    believes(B, fresh(Ta));
    distinct(A, S);
    distinct(A, B);
    distinct(B, S);

MESSAGES
    1. ? -> A: encrypt([Ts, shared_key(Kab, A, B), encrypt([Ts,
         shared_key(Kab, A, B)], Kbs, S)], Kas, S);
    2. ? -> B: [encrypt([Ts, shared_key(Kab, A, B)], Kbs, S),
         encrypt([Ta, shared_key(Kab, A, B)], Kab, A)];

GOALS
    believes(A, shared_key(Kab, A, B));
    believes(B, shared_key(Kab, A, B));
    believes(B, believes(A, shared_key(Kab, A, B)));
    believes(A, believes(B, shared_key(Kab, A, B)));

END;

Final theory representation (size 52) [omitted]

critical properties for this theory:
    believes(A, believes(S, Ts))
    believes(A, believes(S, encrypt([Ts, shared_key(Kab, A, B)],
         Kbs, S)))
    believes(A, believes(S, shared_key(Kab, A, B)))
    believes(A, said(S, Ts))
```

```
    believes(A, shared_key(Kab, A, B))
    believes(A, shared_key(Kas, A, S))
    believes(B, believes(A, Ta))
    believes(B, believes(A, shared_key(Kab, A, B)))
    believes(B, believes(S, Ts))
    believes(B, believes(S, shared_key(Kab, A, B)))
    believes(B, said(A, Ta))
    believes(B, said(S, Ts))
    believes(B, shared_key(Kab, A, B))
    believes(B, shared_key(Kbs, B, S))

desired property: believes(A, shared_key(Kab, A, B))
   is TRUE
desired property: believes(B, shared_key(Kab, A, B))
   is TRUE
desired property: believes(B, believes(A, shared_key(Kab, A,
         B)))
   is TRUE
desired property: believes(A, believes(B, shared_key(Kab, A,
         B)))
   is FALSE

// ----------------------------------------------
PROTOCOL Andrew_RPC_1;  // Logic: BAN

VARIABLES
    A, B: Principal;
    Kab, Kab': SKey;
    Na, Nb, Nb': Field;

ASSUMPTIONS
    believes(A, shared_key(Kab, A, B));
    believes(B, shared_key(Kab, A, B));
    believes(A, controls(B, shared_key(?K, A, B)));
    believes(B, shared_key(Kab', A, B));
    believes(A, fresh(Na));
    believes(B, fresh(Nb));
    believes(B, fresh(Nb'));
    distinct(A, B);

MESSAGES
    1. ? -> B: encrypt(Na, Kab, A);
    2. ? -> A: encrypt([Na, Nb], Kab, B);
    3. ? -> B: encrypt(Nb, Kab, A);
    4. ? -> A: encrypt([shared_key(Kab', A, B), Nb'], Kab, B);
```

```
GOALS
    believes(B, shared_key(Kab’, A, B));
    believes(A, said(B, [shared_key(Kab’, A, B), Nb’]));
    believes(B, believes(A, Nb));
    believes(A, believes(B, [Na, Nb]));
    believes(A, shared_key(Kab’, A, B));
    believes(A, believes(B, shared_key(Kab’, A, B)));
    believes(B, believes(A, shared_key(Kab’, A, B)));

END;

Final theory representation (size 32): [omitted]

critical properties for this theory:
    believes(A, believes(B, Na))
    believes(A, believes(B, Nb))
    believes(A, said(B, Na))
    believes(A, said(B, Nb))
    believes(A, said(B, Nb’))
    believes(B, believes(A, Nb))
    believes(B, said(A, Na))
    believes(B, said(A, Nb))

desired property: believes(B, shared_key(Kab’, A, B))
   is TRUE
desired property: believes(A, said(B, [shared_key(Kab’, A, B),
         Nb’]))
   is TRUE
desired property: believes(B, believes(A, Nb))
   is TRUE
desired property: believes(A, believes(B, [Na, Nb]))
   is TRUE
desired property: believes(A, shared_key(Kab’, A, B))
   is FALSE
desired property: believes(A, believes(B, shared_key(Kab’, A,
         B)))
   is FALSE
desired property: believes(B, believes(A, shared_key(Kab’, A,
         B)))
   is FALSE

// ----------------------------------------------
PROTOCOL Andrew_RPC_2;  // Logic: BAN

VARIABLES
    A, B: Principal;
```

```
    Kab, Kab’: SKey;
    Na, Nb, Nb’: Field;

ASSUMPTIONS
    believes(A, shared_key(Kab, A, B));
    believes(B, shared_key(Kab, A, B));
    believes(A, controls(B, shared_key(?K, A, B)));
    believes(B, shared_key(Kab’, A, B));
    believes(A, fresh(Na));
    believes(B, fresh(Nb));
    believes(B, fresh(Nb’));
    distinct(A, B);

MESSAGES
    1. ? -> B: encrypt(Na, Kab, A);
    2. ? -> A: encrypt([Na, Nb], Kab, B);
    3. ? -> B: encrypt(Nb, Kab, A);
    4. ? -> A: encrypt([shared_key(Kab’, A, B), Nb’, Na], Kab,
         B);

GOALS
    believes(B, shared_key(Kab’, A, B));
    believes(A, shared_key(Kab’, A, B));
    believes(A, believes(B, shared_key(Kab’, A, B)));
    believes(B, believes(A, shared_key(Kab’, A, B)));

END;

Final theory representation (size 39): [omitted]

critical properties for this theory:
    believes(A, believes(B, Na))
    believes(A, believes(B, Nb))
    believes(A, believes(B, Nb’))
    believes(A, believes(B, shared_key(Kab’, A, B)))
    believes(A, said(B, Na))
    believes(A, said(B, Nb))
    believes(A, said(B, Nb’))
    believes(A, shared_key(Kab’, A, B))
    believes(B, believes(A, Nb))
    believes(B, said(A, Na))
    believes(B, said(A, Nb))

desired property: believes(B, shared_key(Kab’, A, B))
   is TRUE
```

```
desired property: believes(A, shared_key(Kab’, A, B))
   is TRUE
desired property: believes(A, believes(B, shared_key(Kab’, A,
         B)))
   is TRUE
desired property: believes(B, believes(A, shared_key(Kab’, A,
         B)))
   is FALSE

// -----------------------------------------------
PROTOCOL Andrew_RPC_3;  // Logic: BAN

VARIABLES
    A, B: Principal;
    Kab, Kab’: SKey;
    Na: Field;

ASSUMPTIONS
    believes(A, shared_key(Kab, A, B));
    believes(B, shared_key(Kab, A, B));
    believes(A, controls(B, shared_key(?K, A, B)));
    believes(B, shared_key(Kab’, A, B));
    believes(A, fresh(Na));
    believes(B, fresh(Nb));
    believes(B, fresh(Nb’));
    distinct(A, B);
    believes(B, fresh(Kab’));

MESSAGES
    1. ? -> A: encrypt([Na, shared_key(Kab’, A, B)], Kab, B);
    2. ? -> B: encrypt(shared_key(Kab’, A, B), Kab’, A);

GOALS
    believes(B, shared_key(Kab’, A, B));
    believes(A, shared_key(Kab’, A, B));
    believes(A, believes(B, shared_key(Kab’, A, B)));
    believes(B, believes(A, shared_key(Kab’, A, B)));

END;

Final theory representation (size 24): [omitted]

critical properties for this theory:
    believes(A, believes(B, Na))
    believes(A, believes(B, shared_key(Kab’, A, B)))
    believes(A, said(B, Na))
```

```
    believes(A, shared_key(Kab', A, B))
    believes(B, believes(A, shared_key(Kab', A, B)))

desired property: believes(B, shared_key(Kab', A, B))
   is TRUE
desired property: believes(A, shared_key(Kab', A, B))
   is TRUE
desired property: believes(A, believes(B, shared_key(Kab', A,
         B)))
   is TRUE
desired property: believes(B, believes(A, shared_key(Kab', A,
         B)))
   is TRUE

// ----------------------------------------------
PROTOCOL Needham_Schroeder_1;  // Logic: BAN

VARIABLES
    A, B, S: Principal;
    Ka, Kb, Ks': PKey;
    Na, Nb: Field;

ASSUMPTIONS
    believes(A, public_key(Ka, A));
    believes(A, public_key(Ks, S));
    believes(B, public_key(Kb, B));
    believes(B, public_key(Ks, S));
    believes(S, public_key(Ka, A));
    believes(S, public_key(Kb, B));
    believes(S, public_key(Ks, S));
    believes(A, controls(S, public_key(?K, B)));
    believes(B, controls(S, public_key(?K, A)));
    believes(A, fresh(Na));
    believes(B, fresh(Nb));
    believes(A, secret(Na, A, B));
    believes(B, secret(Nb, A, B));
    believes(A, fresh(public_key(Kb, B)));
    believes(B, fresh(public_key(Ka, A)));
    distinct(A, B);
    distinct(A, S);
    distinct(B, S);
    inv(Ks, Ks');

MESSAGES
    1. ? -> A: encrypt(public_key(Kb, B), Ks', S);
    2. ? -> B: encrypt(Na, Kb, A);
```

```
    3. ? -> B: encrypt(public_key(Ka, A), Ks', S);
    4. ? -> A: encrypt(combine(secret(Nb, A, B), Na), Ka, B);
    5. ? -> B: encrypt(combine(secret(Na, A, B), Nb), Kb, A);

GOALS
    believes(A, public_key(Kb, B));
    believes(B, public_key(Ka, A));
    believes(A, believes(B, secret(Nb, A, B)));
    believes(B, believes(A, secret(Na, A, B)));

END;

Final theory representation (size 41): [omitted]

critical properties for this theory:
    believes(A, believes(B, secret(Nb, A, B)))
    believes(A, believes(S, public_key(Kb, B)))
    believes(A, public_key(Kb, B))
    believes(B, believes(A, secret(Na, A, B)))
    believes(B, believes(S, public_key(Ka, A)))
    believes(B, public_key(Ka, A))

desired property: believes(A, public_key(Kb, B))
   is TRUE
desired property: believes(B, public_key(Ka, A))
   is TRUE
desired property: believes(A, believes(B, secret(Nb, A, B)))
   is TRUE
desired property: believes(B, believes(A, secret(Na, A, B)))
   is TRUE

// ----------------------------------------------
PROTOCOL Needham_Schroeder_2;  // Logic: BAN

VARIABLES
    A, B, S: Principal;
    Ka, Kb, Ks': PKey;
    Na, Nb, Ts: Field;

ASSUMPTIONS
    believes(A, public_key(Ka, A));
    believes(A, public_key(Ks, S));
    believes(B, public_key(Kb, B));
    believes(B, public_key(Ks, S));
    believes(S, public_key(Ka, A));
    believes(S, public_key(Kb, B));
```

```
    believes(S, public_key(Ks, S));
    believes(A, controls(S, public_key(?K, B)));
    believes(B, controls(S, public_key(?K, A)));
    believes(A, fresh(Na));
    believes(B, fresh(Nb));
    believes(A, fresh(Ts));
    believes(B, fresh(Ts));
    believes(A, secret(Na, A, B));
    believes(B, secret(Nb, A, B));
    distinct(A, B);
    distinct(A, S);
    distinct(B, S);
    inv(Ks, Ks');

MESSAGES
    1. ? -> A: encrypt([public_key(Kb, B), Ts], Ks', S);
    2. ? -> B: encrypt(Na, Kb, A);
    3. ? -> B: encrypt([public_key(Ka, A), Ts], Ks', S);
    4. ? -> A: encrypt(combine(secret(Nb, A, B), Na), Ka, B);
    5. ? -> B: encrypt(combine(secret(Na, A, B), Nb), Kb, A);

GOALS
    believes(A, public_key(Kb, B));
    believes(B, public_key(Ka, A));
    believes(A, believes(B, secret(Nb, A, B)));
    believes(B, believes(A, secret(Na, A, B)));

END;

Final theory representation (size 53): [omitted]

critical properties for this theory:
    believes(A, believes(B, secret(Nb, A, B)))
    believes(A, believes(S, Ts))
    believes(A, believes(S, public_key(Kb, B)))
    believes(A, public_key(Kb, B))
    believes(A, said(S, Ts))
    believes(B, believes(A, secret(Na, A, B)))
    believes(B, believes(S, Ts))
    believes(B, believes(S, public_key(Ka, A)))
    believes(B, public_key(Ka, A))
    believes(B, said(S, Ts))

desired property: believes(A, public_key(Kb, B))
   is TRUE
```

```
desired property: believes(B, public_key(Ka, A))
   is TRUE
desired property: believes(A, believes(B, secret(Nb, A, B)))
   is TRUE
desired property: believes(B, believes(A, secret(Na, A, B)))
   is TRUE

// ----------------------------------------------
PROTOCOL X_509_1;  // Logic: BAN

VARIABLES
    A, B: Principal;
    Ka, Ka', Kb, Kb': PKey;
    Na, Nb, Ta, Tb, Xa, Xb, Ya, Yb: Field;

ASSUMPTIONS
    believes(A, public_key(Ka, A));
    believes(B, public_key(Kb, B));
    believes(A, public_key(Kb, B));
    believes(B, public_key(Ka, A));
    believes(A, fresh(Na));
    believes(B, fresh(Nb));
    believes(A, fresh(Tb));
    believes(B, fresh(Ta));
    believes(A, secret(Na, A, B));
    believes(B, secret(Nb, A, B));
    distinct(A, B);
    inv(Ka, Ka');
    inv(Kb, Kb');

MESSAGES
    1. ? -> B: encrypt([Ta, Na, Xa, encrypt(Ya, Kb, A)], Ka',
         A);
    2. ? -> A: encrypt([Tb, Nb, Na, Xb, encrypt(Yb, Ka, B)],
         Kb', B);
    3. ? -> B: encrypt(Nb, Ka', A);

GOALS
    believes(A, believes(B, Xb));
    believes(B, believes(A, Xa));
    believes(A, believes(B, Yb));
    believes(B, believes(A, Ya));

END;

Final theory representation (size 69): [omitted]
```

```
critical properties for this theory:
    believes(A, believes(B, Na))
    believes(A, believes(B, Nb))
    believes(A, believes(B, Tb))
    believes(A, believes(B, Xb))
    believes(A, believes(B, encrypt(Yb, Ka, B)))
    believes(A, said(B, Na))
    believes(A, said(B, Nb))
    believes(A, said(B, Tb))
    believes(A, said(B, Xb))
    believes(B, believes(A, Na))
    believes(B, believes(A, Nb))
    believes(B, believes(A, Ta))
    believes(B, believes(A, Xa))
    believes(B, believes(A, encrypt(Ya, Kb, A)))
    believes(B, said(A, Na))
    believes(B, said(A, Nb))
    believes(B, said(A, Ta))
    believes(B, said(A, Xa))

desired property: believes(A, believes(B, Xb))
   is TRUE
desired property: believes(B, believes(A, Xa))
   is TRUE
desired property: believes(A, believes(B, Yb))
   is FALSE
desired property: believes(B, believes(A, Ya))
   is FALSE

// ----------------------------------------------
PROTOCOL X_509_2;  // Logic: BAN

VARIABLES
    A, B: Principal;
    Ka, Ka', Kb, Kb': PKey;
    Na, Nb, Ta, Xa, Xb, Ya, Yb: Field;

ASSUMPTIONS
    believes(A, public_key(Ka, A));
    believes(B, public_key(Kb, B));
    believes(A, public_key(Kb, B));
    believes(B, public_key(Ka, A));
    believes(A, fresh(Na));
    believes(B, fresh(Nb));
    believes(B, fresh(Ta));
```

```
    believes(A, secret(Na, A, B));
    believes(B, secret(Nb, A, B));
    distinct(A, B);
    inv(Ka, Ka');
    inv(Kb, Kb');

MESSAGES
    1. ? -> B: encrypt([Ta, Na, Xa, encrypt(encrypt([Ya, Ta],
         Ka', A), Kb, A)], Ka', A);
    2. ? -> A: encrypt([Nb, Na, Xb, encrypt(encrypt([Yb, Na],
         Kb', B), Ka, B)], Kb', B);
    3. ? -> B: encrypt(Nb, Ka', A);

GOALS
    believes(A, believes(B, Xb));
    believes(B, believes(A, Xa));
    believes(A, believes(B, Yb));
    believes(B, believes(A, Ya));

END;

Final theory representation (size 74): [omitted]

critical properties for this theory:
    believes(A, believes(B, Na))
    believes(A, believes(B, Nb))
    believes(A, believes(B, Xb))
    believes(A, believes(B, Yb))
    believes(A, believes(B, encrypt(encrypt([Na, Yb], Kb', B),
         Ka, B)))
    believes(A, said(B, Na))
    believes(A, said(B, Nb))
    believes(A, said(B, Xb))
    believes(A, said(B, Yb))
    believes(B, believes(A, Na))
    believes(B, believes(A, Nb))
    believes(B, believes(A, Ta))
    believes(B, believes(A, Xa))
    believes(B, believes(A, Ya))
    believes(B, believes(A, encrypt(encrypt([Ta, Ya], Ka', A),
         Kb, A)))
    believes(B, said(A, Na))
    believes(B, said(A, Nb))
    believes(B, said(A, Ta))
    believes(B, said(A, Xa))
```

```
    believes(B, said(A, Ya))

desired property: believes(A, believes(B, Xb))
   is TRUE
desired property: believes(B, believes(A, Xa))
   is TRUE
desired property: believes(A, believes(B, Yb))
   is TRUE
desired property: believes(B, believes(A, Ya))
   is TRUE

// ----------------------------------------------
PROTOCOL Wide_Mouth_Frog_1;  // Logic: BAN

VARIABLES
    A, B, S: Principal;
    Kab, Kat, Kbt: SKey;
    Ta, Ts: Field;

ASSUMPTIONS
    believes(A, shared_key(Kat, A, S));
    believes(S, shared_key(Kat, A, S));
    believes(B, shared_key(Kbt, B, S));
    believes(S, shared_key(Kbt, B, S));
    believes(A, shared_key(Kab, A, B));
    believes(S, fresh(Ta));
    believes(B, fresh(Ts));
    believes(S, controls(A, shared_key(?K, A, B)));
    believes(B, controls(S, shared_key(?K, A, B)));
    distinct(A, B);
    distinct(A, S);
    distinct(B, S);

MESSAGES
    1. ? -> S: encrypt([Ta, shared_key(Kab, A, B)], Kat, A);
    2. ? -> B: encrypt([Ts, shared_key(Kab, A, B)], Kbt, S);

GOALS
    believes(A, shared_key(Kab, A, B));
    believes(B, shared_key(Kab, A, B));
    believes(B, believes(A, shared_key(Kab, A, B)));
    believes(A, believes(B, shared_key(Kab, A, B)));

END;

Final theory representation (size 34): [omitted]
```

```
critical properties for this theory:
    believes(B, believes(S, Ts))
    believes(B, believes(S, shared_key(Kab, A, B)))
    believes(B, said(S, Ts))
    believes(B, shared_key(Kab, A, B))
    believes(S, believes(A, Ta))
    believes(S, believes(A, shared_key(Kab, A, B)))
    believes(S, said(A, Ta))
    believes(S, shared_key(Kab, A, B))

desired property: believes(A, shared_key(Kab, A, B))
   is TRUE
desired property: believes(B, shared_key(Kab, A, B))
   is TRUE
desired property: believes(B, believes(A, shared_key(Kab, A,
         B)))
   is FALSE
desired property: believes(A, believes(B, shared_key(Kab, A,
         B)))
   is FALSE

// ----------------------------------------------
PROTOCOL Wide_Mouth_Frog_2;  // Logic: BAN

VARIABLES
    A, B, S: Principal;
    Kab, Kat, Kbt: SKey;
    Ta, Ts: Field;

ASSUMPTIONS
    believes(A, shared_key(Kat, A, S));
    believes(S, shared_key(Kat, A, S));
    believes(B, shared_key(Kbt, B, S));
    believes(S, shared_key(Kbt, B, S));
    believes(A, shared_key(Kab, A, B));
    believes(S, fresh(Ta));
    believes(B, fresh(Ts));
    believes(B, controls(S, believes(A, shared_key(?K, A,
         B))));
    believes(B, controls(A, shared_key(?K, A, B)));
    distinct(A, B);
    distinct(A, S);
    distinct(B, S);

MESSAGES
```

```
    1. ? -> S: encrypt([Ta, shared_key(Kab, A, B)], Kat, A);
    2. ? -> B: encrypt([Ts, believes(A, shared_key(Kab, A, B))],
         Kbt, S);

GOALS
    believes(A, shared_key(Kab, A, B));
    believes(B, shared_key(Kab, A, B));
    believes(B, believes(A, shared_key(Kab, A, B)));
    believes(A, believes(B, shared_key(Kab, A, B)));

END;

Final theory representation (size 34): [omitted]

critical properties for this theory:
    believes(B, believes(A, shared_key(Kab, A, B)))
    believes(B, believes(S, Ts))
    believes(B, believes(S, believes(A, shared_key(Kab, A,
         B))))
    believes(B, said(S, Ts))
    believes(B, shared_key(Kab, A, B))
    believes(S, believes(A, Ta))
    believes(S, believes(A, shared_key(Kab, A, B)))
    believes(S, said(A, Ta))

desired property: believes(A, shared_key(Kab, A, B))
   is TRUE
desired property: believes(B, shared_key(Kab, A, B))
   is TRUE
desired property: believes(B, believes(A, shared_key(Kab, A,
         B)))
   is TRUE
desired property: believes(A, believes(B, shared_key(Kab, A,
         B)))
   is FALSE

// ----------------------------------------------
PROTOCOL Yahalom_2;  // Logic: BAN

VARIABLES
    A, B, S: Principal;
    Kab, Kas, Kbs: SKey;
    Ra, Rb: Field;

ASSUMPTIONS
    believes(A, shared_key(Kas, A, S));
```

```
    believes(B, shared_key(Kbs, B, S));
    believes(A, fresh(Ra));
    believes(B, fresh(Rb));
    believes(A, controls(S, shared_key(?K, A, B)));
    believes(B, controls(S, shared_key(?K, A, B)));
    distinct(A, B);
    distinct(A, S);
    distinct(B, S);

MESSAGES
    1. ? -> S: encrypt([Ra, A], Kbs, B);
    2. ? -> A: encrypt([Ra, shared_key(Kab, A, B)], Kas, S);
    3. ? -> A: encrypt([Rb, shared_key(Kab, A, B)], Kbs, S);
    4. ? -> B: encrypt([Rb, shared_key(Kab, A, B)], Kbs, S);
    5. ? -> B: encrypt([Rb, shared_key(Kab, A, B)], Kab, A);

GOALS
    believes(A, shared_key(Kab, A, B));
    believes(B, shared_key(Kab, A, B));
    believes(B, believes(A, shared_key(Kab, A, B)));
    believes(A, believes(B, shared_key(Kab, A, B)));

END;

Final theory representation (size 40): [omitted]

critical properties for this theory:
    believes(A, believes(S, Ra))
    believes(A, believes(S, shared_key(Kab, A, B)))
    believes(A, said(S, Ra))
    believes(A, shared_key(Kab, A, B))
    believes(B, believes(A, Rb))
    believes(B, believes(A, shared_key(Kab, A, B)))
    believes(B, believes(S, Rb))
    believes(B, believes(S, shared_key(Kab, A, B)))
    believes(B, said(A, Rb))
    believes(B, said(S, Rb))
    believes(B, shared_key(Kab, A, B))

desired property: believes(A, shared_key(Kab, A, B))
   is TRUE
desired property: believes(B, shared_key(Kab, A, B))
   is TRUE
desired property: believes(B, believes(A, shared_key(Kab, A,
         B)))
```

```
   is TRUE
desired property: believes(A, believes(B, shared_key(Kab, A,
         B)))
   is FALSE

// -------------------------------------------
PROTOCOL Yahalom_3;  // Logic: BAN

VARIABLES
    A, B, S: Principal;
    Kab, Kas, Kbs: SKey;
    Ra, Rb: Field;

ASSUMPTIONS
    believes(A, shared_key(Kas, A, S));
    believes(B, shared_key(Kbs, B, S));
    believes(S, shared_key(Kas, A, S));
    believes(S, shared_key(Kbs, B, S));
    believes(S, shared_key(Kab, A, B));
    believes(A, fresh(Ra));
    believes(B, fresh(Rb));
    believes(A, controls(S, shared_key(?K, A, B)));
    believes(B, controls(S, shared_key(?K, A, B)));
    believes(S, fresh(shared_key(Kab, A, B)));
    believes(B, controls(S, fresh(shared_key(?K, A, B))));
    believes(B, controls(A, believes(S, fresh(shared_key(?K, A,
         B)))));
    believes(A, controls(S, said(B, ?N)));
    believes(B, secret(Rb, A, B));
    distinct(A, B);
    distinct(A, S);
    distinct(B, S);

MESSAGES
    1. ? -> S: encrypt([Ra, Rb], Kbs, B);
    2. ? -> A: encrypt([shared_key(Kab, A, B),
         fresh(shared_key(Kab, A, B)), Ra, Rb, said(B, Ra)],
         Kas, S);
    3. ? -> A: encrypt(shared_key(Kab, A, B), Kbs, S);
    4. ? -> B: encrypt(shared_key(Kab, A, B), Kbs, S);
    5. ? -> B: encrypt(combine([Rb, shared_key(Kab, A, B),
         believes(S, fresh(shared_key(Kab, A, B)))], Rb), Kab,
         A);

GOALS
    believes(A, shared_key(Kab, A, B));
```

```
    believes(B, shared_key(Kab, A, B));
    believes(B, believes(A, shared_key(Kab, A, B)));
    believes(A, believes(B, Ra));

END;

Final theory representation (size 60): [omitted]

critical properties for this theory:
    believes(A, believes(B, Ra))
    believes(A, believes(S, Ra))
    believes(A, believes(S, Rb))
    believes(A, believes(S, fresh(shared_key(Kab, A, B))))
    believes(A, believes(S, said(B, Ra)))
    believes(A, believes(S, shared_key(Kab, A, B)))
    believes(A, said(B, Ra))
    believes(A, said(S, Ra))
    believes(A, said(S, Rb))
    believes(A, shared_key(Kab, A, B))
    believes(S, said(B, Ra))
    believes(S, said(B, Rb))

desired property: believes(A, shared_key(Kab, A, B))
   is TRUE
desired property: believes(B, shared_key(Kab, A, B))
   is FALSE
desired property: believes(B, believes(A, shared_key(Kab, A,
         B)))
   is FALSE
desired property: believes(A, believes(B, Ra))
   is TRUE

// ------------------------------------------------
PROTOCOL Yahalom_4;  // Logic: BAN

VARIABLES
    A, B, S: Principal;
    Kab, Kas, Kbs: SKey;
    Ra, Rb: Field;

ASSUMPTIONS
    believes(A, shared_key(Kas, A, S));
    believes(B, shared_key(Kbs, B, S));
    believes(S, shared_key(Kas, A, S));
    believes(S, shared_key(Kbs, B, S));
    believes(S, shared_key(Kab, A, B));
```

```
    believes(A, fresh(Ra));
    believes(B, fresh(Rb));
    believes(A, controls(S, shared_key(?K, A, B)));
    believes(B, controls(S, shared_key(?K, A, B)));
    believes(S, fresh(shared_key(Kab, A, B)));
    believes(B, controls(S, fresh(shared_key(?K, A, B))));
    believes(B, controls(A, believes(S, fresh(shared_key(?K, A,
         B))))));
    believes(A, controls(S, said(B, ?N)));
    believes(B, secret(Rb, A, B));
    distinct(A, B);
    distinct(A, S);
    distinct(B, S);

MESSAGES
    1. ? -> S: encrypt([A, Ra], Kbs, B);
    2. ? -> A: encrypt([shared_key(Kab, A, B), said(B, Ra), Ra],
         Kas, S);
    3. ? -> A: encrypt([shared_key(Kab, A, B), Rb], Kbs, S);
    4. ? -> B: encrypt([shared_key(Kab, A, B), Rb], Kbs, S);
    5. ? -> B: encrypt([Rb, shared_key(Kab, A, B)], Kab, A);

GOALS
    believes(A, shared_key(Kab, A, B));
    believes(B, shared_key(Kab, A, B));
    believes(B, believes(A, shared_key(Kab, A, B)));
    believes(A, believes(B, Ra));

END;

Final theory representation (size 62): [omitted]

critical properties for this theory:
    believes(A, believes(B, Ra))
    believes(A, believes(S, Ra))
    believes(A, believes(S, said(B, Ra)))
    believes(A, believes(S, shared_key(Kab, A, B)))
    believes(A, said(B, Ra))
    believes(A, said(S, Ra))
    believes(A, shared_key(Kab, A, B))
    believes(B, believes(A, Rb))
    believes(B, believes(A, shared_key(Kab, A, B)))
    believes(B, believes(S, Rb))
    believes(B, believes(S, shared_key(Kab, A, B)))
    believes(B, said(A, Rb))
```

```
    believes(B, said(S, Rb))
    believes(B, shared_key(Kab, A, B))
    believes(S, said(B, A))
    believes(S, said(B, Ra))

desired property: believes(A, shared_key(Kab, A, B))
   is TRUE
desired property: believes(B, shared_key(Kab, A, B))
   is TRUE
desired property: believes(B, believes(A, shared_key(Kab, A,
         B)))
   is TRUE
desired property: believes(A, believes(B, Ra))
   is TRUE
```

B.2 AUTLOG

AUTLOG, with sample protocol specifications and REVERE analyses:

```
LOGIC AUTLOG;

REWRITES
  comma_commutative:
    comma(?X, ?Y) = comma(?Y, ?X)
  comma_associative_1:
    comma(comma(?X, ?Y), ?Z) = comma(?X, comma(?Y, ?Z))
  comma_associative_2:
    comma(?X, comma(?Y, ?Z)) = comma(comma(?X, ?Y), ?Z)
  shared_key_commutative:
    shared_key(?K, ?Q, ?R) = shared_key(?K, ?R, ?Q)
  secret_commutative:
    secret(?Y, ?Q, ?R) = secret(?Y, ?R, ?Q)

S-RULES
  seeing_list:
     sees(?P, comma(?X, ?Y))
    -------------------------
     sees(?P, ?X)

  list_said:
     believes(?P, said(?Q, comma(?X, ?Y)))
    ---------------------------------------
     believes(?P, said(?Q, ?X))
```

```
list_rec_said:
   believes(?P, says(?Q, comma(?X, ?Y)))
  ---------------------------------------
   believes(?P, says(?Q, ?X))

nonce_verification:
   believes(?P, fresh(?X))
   believes(?P, said(?Q, ?X))
  ----------------------------
   believes(?P, says(?Q, ?X))

jurisdiction:
   believes(?P, controls(?Q, ?X))
   believes(?P, says(?Q, ?X))
  --------------------------------
   believes(?P, ?X)

seeing_shared:
   sees(?P, shared_key(?K, ?P, ?Q))
   sees(?P, encrypt(?X, ?K, ?B))
  ----------------------------------
   sees(?P, ?X)

auth_shared:
   believes(?P, shared_key(?K, ?Q, ?P))
   sees(?P, encrypt(?X, ?K, ?R))
   believes(?P, recognizable(?X))
  ---------------------------------------------
   believes(?P, said(?Q, ?X))
   believes(?P, said(?Q, ?K))
   believes(?P, said(?Q, encrypt(?X, ?K, ?R)))

key_shared:
   sees(?P, encrypt(?X, ?K, ?R))
   believes(?P, shared_key(?K, ?P, ?Q))
   believes(?P, says(?Q, ?X))
  ------------------------------------------------
   believes(?P, says(?Q, shared_key(?K, ?P, ?Q)))

contents_shared:
   believes(?P, says(?Q, encrypt(?X, ?K, ?R)))
   believes(?P, shared_key(?K, ?P, ?Q))
  ---------------------------------------------
   believes(?P, says(?Q, ?X))
```

```
auth_mac:
   believes(?P, shared_key(?K, ?Q, ?P))
   sees(?P, mac(?K, ?X))
   sees(?P, ?X)
  --------------------------------------
   believes(?P, said(?Q, ?X))
   believes(?P, said(?Q, ?K))
   believes(?P, said(?Q, mac(?K, ?X)))

key_mac:
   sees(?P, mac(?K, ?X))
   believes(?P, shared_key(?K, ?P, ?Q))
   believes(?P, says(?Q, ?X))
  ------------------------------------------------
   believes(?P, says(?Q, shared_key(?K, ?P, ?Q)))

contents_mac:
   believes(?P, says(?Q, mac(?K, ?X)))
   believes(?P, shared_key(?K, ?P, ?Q))
  --------------------------------------
   believes(?P, says(?Q, ?X))

seeing_secret:
   sees(?P, combine(?X, ?Y))
  ---------------------------
   sees(?P, ?X)

auth_secret:
   believes(?P, secret(?Y, ?Q, ?P))
   sees(?P, combine(?X, ?Y))
  -----------------------------------------
   believes(?P, said(?Q, ?X))
   believes(?P, said(?Q, ?Y))
   believes(?P, said(?Q, combine(?X, ?Y)))

key_secret:
   sees(?P, combine(?X, ?Y))
   believes(?P, secret(?Y, ?P, ?Q))
   believes(?P, says(?Q, ?X))
  --------------------------------------------
   believes(?P, says(?Q, secret(?Y, ?P, ?Q)))

contents_secret:
   believes(?P, says(?Q, combine(?X, ?Y)))
   believes(?P, secret(?Y, ?P, ?Q))
```

```
   -----------------------------------------
    believes(?P, says(?Q, ?X))

seeing_public:
   sees(?P, public_key(?K, ?P))
   sees(?P, encrypt(?X, ?K, ?R))
  -------------------------------
   sees(?P, ?X)

seeing_sig:
   sees(?P, public_key(?K1, ?Q))
   sees(?P, encrypt(?X, ?K2, ?B))
   inv(?K1, ?K2)
  --------------------------------
   sees(?P, ?X)

auth_sig:
   sees(?P, encrypt(?X, ?K2, ?R))
   believes(?P, public_key(?K1, ?Q))
   believes(?P, recognizable(?X))
   inv(?K1, ?K2)
  ----------------------------------------------
   believes(?P, said(?Q, ?X))
   believes(?P, said(?Q, ?K2))
   believes(?P, said(?Q, encrypt(?X, ?K2, ?R)))

key_sig:
   sees(?P, encrypt(?X, ?K2, ?R))
   believes(?P, public_key(?K1, ?Q))
   believes(?P, says(?Q, ?X))
   inv(?K1, ?K2)
  ---------------------------------------------
   believes(?P, says(?Q, public_key(?K1, ?Q)))

contents_sig:
   believes(?P, says(?Q, encrypt(?X, ?K2, ?R)))
   believes(?P, public_key(?K1, ?Q))
   inv(?K1, ?K2)
  ----------------------------------------------
   believes(?P, says(?Q, ?X))

contents_hash:
   believes(?P, said(?Q, hash(?X)))
   sees(?P, ?X)
  ----------------------------------
```

```
     believes(?P, said(?Q, ?X))

G-RULES
  freshness_list:
     believes(?P, fresh(?X))
    ------------------------------------
     believes(?P, fresh(comma(?X, ?Y)))

  recognizing_list:
     believes(?P, recognizable(?X))
    -------------------------------------------
     believes(?P, recognizable(comma(?X, ?Y)))

  freshness_shared_1:
     believes(?P, fresh(?X))
     sees(?P, shared_key(?K, ?P, ?Q))
    ------------------------------------------
     believes(?P, fresh(encrypt(?X, ?K, ?R)))

  freshness_shared_2:
     believes(?P, fresh(shared_key(?K, ?P, ?Q)))
     sees(?P, shared_key(?K, ?P, ?Q))
    ---------------------------------------------
     believes(?P, fresh(encrypt(?X, ?K, ?R)))

  recognizing_shared:
     believes(?P, recognizable(?X))
     sees(?P, shared_key(?K, ?P, ?Q))
    -------------------------------------------------
     believes(?P, recognizable(encrypt(?X, ?K, ?R)))

  freshness_mac_1:
     believes(?P, fresh(?X))
     sees(?P, shared_key(?K, ?P, ?Q))
    ----------------------------------
     believes(?P, fresh(mac(?K, ?X)))

  freshness_mac_2:
     believes(?P, fresh(shared_key(?K, ?P, ?Q)))
     sees(?P, shared_key(?K, ?P, ?Q))
    ---------------------------------------------
     believes(?P, fresh(mac(?K, ?X)))

  recognizing_mac:
     believes(?P, recognizable(?X))
```

```
    sees(?P, shared_key(?K, ?P, ?Q))
   -----------------------------------------
    believes(?P, recognizable(mac(?K, ?X)))

freshness_secret_1:
    believes(?P, fresh(?X))
   --------------------------------------
    believes(?P, fresh(combine(?X, ?Y)))

freshness_secret_2:
    believes(?P, fresh(secret(?Y, ?P, ?Q)))
   -----------------------------------------
    believes(?P, fresh(combine(?X, ?Y)))

recognizing_secret:
    believes(?P, recognizable(?X))
   ---------------------------------------------
    believes(?P, recognizable(combine(?X, ?Y)))

freshness_public_1:
    believes(?P, fresh(?X))
    sees(?P, public_key(?K, ?Q))
   ------------------------------------------
    believes(?P, fresh(encrypt(?X, ?K, ?R)))

freshness_public_2:
    believes(?P, fresh(public_key(?K, ?Q)))
    sees(?P, public_key(?K, ?Q))
   ------------------------------------------
    believes(?P, fresh(encrypt(?X, ?K, ?R)))

recognizing_public:
    believes(?P, recognizable(?X))
    believes(?P, public_key(?K, ?P))
   -------------------------------------------------
    believes(?P, recognizable(encrypt(?X, ?K, ?R)))

freshness_sig_1:
    believes(?P, fresh(?X))
    sees(?P, public_key(?K1, ?Q))
    inv(?K1, ?K2)
   -------------------------------------------
    believes(?P, fresh(encrypt(?X, ?K2, ?R)))

freshness_sig_2:
```

```
     believes(?P, fresh(public_key(?K1, ?Q)))
     sees(?P, public_key(?K1, ?Q))
     inv(?K1, ?K2)
    -------------------------------------------
     believes(?P, fresh(encrypt(?X, ?K2, ?R)))

  recognizing_sig:
     believes(?P, recognizable(?X))
     sees(?P, public_key(?K1, ?Q))
     inv(?K1, ?K2)
    --------------------------------------------------
     believes(?P, recognizable(encrypt(?X, ?K2, ?R)))

  freshness_hash:
     believes(?P, fresh(?X))
    -------------------------------
     believes(?P, fresh(hash(?X)))

  recognizing_hash:
     believes(?P, recognizable(?X))
    --------------------------------------
     believes(?P, recognizable(hash(?X)))

END;

// ----------------------------------------------
PROTOCOL Challenge_1;  // Logic: AUTLOG

VARIABLES
    A, B: Principal;
    Kab: SKey;
    Rb: Field;

ASSUMPTIONS
    believes(B, fresh(Rb));
    believes(B, secret(Rb, A, B));

MESSAGES
    1. ? -> A: encrypt(secret(Rb, A, B), Kab, B);
    2. ? -> B: combine(Rb, Rb);

GOALS
    believes(B, says(A, Rb));
    believes(A, said(B, secret(Rb, A, B)));

END;
```

```
Final theory representation (size 10):
    believes(B, fresh(Rb))
    believes(B, said(A, Rb))
    believes(B, said(A, combine(Rb, Rb)))
    believes(B, says(A, Rb))
    believes(B, says(A, combine(Rb, Rb)))
    believes(B, says(A, secret(Rb, A, B)))
    believes(B, secret(Rb, A, B))
    sees(A, encrypt(secret(Rb, A, B), Kab, B))
    sees(B, Rb)
    sees(B, combine(Rb, Rb))
critical properties for this theory:
    believes(B, said(A, Rb))
    believes(B, says(A, Rb))
    believes(B, says(A, combine(Rb, Rb)))
    believes(B, says(A, secret(Rb, A, B)))

desired property: believes(B, says(A, Rb))
   is TRUE
desired property: believes(A, said(B, secret(Rb, A, B)))
   is FALSE

// ---------------------------------------------
PROTOCOL Challenge_2;  // Logic: AUTLOG

VARIABLES
    A, B: Principal;
    Kab: SKey;
    Rb: Field;

ASSUMPTIONS
    believes(B, fresh(Rb));
    believes(B, recognizable(Rb));
    believes(B, shared_key(Kab, A, B));
    sees(B, shared_key(Kab, A, B));

MESSAGES
    1. ? -> A: Rb;
    2. ? -> B: encrypt(Rb, Kab, A);

GOALS
    believes(B, says(A, Rb));

END;
```

```
Final theory representation (size 13):
    believes(B, fresh(Rb))
    believes(B, recognizable(Rb))
    believes(B, said(A, Kab))
    believes(B, said(A, Rb))
    believes(B, said(A, encrypt(Rb, Kab, A)))
    believes(B, says(A, Rb))
    believes(B, says(A, encrypt(Rb, Kab, A)))
    believes(B, says(A, shared_key(Kab, A, B)))
    believes(B, shared_key(Kab, A, B))
    sees(A, Rb)
    sees(B, Rb)
    sees(B, encrypt(Rb, Kab, A))
    sees(B, shared_key(Kab, A, B))
critical properties for this theory:
    believes(B, said(A, Kab))
    believes(B, said(A, Rb))
    believes(B, says(A, Rb))
    believes(B, says(A, encrypt(Rb, Kab, A)))
    believes(B, says(A, shared_key(Kab, A, B)))

desired property: believes(B, says(A, Rb))
   is TRUE

// ----------------------------------------------
PROTOCOL Kerberos_Autlog;  // Logic: AUTLOG

VARIABLES
    A, B, S: Principal;
    Kab, Kas, Kbs: SKey;
    Ta, Ts: Field;

ASSUMPTIONS
    believes(A, shared_key(Kas, S, A));
    believes(B, shared_key(Kbs, S, B));
    believes(S, shared_key(Kas, A, S));
    believes(S, shared_key(Kbs, B, S));
    believes(S, shared_key(Kab, A, B));
    believes(A, controls(S, shared_key(?K, A, B)));
    believes(B, controls(S, shared_key(?K, A, B)));
    believes(A, fresh(Ts));
    believes(B, fresh(Ts));
    believes(A, fresh(Ta));
    believes(B, fresh(Ta));
    believes(A, recognizable(shared_key(Kab, A, B)));
    believes(B, recognizable(shared_key(Kab, A, B)));
```

```
    sees(A, shared_key(Kas, S, A));
    sees(B, shared_key(Kbs, S, B));
    sees(S, shared_key(Kas, A, S));
    sees(S, shared_key(Kbs, B, S));
    sees(S, shared_key(Kab, A, B));

MESSAGES
    1. ? -> A: encrypt([Ts, shared_key(Kab, A, B), encrypt([Ts,
         shared_key(Kab, A, B)], Kbs, S)], Kas, S);
    2. ? -> B: [encrypt([Ts, shared_key(Kab, A, B)], Kbs, S),
         encrypt([Ta, shared_key(Kab, A, B)], Kab, A)];
    3. ? -> A: encrypt([Ta, shared_key(Kab, A, B)], Kab, B);

GOALS
    believes(A, shared_key(Kab, A, B));
    believes(B, shared_key(Kab, A, B));
    believes(B, says(A, shared_key(Kab, A, B)));
    believes(A, says(B, shared_key(Kab, A, B)));

END;

Final theory representation (size 79): [omitted]

critical properties for this theory:
    believes(A, said(B, Kab))
    believes(A, said(B, Ta))
    believes(A, said(S, Kas))
    believes(A, said(S, Ts))
    believes(A, says(B, Ta))
    believes(A, says(B, encrypt([Ta, shared_key(Kab, A, B)],
         Kab, B)))
    believes(A, says(B, shared_key(Kab, A, B)))
    believes(A, says(S, Ts))
    believes(A, says(S, encrypt([Ts, encrypt([Ts,
         shared_key(Kab, A, B)], Kbs, S), shared_key(Kab, A,
         B)], Kas, S)))
    believes(A, says(S, encrypt([Ts, shared_key(Kab, A, B)],
         Kbs, S)))
    believes(A, says(S, shared_key(Kab, A, B)))
    believes(A, says(S, shared_key(Kas, A, S)))
    believes(A, shared_key(Kab, A, B))
    believes(A, shared_key(Kas, A, S))
    believes(B, said(A, Kab))
    believes(B, said(A, Ta))
    believes(B, said(S, Kbs))
```

```
    believes(B, said(S, Ts))
    believes(B, says(A, Ta))
    believes(B, says(A, encrypt([Ta, shared_key(Kab, A, B)],
         Kab, A)))
    believes(B, says(A, shared_key(Kab, A, B)))
    believes(B, says(S, Ts))
    believes(B, says(S, encrypt([Ts, shared_key(Kab, A, B)],
         Kbs, S)))
    believes(B, says(S, shared_key(Kab, A, B)))
    believes(B, says(S, shared_key(Kbs, B, S)))
    believes(B, shared_key(Kab, A, B))
    believes(B, shared_key(Kbs, B, S))

desired property: believes(A, shared_key(Kab, A, B))
   is TRUE
desired property: believes(B, shared_key(Kab, A, B))
   is TRUE
desired property: believes(B, says(A, shared_key(Kab, A, B)))
   is TRUE
desired property: believes(A, says(B, shared_key(Kab, A, B)))
   is TRUE
```

B.3 Accountability

Kailar's accountability logic, with sample protocol specifications and REVERE analyses:

```
LOGIC Accountability;

REWRITES
  comma_commutative:
    comma(?X, ?Y) = comma(?Y, ?X)
  comma_associative_1:
    comma(comma(?X, ?Y), ?Z) = comma(?X, comma(?Y, ?Z))
  comma_associative_2:
    comma(?X, comma(?Y, ?Z)) = comma(comma(?X, ?Y), ?Z)

S-RULES
  inf:
     implies(?X, ?Y)
     can_prove(?P, ?X)
    -------------------
     can_prove(?P, ?Y)
```

```
  conj:
     can_prove(?P, comma(?X, ?Y))
    ------------------------------
     can_prove(?P, ?X)

  sign:
     receives(?P, signed_with(?M, ?K'))
     can_prove(?P, authenticates(?K, ?Q))
     inv(?K, ?K')
    --------------------------------------
     can_prove(?P, says(?Q, ?M))

  extract_comma_1:
     can_prove(?P, says(?Q, comma(?X, ?Y)))
    ----------------------------------------
     can_prove(?P, says(?Q, ?X))

  extract_comma_2:
     receives(?P, comma(?X, ?Y))
    -----------------------------
     receives(?P, ?X)

  extract_signed:
     receives(?P, signed_with(?X, ?K))
    -----------------------------------
     receives(?P, ?X)

  trust:
     can_prove(?P, says(?Q, ?X))
     can_prove(?P, is_trusted_on(?Q, ?X))
    --------------------------------------
     can_prove(?P, ?X)

END;

// ----------------------------------------------
PROTOCOL Netbill_1;  // Logic: Accountability

VARIABLES
    E, P, S: Principal;
    Kb', Ke, Ke', Ks, Ks': PKey;
    Service, ServiceAck: Field;

ASSUMPTIONS
    can_prove(S, authenticates(Ke, E));
    can_prove(E, authenticates(Ks, S));
```

```
    can_prove(S, authenticates(Kb, B));
    can_prove(E, authenticates(Kb, B));
    implies(says(S, Price(?Amt)), AgreesTo(S, Price(?Amt)));
    implies(says(E, Price(?Amt)), AgreesTo(E, Price(?Amt)));
    implies(says(S, Service), RendersItem(S));
    implies(says(E, ServiceAck), ReceivesItem(E));
    knows_key(S, Ks’);
    knows_key(E, Ke’);
    knows_key(B, Kb’);
    inv(Ks, Ks’);
    inv(Ke, Ke’);
    inv(Kb, Kb’);

MESSAGES
    1. ? -> E: signed_with(Price(P), Ks’);
    2. ? -> S: signed_with([signed_with(Price(P), Ks’),
         Price(P)], Ke’);
    3. ? -> E: signed_with(Service, Ks’);
    4. ? -> S: signed_with(ServiceAck, Ke’);
    5. ? -> S:
         [signed_with(encrypt([signed_with(
        [signed_with(Price(P), Ks’), Price(P)], Ke’),
         signed_with(ServiceAck, Ke’)], Ks), Kb’),
         signed_with(encrypt([signed_with([signed_with(Price(P),
         Ks’), Price(P)], Ke’), signed_with(ServiceAck, Ke’)],
         Ke), Kb’)];
    6. ? -> E:
         signed_with(encrypt([signed_with([signed_with(Price(P),
         Ks’), Price(P)], Ke’), signed_with(ServiceAck, Ke’)],
         Ke), Kb’);

GOALS
    can_prove(E, AgreesTo(S, Price(P)));
    can_prove(S, AgreesTo(E, Price(P)));
    can_prove(E, RendersItem(S));
    can_prove(S, ReceivesItem(E));
    can_prove(E, says(B, [signed_with([signed_with(Price(P),
         Ks’), Price(P)], Ke’), signed_with(ServiceAck,
         Ke’)]));
    can_prove(S, says(B, [signed_with([signed_with(Price(P),
         Ks’), Price(P)], Ke’), signed_with(ServiceAck,
         Ke’)]));

END;
```

```
Final theory representation (size 44):
    can_prove(E, AgreesTo(S, Price(P)))
    can_prove(E, RendersItem(S))
    can_prove(E, authenticates(Kb, B))
    can_prove(E, authenticates(Ks, S))
    can_prove(E, says(B, encrypt([signed_with(ServiceAck, Ke'),
         signed_with([Price(P), signed_with(Price(P), Ks')],
         Ke')], Ke)))
    can_prove(E, says(S, Price(P)))
    can_prove(E, says(S, Service))
    can_prove(S, AgreesTo(E, Price(P)))
    can_prove(S, ReceivesItem(E))
    can_prove(S, authenticates(Kb, B))
    can_prove(S, authenticates(Ke, E))
    can_prove(S, says(B, encrypt([signed_with(ServiceAck, Ke'),
         signed_with([Price(P), signed_with(Price(P), Ks')],
         Ke')], Ke)))
    can_prove(S, says(B, encrypt([signed_with(ServiceAck, Ke'),
         signed_with([Price(P), signed_with(Price(P), Ks')],
         Ke')], Ks)))
    can_prove(S, says(E, Price(P)))
    can_prove(S, says(E, ServiceAck))
    can_prove(S, says(E, [Price(P), signed_with(Price(P),
         Ks')]))
    can_prove(S, says(E, signed_with(Price(P), Ks')))
    implies(says(E, Price(?CAN0)), AgreesTo(E, Price(?CAN0)))
    implies(says(E, ServiceAck), ReceivesItem(E))
    implies(says(S, Price(?CAN0)), AgreesTo(S, Price(?CAN0)))
    implies(says(S, Service), RendersItem(S))
    inv(Kb, Kb')
    inv(Ke, Ke')
    inv(Ks, Ks')
    knows_key(B, Kb')
    knows_key(E, Ke')
    knows_key(S, Ks')
    receives(E, Price(P))
    receives(E, Service)
    receives(E, encrypt([signed_with(ServiceAck, Ke'),
         signed_with([Price(P), signed_with(Price(P), Ks')],
         Ke')], Ke))
    receives(E, signed_with(Price(P), Ks'))
    receives(E, signed_with(Service, Ks'))
    receives(E, signed_with(encrypt([signed_with(ServiceAck,
         Ke'), signed_with([Price(P), signed_with(Price(P),
         Ks')], Ke')], Ke), Kb'))
```

```
    receives(S, Price(P))
    receives(S, ServiceAck)
    receives(S, [Price(P), signed_with(Price(P), Ks’)])
    receives(S, [signed_with(encrypt([signed_with(ServiceAck,
         Ke’), signed_with([Price(P), signed_with(Price(P),
         Ks’)], Ke’)], Ke), Kb’),
         signed_with(encrypt([signed_with(ServiceAck, Ke’),
         signed_with([Price(P), signed_with(Price(P), Ks’)],
         Ke’)], Ks), Kb’)])
    receives(S, encrypt([signed_with(ServiceAck, Ke’),
         signed_with([Price(P), signed_with(Price(P), Ks’)],
         Ke’)], Ke))
    receives(S, encrypt([signed_with(ServiceAck, Ke’),
         signed_with([Price(P), signed_with(Price(P), Ks’)],
         Ke’)], Ks))
    receives(S, signed_with(Price(P), Ks’))
    receives(S, signed_with(ServiceAck, Ke’))
    receives(S, signed_with([Price(P), signed_with(Price(P),
         Ks’)], Ke’))
    receives(S, signed_with(encrypt([signed_with(ServiceAck,
         Ke’), signed_with([Price(P), signed_with(Price(P),
         Ks’)], Ke’)], Ke), Kb’))
    receives(S, signed_with(encrypt([signed_with(ServiceAck,
         Ke’), signed_with([Price(P), signed_with(Price(P),
         Ks’)], Ke’)], Ks), Kb’))
critical properties for this theory:
    can_prove(E, AgreesTo(S, Price(P)))
    can_prove(E, RendersItem(S))
    can_prove(E, says(B, encrypt([signed_with(ServiceAck, Ke’),
         signed_with([Price(P), signed_with(Price(P), Ks’)],
         Ke’)], Ke)))
    can_prove(E, says(S, Price(P)))
    can_prove(E, says(S, Service))
    can_prove(S, AgreesTo(E, Price(P)))
    can_prove(S, ReceivesItem(E))
    can_prove(S, says(B, encrypt([signed_with(ServiceAck, Ke’),
         signed_with([Price(P), signed_with(Price(P), Ks’)],
         Ke’)], Ke)))
    can_prove(S, says(B, encrypt([signed_with(ServiceAck, Ke’),
         signed_with([Price(P), signed_with(Price(P), Ks’)],
         Ke’)], Ks)))
    can_prove(S, says(E, Price(P)))
    can_prove(S, says(E, ServiceAck))
    can_prove(S, says(E, signed_with(Price(P), Ks’)))
```

```
desired property: can_prove(E, AgreesTo(S, Price(P)))
   is TRUE
desired property: can_prove(S, AgreesTo(E, Price(P)))
   is TRUE
desired property: can_prove(E, RendersItem(S))
   is TRUE
desired property: can_prove(S, ReceivesItem(E))
   is TRUE
desired property: can_prove(E, says(B,
         [signed_with([signed_with(Price(P), Ks'), Price(P)],
         Ke'), signed_with(ServiceAck, Ke')]))
   is FALSE
desired property: can_prove(S, says(B,
         [signed_with([signed_with(Price(P), Ks'), Price(P)],
         Ke'), signed_with(ServiceAck, Ke')]))
   is FALSE

// ----------------------------------------------
PROTOCOL Netbill_1a;  // Logic: Accountability

VARIABLES
    B, E, P, S: Principal;
    Kb, Kb', Ke, Ke', Ks, Ks': PKey;
    Service, ServiceAck: Field;

ASSUMPTIONS
    can_prove(S, authenticates(Ke, E));
    can_prove(E, authenticates(Ks, S));
    can_prove(S, authenticates(Kb, B));
    can_prove(E, authenticates(Kb, B));
    implies(says(S, Price(?Amt)), AgreesTo(S, Price(?Amt)));
    implies(says(E, Price(?Amt)), AgreesTo(E, Price(?Amt)));
    implies(says(S, Service), RendersItem(S));
    implies(says(E, ServiceAck), ReceivesItem(E));
    knows_key(S, Ks');
    knows_key(E, Ke');
    knows_key(B, Kb');
    inv(Ks, Ks');
    inv(Ke, Ke');
    inv(Kb, Kb');

MESSAGES
    1. ? -> E: signed_with(Price(P), Ks');
    2. ? -> S: signed_with([signed_with(Price(P), Ks'),
         Price(P)], Ke');
    3. ? -> E: signed_with(Service, Ks');
```

```
    4. ? -> S: signed_with(ServiceAck, Ke');
    5. ? -> B:
         encrypt(signed_with([signed_with([signed_with(Price(P),
         Ks'), Price(P)], Ke'), signed_with(ServiceAck, Ke')],
         Ks'), Kb);
    6. ? -> S:
         [encrypt(signed_with([signed_with(
        [signed_with(Price(P), Ks'), Price(P)], Ke'),
         signed_with(ServiceAck, Ke')], Kb'), Ks),
         encrypt(signed_with([signed_with([signed_with(Price(P),
         Ks'), Price(P)], Ke'), signed_with(ServiceAck, Ke')],
         Kb'), Ke)];
    7. ? -> E:
         encrypt(signed_with([signed_with([signed_with(Price(P),
         Ks'), Price(P)], Ke'), signed_with(ServiceAck, Ke')],
         Kb'), Ke);

GOALS
    can_prove(E, AgreesTo(S, Price(P)));
    can_prove(S, AgreesTo(E, Price(P)));
    can_prove(E, RendersItem(S));
    can_prove(S, ReceivesItem(E));
    can_prove(E, says(B, [signed_with([signed_with(Price(P),
         Ks'), Price(P)], Ke'), signed_with(ServiceAck,
         Ke')]));
    can_prove(S, says(B, [signed_with([signed_with(Price(P),
         Ks'), Price(P)], Ke'), signed_with(ServiceAck,
         Ke')]));

END;

Final theory representation (size 39): [omitted]

critical properties for this theory:
    can_prove(E, AgreesTo(S, Price(P)))
    can_prove(E, RendersItem(S))
    can_prove(E, says(S, Price(P)))
    can_prove(E, says(S, Service))
    can_prove(S, AgreesTo(E, Price(P)))
    can_prove(S, ReceivesItem(E))
    can_prove(S, says(E, Price(P)))
    can_prove(S, says(E, ServiceAck))
    can_prove(S, says(E, signed_with(Price(P), Ks')))

desired property: can_prove(E, AgreesTo(S, Price(P)))
```

```
   is TRUE
desired property: can_prove(S, AgreesTo(E, Price(P)))
   is TRUE
desired property: can_prove(E, RendersItem(S))
   is TRUE
desired property: can_prove(S, ReceivesItem(E))
   is TRUE
desired property: can_prove(E, says(B,
         [signed_with([signed_with(Price(P), Ks'), Price(P)],
         Ke'), signed_with(ServiceAck, Ke')]))
   is FALSE
desired property: can_prove(S, says(B,
         [signed_with([signed_with(Price(P), Ks'), Price(P)],
         Ke'), signed_with(ServiceAck, Ke')]))
   is FALSE

// ------------------------------------------------
PROTOCOL Netbill_2;  // Logic: Accountability

VARIABLES
    B, E, P, S: Principal;
    Keb, Kes: SKey;
    Kb, Kb', Ks, Ks': PKey;
    Service, ServiceAck: Field;

ASSUMPTIONS
    can_prove(S, authenticates(Ke, E));
    can_prove(E, authenticates(Ks, S));
    can_prove(S, authenticates(Kb, B));
    can_prove(E, authenticates(Kb, B));
    implies(says(S, Price(?Amt)), AgreesTo(S, Price(?Amt)));
    implies(says(E, Price(?Amt)), AgreesTo(E, Price(?Amt)));
    implies(says(S, Service), RendersItem(S));
    implies(says(E, ServiceAck), ReceivesItem(E));
    knows_key(S, Ks');
    knows_key(E, Ke');
    knows_key(B, Kb');
    knows_key(S, Kes);
    knows_key(E, Kes);
    knows_key(B, Keb);
    knows_key(E, Keb);
    inv(Keb, Keb);
    inv(Kes, Kes);
    inv(Ks, Ks');
    inv(Ke, Ke');
    inv(Kb, Kb');
```

```
MESSAGES
    1. ? -> E: signed_with(Price(P), Ks');
    2. ? -> S: [encrypt([signed_with(Price(P), Ks'), Price(P)],
         Keb), encrypt(Price(P), Kes)];
    3. ? -> E: signed_with(Service, Ks');
    4. ? -> S: [encrypt(ServiceAck, Kes), encrypt(ServiceAck,
         Keb)];
    5. ? -> B:
         signed_with(encrypt([encrypt([signed_with(Price(P),
         Ks'), Price(P)], Keb), signed_with(Price(P), Ks'),
         signed_with(encrypt(ServiceAck, Keb), Ks')], Kb),
         Ks');
    6. ? -> S:
         [signed_with(encrypt([encrypt([signed_with(Price(P),
         Ks'), Price(P)], Keb), signed_with(Price(P), Ks'),
         signed_with(encrypt(ServiceAck, Keb), Ks')], Keb),
         Kb'),
         signed_with(encrypt([encrypt([signed_with(Price(P),
         Ks'), Price(P)], Keb), signed_with(Price(P), Ks'),
         signed_with(encrypt(ServiceAck, Keb), Ks')], Ks),
         Kb')];
    7. ? -> E:
         signed_with(encrypt([encrypt([signed_with(Price(P),
         Ks'), Price(P)], Keb), signed_with(Price(P), Ks'),
         signed_with(encrypt(ServiceAck, Keb), Ks')], Keb),
         Kb');

GOALS
    can_prove(E, AgreesTo(S, Price(P)));
    can_prove(S, AgreesTo(E, Price(P)));
    can_prove(E, RendersItem(S));
    can_prove(S, ReceivesItem(E));
    can_prove(E, says(B, [encrypt([signed_with(Price(P), Ks'),
         Price(P)], Keb), signed_with(Price(P), Ks'),
         signed_with(encrypt(ServiceAck, Keb), Ks')]));
    can_prove(S, says(B, [encrypt([signed_with(Price(P), Ks'),
         Price(P)], Keb), signed_with(Price(P), Ks'),
         signed_with(encrypt(ServiceAck, Keb), Ks')]));

END;

Final theory representation (size 46): [omitted]

critical properties for this theory:
```

```
    can_prove(E, AgreesTo(S, Price(P)))
    can_prove(E, RendersItem(S))
    can_prove(E, says(B, encrypt([encrypt([Price(P),
         signed_with(Price(P), Ks')], Keb),
         signed_with(Price(P), Ks'),
         signed_with(encrypt(ServiceAck, Keb), Ks')], Keb)))
    can_prove(E, says(S, Price(P)))
    can_prove(E, says(S, Service))
    can_prove(S, says(B, encrypt([encrypt([Price(P),
         signed_with(Price(P), Ks')], Keb),
         signed_with(Price(P), Ks'),
         signed_with(encrypt(ServiceAck, Keb), Ks')], Keb)))
    can_prove(S, says(B, encrypt([encrypt([Price(P),
         signed_with(Price(P), Ks')], Keb),
         signed_with(Price(P), Ks'),
         signed_with(encrypt(ServiceAck, Keb), Ks')], Ks)))

desired property: can_prove(E, AgreesTo(S, Price(P)))
   is TRUE
desired property: can_prove(S, AgreesTo(E, Price(P)))
   is FALSE
desired property: can_prove(E, RendersItem(S))
   is TRUE
desired property: can_prove(S, ReceivesItem(E))
   is FALSE
desired property: can_prove(E, says(B,
         [encrypt([signed_with(Price(P), Ks'), Price(P)], Keb),
         signed_with(Price(P), Ks'),
         signed_with(encrypt(ServiceAck, Keb), Ks')]))
   is FALSE
desired property: can_prove(S, says(B,
         [encrypt([signed_with(Price(P), Ks'), Price(P)], Keb),
         signed_with(Price(P), Ks'),
         signed_with(encrypt(ServiceAck, Keb), Ks')]))
   is FALSE

// ---------------------------------------------
PROTOCOL Netbill_2a;  // Logic: Accountability

VARIABLES
    B, E, P, S: Principal;
    Keb: SKey;
    Kb, Kb', Ke', Ks, Ks': PKey;
    Service, ServiceAck: Field;

ASSUMPTIONS
```

```
    can_prove(S, authenticates(Ke, E));
    can_prove(E, authenticates(Ks, S));
    can_prove(S, authenticates(Kb, B));
    can_prove(E, authenticates(Kb, B));
    implies(says(S, Price(?Amt)), AgreesTo(S, Price(?Amt)));
    implies(says(E, Price(?Amt)), AgreesTo(E, Price(?Amt)));
    implies(says(S, Service), RendersItem(S));
    implies(says(E, ServiceAck), ReceivesItem(E));
    knows_key(S, Ks');
    knows_key(E, Ke');
    knows_key(B, Kb');
    knows_key(S, Kes);
    knows_key(E, Kes);
    knows_key(B, Keb);
    knows_key(E, Keb);
    inv(Keb, Keb);
    inv(Kes, Kes);
    inv(Ks, Ks');
    inv(Ke, Ke');
    inv(Kb, Kb');

MESSAGES
    1. ? -> E: signed_with(Price(P), Ks');
    2. ? -> S: [encrypt([signed_with(Price(P), Ks'), Price(P)],
         Keb), signed_with(Price(P), Ke')];
    3. ? -> E: signed_with(Service, Ks');
    4. ? -> S: [signed_with(ServiceAck, Ke'),
         encrypt(ServiceAck, Keb)];
    5. ? -> B:
         signed_with(encrypt([encrypt([signed_with(Price(P),
         Ks'), Price(P)], Keb), signed_with(Price(P), Ks'),
         signed_with(encrypt(ServiceAck, Keb), Ks')], Kb),
         Ks');
    6. ? -> S:
         [signed_with(encrypt([encrypt([signed_with(Price(P),
         Ks'), Price(P)], Keb), signed_with(Price(P), Ks'),
         signed_with(encrypt(ServiceAck, Keb), Ks')], Keb),
         Kb'),
         signed_with(encrypt([encrypt([signed_with(Price(P),
         Ks'), Price(P)], Keb), signed_with(Price(P), Ks'),
         signed_with(encrypt(ServiceAck, Keb), Ks')], Ks),
         Kb')];
    7. ? -> E:
         signed_with(encrypt([encrypt([signed_with(Price(P),
         Ks'), Price(P)], Keb), signed_with(Price(P), Ks'),
```

```
         signed_with(encrypt(ServiceAck, Keb), Ks')], Keb),
         Kb');

GOALS
    can_prove(E, AgreesTo(S, Price(P)));
    can_prove(S, AgreesTo(E, Price(P)));
    can_prove(E, RendersItem(S));
    can_prove(S, ReceivesItem(E));
    can_prove(E, says(B, [encrypt([signed_with(Price(P), Ks'),
         Price(P)], Keb), signed_with(Price(P), Ks'),
         signed_with(encrypt(ServiceAck, Keb), Ks')]));
    can_prove(S, says(B, [encrypt([signed_with(Price(P), Ks'),
         Price(P)], Keb), signed_with(Price(P), Ks'),
         signed_with(encrypt(ServiceAck, Keb), Ks')]));

END;

Final theory representation (size 52): [omitted]

critical properties for this theory:
    can_prove(E, AgreesTo(S, Price(P)))
    can_prove(E, RendersItem(S))
    can_prove(E, says(B, encrypt([encrypt([Price(P),
         signed_with(Price(P), Ks')], Keb),
         signed_with(Price(P), Ks'),
         signed_with(encrypt(ServiceAck, Keb), Ks')], Keb)))
    can_prove(E, says(S, Price(P)))
    can_prove(E, says(S, Service))
    can_prove(S, AgreesTo(E, Price(P)))
    can_prove(S, ReceivesItem(E))
    can_prove(S, says(B, encrypt([encrypt([Price(P),
         signed_with(Price(P), Ks')], Keb),
         signed_with(Price(P), Ks'),
         signed_with(encrypt(ServiceAck, Keb), Ks')], Keb)))
    can_prove(S, says(B, encrypt([encrypt([Price(P),
         signed_with(Price(P), Ks')], Keb),
         signed_with(Price(P), Ks'),
         signed_with(encrypt(ServiceAck, Keb), Ks')], Ks)))
    can_prove(S, says(E, Price(P)))
    can_prove(S, says(E, ServiceAck))

desired property: can_prove(E, AgreesTo(S, Price(P)))
   is TRUE
desired property: can_prove(S, AgreesTo(E, Price(P)))
   is TRUE
```

```
desired property: can_prove(E, RendersItem(S))
   is TRUE
desired property: can_prove(S, ReceivesItem(E))
   is TRUE
desired property: can_prove(E, says(B,
         [encrypt([signed_with(Price(P), Ks'), Price(P)], Keb),
         signed_with(Price(P), Ks'),
         signed_with(encrypt(ServiceAck, Keb), Ks')]))
   is FALSE
desired property: can_prove(S, says(B,
         [encrypt([signed_with(Price(P), Ks'), Price(P)], Keb),
         signed_with(Price(P), Ks'),
         signed_with(encrypt(ServiceAck, Keb), Ks')]))
   is FALSE

// ----------------------------------------------
PROTOCOL SPX;  // Logic: Accountability

VARIABLES
    C, S: Principal;
    Kcdc', Kdel, Kta1', Kta2': SKey;
    Kc, Kc', Ks: PKey;

ASSUMPTIONS
    can_prove(C, authenticates(Kcdc, CDC));
    can_prove(S, authenticates(Kcdc, CDC));
    can_prove(C, authenticates(Kta1, TA1));
    can_prove(S, authenticates(Kta2, TA2));
    can_prove(C, is_trusted_on(CDC, ?X));
    can_prove(S, is_trusted_on(CDC, ?X));
    can_prove(C, is_trusted_on(TA1, ?X));
    can_prove(S, is_trusted_on(TA2, ?X));
    can_prove(S, is_trusted_on(C, authenticates(?K, C)));
    knows_key(S, Ks');
    knows_key(C, Kc');
    knows_key(CDC, Kcdc');
    inv(Ks, Ks');
    inv(Kc, Kc');
    inv(Kdel, Kdel');
    inv(Kcdc, Kcdc');
    inv(Kta1, Kta1');
    inv(Kta2, Kta2');

MESSAGES
    1. ? -> C: signed_with(signed_with(authenticates(Ks, S),
         Kta1'), Kcdc');
```

```
    2. ? -> S: signed_with(authenticates(Kdel, C), Kc');
    3. ? -> S: signed_with(signed_with(authenticates(Kc, C),
         Kta2'), Kcdc');

GOALS
    can_prove(C, authenticates(Ks, S));
    can_prove(S, authenticates(Kc, C));
    can_prove(S, authenticates(Kdel, C));

END;

Final theory representation (size 36): [omitted]

critical properties for this theory:
    can_prove(C, authenticates(Ks, S))
    can_prove(C, is_trusted_on(CDC, ?CAN0))
    can_prove(C, is_trusted_on(TA1, ?CAN0))
    can_prove(C, says(CDC, signed_with(authenticates(Ks, S),
         Kta1')))
    can_prove(C, says(TA1, authenticates(Ks, S)))
    can_prove(C, signed_with(authenticates(Ks, S), Kta1'))
    can_prove(S, authenticates(Kc, C))
    can_prove(S, authenticates(Kdel, C))
    can_prove(S, is_trusted_on(C, authenticates(?CAN0, C)))
    can_prove(S, is_trusted_on(CDC, ?CAN0))
    can_prove(S, is_trusted_on(TA2, ?CAN0))
    can_prove(S, says(C, authenticates(Kdel, C)))
    can_prove(S, says(CDC, signed_with(authenticates(Kc, C),
         Kta2')))
    can_prove(S, says(TA2, authenticates(Kc, C)))
    can_prove(S, signed_with(authenticates(Kc, C), Kta2'))

desired property: can_prove(C, authenticates(Ks, S))
   is TRUE
desired property: can_prove(S, authenticates(Kc, C))
   is TRUE
desired property: can_prove(S, authenticates(Kdel, C))
   is TRUE
```

B.4 RV

RV rules and rewrites, with sample protocol specifications and REVERE analyses:

```
LOGIC RV;
```

```
REWRITES
  comma_commutative:
    comma(?X, ?Y) = comma(?Y, ?X)
  comma_associative_1:
    comma(comma(?X, ?Y), ?Z) = comma(?X, comma(?Y, ?Z))
  comma_associative_2:
    comma(?X, comma(?Y, ?Z)) = comma(comma(?X, ?Y), ?Z)
  seq_associative_1:
    seq(seq(?X, ?Y), ?Z) = seq(?X, seq(?Y, ?Z))
  seq_associative_2:
    seq(?X, seq(?Y, ?Z)) = seq(seq(?X, ?Y), ?Z)
  shared_key_commutative:
    shared_key(?K, ?Q, ?R) = shared_key(?K, ?R, ?Q)
  secret_commutative:
    secret(?Y, ?Q, ?R) = secret(?Y, ?R, ?Q)

S-RULES
  seeing_list:
     sees(?P, comma(?X, ?Y))
    -------------------------
     sees(?P, ?X)

  seeing_seq:
     sees(?P, seq(?X, ?Y))
    -----------------------
     sees(?P, ?X)
     sees(?P, ?Y)

  seeing_tagged:
     sees(?P, tagged(?T, ?Y))
    --------------------------
     sees(?P, ?Y)

  list_said:
     believes(?P, said(?Q, comma(?X, ?Y)))
    ---------------------------------------
     believes(?P, said(?Q, ?X))

  list_says:
     believes(?P, says(?Q, comma(?X, ?Y)))
    ---------------------------------------
     believes(?P, says(?Q, ?X))

  nonce_verification:
```

```
    believes(?P, fresh(?X))
    believes(?P, said(?Q, ?X))
   ----------------------------
    believes(?P, says(?Q, ?X))

jurisdiction:
    believes(?P, controls(?Q, ?X))
    believes(?P, says(?Q, ?X))
   --------------------------------
    believes(?P, ?X)

seeing_shared:
    sees(?P, shared_key(?K, ?Q, ?R))
    sees(?P, encrypt(?X, ?K))
   ----------------------------------
    sees(?P, ?X)

auth_shared:
    believes(?P, shared_key(?K, ?Q, ?P))
    sees(?P, encrypt(?X, ?K))
   -----------------------------------------
    believes(?P, said(?Q, ?X))
    believes(?P, said(?Q, ?K))
    believes(?P, said(?Q, encrypt(?X, ?K)))

key_shared:
    sees(?P, encrypt(?X, ?K))
    believes(?P, shared_key(?K, ?P, ?Q))
    believes(?P, says(?Q, ?X))
   ------------------------------------------------
    believes(?P, says(?Q, shared_key(?K, ?P, ?Q)))

contents_shared:
    believes(?P, says(?Q, encrypt(?X, ?K)))
    believes(?P, shared_key(?K, ?P, ?Q))
   -----------------------------------------
    believes(?P, says(?Q, ?X))

auth_mac:
    believes(?P, shared_key(?K, ?Q, ?P))
    sees(?P, mac(?K, ?X))
    sees(?P, ?X)
   --------------------------------------
    believes(?P, said(?Q, ?X))
    believes(?P, said(?Q, ?K))
```

```
   believes(?P, said(?Q, mac(?K, ?X)))

key_mac:
   sees(?P, mac(?K, ?X))
   believes(?P, shared_key(?K, ?P, ?Q))
   believes(?P, says(?Q, ?X))
  ------------------------------------------------
   believes(?P, says(?Q, shared_key(?K, ?P, ?Q)))

contents_mac:
   believes(?P, says(?Q, mac(?K, ?X)))
   believes(?P, shared_key(?K, ?P, ?Q))
  --------------------------------------
   believes(?P, says(?Q, ?X))

seeing_secret:
   sees(?P, combine(?X, ?Y))
  ---------------------------
   sees(?P, ?X)

auth_secret:
   believes(?P, secret(?Y, ?Q, ?P))
   sees(?P, combine(?X, ?Y))
  -----------------------------------------
   believes(?P, said(?Q, ?X))
   believes(?P, said(?Q, ?Y))
   believes(?P, said(?Q, combine(?X, ?Y)))

key_secret:
   sees(?P, combine(?X, ?Y))
   believes(?P, secret(?Y, ?P, ?Q))
   believes(?P, says(?Q, ?X))
  --------------------------------------------
   believes(?P, says(?Q, secret(?Y, ?P, ?Q)))

contents_secret:
   believes(?P, says(?Q, combine(?X, ?Y)))
   believes(?P, secret(?Y, ?P, ?Q))
  -----------------------------------------
   believes(?P, says(?Q, ?X))

seeing_public:
   sees(?P, public_key(?K, ?P))
   sees(?P, encrypt(?X, ?K))
  ------------------------------
```

```
    sees(?P, ?X)

  seeing_sig:
    sees(?P, public_key(?K1, ?Q))
    sees(?P, encrypt(?X, ?K2))
    inv(?K1, ?K2)
   -------------------------------
    sees(?P, ?X)

  auth_sig:
    sees(?P, encrypt(?X, ?K2))
    believes(?P, public_key(?K1, ?Q))
    inv(?K1, ?K2)
   ------------------------------------------
    believes(?P, said(?Q, ?X))
    believes(?P, said(?Q, ?K2))
    believes(?P, said(?Q, encrypt(?X, ?K2)))

  key_sig:
    sees(?P, encrypt(?X, ?K2))
    believes(?P, public_key(?K1, ?Q))
    believes(?P, says(?Q, ?X))
    inv(?K1, ?K2)
   ---------------------------------------------
    believes(?P, says(?Q, public_key(?K1, ?Q)))

  contents_sig:
    believes(?P, says(?Q, encrypt(?X, ?K2)))
    believes(?P, public_key(?K1, ?Q))
    inv(?K1, ?K2)
   ------------------------------------------
    believes(?P, says(?Q, ?X))

  contents_hash:
    believes(?P, said(?Q, hash(?X)))
    sees(?P, ?X)
   ----------------------------------
    believes(?P, said(?Q, ?X))

  maysee_shared_key:
    sees(?P, shared_key(?Q, ?R))
   ------------------------------
    believes(?P, maysee(?Q, ?K))

  maysee_secret:
```

```
      sees(?P, secret(?Y, ?Q, ?R))
     ------------------------------
      believes(?P, maysee(?Q, ?Y))

  maysee_privkey:
      sees(?P, public_key(?K1, ?Q))
      inv(?K1, ?K2)
     -------------------------------
      believes(?P, maysee(?Q, ?K2))

  maysee_seeing_is_believing:
      sees(?P, maysee(?Q, ?X))
     ------------------------------
      believes(?P, maysee(?Q, ?X))

  maysee_sees_maysee:
      sees(?P, sees(?Q, ?X))
     ------------------------------
      believes(?P, maysee(?Q, ?X))

  maysee_comma:
      believes(?P, maysee(?Q, comma(?X, ?Y)))
     -----------------------------------------
      believes(?P, maysee(?Q, ?X))

  maysee_pubkey:
      sees(?P, public_key(?K, ?Q))
     ------------------------------
      believes(?P, maysee(?I, ?K))

G-RULES
  freshness_list:
      believes(?P, fresh(?X))
     ------------------------------------
      believes(?P, fresh(comma(?X, ?Y)))

  freshness_seq_1:
      believes(?P, fresh(?X))
     ----------------------------------
      believes(?P, fresh(seq(?X, ?Y)))

  freshness_seq_2:
      believes(?P, fresh(?X))
     ----------------------------------
      believes(?P, fresh(seq(?Y, ?X)))
```

```
freshness_tagged:
   believes(?P, fresh(?X))
  -------------------------------------
   believes(?P, fresh(tagged(?T, ?X)))

freshness_shared_1:
   believes(?P, fresh(?X))
   sees(?P, shared_key(?K, ?P, ?Q))
  --------------------------------------
   believes(?P, fresh(encrypt(?X, ?K)))

freshness_shared_2:
   believes(?P, fresh(shared_key(?K, ?P, ?Q)))
   sees(?P, shared_key(?K, ?P, ?Q))
  ---------------------------------------------
   believes(?P, fresh(encrypt(?X, ?K)))

freshness_mac_1:
   believes(?P, fresh(?X))
   sees(?P, shared_key(?K, ?P, ?Q))
  ----------------------------------
   believes(?P, fresh(mac(?K, ?X)))

freshness_mac_2:
   believes(?P, fresh(shared_key(?K, ?P, ?Q)))
   sees(?P, shared_key(?K, ?P, ?Q))
  ---------------------------------------------
   believes(?P, fresh(mac(?K, ?X)))

freshness_secret_1:
   believes(?P, fresh(?X))
  --------------------------------------
   believes(?P, fresh(combine(?X, ?Y)))

freshness_secret_2:
   believes(?P, fresh(secret(?Y, ?P, ?Q)))
  -----------------------------------------
   believes(?P, fresh(combine(?X, ?Y)))

freshness_public_1:
   believes(?P, fresh(?X))
   sees(?P, public_key(?K, ?Q))
  --------------------------------------
   believes(?P, fresh(encrypt(?X, ?K)))
```

```
freshness_public_2:
   believes(?P, fresh(public_key(?K, ?Q)))
   sees(?P, public_key(?K, ?Q))
  -----------------------------------------
   believes(?P, fresh(encrypt(?X, ?K)))

freshness_sig_1:
   believes(?P, fresh(?X))
   sees(?P, public_key(?K1, ?Q))
   inv(?K1, ?K2)
  ---------------------------------------
   believes(?P, fresh(encrypt(?X, ?K2)))

freshness_sig_2:
   believes(?P, fresh(public_key(?K1, ?Q)))
   sees(?P, public_key(?K1, ?Q))
   inv(?K1, ?K2)
  ------------------------------------------
   believes(?P, fresh(encrypt(?X, ?K2)))

freshness_hash:
   believes(?P, fresh(?X))
  -------------------------------
   believes(?P, fresh(hash(?X)))

introspection_seeing:
   sees(?P, ?X)
  ----------------------------
   believes(?P, sees(?P, ?X))

maysee_encrypt_shared:
   believes(?P, maysee(?Q, ?X))
   believes(?P, maysee(?R, ?X))
   believes(?P, shared_key(?K, ?Q, ?R))
  -------------------------------------------
   believes(?P, maysee(?I, encrypt(?X, ?K)))

maysee_encrypt_public:
   believes(?P, maysee(?Q, ?X))
   believes(?P, public_key(?K, ?Q))
  -------------------------------------------
   believes(?P, maysee(?I, encrypt(?X, ?K)))

maysee_encrypt:
```

```
   believes(?P, maysee(?Q, ?X))
  -------------------------------------------
   believes(?P, maysee(?Q, encrypt(?X, ?K)))

maysee_concat:
   believes(?P, maysee(?Q, ?X))
   believes(?P, maysee(?Q, ?Y))
  ---------------------------------------
   believes(?P, maysee(?Q, seq(?X, ?Y)))

has_sees:
   sees(?P, ?X)
  --------------
   has(?P, ?X)

has_seq:
   has(?P, ?X)
   has(?P, ?Y)
  ----------------------
   has(?P, seq(?X, ?Y))

has_tagged:
   has(?P, ?X)
  -------------------------
   has(?P, tagged(?Y, ?X))

has_encrypt:
   has(?P, ?X)
   has(?P, ?K)
  --------------------------
   has(?P, encrypt(?X, ?K))

has_pubkey:
   believes(?P, public_key(?K, ?Q))
  ----------------------------------
   has(?P, ?K)

has_privkey:
   believes(?P, public_key(?K1, ?P))
   inv(?K1, ?K2)
  -----------------------------------
   has(?P, ?K2)

legit_seq:
   believes(?Q, legit(?X))
```

```
     believes(?Q, legit(?Y))
    -----------------------------------
     believes(?Q, legit(seq(?X, ?Y)))

  legit_encrypt:
     believes(?Q, legit(?X))
     believes(?Q, public_key(?K, ?P))
    ---------------------------------------
     believes(?Q, legit(encrypt(?X, ?K)))

END;

// ----------------------------------------------
PROTOCOL Needham_Schroeder_Pub_1;  // Logic: RV

VARIABLES
   A, B, S: Principal;
   Ka, Kb, Ks, Ks': PKey;
   Na, Nb, msg6_tag, msg7_tag: Field;


ASSUMPTIONS
   believes(A, public_key(Ka, A));
   believes(A, public_key(Ks, S));
   believes(B, public_key(Kb, B));
   believes(B, public_key(Ks, S));
   believes(S, public_key(Ka, A));
   believes(S, public_key(Kb, B));
   believes(S, public_key(Ks, S));

   sees(A, public_key(Ka, A));
   sees(A, public_key(Ks, S));
   sees(B, public_key(Kb, B));
   sees(B, public_key(Ks, S));
   sees(S, public_key(Ka, A));
   sees(S, public_key(Kb, B));
   sees(S, public_key(Ks, S));

   believes(A, controls(S, public_key(?K, B)));
   believes(B, controls(S, public_key(?K, A)));

   believes(A, fresh(Na));
   believes(B, fresh(Nb));

   believes(A, secret(Na, A, B));
   believes(B, secret(Nb, A, B));
```

```
// As in the BAN analysis, these two assumptions represent the
// protocol weakness that each principal must assume that the
// message containing the public key of the other principal is
// fresh.
believes(A, fresh(public_key(Kb, B)));
believes(B, fresh(public_key(Ka, A)));

inv(Ks, Ks');

has(P, P);
has(A, B);
has(A, Na);
has(B, Nb);

// === Interpretations ===

// messages 2 & 5 (public key certs from the server)
interp( // conclusion
        believes(?Q, says(?P, public_key(?K, ?R))),
        // premises
        believes(?Q, says(?P, tagged(conc_tag, seq(?K, ?R))))
        );

// message 3 needs no interpretation

// two-step interpretation of message 6

//   NOTE: w/o msg6_tag, this interpretation could also
//         be applied to message 3; that represents an
//         (arguable) weakness in the protocol.  If A
//         initiates the protocol with A, then an intruder
//         can pose as (the other) A by replaying message
//         3 as message 6.  Then A believes that "A" is a
//         nonce that (the other) A believes to be secret,
//         when in fact the other A (rightfully) does not
//         believe this.  Not a very practical attack.

interp( // conclusion
        sees(?Q, combine(tagged(msg6_tag, ?N2), ?N1)),
        // premises
        sees(?Q, seq(?N1, ?N2))
        );

interp( // conclusion
```

```
          believes(?Q, says(?P, secret(?N2, ?P, ?Q))),
          // premises
          believes(?Q, says(?P, tagged(msg6_tag, ?N2)))
          );

   // two-step interpretation of message 7

   interp( // conclusion
          sees(?Q, combine(tagged(msg7_tag, ?N1), ?N1)),
          // premises
          sees(?Q, ?N1)
          );

   // Note: conclusion includes a protocol-instance
   //       variable (Na)
   interp( // conclusion
          believes(?Q, says(?P,
      comma(secret(Na, ?P, ?Q),
                   says(?Q, secret(?N2, ?P, ?Q))))),
          // premises
          believes(?Q, says(?P, tagged(msg7_tag, ?N2)))
          );

MESSAGES
   // concrete messages
   1. A -> S: seq(A, B);
   2. S -> A: encrypt(seq(Kb, B), Ks');
   3. A -> B: encrypt(seq(Na, A), Kb);
   4. B -> S: seq(B, A);
   5. S -> B: encrypt(seq(Ka, A), Ks');
   6. B -> A: encrypt(seq(Na, Nb), Ka);
   7. A -> B: encrypt(Nb, Kb);

GOALS
   // these two do *not* hold unless the pk certs are
   // initially believed fresh
   believes(A, public_key(Kb, B));
   believes(B, public_key(Ka, A));

   believes(A, says(B, secret(Nb, A, B)));
   believes(B, says(A, secret(Na, A, B)));

   believes(B, says(A, says(B, secret(Nb, A, B))));

END;
```

```
Final theory representation (size 155): [omitted]

desired property: believes(A, public_key(Kb, B))
   is TRUE
desired property: believes(B, public_key(Ka, A))
   is TRUE
desired property: believes(A, says(B, secret(Nb, A, B)))
   is TRUE
desired property: believes(B, says(A, secret(Na, A, B)))
   is TRUE
desired property: believes(B, says(A, says(B, secret(Nb, A, B))))
   is TRUE

Interpretation rules: INVALID [I3 violated by rule 5]
Honesty check: PASS
Secrecy check: PASS
Feasibility check: PASS
```

Bibliography

[AN96] Martín Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, January 1996.

[ASN94] Abstract Syntax Notation One (ASN.1): Specification of basic notation. ITU-T Recommendation X.680, 1994.

[AT91] Martín Abadi and Mark R. Tuttle. A semantics for a logic of authentication (extended abstract). In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, pages 201–216, August 1991.

[BAN89] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. Technical Report SRC-39, DEC SRC, 1989.

[BAN90] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

[Bar77] Jon Barwise. *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, chapter A.1. North-Holland, Amsterdam, 1977.

[BCM$^+$90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, , and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. Fifth Annual IEEE Symposium on Logic in Computer Science (LICS)*, 1990.

[Ben86] Jon Bentley. Little languages. *Communications of the ACM*, 29(8):711–721, 1986.

[BL76] 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.

[BM79] R. S. Boyer and J. S. Moore. *A Computational Logic*. ACM monograph series. Academic Press, New York, 1979.

[BP97] G. Bella and L. C. Paulson. Using Isabelle to prove properties of the Kerberos authentication system. In *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols*, September 1997.

[Bra96] Stephen H. Brackin. A HOL extension of GNY for automatically analyzing cryptographic protocols. In *Proceedings of the Ninth IEEE Computer Security Foundations Workshop*, pages 62–75, June 1996.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986. Reprinted in M. Yoeli, Formal Verification of Hardware Design, IEEE Computer Society Press, 1990, pp. 253–267.

[CFN88] D. Chaum, A. Fiat, and M. Naor. Untraceable electronic cash. In *Proceedings of Advances in Cryptology—CRYPTO '88*, pages 319–327, 1988.

[CJM98] Edmund Clarke, Somesh Jha, and Will Marrero. Using state space exploration and a natural deduction style message derivation engine to verify security protocols. In *Proc. IFIP Working Conference on Programming Concepts and Methods (PROCOMET)*, 1998.

[CL73] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York, 1973.

[CM81] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, 1981.

[DDHY92] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[DG79] Burton Dreben and Warren D. Goldfarb. *The Decision Problem: Solvable Classes of Quantificational Formulas.* Addison-Wesley, Reading, Mass., 1979.

[DS81] Dorothy E. Denning and Giovanni Maria Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, August 1981.

[DY81] D. Dolev and A. C. Yao. On the security of public key protocols (extended abstract). In *22nd Annual Symposium on Foundations of Computer Science*, pages 350–357, October 1981.

[FKK96] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL protocol, version 3.0. IETF Internet Draft `draft-ietf-tls-ssl-version3-00.txt`, available at `http://home.netscape.com/products/security/ssl`, November 1996.

[For82] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[GM93] M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, Cambridge, UK, 1993.

[GMT+80] Susan L. Gerhart, David R. Musser, D. H. Thompson, D. A. Baker, R. L. Bates, R. W. Erickson, R. L. London, D. G. Taylor, and D. S. Wile. An overview of AFFIRM: A specification and verification system. In S. H. Lavington, editor, *Proceedings of IFIP Congress 80*, pages 343–347, Tokyo, Japan, October 1980. North-Holland.

[GNY90] L. Gong, R. Needham, and R. Yahalom. Reasoning about belief in cryptographic protocols. In *Proc. IEEE Symposium on Security and Privacy*, pages 234–248, May 1990.

[Gro99] IETF Secure Shell (secsh) Working Group. Secure shell (secsh) charter. Available at `http://www.ietf.org/html.charters/secsh-charter.html`, March 1999.

[HTWW96] Nevin Heintze, Doug Tygar, Jeannette Wing, and Hao-Chi Wong. Model checking electronic commerce protocols. In *Proceedings of* *the Second USENIX Workshop on Electronic Commerce*, pages 147–164, 1996.

[IM95] J. W. Gray III and J. McLean. Using temporal logic to specify and verify cryptographic protocols. In *Proceedings of the Eighth IEEE Computer Security Foundations Workshop*, pages 108–116, June 1995.

[Kai96] Rajashekar Kailar. Accountability in electronic commerce protocols. *IEEE Transactions on Software Engineering*, 22(5):313–328, May 1996.

[KMM94] R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7(2):79–130, 1994.

[KW94] Volker Kessler and Gabriele Wedel. AUTLOG—an advanced logic of authentication. In *Proceedings of the Computer Security Foundations Workshop VII*, pages 90–99. IEEE Comput. Soc., June 1994.

[Low95] G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, November 1995.

[Low96] 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.

[LR97] Gavin Lowe and Bill Roscoe. Using CSP to detect errors in the TMN protocol. *IEEE Transactions on Software Engineering*, 23(10):659–669, 1997.

[LRS79] K. N. Levitt, L. Robinson, and B. A. Silverberg. The HDM handbook, vols. 1–3. Technical report, SRI International, Menlo Park, California, 1979.

[LS97] Naomi Lindenstrauss and Yehoshua Sagiv. Automatic termination analysis of logic programs. In *Proc. 14th International Conference on Logic Programming*, 1997.

[LSSE80] 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.

[Mao95] Wenbo Mao. An augmentation of BAN-like logics. In *Proceedings of the Eighth IEEE Computer Security Foundations Workshop*, pages 44–56, June 1995.

[MB94] Wenbo Mao and Colin Boyd. Development of authentication protocols: Some misconceptions and a new approach. In *Proceedings of the Seventh IEEE Computer Security Foundations Workshop*, 1994.

[MCJ97] W. Marrero, E. M. Clarke, and S. Jha. Model checking for security protocols. Technical Report CMU-CS-97-139, Carnegie Mellon University, May 1997.

[McM92] K. L. McMillan. *Symbolic model checking—an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, 1992.

[Mea94] Catherine Meadows. A model of computation for the NRL protocol analyzer. In *Proceedings of the Seventh IEEE Computer Security Foundations Workshop*, pages 84–89, June 1994.

[Mea98] Catherine Meadows. Using the NRL protocol analyzer to examine protocol suites. In *Proc. Workshop on Formal Methods and Security Protocols*, Indianapolis, June 1998.

[Mil84] Jonathan K. Millen. The Interrogator: a tool for cryptographic protocol security. In *Proc. IEEE Symposium on Security and Privacy*, April 1984.

[Mil97] Jonathan K. Millen. CAPSL: Common authentication protocol specification language. available at `http://www.jcompsec.mews.org/capsl/`, July 1997.

[Mil99] Jonathan K. Millen. A necessarily parallel attack. draft paper, available at `http://www.csl.sri.com/~millen/`, February 1999.

[MMS97] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murϕ. In *Proc. IEEE Symposium on Security and Privacy*, pages 141–151, May 1997.

[MNSS87] 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.

[Mos89] Louise E. Moser. A logic of knowledge and belief for reasoning about computer security. In *Proceedings of the Second IEEE Computer Security Foundations Workshop*, pages 84–89, 1989.

[MSS98] J. C. Mitchell, V. Shmatikov, and U. Stern. Finite-state analysis of SSL 3.0. In *Proc. of the Seventh USENIX Security Symposium*, pages 257, 201–215, January 1998.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[Nes90] D. Nessett. A critique of the Burrows, Abadi, and Needham logic. *ACM Operating Systems Review*, 24(2):35–38, April 1990.

[NN93] P. Nivela and R. Nieuwenhuis. Saturation of first-order (constrained) clauses with the Saturate system. In *Proceedings of the Fifth International Conference on Rewriting Techniques and Applications*, pages 436–440, June 1993.

[NS78] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[NS93] B. Neuman and S. Stubblebine. A note on the use of timestamps as nonces. *ACM Operating Systems Review*, 27:10–14, April 1993.

[OR87] D. Otway and O. Rees. Efficient and timely mutual authentication. *Operating Systems Review*, 21(1):8–10, 1987.

[ORSvH95] 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.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[Pau96] Lawrence C. Paulson. Proving properties of security protocols by induction. Technical report, University of Cambridge, December 1996.

[Plo72] G. D. Plotkin. Building-in equational theories. *Machine Intelligence*, 7:73–90, 1972.

[Ros94] 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.

[Ros95] A. W. Roscoe. CSP and determinism in security modelling. In *Proc. IEEE Symposium on Security and Privacy*, pages 114–127, 1995.

[RS96] E. Rescorla and A. Schiffman. The Secure HyperText Transfer Protocol. IETF Internet Draft `draft-ietf-wts-shttp-03.txt`, available at `http://www.eit.com/projects/s-http`, July 1996.

[ST95] Marvin Sirbu and J. D. Tygar. Netbill: An internet commerce system optimized for network delivered services. In *Digest of Papers for COMPCON'95: Technologies for the Information Superhighway*, pages 20–25, March 1995.

[SvO94] P. F. Syverson and P. C. van Oorschot. On unifying some cryptographic protocol logics. In *Proceedings of the 1994 IEEE Symp. on Security and Privacy*, pages 14–28, 1994.

[Syv91] Paul Syverson. The use of logic in the analysis of cryptographic protocols. In *Proc. IEEE Symposium on Security and Privacy*, pages 156–170, May 1991.

[VCP+95] Manuela Veloso, Jaime Carbonell, Alicia Perez, Daniel Borrajo, Eugene Fink, and Jim Blythe. Integrating planning and learning: The PRODIGY architecture. *Journal of Theoretical and Experimental Artificial Intelligence*, 7(1), 1995.

[VM96] Visa and MasterCard. Secure Electronic Transaction (SET) specification. Available at `http://www.visa.com/cgi-bin/vee/nt/ecomm/set/setprot.html`, June 1996.

[WL92a] Thomas Y. C. Woo and Simon S. Lam. Authentication for distributed systems. *Computer*, 25(1):39–52, January 1992.

[WL92b] Thomas Y. C. Woo and Simon S. Lam. ‘Authentication’ revisited. *Computer*, 25(3):10, March 1992.

[WL93] Thomas Y. C. Woo and Simon S. Lam. A semantic model for authentication protocols. In *Proc. IEEE Symposium on Security and Privacy*, pages 178–194, May 1993.