AFRL-IF-RS-TR-2003-63
Final Technical Report
March 2003

SURVIVABLE LOOSELY COUPLED ARCHITECTURES

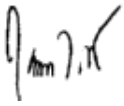
SRI International

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2003-63 has been reviewed and is approved for publication.

APPROVED: 

JAMES L. SIDORAN
Project Engineer

FOR THE DIRECTOR: 

WARREN H. DEBANY, Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	MARCH 2003	Final Aug 96 – Dec 99

4. TITLE AND SUBTITLE
SURVIVABLE LOOSELY COUPLED ARCHITECTURES

5. FUNDING NUMBERS
C - F30602-96-C-0291
PE - 62301E
PR - D985
TA - 02
WU - 01

6. AUTHOR(S)
John Rushby, Dawn Xiaodong Song, Jonathan K. Millen, Harald Rueb, and Veronique Cortier

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
SRI International
333 Ravenswood Avenue
Menlo Park California 94022

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory/IFGB
525 Brooks Road
Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2003-63

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: James L. Sidoran/IFGB/(315) 330-3174/ James.Sidoran@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

The objective of this research was to develop mechanisms and methods of analysis to support construction of survivable systems where survivable means systems able to withstand multiple kinds of faults among their components, including those induced deliberately by an active attacker. One class of architectures for survivability builds on classical methods for fault tolerance, in which replication and voting are used to mask faults. An alternative class of methods requires less tight coordination, giving rise to loosely coupled architectures. Mechanisms that support survivability in loosely coupled architectures are typically based on cryptography, and much of the work performed in this project focused on development of suitable cryptographic protocols and on their formal verification. In the course of the project, the state of the art was advanced from one where formal verification of these protocols was a tour de force to one where it may be considered routine and available for general deployment. The outputs of this research are documented in a series of technical papers (with associated abstracts) that follow.

14. SUBJECT TERMS
Survivable Systems, Loosely Coupled Architectures, Fault Tolerant Methods, Cryptographic Protocols

15. NUMBER OF PAGES
131

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

Part I: Introduction

This report covers the period August 28, 1996 through December 31, 1999, and documents work performed by SRI International for Rome Laboratory Contract F30602-96-C-0291, Arpa Order E301.

The objective of this research was to develop mechanisms and methods of analysis to support construction of survivable systems. By survivable systems, we mean those able to withstand multiple kinds of faults among their components, including those induced deliberately by an active attacker. One class of architectures for survivability builds on classical methods for fault tolerance, in which replication and voting are used to mask faults. These methods, however, require tight coordination among the replicas and may fail to make progress if certain connectivity requirements are not satisfied (e.g., if there is no majority clique in a partitioned network). An alternative class of methods requires less tight coordination, giving rise to *loosely coupled* architectures.

Mechanisms that support survivability in loosely coupled architectures are typically based on cryptography, and much of the work performed in this project focused on development of suitable cryptographic protocols and on their formal verification. In the course of the project we advanced the state of the art from one where formal verification of these protocols was a *tour de force* to one where it may be considered routine and available for general deployment. The outputs of this research are documented in a series of technical papers that are collected in Part II of this report. Below, we provide an index and abstracts for these papers. Several of them were selected for presentation at major scientific conferences, and we also provide citations for these publications.

Secure Auctions in a Publish/Subscribe System (page 6).

The project began with the design and verification of a protocol for fault-tolerant and secure service for sealed-bid auctions in a loosely coupled system.

Abstract We present an approach to provide a fault-tolerant and secure service for sealed-bid auctions. The solution is designed for a loosely coupled publish-subscribe system. It employs multiple auction servers and achieves validity and security properties through application of secret-sharing methods and public-key encryption and signatures. It can tolerate Byzantine failures of one third of the auction servers and any number of bidders. A verification of the desired properties has been machine checked using PVS. This work also provides insight and useful experience in techniques for specifying and verifying this type of system.

A Necessarily Concurrent Attack (page 2 4). Published as [1].

The project next established that certain classes of attacks on protocols can be mounted in a concurrent system but not in a sequential system. Thus, the intruder is strictly more powerful in a concurrent system, and the burden of verification correspondingly greater.

Abstract An artificial protocol called the "ffgg" protocol is constructed, with an assumed security objective to keep a certain data item secret. A message modification attack is given that exposes the data item; in this attack there are two concurrently running responder processes belonging to the same agent. To show that a concurrent attack is necessary, we use an inductive approach to prove that the protocol is secure under the assumption that this kind of concurrency is excluded.

Protocol-Independent Secrecy (page 51). Published as [2].

Formal verifications of cryptographic protocols have previously been monolithic. This paper introduces a decomposition method for dividing the verification into two components, thereby allowing reuse and reducing the overall effort required.

Abstract Inductive proofs of secrecy invariants for cryptographic protocols can be facilitated by separating the protocol-dependent part from the protocol-independent part. Our Secrecy theorem encapsulates the use of induction so that the discharge of protocol-specific proof obligations is reduced to first-order reasoning. Secrecy proofs for Otway-Rees and the corrected Needham-Schroeder protocol are given.

Local Secrecy for State-Based Models (page 6 8). Published as [3].

The next paper illustrates the verification techniques introduced by the previous paper, using extracts from actual verifications performed using SRI's PVS verification system.

Abstract Proofs of secrecy invariants for cryptographic protocols can be facilitated by separating the protocol-dependent part from the protocol-independent part. Our Secrecy theorem encapsulates the use of induction so that the discharge of protocol-specific proof obligations is reduced to first-order reasoning. The theorem has been proved and applied in the PVS environment with supporting protocol representation theories based on a state-transition model. This technique has been successfully applied to both standard benchmark examples and to parts of the verification of the Enclave group management system.

Proving secrecy is easy enough (page 80). Published as [4].

The decomposition method developed in the previous two papers facilitates systematic development of secrecy proofs. The next paper presents the culmination of this element of the research: a completely systematic method that allows easy verification of challenging cryptographic protocols.

Abstract We develop a systematic proof procedure for establishing secrecy results for cryptographic protocols. Part of the procedure is to reduce messages to simplified constituents, and its core is a search procedure for establishing secrecy results. This procedure is sound but incomplete in that it may fail to establish secrecy for some secure protocols. However, it is amenable to mechanization, and it also has a convenient visual representation. We demonstrate the utility of our procedure with secrecy proofs for standard benchmarks such as the Yahalom protocol.

An Overview of Formal Verification for the Time-Triggered Architecture (page 104). Published as [5].

In parallel to formal verification of cryptographic protocols, we also performed research on formal verification of algorithms for the Time-Triggered Architecture (TTA), which is being adopted for critical control applications in both civil and military domains (for example, it is used in a new engine controller for the F16). Although not loosely coupled, we considered that the very well defined verification challenges presented by TTA would provide an excellent driver for development of new techniques. This proved to be the case, as the diagrammatic formal verification method developed for the TTA membership algorithm was subsequently applied successfully (in another DARPA project) to the loosely coupled Enclaves architecture [6].

Abstract We describe formal verification of some of the key algorithms in the Time-Triggered Architecture (TTA) for real-time safety-critical control applications. Some of these algorithms pose formidable challenges to current techniques and have been formally verified only in simplified form or under restricted fault assumptions. We describe what has been done and what remains to be done and indicate some directions that seem promising for the remaining cases and for increasing the automation that can be applied. We also describe the larger challenges posed by formal verification of the interaction of the constituent algorithms and of their emergent properties.

Bibliography

[1] Jonathan K. Millen. A necessarily parallel attack. In Nevin Heintze and Edmund Clarke, editors, *Workshop on Formal Methods and Security Protocols (Part of the Federated Logic Conference, FLoC)*, Trento, Italy, July 1999.

[2] Jon Millen and Harald Rueß. Protocol-independent secrecy. In Michael Reiter and Roger Needham, editors, *Proceedings of the Symposium on Security and Privacy*, pages 110–119, Oakland, CA, May 2000. IEEE Computer Society.

[3] Harald Rueß and Jonathan Millen. Local secrecy for state-based models. In *Workshop on Formal Methods and Computer Security (held in association with the Conference on Computer Aided Verification, CAV)*, Chicago, IL, July 2000.

[4] Véronique Cortier, Jon Millen, and Harald Rueß. Proving secrecy is easy enough. In *14th Computer Security Foundations Workshop*, pages 97–108, Cape Breton, Novia Scotia, Canada, June 2001. IEEE Computer Society.

[5] John Rushby. An overview of formal verification for the time-triggered architecture. In Werner Damm and Ernst-Rüdiger Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 2469 of *Lecture Notes in Computer Science*, pages 83–105, Oldenburg, Germany, November 2002. Springer-Verlag.

[6] B. Dutertre, H. Saïdi, and V. Stavridou. Intrusion-tolerant group management in Enclaves. In *The International Conference on Dependable Systems and Networks*, pages 203–212, Goteborg, Sweden, July 2001. IEEE Computer Society.

Part II

Technical Papers

Secure Auctions in a Publish/Subscribe System *

Dawn Xiaodong Song
Carnegie Mellon University
skyxd@cs.cmu.edu

Jonathan K. Millen
SRI International
millen@csl.sri.com

Abstract

We present an approach to provide a fault-tolerant and secure service for sealed-bid auctions. The solution is designed for a loosely coupled publish/subscribe system. It employs multiple auction servers and achieves validity and security properties through application of secret-sharing methods and public-key encryption and signatures. It can tolerate Byzantine failures of one third of the auction servers and any number of bidders. A verification of the desired properties has been machine-checked using PVS. This work also provides insight and useful experience in techniques for specifying and verifying this type of system.

1 Introduction

The transition from traditional financial procedures to novel electronic and digital procedures is taking place worldwide at a surprisingly high speed. Electronic commerce systems, such as electronic trading, electronic banking, and electronic exchanges are becoming critical systems for society. As is the case with the traditional forms of critical systems, electronic commerce systems often require safety and reliability guarantees. They must be scalable and adaptable. They also require security properties such as secrecy, anonymity, and non-repudiation.

It's also commonly agreed that formal specification and verification are needed to provide solutions of this kind [15] [8] [10]. Many hand-checked protocols are found to be flawed via formal methods after they are proposed [5] [9] [6]. But there is still a lack of instructive experience and a systematic way of combining system building blocks and formal specification and verification techniques to provide a real solution.

Motivated by these problems, we studied one of these electronic commerce systems, sealed-bid secure auction service. A sealed-bid auction is one in which secret bids are issued for a certain item, and when the bidding is closed, the bids will be opened and the winner will be chosen according to certain publicly known rules. Sealed-bid auctions are

*This work was supported by the U.S. Government under contract no. F30602-96-C-0291

used in auctioning of various contracts, and in the sale of different types of goods, such as artwork and real estate [4] [14].

Besides efficiency and scalability, sealed-bid auctions have strong security requirements. The identity of the bidders and the contents of the bids should not be revealed until the bidding is closed. After the bidding is closed, no more bids should be accepted as valid bids. The auction service should be able to tolerate a certain degree of corruption of the insiders in the auction house and the maliciousness of some bidders. In an internet environment, it is necessary to provide the required functional and security properties in the face of unreliable network communication and random failures of important components such as auction servers.

Franklin and Reiter have given a solution in the context of monetary bids [4]. Their solution is focused on using a cryptographic technique to provide protections to monetary bids, such as digital cash bids. It inherits certain properties from the digital cash scheme used for the bids. In their solution, every bidding message and auction server synchronization message requires atomic multicast [13] primitives, which can be a bottleneck in a large system.

In this paper we present a new approach which is built on a loosely coupled architecture and does not require atomic multicast. Loosely coupled publish/subscribe architectures have been widely used for scalable, adaptable distributed systems [11]. Their flexibility makes them a desirable infrastructure for many applications, but they generally lack fault tolerance and security support in malicious environments. Our challenge is to integrate fault tolerance and security in a loosely coupled publish/subscribe architecture in a systematic way and use formal specification and verification to increase the assurance of the design correctness [17].

Our solution is based on the the direct application of secret sharing and public key encryption. It can tolerate Byzantine failures of one third of the auction servers and any number of bidders. It provides a bid receipt service, which is often desirable in financial activities, and can be used by the bidder to prove that a bid was entered before the bidding was closed. We use PVS for formal specification and verification of the system and the properties [12]. A resulting prototype is in process to demonstrate the efficiency and scalability of the system.

The rest of the paper is organized as follows: the desired properties of the auction are summarized in the next section. In Section 3, we present the basic building blocks of the system and the cryptographic primitives needed in the design. In Section 4, we give an informal description of the protocol in detail. In Section 5, we give an overview of the formal specification of the system and some abstraction techniques. In Section 6, we list the desired system properties as specified in PVS and explain how we used PVS to prove these properties. Some issues are discussed in Section 7.

2 Auction Properties

The auction scheme is designed for any number of bidders and auction servers (also called auctioneers). Some of the auction servers and bidders may be faulty by either intentionally or incompetently failing to follow the specification of the protocol. The failure model and other environmental assumptions are discussed in detail later.

The desired properties of the auction are as follows:

1. The bidding period starts only if at least one good auction server decides that it should.
2. A good auction server stops accepting bids only after at least one other good auction server decides that the bidding period should be closed.
3. The identity of the bidders and the content of their bids are not revealed until the bidding is closed.
4. After the bidding period is closed, no more bids are accepted as valid.
5. Bidders are provided with evidence to prove that their bids are accepted before the bidding is closed.
6. Winning bid will be determined according to certain publicly known rules.

At the end of the auction, a winning bid is selected. Guarantees regarding the authenticity, nonrepudiation, and collectability of the bids are not provided by the protocol itself, but those issues can be addressed separately through construction of the bid contents.

3 Building Blocks

The three architectural components of the system are:

- a loosely coupled publish/subscribe system,
- a set of cryptographic primitives, and
- an auction protocol.

The first two of these are summarized below. The principal contribution of this paper is the design and verification of the auction protocol, as described in subsequent sections.

3.1 System Characteristics

3.1.1 Loosely Coupled Systems

Loosely coupled systems have been developed to meet the need for large-scale survivable distributed systems [11]. The distinction between a loosely coupled system and a tightly coupled one lies in the way they handle process groups [1]. In a tightly coupled system there is a strong notion of group, sharing a common view of the group membership and the state of the system. A tightly coupled system often requires reliable multicast and atomic multicast [13]. The group membership protocol and reliable and atomic multicast primitives are complex and expensive to implement and can be a bottleneck of a system.

Loosely coupled systems, by contrast, do not need a strong notion of group membership. Instead of atomic multicast, they often use a publish/subscribe infrastructure where components acting in the role of publishers or subscribers communicate through a virtual bus (often called an "infobus"). Their great flexibility, adaptability and efficiency have made such systems suitable for very large and wide-area networks.

3.1.2 Publish/Subscribe Architecture

In a publish/subscribe system, messages have a subject and a content field. Publishers publish messages under certain subjects. Subscribers subscribe to subjects of interest and receive the messages that are published under those subjects. Publish/subscribe systems are flexible because the subjects and contents of messages are minimally constrained by the core communication architecture. Subjects may have hierarchically organized, application-defined modifiers or subtopics, and the format of the message content can be defined freely according to the needs of the applications. Publish/subscribe also provides anonymity of publishers and subscribers.

For the auction scheme, there will be an auction subject, with modifiers identifying a particular auction and indicating whether the message is intended for auction servers or bidders. Auction Servers and bidders both publish and subscribe to appropriate message subjects as defined by the protocol. For each particular auction, there is a fixed set of auction servers of known size.

The subject field and subscription mechanism cannot be depended upon to support security objectives such as authenticating authorized publishers or restricting distribution of particular types of messages. For these and other security functions, we make use of additional cryptographic services.

3.1.3 Failure Model

The failure model has two aspects: the reliability of message delivery in the network and the correctness of infobus clients, either auction servers or bidders.

The network is not assumed to be totally reliable. Messages can be delayed or lost or received out of order. However, the protocol is not designed for arbitrary network failure or indefinite denial of message delivery. It would not make sense to assume that an attacker can intercept any and all messages, since then the attacker can simply intercept all bidding messages from other bidders and only let its own bid go through.

It is assumed that published messages will be delivered to a sufficiently large portion of the network within a bounded time. That is, any routing failures or denial of service attacks, whether they are permanent or intermittent, can affect only relatively small segments of the network. By "relatively small," we refer to the proportion of auction servers that may be affected. Since nothing is said about order of delivery, this assumption does not fall precisely into previously defined categories of "unreliable" or "reliable" communication in sources such as [4] and [11].

The second aspect of the failure model is the possible dishonesty of auction servers, possibly in collusion with bidders. We adopt a Byzantine failure model in which faulty auction servers may depart from the auction server protocol, withhold messages expected from it, subscribe to all auction-related messages, and publish all kinds of auction-related messages. A bidder may also be faulty and misbehave by submitting improper bids or publishing them at improper times.

A "good" auction server is one that is not faulty *and* lies in a segment of the network where messages published to other good auction servers will be received by all good auction servers in a bounded time. In practice, it may be necessary to send messages repeatedly to ensure delivery, and this can be a normal function of the basic publish/subscribe transmission protocol. The bounded-time assumption is discussed further in the Issues section at the end.

We assume that at most a specified number t of the n auction servers are not good, and that $n \geq 3t + 1$. Any number of bidders may be faulty or isolated in parts of the network behind unreliable routers. Some bids may be lost for this reason.

3.2 Security Support

3.2.1 Public Key Infrastructure

The protocol will make use of a public-key cryptosystem that must be used by auction servers and bidders for encrypting and signing messages, as called for in the protocol. We assume that there is a certification authority that can provide public key certificates prior to the auction. Implementing a practical public-key certification infrastructure is nontrivial, but this task is separable from the conduct of the auction. In fact, there may be many services other than an auction service that would make use of common key management facilities.

One auction-service-specific function is required of the certificate authority: the certificate for an auction server's public key should indicate that its role as an auction server is authorized.

3.2.2 Secret Sharing

We also need a threshold secret sharing scheme. An (m, n)-threshold scheme permits a message to be projected onto n *shares* such that any m of them can be combined to reconstruct the original message, but less than m of them cannot. Several algorithms for this are given in Section 23.2 of [19].

4 Protocol Description

We assume that there are a set of n auction servers, denoted by S_1,...,S_n. The number n is fixed for a given auction. We assume that $n \geq 3t + 1$. For brevity, we refer to auction servers as "servers," though technically they are "clients" on the infobus. S_i has server ID s_i. There may be any number of bidders B_j, with identifiers b_j. The auction has a unique auction ID, denoted as *aid*.

All messages relating to this auction are published under an "auction" subject qualified by the auction ID. Some messages are intended solely for auction servers or bidders, and for efficiency that fact may be indicated as a subject modification as well. From an abstract or security point of view, it does not matter whether a field is part of the subject or part of the content of a message, and we assume that hostile parties can eavesdrop on all messages.

For simplicity of the representation, we introduce some shorthand denotations.

For any message a, $[a]^i$ is the encryption of a by server S_i's public key. It is assumed that any auction participant can look up and use S_i's public key given s_i.

For any message a, $[a]_i$ is a signed by server S_i's private key. We assume that a is recoverable from $[a]_i$, and that the signature can be checked by any participant given s_i.

All server messages in the protocol are signed, so that other servers will know they are authentic. This is important to determine subsequent server actions and to justify inferences about the state of good servers. Authentication of bids is not indicated in the protocol because it affects only the internal structure of bids, and it matters only for bid evaluation, which occurs after the protocol as specified has concluded.

We use a $(t+1, n)$-threshold sharing scheme, where t is the maximum tolerable number of faulty servers. $\mathsf{SSF}_i(s)$ is the ith share of a secret s.

A server's state transitions are depicted in Figure 1. A bidder's state transitions are depicted in Figure 2.

S.1 Starting the bidding

When server S_i decides that the bidding should be started, it publishes a *start* message: $\mathit{aid}, s_i, [\mathit{aid}, \mathsf{start}]_i$. When S_i has received start messages from at least $t + 1$ different other servers, it considers the bidding started and starts to accept bidding messages from bidders.

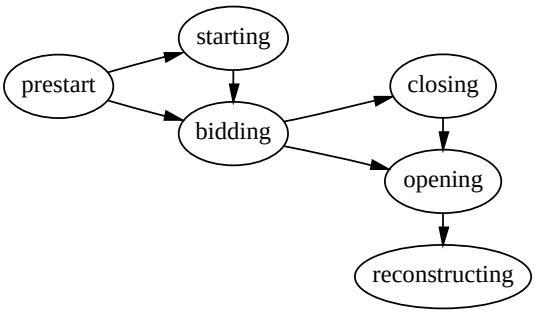


Figure 1: Server State Transitions

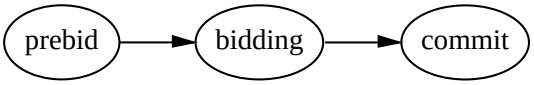


Figure 2: Bidder State Transitions

B.1 Submitting bids

Suppose a bidder B_j decides to submit a bid y_j. The format of y_j will be discussed later.

B_j breaks y_j into shares $x_{ij} = \mathsf{SSF}_i(y_j)$, for $i = 1, ..., n$. Then B_j generates the *bid* message:

$$M_j = \mathit{aid}, b_j, [x_{1j}]^1, ..., [x_{nj}]^n.$$

This message is published to all servers.

S.2 During bidding

When server S_i receives a bid M_j from a bidder B_j during the bidding, it publishes a *receipt*:

$$\mathit{aid}, b_j, s_i, [\mathsf{hash}(\mathit{aid}, M_j)]_i$$

where hash may be any standard one-way hashing function.

B.2 Committing the bids

When B_j receives a receipt from S_i, it checks the validity of the receipt by checking the signature on the hash value. After B_j receives valid receipts from at least $2t + 1$ different servers, it enters its *commit* phase. Until then, it will either wait or periodically retry submitting the bid. We assume, essentially as part of the definition of a "good" server, that all good servers will eventually receive and acknowledge a correctly formatted bid.

S.3 Closing the bidding

When S_i decides that the bidding should be closed, it publishes a signed *close* message:

$$\mathit{aid}, s_i, [\mathit{aid}, \mathsf{close}]_i.$$

When S_i has received close messages from at least $t + 1$ different other servers, it considers the bidding closed and stops accepting any more bidding messages from the bidders.

Suppose S_i received L_i bids in total. Let R_i be the set of indices of the bidders whose bids were received by S_i. Thus, R_i is of size L_i. For each $k \in R_i$, S_i decrypts its share of B_k's bid, namely x_{ik}.

It then publishes a *fingerprint* of the set of bids that it has received:

$$\mathit{aid}, s_i, [\mathsf{hash}(\mathit{aid}, \{(b_k, x_{ik}, M_k)\}_{k \in R_i})]_i$$

.

The fingerprint contains a signed hash of a list of triples, one for each received bid; each triple has the bidder ID, S_i's bid share, and the complete bid message. (Faulty servers may or may not send out a fingerprint message, but if they do, it is received by all good servers.)

S.4 Opening the bids

After a bounded time, all the good servers should have stopped receiving bids, and have published their fingerprints. Since there are at most t faulty servers, there are at least $n - t$ fingerprints published.

After a bounded additional time, each good server S_i will have received fingerprint messages from all other good servers. They republish all the fingerprint messages that they have received. The inconsistent messages will be considered as from faulty servers and will be discarded. So all good servers will have the same set of fingerprint messages. Then it publishes its *bid-set* message, containing the information that was hashed to compute the fingerprint. The bid-set message is:

$$aid, s_i, [\{(b_k, x_{ik}, M_k)\}_{k \in R_i}]_i.$$

S.5 Reconstructing the bids

After another bounded additional interval, each good server S_i will have received all bid-set messages sent by all other good servers.

When S_i receives a bid-set message from S_j, it first checks whether it matches the fingerprint from S_j by computing the hash value. If they don't match, it means S_j is faulty and that bid-set message is discarded by S_i (and all other) good servers.They republish all the bid-set messages that they have received. The inconsistent messages will be considered as from faulty servers and will be discarded. So all good servers will have the same set of bid-set messages.

S_i reconstructs the bid from B_j as follows. Let T_{ik} be the set of indices of servers S_j from whom a bid-set message with M_k has been received by S_i.

For each index $k \in T_{ik}$, S_i can extract S_j's share of B_k's bid y_k, namely x_{jk}, from the jth bid-set message, compute $[x_{jk}]^j$, and compare this with the value from the bid message M_k. If they match, the share x_{jk} is valid and can be used to reconstruct the bid y_k.

If T_{ik} contains at least $t + 1$ elements, then S_i combines those $t + 1$ shares to construct a value y'_k that should be equal to the bid y_k.

If there exists any j such that $[\mathsf{SSF}_j(y'_k)]^j \neq [x_{jk}]^j$, where $[x_{jk}]^j$ is taken from the bid message M_k, then S_i discards the bid from B_k.

In this way, S_i reconstructs a set of bids and selects a winner according to the publicly-known rule for the auction.

All the good servers will reconstruct exactly the same set of bids, because each of them received the same set of bid-set messages. The majority of the servers will agree on a selection, since good servers are in the majority, and that selection is declared the winner of the auction. Issues such as authentication and enforcement of the bids will be discussed in a later section.

5 Formal Specification of the System

The secure auction service system is a distributed system composed of asynchronous processes, namely, the auction servers and bidders. Systems and most programming language structures can be modeled as state machines [18]. A state machine consists of some encoding of the system state, and the next-state transition relationship.

Compositional reasoning and verification are often necessary and desired to simplify the complexity of a verification [3]. The state of a distributed system can be viewed as the composition of the local states of its component processes. The state transition relation, as well, can be decomposed into local state transitions per component.

Abstraction of the system structure, including communication and cryptographic primitives, is necessary for protocol level specification and verification. In this section, we describe how we use composition and abstraction techniques for the system specification.

For the secure auction service system, the global state is the composition of the local states of the components representing auction servers and bidders. Each of these components operates asynchronously according to a local state transition relation. There are two local transition relations, one for auction servers and one for bidders.

All auction servers have the same state structure, and so do all of the bidders. These structures are described in the next subsection.

The infobus is modeled using local state variables that record the sets of messages that have been published by each participant. The state of the infobus is the union of all of these locally-defined sets.

The global state structure is summarized schematically in Figure 3. The figure shows how the auction server state and the bidder state are decomposed into state variables. The infobus state also has components, each of which is derived as the union of corresponding local state components.

5.1 Abstraction of the Auction Server

An auction server is a local state machine with the state variables shown in Figure 3. The phase variable has one of the values *prestart, starting, bidding, closing, opening, reconstructing*. *wantStart* and *wantClose* are boolean variables that indicates when the auction server decides that it's time to start or close, respectively.

start_buffer, close_buffer, bids_buffer, fingerprint_buffer and *bidset_buffer* are sets of IDs identifying servers and bidders from whom messages of these kinds have been received.

openBid is the set of IDs identifying bidders whose bid shares have been opened by this auction server, i.e., those that are included in its bid-set message.

holdShare_buffer is the set of all the shares that the auction server can decrypt from its *bids_buffer*. *holdBid_buffer* is the set of all the bids that the auction server reconstructs at the end.

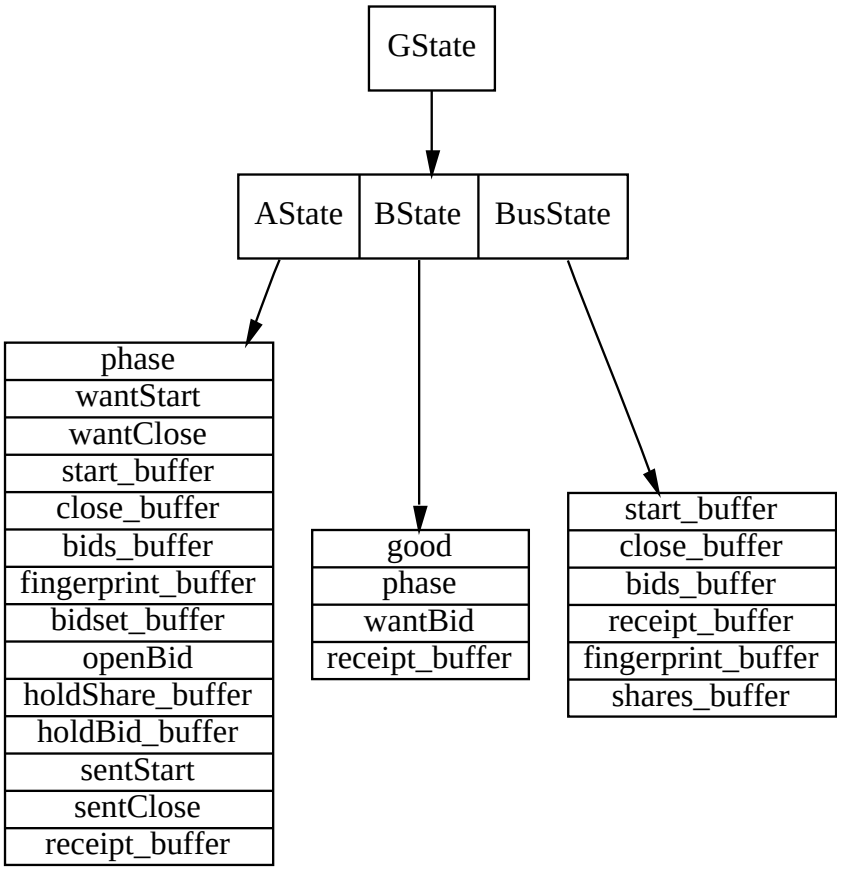


Figure 3: Global State Structure

sendStart and *sendClose* are boolean variables that indicate when the auction server has already sent out start or close messages. *receipt_buffer* is the set of all the IDs of bidders whose bid it has acknowledged by a receipt.

5.2 Abstraction of the Bidder

A bidder is a local state machine with the local state variables shown in Figure 3. *good* is a boolean flag that indicates whether the bidder is "good," that is, if it follows the protocol specification. The *phase* variable has one of the values *prebid, bidding, commit*. *wantBid* is a boolean variable that indicates that when the bidder decides that it's time to submit its bid. *receipt_buffer* is the set of IDs of servers from whom the bidder has received a receipt.

5.3 Abstraction of the Publish/Subscribe Communication

The bus has a state with six components. Each component is a set of IDs of servers or bidders who have published messages of each type: *start_buffer, close_buffer, bids_buffer, receipt_buffer, fingerprint_buffer and shares_buffer*. These sets are computed from corresponding state variables in the local states of the servers and bidders.

In any state transition in which a message is published, that fact is recorded in the local state of the publisher, and appears also by definition in the state of the bus. A message can be received (as indicated in a local state variable) only if the message has previously been published, as recorded in the current bus state. This is a fact about the construction of the next-state transition relation. Also, by construction, each buffer set is nondecreasing.

While some state variables contain sets of messages, such messages are formalized as elements of a primitive type, so that the actual contents and formats of protocol messages are not explicitly represented in the specification. Instead, their essential properties are axiomatized.

6 Formal Specification and Verification of Security Properties in PVS

This section describes how the auction protocol was specified and verified using the PVS environment.

6.1 PVS Overview

PVS is a integrated environment for specification and automated verification developed at SRI [12]. PVS specification language is based on higher-order logic with a richly expressive type system. It supports standard theories of integers, sets, functions, and relations, as well as the ability to construct new abstract data types. The PVS theorem prover consists of a

powerful collection of inference steps augmented with a library of decision procedures and the ability to add user-defined proof strategies.

A PVS specification is divided into *theories*, each defining a related set of data types and stating axioms and theorems about them. Data type declarations resemble those in a strongly-typed programming language. The bulk of the auction service specification is in a single theory that introduces types for the state data structures summarized above.

The subsections below show how the essential property of the shared secret function is axiomatized and how the auction service properties are stated. A few remarks about PVS notation should be sufficient to read these formulas.

The new data types include *ID*, *GID*, *BID*, and *trace*. The ID type consists of all auction server IDs, with a subtype GID of good server IDs. The BID type is for bidder IDs. A trace is, by definition, a sequence of global states beginning with an initial state and such that each consecutive pair of states is consistent with the transition relation.

Components of a structure are accessed by using the component names as functions. Local states are obtained from a global state by indexing on the ID, so that, for example, the holdBid component of server i in the global state g is
`holdBid(astate(g)(i))`.

In PVS, a set can be represented by a boolean function. Thus, the formula $x \in \{y|G(y)\}$ would appear in the specification as `G(x)`, and the set itself is written `(G)`.

The shared-secret function invocation $\mathsf{SSF}_i(j)$ is written `SSF(i)(j)`, and `card` is the cardinality function.

6.2 Axiomatization of the Shared Secret Function

The mathematical properties of the threshold sharing scheme are captured by the following axiom, stating that at least $t+1$ shares of a bid must be held by a server S_i, as indicated in its *holdShare* state variable, in order for that server to hold the reconstructed bid, as indicated in its *holdBid* state variable. This is stated as true for every global state in a trace. The contents of *holdBid* are not affected or constrained by any other part of the specification.

```
holdBid_true: AXIOM
   FORALL (trace1:trace,j:nat,i:ID,b:BID):
holdBid(astate(trace1(j))(i))(myBids(b)) AND
good(bstate(trace1(j))(b)) <=>
(EXISTS (y:finite_set[below[N]]):
(FORALL(a:(y)):
holdShare(astate(trace1(j))(i))(SSF(myBids(b))(a))) AND
card(y)>t)
```

6.3 Invariants

The desired properties of the system are invariants; they are true of every reachable state, i.e., every state in a trace. They are proved inductively by showing that they are true in an initial state and preserved by all state transitions.

- ```
  Safe1: THEOREM
    FORALL (trace1: trace, j:nat, gid:GID):
  phase(astate(trace1(j))(gid))=bidding =>
  EXISTS (i:GID): (wantStart(astate(trace1(j))(i))
  ```

  The bidding period starts only after a good auction server decides that it should start.

- ```
  Safe2: THEOREM
    FORALL (trace1: trace, j:nat, gid:GID):
  phase(astate(trace1(j))(gid))=opening =>
  EXISTS (i:GID): (wantClose(astate(trace1(j))(i)))
  ```

  A good auction server stops accepting bids only after some good auction server decides that the bidding period should be closed.

- ```
   pss: THEOREM
  FORALL (trace1:trace,j:nat,i:ID,b:BID):
  holdBid(astate(trace1(j))(i))(myBids(b)) AND
  good(bstate(trace1(j))(b))  =>
  CLOSE_bid(trace1(j))
  ```

  Before the bidding is closed, the identity of the bidder and the bids of the bidder are not revealed. CLOSE_bid(g) is defined as true if all good servers in global state g have reached at least the opening phase.

- ```
   Uniform: THEOREM
  FORALL (trace1:trace,j:nat,i1:GID,i2:GID,b:BID):
  (holdBid(astate(trace1(j))(i1))(myBids(b)) AND i1/=i2
  AND Open_bid(trace1(j))
  AND good(bstate(trace1(j))(b))) =>
  holdBid(astate(trace1(j))(i2))(myBids(b))
  ```

  After the bids are reconstructed, all the good servers reconstruct the same set of bids.

- ```
   Close1: THEOREM
  FORALL (trace1:trace,j:nat,i:GID,b:BID):
  (holdBid(astate(trace1(j))(i))(myBids(b)) AND
  good(bstate(trace1(j))(b))) =>
  validBid(trace1(j))(b)
  ```
  ```

After the bidding period is closed, no more bids can be accepted as valid bids. validBid(g)(b) is defined as true in a global state g if the bid from b has been accepted by at least one good server.

- ```
  commit: THEOREM
    FORALL (trace1:trace,i:GID,b:BID,j:nat):
  phase(bstate(trace1(j))(b))=commit
  AND good(bstate(trace1(j))(b))
  AND Open_bid(trace1(j))
   => holdBid(astate(trace1(j))(i))
  ```

  If a good bidder commits, its bid is guaranteed to be reconstructed and taken into final consideration as a in-time bid.

7 Design and Modeling Issues

This section discusses some issues regarding assumptions and design choices that were made in the present protocol design.

7.1 Other properties of the Auction

At the conclusion of the protocol as presented, all good servers have opened the same set of bids and agreed on a winner. The identity of the bidder supplying that bid is not guaranteed by the protocol. Any authentication or nonrepudiation if needed can be provided by some other cryptographic primitives and the format of the bids which is application-specific.

7.2 Delivery of electronic goods

If the object of the auction is in electronic form, such as software or a postscript file, our original approach can be extended to secure delivery as follows. Every bidder will include a public key in its bid. Then the goods can be transmitted confidentially to the winner by using the public key provided in the winner's bid. This public key need not be certified, because it is in the interests of the winner to provide the correct key, and the good servers will agree on its value.

We might also ask where the file to be awarded was held prior to delivery to the winner. Rather than trust any one server to hold it, it can be split using a $(t+1, n)$ secret-sharing scheme among all servers. Each server will publish its own share encrypted by the winner's public key so that the winning bidder will receive enough shares to reconstruct the item, and a collusion of faulty servers will not be able to reconstruct it.
  ```

7.3 Externally Triggered Transitions

Certain state transitions occur as a result of the passage of time, based on assumptions about the reliability of good servers and network message delivery. Good servers decide to start the bidding and close the bidding according to a predefined date/time schedule for the auction or some external event. They consult a local system clock or receive some other events to trigger those state changes. The triggering events may be out of synchronization, but the protocol compensates for this by forcing good servers to undergo the phase change when it has received signal messages from $t + 1$ other servers. The number $t + 1$ means that at least one good server has sent out its signal.

Event-triggered state changes are indicated with boolean state variables. In the specification, they are set nondeterministically.

7.4 Time Bounds

A good server opens bids only when it knows that all good servers have stopped accepting bids and published their fingerprints. This knowledge comes not from having received any particular number of close or fingerprint messages, but rather from the time bound on actions of good servers and delivery of their messages. The transition to opening bids is triggered in the specification by a predicate on the global state testing whether all good servers have published their fingerprint messages.

The assumption that good servers can send messages to one another within a known time bound is a strong but reasonable assumption. The protocol will fail if some global outage (internet worms, satellite failure, etc.) affects a large portion of the network for an excessive time. We are investigating whether we can weaken the delivery assumption by making use of failure detectors or by assuming instead partial synchrony, where a time bound exists but is not known [2]. Alternatively, it may be adequate to recognize, when a known time bound passes, that an insufficient number of good servers has responded, and declare the auction invalid without compromising the bids. In the present protocol, if too many servers go out of communication, it is a liveness rather than a safety or security problem, since the bids will remain secret.

8 Conclusions

The motivation for this work was to understand whether it is possible to integrate fault-tolerance and security into loosely coupled publish/subscribe systems and to combine the system building blocks with formal techniques to provide possible solutions for electronic commerce systems, particularly a secure auction service.

We have accomplished these goals, and gained assurance in the correctness of the design through the use of an established specification and verification facility. One of the beneficial consequences of the verification activity was a better understanding of what assumptions to

make about message delivery, leading us to a different category of "reliable" transmission that is reasonable for a publish/subscribe system.

We are in the process of implementing a prototype system demonstrating the design, using the Java Infobus application program interface.

Acknowledgements

Thanks to John Rushby for helpful discussions and advice. Thanks to Sergey Berezin and others at SRI for help with PVS.

References

[1] K. P. Birman. The process group approach to reliable distributed computing. *Comm. ACM*, 1993.

[2] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 1988.

[3] E.Clarke, D.Long, and K.McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 353–362, 1989.

[4] M. K. Franklin and M. K. Reiter. The design and implementation of a secure auction service. In *IEEE Security and Privacy Symposium*, pages 2–14. IEEE Computer Society, 1995.

[5] G.Lowe. Breaking and fixing the needham-schroeder public-key protocol using FDR. In *Proceedings of TACAS, Lecture Notes in Computer Science*, volume 1055, 1996.

[6] Li Gong, R.Needham, and R.Yahalom. Reasoning about belief in cryptographic protocols. In *Proceedings of 1990 IEEE Symposium on Research in Security and Privacy*, 1990.

[7] L.Lamport and M.Pease. The byzantine generals problem. *ACM TOPLAS*, 1982.

[8] L.Paulson. Proving properties of security protocols by induction. In *10th IEEE Computer Security Foundations Workshop*, 1997.

[9] M.Burrows, M.Abadi, and R.Needham. A logic of authentication. In *Proceedings of the Royal Society*, volume 426 of *A*, pages 233–271, 1989.

[10] Catherine Meadows. The NRL protocol analyzer: an overview. *Journal of Logic Programming*, 1996.

[11] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus - an architecture for extensible distributed systems. *ACM Operating Systems Review*, 27(5):58–68, 1993.

[12] S. Owre, J. M. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. on Software Engineering*, 21(2):107–125, February 1995.

[13] Michael Reiter. Secure agreement protocols: Reliable and atomic group multicast in rampart. In *2nd ACM Conference on Computer and Communications Security*, November 1994.

[14] R.McAfee and J.McMillan. Auctions and bidding. *Journal of Economic Literature*, 1987.

[15] John Rushby. Formal methods and their role in the certification of critical systems. Technical Report SRI-CSL-95-1, Computer Science Laboratory, SRI International, Menlo Park, CA, March 1995.

[16] John Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. In Mario Dal Cin, Catherine Meadows, and William H. Sanders, editors, *Dependable Computing for Critical Applications—6*, volume 11 of *Dependable Computing and Fault Tolerant Systems*, pages 203–222, Garmisch-Partenkirchen, Germany, March 1997. IEEE Computer Society.

[17] John Rushby and Friedrich von Henke. Formal verification of algorithms for critical systems. *IEEE Transactions on Software Engineering*, 19(1):13–23, 1993.

[18] Fred Schneider. Implementing fault-tolerant services using state machine approach: a tutorial. *ACM Computing Serveys*, 1990.

[19] B. Schneier. *Applied Cryptography*. Wiley, 1996.

A Necessarily Concurrent Attack *

Jonathan K. Millen
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

millen@csl.sri.com
Phone: +1 (415) 859-2358 Fax: +1 (415) 859-2844

Abstract

An artificial protocol called the "ffgg" protocol is constructed, with an assumed security objective to keep a certain data item secret. A message modification attack is given that exposes the data item; in this attack there are two concurrently-running responder processes belonging to the same agent. To show that a concurrent attack is necessary, we use an inductive approach to prove that the protocol is secure under the assumption that this kind of concurrency is excluded.

1 Introduction

Model checking has proved to be a successful way to find vulnerabilities in cryptographic protocols. See, for example, [2, 3, 6]. If a model checker fails to find an attack, however, it may only mean that there is no attack on the particular finite system analyzed. We would like to know under what conditions an analysis of a finite system is, or is not, sufficient to justify a security claim for a protocol in a network environment with an unbounded number of concurrent and past runs of this and other protocols. Under certain restrictive assumptions about the protocol, Lowe has shown that it is sufficient to analyze a system with one honest agent in each role, each of whom can run the protocol just once with the other honest agents [1]. The purpose of this paper is to show that for some other protocols, it is necessary to analyze a system with at least two processes running the same role for the same agent.

Furthermore, the two processes must run concurrently; that is, the protocol is secure if the two processes are serialized. We call this *role concurrency* to distinguish it from the

*This work was supported by ARPA under Arpa Order E301, Air Force Rome Laboratory contract no. F30602-96-C-0291

normal concurrency of the communicating processes in complementary roles. It should also be distinguished from the normal concurrency of independent protocol sessions involving disjoint sets of agents. Role concurrency is significant because state exploration techniques encounter a combinatorial explosion with concurrent processes that is avoided if they can be serialized.

An artificial protocol called the "ffgg" protocol is constructed, with an assumed security objective to keep a certain data item secret. A message modification attack is given that exposes the data item; in this attack there are two concurrently-running responder processes belonging to the same agent. To show that a concurrent attack is necessary, we use an inductive approach to prove that the protocol is secure under the assumption that role concurrency is excluded. The proof technique is based primarily on Paulson's work [5], but it borrows the "ideal" concept from the Thayer, Herzog and Guttman paper [7], and the proof is constructed and checked in the PVS verification environment [4].

2 The ffgg Protocol

In this protocol, A and B are agents (sometimes called "principals"), N_1 and N_2 are nonces, M is a secret message, and *PKB* is B's public key.

1. $A \rightarrow B : A$
2. $B \rightarrow A : N_1, N_2$
3. $A \rightarrow B : \{N_1, N_2, M\}_{PKB}$
4. $B \rightarrow A : N_1, N_2, \{N_2, M, N_1\}_{PKB}$

When B receives message 3, B performs only certain limited checks and computations to form the reply. B checks N_1, and extracts N_2 but *does not check it* against the original value generated by B. We also assume that the type of M is not discernibly different from that of N_2.

The use of *PKB* rather than *PKA* in the last message is not a misprint. We do not claim that this protocol is suitable for any application, only that it poses an interesting problem for analysis.

We call this the "ffgg" protocol because the responder B has two state transitions: the first, or f-transition, is to reply to message 1 with message 2, and the second, or g-transition, is to reply to message 3 with message 4. In the attack scenario, there is another B responder doing f' and g' transitions, and these are interleaved concurrently with f and g in the pattern $ff'gg'$.

3 The Concurrent Attack

A message-modification attack that exposes the secret data item M is presented below.

An agent identifier in parentheses indicates interference by the attacker: if the source is in parentheses, the message has been forged or modified by the attacker. If the destination is in parentheses, the message is intercepted before it reaches the named destination.

There are two responder processes running for agent B; the second process is associated with primed symbols B', N_1', N_2'. Note that, because the second responder process is running on behalf of the same agent B, it still uses the same public key *PKB*.

$$
\begin{array}{ll}
1.\ A \to B : A & \\
 & 1'.\ (A) \to B' : A \\
2a.\ B \to (A) : N_1, N_2 & \\
 & 2'.\ B' \to (A) : N_1', N_2' \\
2b.\ (B) \to A : N_1, N_1' & \\
3.\ A \to B : \{N_1, N_1', M\}_{PKB} & \\
4.\ B \to (A) : N_1, N_1', \{N_1', M, N_1\}_{PKB} & \\
 & 3'.\ (A) \to B' : \{N_1', M, N_1\}_{PKB} \\
 & 4'.\ B' \to (A) : N_1', M, \{M, N_1, N_1'\}_{PKB}
\end{array}
$$

Having shown that there is a concurrent attack, we must now establish that role concurrency is a necessary feature of any successful attack. That is, we must prove that the protocol is secure if role concurrency is disallowed. We do this by setting out to prove that the protocol is secure as it stands, and reducing the proof to one remaining case that fails only if role concurrency is allowed.

4 The Modelling Approach

We apply Paulson's modelling approach. The network environment is captured by several rules that permit message events to be appended to a trace. Most of the rules represent state transitions by agents following a specified role in the protocol. Another rule represents the fabrication of message events by the "Spy."

The content of a message is a *field* in the set $\mathcal{F}$ of one of several subtypes: an *agent* in the set $\mathcal{A}$, a *nonce* in the set $\mathcal{N}$, a *key* in the set $\mathcal{K}$, or it could be computed as a *concatenation* (X, Y) for any $X, Y \in \mathcal{F}$ or a *ciphertext* $\{X\}_K$ for $X \in \mathcal{F}$ and $K \in \mathcal{K}$.

In a more general context, a distinction would be made between public keys and symmetric keys, and other types of fields or computations might be needed.

A *message event* has the form $A \to B : X$, where A and B are the source and destination agents, respectively, and the field X is the message content. When X is a concatenation occurring as a message content, we omit the outer parentheses. We also omit the outer parentheses when the concatenation is being encrypted, and there are brackets around it. When we write a multiple concatenation like (E, F, G), this is supposed to be interpreted as a nested binary concatenation that is parsed right-associatively, as $(E, (F, G))$.

We are taking liberties with Paulson's terminology – for example, Paulson would write an event as Says ABX, and the type of X was called "message." Our variance in terminology is for two reasons: in most of the text we wish to stay as close as possible to conventional protocol notation, and the machine-checked proofs use a different PVS formulation.

The essential aspects of Paulson's model have been retained, however. In particular, the trace only contains send events – no receive events. Also, the source agent name A is the true source and would be "Spy" for messages generated by the Spy. This address is not visible to the destination agent, who can see only the message content X.

A *trace* is a sequence of message events. A *protocol* is a triple (T, S, I) where T is a set of traces, S is a set of secret fields, and I is a set of fields interpreted as the initial knowledge of the Spy. Thus, a protocol, by our definition, includes a secrecy policy and an assumption about what information a potential attacker might have.

We assume that T is prefix-closed (if $st \in T$ then $s \in T$). The elements of S are atomic data items (keys and nonces) that are required to be protected from disclosure to the Spy. Thus, $S \cap I = \emptyset$ (otherwise the game is over). Actually, we need a stronger condition given in the next section.

5 The Secrecy Policy

The secrecy policy for the protocol (T, S, I) is that if $A \rightarrow B : X$ occurs in some trace $t \in T$, then $X \notin S$. (This is what we want, because if the Spy ever obtains a secret item X, it can transmit it as a message.) This secrecy assertion is proved inductively as an invariant. It is clearly true for the null trace, and our objective is to show that the invariant is preserved by each of the rules for appending events to traces.

As is often the case with inductive proofs, the invariant has to be strengthened to carry out the induction. The invariant that we will actually prove is that $X \notin I_k[S]$, where $I_k[S]$ is a Thayer-Herzog-Guttman *ideal*, a set of fields that includes S and which is closed under concatenation with any fields and encryption with keys in k.

As defined in [7], $I_k[S]$ is the smallest set of fields including S such that for all $X \in I_k[S]$, keys $K \in k$, and fields Y,

1. $(X, Y) \in I_k[S]$
2. $(Y, X) \in I_k[S]$
3. $\{X\}_K \in I_k[S]$

The reason for protecting the whole ideal is that compromising any element of the ideal effectively compromises some element of S.

For our purposes, k is the set of keys whose corresponding inverse keys are not in S. Thus, we use the special ideal:

$J[S] = I_k[S]$ where $k = \{K | K^{-1} \notin S\}$.

More generally, an ideal should be closed under all transformations that are reversible by the spy.

We remarked in the previous section that the condition $I \cap S = \emptyset$ is not enough to make (T, S, I) a protocol. This is because the spy's initial knowledge should not contain anything in $J[S]$. Thus, we will assume that

$$I \cap J[S] = \emptyset.$$

6 Modelling the Spy

Given a trace representing the history of messages already sent, the spy can examine the contents of all messages, analyze them by decrypting fi elds for which he has the appropriate key, and synthesize new messages from the fi elds thus obtained. Paulson introduced the set functions sees, parts, analz, and synth to describe these activities.

Given a trace t, the message contents seen by the spy form the set of fi elds:

$$\mathsf{sees}(t) = \{X | (\exists A, B)(A \to B : X) \in t\}.$$

Notation. We are using $\in$ to denote occurrence of a message in a trace as well as for set membership. We will abbreviate $\mathsf{sees}(t)$ with an underline: sees(t) = $\underline{t}$. Also, if $m = (A \to B : X)$, we will write $\underline{m} = X$. Thus, $\underline{t} = \{\underline{m} | m \in t\}$.

The parts of a set G of fi elds include the components of concatenations and the plaintext of encrypted fi elds.

$X \in \mathsf{parts}(G)$ iff
$X \in G$ or
$(\exists Y)((X, Y) \in \mathsf{parts}(G)$ or $(Y, X) \in \mathsf{parts}(G))$ or
$(\exists K)\{X\}_K \in \mathsf{parts}(G)$.

The spy cannot analyze out all the parts, only those for which he has the needed keys. The spy-visible subset of parts is analz.

$X \in \mathsf{analz}(G)$ iff
$X \in G$ or
$(\exists Y)((X, Y) \in \mathsf{parts}(G)$ or $(Y, X) \in \mathsf{analz}(G))$ or
$(\exists K)(\{X\}_K \in \mathsf{analz}(G)$ and $K^{-1} \in \mathsf{analz}(G))$.

Fields are synthesized from existing ones by concatenating them and encrypting them.

$X \in \mathsf{synth}(G)$ iff
$X \in G$ or
$(\exists Y, Z \in G)X = (Y, Z)$ or
$(\exists K, Y \in G)X = \{Y\}_K$.

Paulson showed that each of these three operators is idempotent: that is, $\mathsf{parts}(\mathsf{parts}(G)) = \mathsf{parts}(G), \mathsf{analz}(\mathsf{analz}(G)) = \mathsf{analz}(G)$, and $\mathsf{synth}(\mathsf{synth}(G)) = \mathsf{synth}(G)$.

Given a trace t representing a history of message events, the spy can fabricate new message events Spy $\rightarrow B : X$ to any agent B provided that X can be synthesized from fields analyzed from t or already known initially. Suppose the spy initially knows the fields in I. The predicate Fake gives the rule for creating spy-fabricated messages:

Fake(m, t, I) iff
$(\exists B, X) B \in \mathcal{A}$ and
m = Spy $\rightarrow B : X$ and
$X \in \mathsf{synth}(\mathsf{analz}(\underline{t} \cup I))$.

As usual, the spy is assumed to know the identities of all agents, so they do not have to be included in I. Other fields such as the spy's own secret keys have to be put into I because we don't have a general protocol-independent way to express them.

7 The Secrecy Theorem

The proof of security has a general protocol-independent part and a protocol-specific part. The "Secrecy Theorem" given in this section is the protocol-independent part.

A trace is called *safe* if no field analyzable from it, using fields in I, is in the secret ideal. If every trace in a protocol is safe, then no message exposes any field in S, and the protocol is secure.

Let $\overline{J[S]} = \{X \in \text{field} \mid X \notin J[S]\}$.

t is (S, I)-*safe* iff
$\mathsf{analz}(\underline{t} \cup I) \subseteq \overline{J[S]}$.

The *public* set $\overline{J[S]}$ has the interesting and useful property that it is closed under application of analz. This makes sense, since $J[S]$ is closed under operations that are reversible by analz. This "analz-closure" property is stated below and proved in the appendix. It plays an important part in the proof of the Secrecy Theorem.

Lemma 1 (Analz-closure) $\mathsf{analz}(\overline{J[S]}) \subseteq \overline{J[S]}$.

A protocol is *event-safe* if, given a safe trace of prior messages, the content of the next message is not in the secret ideal. Proving event safety is a protocol-specific activity. It is essentially the induction step of the overall proof.

A protocol (T, S, I) is *event-safe* iff
$(t$ is (S, I)-safe and $tm \in T) \Rightarrow \underline{m} \in \overline{J[S]}$.

Theorem 2 (Secrecy) *If (T, S, I) is event-safe then t is (S, I)-safe for all $t \in T$.*

The proof of the Secrecy Theorem is in the appendix. A machine-checked PVS proof also exists.

8 The ffgg Protocol Rules

The formal version of the ffgg protocol is a recursive predicate defining a set of traces. The definition says that a message event may be appended to a trace if it is permitted by any of five rules, of which one is the Fake rule and the others correspond to the four messages in the normal protocol sequence. A "rule" is just a predicate relating a possible new message to a prior trace.

ffgg(t) iff
  null(t) or
  $t = sm$ where ffgg(s) and
  (Fake(m, s, I) or
  a1(m, s) or
  bf(m, s) or
  a2(m, s) or
  bg(m, s)).

The message-generating rules are given below. The messages in this version of the protocol are somewhat more elaborate than before. The message number and the identity of the sender have been added to the content of each message (since the actual source will not be visible to the recipient).

Terminology: M is a fixed secret field that may be sent to any agent except the Spy. There is a function pk that maps any agent to its public key.

a1(m, t) iff
  $(\exists A, B) A \neq B$ and
  $m = (A \rightarrow B : 1, A)$

bf(m, t) iff
  $(\exists A, B, C, N_1, N_2)$
  $m = B \rightarrow A : 2, B, N_1, N_2$ and
  $(C \rightarrow B : N_1, A) \in t$ and
  $M \neq N_1 \notin$ parts($\underline{t}$) and
  $M \neq N_2 \notin$ parts($\underline{t}$)

a2(m, t) iff
  $(\exists A, B, C, N_1, N_2, M')$

$m = A \to B : 3, A, \{N_1, N_2, M'\}_{\mathsf{pk}(B)}$ and
$(B \neq \text{Spy}$ or $M' \neq M)$ and
$M' \notin \mathsf{parts}(\underline{t})$ and
$(C \to A : 2, B, N_1, N_2) \in t$

bg(m, t) iff
$(\exists A, B, C, X, Y, N_1, N_2)$
$m = B \to A : 4, N_1, X, \{X, Y, N_1\}_{\mathsf{pk}(B)}$ and
$(C \to B : 3, A, \{N_1, X, Y\}_{\mathsf{pk}(B)}) \in t$ and
$(B \to A : 2, B, N_1, N_2) \in t$ and
$(\forall Z)(B \to A : 4, N_1, Z) \notin t$

The a1 rule should need no explanation, but the rest probably do. Note that the agent identifiers A, B, etc., are variables. These rules can be used by any agents at any time, and the generated trace could be a mixture of any number of sessions.

In bf, the conditions that N_1 and N_2 do not occur in t imply that N_1 and N_2 are fresh; they have not been used before. The fact that N_1 and N_2 are not guessable by the spy is implicit in the fact that they cannot be generated by a Fake message. If we wanted them to be guessable, we would add them to the initial knowledge I. Freshness is, of course, not implemented by reading the trace, but instead by generating nonces randomly.

In bf and most of the other rules, there is nothing to prevent an agent from generating the same message repeatedly, which would not be allowed with a more standard state-transition process specification. The extra messages are not a problem when proving a confidentiality property, since if a secret is not exposed with a liberal protocol specification, it is certainly not exposed with a more restrictive one.

The a2 rule generates a message with a field M' that may or may not be the secret M. The conditions on M' are that M' is fresh and that M' is *not* M if the message is being sent to the Spy.

Rule bg includes a check that message 4 with nonce N_1 has not been sent previously. It also checks that the same agent B has sent message 2 with N_1. Agent B does not have to read the trace to know whether these conditions are satisfied; they would be implemented by saving internal state information.

The protocol ffgg is the triple (T, S, I) where

$T = \{t|$ ffgg$(t)\}$

$S = \{M\} \cup \{\mathsf{pk}(A)^{-1} | A \neq \text{Spy }\}$

$I = \{\mathsf{pk}(\text{Spy})^{-1}\} \cup \{\mathsf{pk}(A) | A \in \mathcal{A}\} \cup \mathcal{A}$

9 Characterizing Concurrency

We know that protocol ffgg is not secure, since an attack has been exhibited. What we can prove is that the protocol fgfg = (T', S, I) is secure, where $T' = \{t|$ ffgg(t) and t is not concurrent$\}$.

By "concurrency" we mean actual rather than potential concurrency. It is a property of a trace indicating that two processes are interleaved in such a way that neither one finishes before the other starts. An initiator process and a responder process normally run concurrently. What we are looking for here is role concurrency in two responder processes running on behalf of the same agent.

We have characterized concurrency, for our purposes, in two ways. First, there is a general observation, stated as an axiom, that concurrency is persistent:

Axiom 1 (Persistence of Concurrency) *If t is a concurrent trace, and s is a trace, then ts is a concurrent trace.*

Second, we can identify role concurrency specifically in the ffgg protocol, when it happens with multiple session processes of the same agent playing the role of responder. Role concurrency has occurred for agent B when the trace contains messages m_1, m_2, m_3, m_4 in that order, but not necessarily consecutively, such that:

$$m_1 = B \rightarrow A : 2, B, N_1, N_2$$

$$m_2 = B \rightarrow A' : 2, B, N_1', N_2'$$

$$m_3 = B \rightarrow A : 4, N_1, X$$

$$m_4 = B \rightarrow A' : 4, N_1', Y$$

Here, the responder role is identified by the message numbers 2 and 4 (not by the use of "B" as the agent variable), and the different session processes are distinguished by different nonces, N_1 *vs.* N_1'. The first nonce in each message identifies the session process because nonces are generated freshly in each session, and the first nonce in message 4 is the same as that in message 2 only if that message belongs to the same session process. Thus, m_1, m_3 belong to one session process and m_2, m_4 belong to another. The agents A and A' do not have to be the same; but they are the same in the ffgg attack.

Role concurrency has still occurred if these messages appear in the order m_2, m_1, m_3, m_4. In either case, the two session processes are concurrent because they are not sequential: neither one finishes before the other starts.

10 Proof Notes

PVS is an interactive environment for writing formal specifications and checking formal proofs. It supports a variety of standard data types useful in mathematics and computer science, and it facilitates the definition of new abstract data types. The PVS proof checker manages the proof construction process and provides simplification and decision procedures to carry out relatively large proof steps.

As encoded in the PVS language, message events and their components were abstract data types, and traces were LISP-like lists. Field-set functions like synth were defined recursively, except for analz, which was defined as an inductive relation in almost exactly the form shown earlier. The needed properties stated by Paulson were all confirmed, along with the new results such as analz-closure.

The bulk of the protocol-specific part of the proof was for the "main lemma," which was the statement that fgfg is event-safe. Proofs are recorded and can be replayed, causing the proof checker to recheck the steps against the current version of the specification files. Proof-checking the main lemma takes about 75 seconds of CPU time. This does not include the checks of previously proved lemmas, which were much shorter.

The proof was primarily a matter of considering, in turn, the rules by which a new message could be generated, and asking how that message could possibly be in $J[S]$. A secret message content could not be generated by the Fake rule, because of the synth-closure lemma below, and Fake messages are synthesized from the prior trace, which was assumed (S, I)-safe. Synth-closure, like Analz-closure, is easy to prove.

Lemma 3 (Synth-closure) $G \subseteq \overline{J[S]} \Rightarrow \mathsf{synth}(G) \subseteq \overline{J[S]}$

The only other rule that could possibly generate a secret message content is bg. This case led to an examination of the prior messages that must have been sent, and the messages that must have preceded them. All cases were eliminated except for one, which exhibits the ffgg-specific condition for role concurrency.

Most of the PVS specifications for the protocol and supporting theories are given in an appendix, and so is the final concurrent case. These specifications are included in the report for reference purposes and are intelligible only to readers familiar with the PVS language.

11 Conclusions

We have given an example protocol, the ffgg protocol, for which role concurrency is necessary to disclose a secrecy compromise. Although the example only exhibits a need for two concurrent processes, it is apparent from the structure of the messages that any degree of concurrency could be forced by inserting more nonces.

We have not addressed non-secrecy policies such as authentication or non-repudiation, but the overall approach of proving the protocol correct except for a concurrent case should still apply.

Concurrency was formalized in a protocol-specific way. It would be desirable to express concurrency more generally. To do so, there would have to be a general way of associating message events with the session process that produced them. That seems to require some foresight in the design of message events when the protocol is specified formally at the process level.

Acknowledgement

The motivation for this example and improvements in its presentation arose from helpful discussions with Grit Denker.

References

[1] G. Lowe, "Towards a completeness result for model checking of security protocols," *1998 Computer Security Foundations Workshop*, IEEE Computer Society, 1998.

[2] W. Marrero, E. Clarke, and S. Jha, "Model checking for security protocols," Carnegie Mellon University, CMU-CS-97-139, 1997.

[3] J. C. Mitchell, M. Mitchell, and U. Stern, "Automated analysis of cryptographic protocols using Murphi," *IEEE Symposium on Security and Privacy*, IEEE Computer Society, 1997, pp. 141-151.

[4] S. Owre, J. Rushby, N. Shankar, and F. von Henke, "Formal verification for fault-tolerant architectures: prolegomena to the design of PVS," *IEEE Trans. Software Eng.* 21(2), Feb. 1995, pp. 107-125.

[5] L. Paulson, "Proving properties of security protocols by induction," *10th IEEE Computer Security Foundations Workshop*, IEEE Computer Society, 1997, pp. 70-83.

[6] A. W. Roscoe, "Modelling and verifying key exchange protocols using FDR," *1995 Computer Security Foundations Workshop*, IEEE Computer Society, 1995, pp. 98-107.

[7] F. J. Thayer, J. Herzog, and J. Guttman, "Honest ideals on strand spaces," *1998 Computer Security Foundations Workshop*, IEEE Computer Society, 1998.

A Proof of Analz-closure

Lemma 1 (Analz-closure) $\mathsf{analz}(\overline{J[S]}) \subseteq \overline{J[S]}$.

Proof. The proof makes use of a fixpoint induction principle based on the definition of analz. It can be shown that if:

1. $R \subseteq T$ and
2. $(X, Y) \in T \Rightarrow X \in T$ and $Y \in T$ and
3. $\{X\}_K \in T$ and $K^{-1} \in T \Rightarrow X \in T$

then $\mathsf{analz}(R) \subseteq T$. In particular, $\mathsf{analz}(T) \subseteq T$ requires only items 2 and 3. For $T = \overline{J[S]}$, these statements are just the contrapositives of the closure properties defining $J[S]$. ■

B Proof of Secrecy Theorem

Theorem 2 (Secrecy) *If (T, S, I) is event-safe then t is (S, I)-safe for all $t \in T$.*

Proof. The proof is by induction on the length of the trace t. The null trace is (S, I)-safe if $\mathsf{analz}(I) \subseteq \overline{J[S]}$. But $I \subseteq \overline{J[S]}$ because (T, S, I) is a protocol. As Paulson noted, analz is monotonic, so

$$\mathsf{analz}(I) \subseteq \mathsf{analz}(\overline{J[S]}).$$

An application of analz-closure completes this case.

Now assume that $t = sm$, and we have the induction hypothesis that s is (S, I)-safe. We must show that

$$\mathsf{analz}(\underline{sm} \cup I) \subseteq \overline{J[S]}.$$

By analz-closure, it suffices to show that

$$\underline{sm} \cup I \subseteq \overline{J[S]}.$$

But $\underline{sm} = \underline{s} \cup \{\underline{m}\}$. Event-safety of (T, S, I) says that

$$\underline{m} \in \overline{J[S]}.$$

Since s is (S, I)-safe, we also have

$$\underline{s} \cup I \subseteq \mathsf{analz}(\underline{s} \cup I) \subseteq \overline{J[S]}.$$ ■

C PVS Theories

This appendix containing PVS specifications is included in the interests of making this report as useful as possible to those who might wish to use it as a starting point for their own approaches.

The proofs in this report required six new PVS theories: *message, parts, ffgg, ideal, protocols*, and *results*. Four of these are not specific to the ffgg protocol and could be used for other protocol proofs, although some modifications and extensions might be needed to handle protocols with new computational operators. These four are: *message, parts, ideal,* and *protocols.*

The theories are listed here essentially as they are in the original files, except that the statements of a number of trivial lemmas (and a few not-so-trivial ones) are omitted from *message* and *parts*, because they are not important in themselves.

A few of the results named in the report, such as Analz-closure, appear in the listing with different names, such as (in this case) Analz_public. There is a comment in the listing in each case. Also, the theories *ffgg* and *results* are actually called *ffgg2* and *results2*. I have resisted the urge to clean up the nomenclature further.

Given that the reader is familiar with the PVS specification language, the next most important thing to point out about the protocol representation used here is that a trace is a list of events, with the last event cons'd to the left end of the list.

```
message: THEORY

  BEGIN

  agent: DATATYPE
    BEGIN
      Server: Server?
      Spy: Spy?
      User (Id: nat): User?
    END agent

  pkey: TYPE+
  skey: TYPE+

  invkey(K1: pkey): pkey
  keypair(K1, K2:pkey): bool = (invkey(K1) = K2)

  pub(A: agent): pkey
  prv(A: agent): pkey
  shr(A: agent): skey

  field: DATATYPE
    BEGIN
      Agent (Name: agent): Agent?
      Nonce (Seq: nat): Nonce?
      Pkey (Pkval: pkey): Pkey?
      Skey (Skval: skey): Skey?
      Hash (Arg: field): Hash?
      Con (Head: field, Tail: field): Con?
      Ped (Pcv: pkey, Ptext: field): Ped?
      Sed (Scv: skey, Stext: field): Sed?
    END field

  event: DATATYPE
    BEGIN
      Said (Src: agent, Dest: agent, Cont: field): Said?
    END event

  trace: TYPE = list[event]
```

```
  Pkey_inv: AXIOM
      FORALL (K: pkey): invkey(invkey(K)) = K

  Inv_pub: AXIOM
      FORALL (A: agent): invkey(pub(A)) = prv(A)

  Unique_pub: AXIOM
      FORALL (A, B: agent): pub(A) = pub(B) => A = B

  Unique_prv: AXIOM
      FORALL (A, B: agent): prv(A) = prv(B) => A = B

  Unique_shr: AXIOM
      FORALL (A, B: agent): shr(A) = shr(B) => A = B

  Pkey_exclusion: AXIOM
      FORALL (A, B: agent): pub(A) /= prv(B)

  Agent_keypair: AXIOM
      FORALL (A: agent): keypair(pub(A), prv(A))

% (some basic lemmas omitted)

  END message

%-------------------------------------------------------------

parts: THEORY

BEGIN

  IMPORTING message

  X, Y, Z, W: VAR field
  H: VAR trace
  Kp: VAR pkey
  Ks: VAR skey
  A, B: VAR agent
  N: VAR nat
  E: VAR event
  S, S1: VAR set[field]
```

```
  depth(X): nat = reduce_nat(
      (lambda (a: agent): 1),           % Agent
      (lambda (n: nat): 1),             % Nonce
      (lambda (k: pkey): 1),            % Pkey
      (lambda (k: skey): 1),            % Skey
      (lambda (n: nat): n + 1),         % Hash
      (lambda (n: nat), (m: nat): n + m + 1), % Con
      (lambda (k: pkey), (n: nat): n + 1),    % Ped
      (lambda (k: skey), (n: nat): n + 1)     % Sed
      )(X)

  Pos_depth: CONJECTURE depth(X) > 0

  part?(X, Y): RECURSIVE bool =
    IF X = Y THEN TRUE
    ELSE CASES Y OF
      Con(Z, W): part?(X, Z) OR part?(X, W),
      Ped(Kp, Z): part?(X, Z),
      Sed(Ks, Z): part?(X, Z)
    ELSE FALSE
    ENDCASES
    ENDIF
    MEASURE (lambda (X, Y): depth(Y))

  parts(S)(X): bool = EXISTS Y: S(Y) AND part?(X, Y)

  sees(H)(X): bool = EXISTS E: member(E, H) AND Cont(E) = X

  parts_tr(H): set[field] = parts(sees(H))

% (Some basic lemmas omitted)

  analz(S)(X): inductive bool = S(X) or
         (exists Kp: analz(S)(Ped(Kp, X))
            and analz(S)(Pkey(invkey(Kp)))) or
         (exists Ks: analz(S)(Sed(Ks, X))
            and analz(S)(Skey(Ks))) or
         (exists Y: analz(S)(Con(X, Y))) or
         (exists Y: analz(S)(Con(Y, X)))
```

```
  analz_tr(H): set[field] = analz(sees(H))

  Analz_induct: LEMMA
    (FORALL S, S1:
       (FORALL X:
          S(X)
            OR (EXISTS Kp: S1(Ped(Kp, X)) AND S1(Pkey(invkey(Kp))))
              OR (EXISTS Ks: S1(Sed(Ks, X)) AND S1(Skey(Ks)))
                OR (EXISTS Y: S1(Con(X, Y))) OR (EX-
ISTS Y: S1(Con(Y, X)))
            IMPLIES S1(X))
      IMPLIES subset?(analz(S), S1))

% (some basic lemmas omitted)

  synth(S)(X): RECURSIVE bool =
      IF S(X) THEN TRUE
      ELSE CASES X OF
            Hash(Y): synth(S)(Y),
            Con(Y, Z): synth(S)(Y) AND synth(S)(Z),
            Ped(Kp, Y): S(Pkey(Kp)) AND synth(S)(Y),
            Sed(Ks, Y): synth(S)(Skey(Ks)) AND synth(S)(Y)
            ELSE FALSE
            ENDCASES
      ENDIF
      MEASURE depth

% (some basic lemmas omitted)

  Fake(E, H, S): bool =
      EXISTS A, X:
      E = Said(Spy, A, X) AND
      synth(analz(union(sees(H), S)))(X)

END parts

%-------------------------------------------------------------

ideal: THEORY

  BEGIN
```

```
  IMPORTING parts

  X, Y, Z: VAR field
  Kp: VAR pkey
  Ks: VAR skey
  NA, NB: VAR nat
  A, B: VAR agent
  H: VAR trace
  S, T: VAR set[field]

  secret(S)(X): RECURSIVE bool =
      IF S(X) THEN TRUE
      ELSE CASES X OF
            Con(Y, Z): secret(S)(Y) OR secret(S)(Z),
            Ped(Kp, Y): secret(S)(Y)
                  AND NOT secret(S)(Pkey(invkey(Kp))),
            Sed(Ks, Y): secret(S)(Y)
                  AND NOT secret(S)(Skey(Ks))
            ELSE FALSE
            ENDCASES
      ENDIF
      MEASURE depth

  % secret(S) is the Thayer-Herzog-Guttman k-ideal I_k[S],
  % where "k" consists of all keys whose in-
verses are not secret.
  % (This is called J[S] in the report.)
  % This is the set of messages from which the spy could
  % analyze an element of S.
  % (We could have used keys not in S, but some day there will
  % be computed keys, e.g., xor(Ks1, Ks2).)

  public(S)(X): bool = NOT secret(S)(X)

  publics(S)(T): bool = subset?(T, public(S))

  basic?(S): bool = FORALL X:
      S(X) => (Nonce?(X) OR Pkey?(X) OR Skey?(X))
```

```
  Synth_rank: LEMMA       % called ‘‘Synth-
closure’’ in the report
      (publics(S)(T) AND basic?(S))
      => publics(S)(synth(T))

  Public_part: LEMMA
      public(S)(X) OR (EXISTS Y: S(Y) AND part?(Y, X))


  Public_trace: LEMMA
      publics(S)(analz(sees(H))) OR
      (EXISTS X: S(X) AND parts_tr(H)(X))

  Public_sub: LEMMA
      FORALL (S, T, T1: set[field]):
      (subset?(T1, T) AND publics(S)(T)) => publics(S)(T1)

  Analz_public: LEMMA        % called ‘‘Analz-
closure’’ in the report
      subset?(analz(public(S)), public(S))

  END ideal

%-------------------------------------------------------------

protocols: THEORY

  BEGIN

  IMPORTING message, parts, ideal

  E: VAR event
  X: VAR field
  H, Hp, H1: VAR trace
  S, I: VAR set[field]
  P: VAR set[trace]

  prefix_closed(P): bool =
      FORALL E, H: P(cons(E, H)) => P(H)

  safe(S, I)(H): bool =
```

```
      publics(S)(analz(union(sees(H), I)))

  protocol(P, S, I): bool =
      prefix_closed(P) AND basic?(S)
      AND safe(S, I)(null)

  extends?(H, Hp): RECURSIVE bool =
    IF H = Hp THEN TRUE
    ELSE CASES H OF
      cons(E, H1): extends?(H1, Hp),
      null: null?(Hp)
      ENDCASES
    ENDIF
    MEASURE LAMBDA (H, Hp): length(H)

  Extends_trans: LEMMA
      (extends?(H, H1) AND extends?(H1, Hp)) => ex-
tends?(H, Hp)

  Extends_sub: LEMMA
      extends?(H, Hp) => subset?(sees(Hp), sees(H))

  First_occ: LEMMA
      parts_tr(H)(X)
      => (EXISTS E, Hp:
      extends?(H, cons(E, Hp))
      AND part?(X, Cont(E))
      AND NOT parts_tr(Hp)(X))

  Extends_closed: LEMMA
      (extends?(H, Hp) AND prefix_closed(P) AND P(H))
      => P(Hp)

  Extends_safe: LEMMA
      (extends?(H, Hp) AND safe(S, I)(H)) => safe(S, I)(Hp)

  Mem_event: LEMMA
      (protocol(P, S, I) AND P(H) AND member(E, H))
      => EXISTS Hp: P(cons(E, Hp)) AND ex-
tends?(H, cons(E, Hp))
```

```
  Mem_event_safe: LEMMA
      (protocol(P, S, I)
      AND safe(S, I)(H) AND P(H)
      AND member(E, H))
      => public(S)(Cont(E))

  event_safe(P, S, I): bool = FORALL E, H:
      (safe(S, I)(H) AND P(cons(E, H)))
      => public(S)(Cont(E))

  Secrecy: THEOREM
      (protocol(P, S, I) AND event_safe(P, S, I))
      => subset?(P, safe(S, I))

  END protocols

%-------------------------------------------------------------

ffgg: THEORY

  BEGIN

  IMPORTING message, parts

  A, B, C: VAR agent
  X, Y, Z: VAR field
  E: VAR event
  H: VAR trace
  N1, N2, Ma: VAR nat
  M: nat      % the secret message (nonce)

  initial(X): bool =
      X = Pkey(prv(Spy)) OR
      (EXISTS A: X = Pkey(pub(A))
            OR X = Agent(A))

  secrets(X): bool =
      X = Nonce(M) OR
      (EXISTS A: A /= Spy AND X = Pkey(prv(A)))

  BigM: AXIOM M > 4
```

```
  a1(E, H): bool =                  % A -> B: 1 A
      EXISTS (A, B):
      A /= B AND
      E = Said(A, B, Con(Nonce(1), Agent(A)))

  bf(E, H): bool =                  % B -> A: 2 B N1 N2
          EXISTS (A, B, C, N1, N2):
      E = Said(B, A, Con(Nonce(2), Con(Agent(B), Con(Nonce(N1), Nonce(N2)))
      AND member(Said(C, B, Con(Nonce(1), Agent(A))), H)
      AND NOT parts_tr(H)(Nonce(N1))
      AND NOT parts_tr(H)(Nonce(N2))
      AND N1 /= M AND N2 /= M

  a2(E, H): bool =                  % A -
> B: 3 A {N1 N2 M}PB
      EXISTS (A, B, C, N1, N2, Ma):
      E = Said(A, B, Con(Nonce(3), Con(Agent(A), Ped(pub(B),
            Con(Nonce(N1), Con(Nonce(N2), Nonce(Ma)))))))
      AND (B /= Spy OR Ma /= M)
      AND NOT parts_tr(H)(Nonce(Ma))
      AND member(Said(C, A, Con(Nonce(2),
            Con(Agent(B), Con(Nonce(N1), Nonce(N2))))), H)

  bg(E, H): bool =                  % B -
> A: 4 N1 N2 {N2 M N1}PB
      EXISTS (A, B, C, X, Y, N1, N2):
      E = Said(B, A, Con(Nonce(4), Con(Nonce(N1),
            Con(X, Ped(pub(B), Con(X, Con(Y, Nonce(N1)))))))
      AND member(Said(C, B, Con(Nonce(3), Con(Agent(A),
            Ped(pub(B), Con(Nonce(N1), Con(X, Y)))))), H)
      AND member(Said(B, A, (Con(Nonce(2), Con(Agent(B),
            Con(Nonce(N1), Nonce(N2)))))), H)     % B checks A and N1
      AND NOT EXISTS Z:             % B has not sent this before
            member(Said(B, A, Con(Nonce(4), Con(Nonce(N1), Z))), H)

  ffgg(Q: trace): RECURSIVE bool =
    CASES Q OF
      cons(E, H) : ffgg(H) AND
            (Fake(E, H, initial)
            OR a1(E, H)
```

```
            OR bf(E, H)
            OR a2(E, H)
            OR bg(E, H)),
       null : TRUE
    ENDCASES MEASURE length


  concurrent?(H): bool      % definition TBS: no ff'g or fg'g

  Conc_persists: LAW
      FORALL E, H: concurrent?(H) => concurrent?(cons(E, H))

  END ffgg

%-------------------------------------------------------------

results: THEORY

  BEGIN

  IMPORTING message, parts, ffgg, ideal, protocols

  A, B, C: VAR agent
  X, Y, Z: VAR field
  E: VAR event
  H: VAR trace
  N1, N2: VAR nat

  Basic_secrets: LEMMA
      basic?((secrets))

  Secret_secrets: LEMMA
      initial(X) => public(secrets)(X)

  Initial_public: LEMMA
      subset?(analz(initial), public(secrets))

  ffgg_prefix: LEMMA
      prefix_closed((ffgg))

  fgfg(H): bool = ffgg(H) AND NOT concurrent?(H)
```

```
fgfg_protocol: LEMMA
    protocol((fgfg), (secrets), (initial))

Mem_pub: LEMMA
    (member(E, H) AND ffgg(H) and safe((secrets),(initial))(H))
    => (public((secrets))(Cont(E)) OR concurrent?(H))

Main_lemma: LEMMA
    event_safe((fgfg), (secrets), (initial))

END results
```

D The Concurrent Case

Below is the last remaining case in the proof that the fgfg example protocol is secure. This is a PVS sequent, with a conjunction of hypotheses above the line and a disjunction of conclusions below the line. At this point in the proof, one hypothesis is hidden, namely, that X is secret. Note, however, that X is exposed in the message named e, which is the last message in the trace cons(e, evs).

The hypotheses in this case indicate that certain messages have occurred and give constraints on their order which imply concurrency, in the sense that two sessions are not serializable.

The names of messages, variables, and initial segments (tails) of the trace were not well chosen, so a few remarks are offered here to help explain the logic. For brevity, we write cons(a, b) as a.b.

By [-1], e.evs is a legal trace of ffgg. The secrecy theorem would say that e.evs is "safe," meaning that no transmitted message is secret.

Let e3 = the message in [-6].

The messages e, e1, e2 and e3 are messages sent by B from two sessions corresponding to the nonces Z1 and N1!2. Each message contains a number (2 or 4) identifying its sequence within the protocol. The next nonce is unique to the session and is consistent between messages 2 and 4. (The protocol checks it.)

By [-1] e follows (in time) any message in evs, which includes all the other messages. e2.Hp1 and e1.Hp2 are right-prefixes of evs by [-7] and [-8]. Note that time goes from right to left in these traces.

e2 follows any message in Hp1, and it must precede e1 since Z1 does not occur in Hp1 by [1].

Then e3 is in Hp2 and hence precedes e1. Thus, {e2, e3} precedes e1 precedes e. e2 and e3 are both type 2, e and e1 are type 4. This implies concurrency since it means neither the Z1 session nor the N1!2 session can complete before the other one begins.

```
Main_lemma.5.3.1.3.2.1.3.1.5.2.2.2.1 :

[-1]    ffgg(cons(e, evs))
[-2]    e = Said(B, A, Con(Nonce(4), Con(Nonce(Z1),
            Con(X, Ped(pub(B), Con(X, Y))))))
[-3]    Said(B, A, (Con(Nonce(2), Con(Agent(B), Con(Nonce(Z1),
            Nonce(N1))))))  = e2
[-4]    e1
          =
        Said(B, A!2,
             Con(Nonce(4),
```

```
                Con(Nonce(N1!2), Con(Nonce(Z1), Ped(pub(B),
                Con(Nonce(Z1), Y!1))))))
[-5]    member(Said(C!2, B,
                 Con(Nonce(3),
                     Con(Agent(A!2),
                         Ped(pub(B),
                             Con(Nonce(N1!2), Con(Nonce(Z1), Y!1))))))),
            Hp2)
[-6]    member(Said(B, A!2, (Con(Nonce(2), Con(Agent(B),
              Con(Nonce(N1!2), Nonce(N2!2))))))), Hp2)
[-7]    extends?(evs, cons(e1, Hp2))
[-8]    extends?(evs, cons(e2, Hp1))
[-9]    protocol((fgfg), (secrets), (initial))
[-10]    safe((secrets), (initial))(evs)
[-11]    ffgg(evs)
  |-------
[1]    parts_tr(Hp1)(Nonce(Z1))
[2]    B = Spy
[3]    concurrent?(cons(e, evs))
```

Protocol-Independent Secrecy*

Presented at 2000 IEEE Symposium on Security and Privacy

Jon Millen and Harald Rueß
SRI International
Menlo Park, CA 94025, USA
{millen,ruess}@csl.sri.com

Abstract

Inductive proofs of secrecy invariants for cryptographic protocols can be facilitated by separating the protocol-dependent part from the protocol-independent part. Our Secrecy theorem encapsulates the use of induction so that the discharge of protocol-specific proof obligations is reduced to first-order reasoning. Secrecy proofs for Otway-Rees and the corrected Needham-Schroeder protocol are given.

1 Introduction

Cryptographic protocols are used to achieve goals like authentication and key distribution. In the analysis of these protocols, however, it is important to establish not only that these goals are actually met, i.e. that 'something good is going to happen', but also to prove that no secrets are being revealed, i.e. it is never the case that 'something bad is happening'. In this paper we concentrate on proving secrecy invariants of cryptographic protocols, since these kinds of proofs have often been found to be the hardest task in analyzing a protocol. More precisely, secrecy has been shown to be undecidable even under very weak assumptions on the protocol [2].

Our starting point is the inductive approach developed by Paulson [7]. In this model, a protocol is a rule for adding message events to a trace of prior events. A trace may involve many interleaved protocol runs. For purposes of analysis it is assumed that protocol messages sent over a network are also accessible to a hostile "spy" who is able to read, alter, and forge messages. Paulson uses a theorem prover to partially automate proofs by developing specialized strategies.

*This work was funded by DARPA through the Air Force Research Laboratory Contract F30602-98-C-0258 and by DARPA through Rome Lab contract F30602-96-C-0291.

The main contribution of this paper is a theorem that reduces secrecy proofs for protocols to first-order reasoning; in particular, discharging these proof obligations does not require any inductions. The trick is to confine the inductions to general, protocol-independent lemmas, so that the protocol-specific part of the proof is minimized.

In order to formulate our results, we borrow the notion of ideals on strand spaces [9], and we show how this concept is useful in a trace model context for stating and proving secrecy invariants. We show how the complement of an ideal, which we call a *coideal*, serves as a catalyst to apply Paulson's calculus-like set operators. Our protocol model is also unusual in that message events are interspersed with "spell" events that generate the short-term secrets in a session and specify which principals are supposed to share them.

We originally intended this investigation to support our work in applying a theorem prover to inductive protocol proofs. However, we discovered that these techniques are so effective that we could perform some proofs by hand using them. Manual proofs have been done before, such as the strand-space proofs in [9] and Schneider's CSP proofs [8], but not for Paulson's trace model, and not for a difficult protocol like Lowe's corrected version of the Needham-Schroeder public key protocol [4]. Examples of secrecy proofs are included here for that protocol and for the Otway-Rees [6] protocol.

2 The Modeling Approach

Our modeling approach closely follows Paulson's [7], although the details of the notation are different. A protocol is a rule for adding messages and other events to a history or trace of past events, represented as a sequence. Encrypted message fields are represented symbolically by terms indicating the key and plaintext field.

2.1 Fields

The modeling task begins by defining the *primitive* data types that may occur as message fields: agents $\mathcal{A}$, keys $\mathcal{K}$, and nonces $\mathcal{N}$. (In another context we might use "principal" instead of "agent.") These sets are assumed to be disjoint, and they are all subtypes (subsets) of the field type $\mathcal{F}$. As a notational convention, variables A, B and variants always stand for agents; K and variants always stand for keys; and N and variants are always nonces. X, Y, and Z are arbitrary fields.

Each agent A has some long-term keys: a public key $\mathsf{pub}(A)$, a corresponding private key $\mathsf{prv}(A)$, and a symmetric key $\mathsf{shr}(A)$. The set of long-term keys is denoted $\mathcal{K}_L$. We assume that short-term keys are symmetric keys, and they are in the set $\mathcal{K}_S$.

The *basic* fields are those in the set $\mathcal{N} \cup \mathcal{K}$. These are the kinds of primitive fields that may be designated as secret according to the policy that the protocol is supposed to uphold. Agents and compound fields are never designated as secret by policy, though some compound fields may have to be protected to maintain the secrecy of some of their components.

Compound fields are constructed by concatenation or encryption. The concatenation of X and Y is the term X, Y. We will add brackets, as $[X, Y]$, when necessary to separate a concatenation from its context to avoid confusion. The concatenation operator is binary but associative, so that it may be viewed as n-ary, and a term like $[X, Y, Z]$ is unambiguous.

The encryption of X using the key K is $\{X\}_K$, regardless of the type of key. Each key K has an inverse K^{-1} such that

$$\{\{X\}_K\}_{K^{-1}} = X \tag{1}$$
$$\{\{X\}_{K^{-1}}\}_K = X \tag{2}$$

in the sense that these terms are regarded as equivalent. For any agent A,

$$\mathsf{pub}(A)^{-1} = \mathsf{prv}(A), \tag{3}$$
$$\mathsf{prv}(A)^{-1} = \mathsf{pub}(A), \tag{4}$$
$$\mathsf{shr}(A)^{-1} = \mathsf{shr}(A). \tag{5}$$

There are two special agents: Srv, a trusted server assumed to hold the symmetric (and thus, shared) key $\mathsf{shr}(A)$ of any agent A; and the intruder Spy.

Given sets $\mathcal{A}$, $\mathcal{K}$, and $\mathcal{N}$, and the operators whose signatures and relations have just been given for them, there is an initial algebra generated by them [5]; this algebra is the *cryptospace* of fields. It is an idealized abstraction of the true set of message fields in several ways. For example, there may be an infinite number of nonces and keys, and repeated encryption with the same key generates an infinite number of values.

2.2 Events

There are two kinds of events: messages and spells. Messages are essentially Paulson's Says events, and spells may be thought of as a variation on the Notes event, but with a different purpose.

A *message* is an event $A \to B : X$, where (as implied by our notational conventions) A and B are agents, and X is a field. The *content* X of a message event M is denoted by $\underline{M}$. The sender A and the receiver B will always be the *true* sender and intended receiver, as in Paulson's model.

A *spell* generates certain session-specific primitive fields and designates them as secret. A spell is an event $S \ddagger L$, where S is a set of short-term basic fields called the *book*, and L,

the so-called *cabal*, is a set of agents who are permitted to share the secrets in S. The book and the cabal of a spell event C are denoted by C_σ and C_α, respectively.

As a notational convention, we use E (and variants) to denote events, while M is a message and C is a spell.

A *trace* is a finite sequence of events. Notationally, variants of H are traces. We indicate trace concatenation or postfixing an event to a trace with juxtaposition, e.g., HE, and $E \in H$ means that E occurs in the sequence H, so that $H = H'EH''$. The empty trace is ϵ.

We extend the notion of a content to traces in the natural way. Spells do not contribute to the content.

$$\underline{H} = \{\underline{M} | M \in H\}.$$

2.3 Inductive Relations

The fundamental operations on sets S of message fields, as introduced by Paulson, are $\mathsf{parts}(S)$, $\mathsf{analz}(S)$, and $\mathsf{synth}(S)$.

Briefly, $\mathsf{parts}(S)$ is the set of all subfields of fields in the set S, including components of concatenations and the plaintext of encryptions (but not the keys). Note that if $X \in \mathsf{parts}(\{Y\})$ then X is a subterm of Y, in the sense of [9], written $X \sqsubseteq Y$. The subterm relation is a partial order.

$\mathsf{analz}(S)$ is the subset of $\mathsf{parts}(S)$ consisting of only those subfields that are accessible to an attacker. These include components of concatenations, and the plaintext of those encryptions where the inverse key is in $\mathsf{analz}(S)$. Thus, $\mathsf{analz}(S)$ is defined to be the smallest set such that

1. $S \subseteq \mathsf{analz}(S)$
2. if $[X, Y] \in \mathsf{analz}(S)$ then $X \in \mathsf{analz}(S)$ and $Y \in \mathsf{analz}(S)$
3. if $\{X\}_K \in \mathsf{analz}(S)$ and $K^{-1} \in \mathsf{analz}(S)$ then $X \in \mathsf{analz}(S)$.

Finally, $\mathsf{synth}(S)$ is the set of fields constructible from S by concatenation and encryption using fields and keys in S.

The following properties are stated, for similarly defined sets, in [7]. They are all proved by straightforward inductions.

Proposition 1 *The set transformers* $\mathsf{parts}(S)$, $\mathsf{analz}(S)$, *and* $\mathsf{synth}(S)$ *are* closure *operators – that is, they are extensive (*$S \subseteq \mathsf{parts}(S)$*), monotonic, and idempotent. Furthermore:*

$$\mathsf{parts}(\mathsf{analz}(S)) \quad = \quad \mathsf{parts}(S) \tag{6}$$

$$\mathsf{analz}(\mathsf{parts}(S)) = \mathsf{parts}(S) \quad (7)$$
$$\mathsf{parts}(\mathsf{synth}(S)) = \mathsf{parts}(S) \cup \mathsf{synth}(S) \quad (8)$$
$$\mathsf{analz}(\mathsf{synth}(S)) = \mathsf{analz}(S) \cup \mathsf{synth}(S) \quad (9)$$

The intruder in our model synthesizes faked messages from analyzable parts of a set of available fields. This motivates the definition of $\mathsf{fake}(S)$.

Definition 1 $\mathsf{fake}(S) = \mathsf{synth}(\mathsf{analz}(S))$

Lemma 1 (Fake-Parts)

$$\mathsf{parts}(\mathsf{fake}(S)) = \mathsf{parts}(S) \cup \mathsf{fake}(S)$$

Proof. Using the equalities in Proposition 1.

$$\mathsf{parts}(\mathsf{fake}(S)) = \mathsf{parts}(\mathsf{analz}(S)) \cup \mathsf{fake}(S) = \mathsf{parts}(S) \cup \mathsf{fake}(S). \blacksquare$$

3 Ideals and Coideals

If the spy ever obtains some secret field X, it can transmit X as the content of a message. Thus, our secrecy policy is that if $A \to B : X$ occurs in some trace, then $X \notin S$, where S is a set of basic secrets.

The invariant that we will actually prove is that $X \notin \mathcal{I}(S)$, where $\mathcal{I}(S)$ is the *ideal* generated by S: the smallest set of fields that includes S and which is closed under concatenation with any fields and under encryption with keys whose inverses are not in S. $\mathcal{I}(S)$ is the k-ideal $I_k[S]$ from [9] where k is the set of keys whose inverses are not in S.

With our choice of k, the ideal is defined as follows:

Definition 2 (Ideal) *$\mathcal{I}(S)$ is the smallest set such that*

1. $S \subseteq \mathcal{I}(S)$
2. *if* $X \in \mathcal{I}(S)$ *or* $Y \in \mathcal{I}(S)$ *then* $[X, Y] \in \mathcal{I}(S)$
3. *if* $X \in \mathcal{I}(S)$ *and* $K^{-1} \notin S$ *then* $\{X\}_K \in \mathcal{I}(S)$

Under the assumption that any term not in the ideal may be already compromised, it is necessary to protect this whole ideal, because compromising any element of the ideal

effectively compromises some element of S. It turns out that protecting this ideal is also sufficient.

The complement of $\mathcal{I}(S)$, which we call a *coideal*, is denoted by $\mathcal{C}(S)$. The coideal $\mathcal{C}(S)$ defines the set of fields that are *public* with respect to the basic secrets S, i.e., fields whose release would not compromise any secrets in S.

The property that makes the notion of "coideal" worth defining is that coideals are closed under attacker analysis, thereby implying that protection of the ideal is sufficient.

Lemma 2 (Analz Closure) *For a set S of fields:*

$$\mathsf{analz}(\mathcal{C}(S)) \quad = \quad \mathcal{C}(S)$$

Proof. The right-to-left inclusion follows from extensivity of $\mathsf{analz}(.)$ (Proposition 1). We apply the smallest-set definition of $\mathsf{analz}(.)$ to show

$$\mathsf{analz}(\mathcal{C}(S)) \subseteq \mathcal{C}(S).$$

We have to show that $\mathcal{C}(S)$ is closed under the two rules that expand $\mathsf{analz}(.)$.

First, suppose $[X, Y] \in \mathcal{C}(S)$. That is, $[X, Y] \notin \mathcal{I}(S)$. Hence neither X nor Y is in $\mathcal{I}(S)$ by definition of the ideal, so both are in $\mathcal{C}(S)$.

Second, suppose $\{X\}_K \in \mathcal{C}(S)$ and $K^{-1} \in S$. $\{X\}_K \notin \mathcal{I}(S)$ implies that either $X \notin \mathcal{I}(S)$ or $K^{-1} \in \mathcal{I}(S)$. The first subcase is trivially finished and the latter subcase contradicts the assumption $K^{-1} \in S$.■

An analogous result does not hold for synthesis in general, but depends on the primitiveness of the elements generating the coideal.

Lemma 3 (Synth Closure) *For a set S of basic fields:*

$$\mathsf{synth}(\mathcal{C}(S)) \quad = \quad \mathcal{C}(S)$$

Proof. The left-to-right inclusion is extensivity (Proposition 1). So it remains to show

$$\mathsf{synth}(\mathcal{C}(S)) \subseteq \mathcal{C}(S).$$

We must show that $\mathcal{C}(S)$ is closed under the two rules that expand $\mathsf{synth}(.)$.

First, let $X \in \mathcal{C}(S)$ and $Y \in \mathcal{C}(S)$. We must show that $[X, Y] \in \mathcal{C}(S)$. Otherwise, $[X, Y] \in \mathcal{I}(S)$, either because $[X, Y] \in S$ or because X or $Y \in \mathcal{I}(S)$. The former cannot be true because S is primitive and the latter would contradict the hypothesis for this case.

Second, let $X \in \mathcal{C}(S)$ and $K \in \mathcal{C}(S)$. We must show that $\{X\}_K \in \mathcal{C}(S)$. Otherwise, $\{X\}_K \in \mathcal{I}(S)$, either because $\{X\}_K \in S$, not possible for primitive S; or partly because $X \in \mathcal{I}(S)$, which contradicts the hypothesis for this case. ■

Lemmas 2, 3 are typically used to reduce proof obligations like $\mathsf{analz}(T) \subseteq \mathcal{C}(S)$ to $T \subseteq \mathcal{C}(S)$; similarly for $\mathsf{synth}(.)$.

4 Protocols and Secrecy

A protocol specifies which messages or spells can be appended to an event trace. A secret in a spell book must be *unused* in the prior trace, in the sense that it is not a part of any message content and it has not occurred as a secret in a prior spell.

Definition 3 (Unused)
If H is a trace, X is unused *in H if X is basic, $X \notin \mathsf{parts}(\underline{H})$, and $X \notin C_\sigma$ for any $C \in H$. The set of unused fields in H is denoted by $\mathsf{unused}(H)$.*

Definition 4 (Protocol)
A protocol *is a binary relation between traces and events, such that if $(H, C) \in P$ then $C_\sigma \subseteq \mathsf{unused}(H)$.*

A plenum is a set of traces that could be generated by a protocol in an environment with intruder activity, given some set of fields I assumed initially held by the intruder.

Definition 5 (Plenum)
If P is a protocol then the plenum $\mathcal{U}_I(P)$ *is the set of traces defined inductively by:*

1. $\epsilon \in \mathcal{U}_I(P)$
2. *If $H \in \mathcal{U}_I(P)$ and $(H, E) \in P$ then $HE \in \mathcal{U}_I(P)$*
3. *If $H \in \mathcal{U}_I(P)$ and E is a message from* Spy *with $\underline{E} \in \mathsf{synth}(\mathsf{analz}(\underline{H} \cup I))$ then $HE \in \mathcal{U}_I(P)$*

A message is called honest *(for $\mathcal{U}_I(P)$) if it has been introduced to a trace by means of rule (2) above, while messages introduced by (3) are* fake.

Because protocol spell books introduce unused secrets, it is easy to show that the spell books of different spells are disjoint.

Lemma 4 (Disjoint Book) *If $C, C' \in H \in \mathcal{U}_I(P)$ then either $C = C'$ or $C_\sigma \cap C'_\sigma = \emptyset$.*

The basic secrets associated with a spell include not only the elements of the spell book but also the long-term secrets of the agents in the cabal.

Definition 6 (Basic Secrets) *Let C be a spell;*

$$\mathsf{S}_C = C_\sigma \cup \{\mathsf{prv}(A) | A \in C_\alpha\} \cup \{\mathsf{shr}(A) | A \in C_\alpha\}$$

A spell is compatible with an initial knowledge set that does not compromise its associated basic secrets, or mention the short-term secrets in its book.

Definition 7 (Compatible Spell) *A spell C is I-compatible if*

1. $I \subseteq \mathcal{C}(\mathsf{S}_C)$ *and*
2. $\mathsf{S}_C \cap \mathsf{parts}(I) = \emptyset$.

A trace is *occult* for an initial I if it protects the basic secrets of any spell compatible with I.

Definition 8 (Occult Trace)
A trace H is I-occult *if, for all I-compatible spell events $C \in H$,*

$$\mathsf{analz}(\underline{H} \cup I) \subseteq \mathcal{C}(\mathsf{S}_C)$$

A protocol is secure with respect to its secrecy policy and the spy's initial knowledge if every trace in the plenum it generates is occult. The secrecy proof for a protocol has a protocol-independent part and a protocol-dependent part. The protocol-dependent part is expressed by the *event-occult* property defi ned below. It says that if the prior trace is occult, the next message event generated by the protocol does not compromise a secret. This has to be proved individually for each protocol.

Definition 9 (Event-Occult)
A protocol P is event-occult *if, for all H, I, and C satisfying the conditions:*

1. $C \in H \in \mathcal{U}_I(P)$ *such that H is I-occult,*
2. $(H, M) \in P$, *and*
3. C *is I-compatible*

it is the case that $\underline{M} \subseteq \mathcal{C}(\mathsf{S}_C)$.

The protocol-independent part of a secrecy proof is the Secrecy theorem. It only has to be proved once.

Theorem 1 (Secrecy)
If P *is event-occult then every trace in* $\mathcal{U}_I(P)$ *is* I*-occult.*

Proof. By induction on the trace H. If $H = \epsilon$ then there is nothing to prove, since H contains no spell.

Consider a trace $HE \in \mathcal{U}_I(P)$. We have $H \in \mathcal{U}_I(P)$ and $(H, E) \in P$. The induction hypothesis is that H is I-occult. For the induction step, we must show that HE is I-occult.

Choose a spell $C \in H$ such that $I \subseteq \mathcal{C}(\mathsf{S}_C)$ and $C_\sigma \cap \mathsf{parts}(I) = \emptyset$. We must show that $\mathsf{analz}(\underline{HE} \cup I) \subseteq \mathcal{C}(\mathsf{S}_C)$.

The event E might be either a message or a spell. Suppose first that E is a message. It might be either honest or fake. In either case $\underline{E} \in \mathcal{C}(\mathsf{S}_C)$. For, if E is honest, this is true because P is event-occult. If E is fake, $\underline{E} \in \mathsf{fake}(\underline{H} \cup I)$. By the induction hypothesis, monotonicity of synth, and the Synth-Closure lemma, we have

$$\mathsf{fake}(\underline{H} \cup I) = \mathsf{synth}(\mathsf{analz}(\underline{H} \cup I)) \subseteq \mathsf{synth}(\mathcal{C}(\mathsf{S}_C)) = \mathcal{C}(\mathsf{S}_C).$$

Now we observe that:

$\underline{HE} \cup I = \{\underline{E}\} \cup \underline{H} \cup I$
$\underline{E} \in \mathcal{C}(\mathsf{S}_C)$ as just shown
$I \subseteq \mathcal{C}(\mathsf{S}_C)$ by choice of C
$\underline{H} \subseteq \mathsf{analz}(\underline{H} \cup I) \subseteq \mathcal{C}(\mathsf{S}_C)$
Hence $\underline{HE} \cup I \subseteq \mathcal{C}(\mathsf{S}_C)$
By monotonicity of $\mathsf{analz}()$ (Proposition 1) and Analz-Closure (Lemma 2) we are done with this case.

Now, let E be a spell. We have

$$\mathsf{analz}(\underline{HE} \cup I) = \mathsf{analz}(\underline{H} \cup I) \subseteq \mathcal{C}(\mathsf{S}_C)$$

■

In the following sections we give examples of proofs of the event-occult property for two protocols, from which we may conclude, by the Secrecy theorem, that their traces are

occult. These are strictly secrecy results, and show only that the secrets generated in a particular run of the protocol are not compromised. Most authors of protocol proofs have noted that the security objectives of a protocol may be undermined in other ways than by compromising secrets, usually due to some failure of authentication. We discuss this concern in the Conclusion.

5 Example: The Otway-Rees Protocol

The Otway-Rees protocol is a good one to begin with because the proof is short. Also this protocol was used as an example in [9], so that one can make a comparison between the effort required here with the effort required to do the secrecy part of the strand-space proof in that paper (which was also fairly short).

The goal of the Otway-Rees protocol is to mutually authenticate an initiator and responder and to distribute a session key generated by the server. One session consists of the four messages in Figure 1. We prove that none of the secrets N_a, N_b, or K are disclosed.

$$
\begin{aligned}
or_1 &= A \to B:\ N, A, B, \{N_a, N, A, B\}_{\mathsf{shr}(A)} \\
or_2 &= B \to \mathsf{Srv}:\ N, A, B, \{N_a, N, A, B\}_{\mathsf{shr}(A)}, \{N_b, N, A, B\}_{\mathsf{shr}(B)} \\
or_3 &= \mathsf{Srv} \to B:\ N, \{N_a, K\}_{\mathsf{shr}(A)}, \{N_b, K\}_{\mathsf{shr}(B)} \\
or_4 &= B \to A:\ N, \{N_a, K\}_{\mathsf{shr}(A)}
\end{aligned}
$$

Figure 1: The Otway-Rees Protocol

The informal rules in Figure 1 are easily, albeit somewhat tediously, encoded in the trace model, in roughly the way Paulson would do it, except for the spell event. The spell is specified by g_0, which generates the two nonces N_a, N_b, and the session key K. Note that the server need not be mentioned in the cabal.

The relations or_1, or_2, or_3 (in Definition 11) on message events M and traces H correspond to the messages in the informal description of the protocol. Relation $or_i(H, M)$ holds when rule or_i is used to generate message M. A message is not sent unless there is a suitable prior history of messages sent and received by the sending agent. Rules that introduce nonces take them from a prior spell with the expected cabal. When an agent uses a secret from a spell book, the agent does not see any of the other secrets in the same spell book, though it might know about them from prior messages.

In general, a trace generated by these rules interleaves the behavior of as many agents as we wish, and any number of concurrent or sequential sessions of the same agents. Also,

once a message is enabled, it can be added to the trace any number of times. This is unrealistic, but it is a possible consequence of attacker behavior, and it does not affect secrecy conclusions.

Definition 10 (Otway-Rees) *OR is the union of the relations*

$$
\begin{aligned}
g_0(H,C) &= (\exists N_a, N_b, K, A, B) \\
&\quad N_a, N_b, K \in \mathsf{unused}(H) \\
&\quad \wedge\ C = \{N_a, N_b, K\} \ddagger \{A, B\} \\
or_1(H,M) &= (\exists A, B, N, N_a, C) \\
&\quad N_a \in C \wedge C_\alpha = \{A, B\} \\
&\quad \wedge\ M = A \to B :\ N, A, B, \{N_a, N, A, B\}_{\mathsf{shr}(A)} \\
or_2(H,M) &= (\exists A, A', B, X, N, N_b, C) \\
&\quad N_b \in C \wedge C_\alpha = \{A, B\} \\
&\quad \wedge\ A' \to B :\ N, A, B, X \in H \\
&\quad \wedge\ \ M = B \to \mathsf{Srv} :\ N, A, B, X, \{N_b, N, A, B\}_{\mathsf{shr}(B)} \\
or_3(H,M) &= (\exists A, B, B', N, N_a, N_b, K, C) \\
&\quad K \in C \wedge C_\alpha = \{A, B\} \\
&\quad \wedge\ \ B' \to \mathsf{Srv} :\ N, A, B, \{N_a, N, A, B\}_{\mathsf{shr}(A)}, \{N_b, N, A, B\}_{\mathsf{shr}(B)} \in H \\
&\quad \wedge\ \ M = \mathsf{Srv} \to B :\ N, \{N_a, K\}_{\mathsf{shr}(A)}, \{N_b, K\}_{\mathsf{shr}(B)} \\
or_4(H,M) &= (\exists A, A', BN, Y, N_a, N_b, K) \\
&\quad B \to \mathsf{Srv} :\ N, A, B, X, \{N_b, N, A, B\}_{\mathsf{shr}(B)} \in H \\
&\quad \wedge\ \mathsf{Srv} \to B :\ N, Y, \{N_b, K\}_{\mathsf{shr}(B)} \\
&\quad \wedge\ \ M = B \to A :\ N, Y
\end{aligned}
$$

OR is a protocol in the sense of Definition 4, since g_0 only puts previously unused fields into the book. From the Secrecy Theorem 1 and the following lemma it follows that OR is secure.

Theorem 2 *The OR protocol is event-occult.*

Proof. Let P be OR and choose I. Let $(H, M) \in P$, where $H \in \mathcal{U}_I(P)$ such that H is I-occult. Let $C \in H$ such that $I \subseteq \mathcal{C}(\mathsf{S}_C)$ and $C_\sigma \cap \mathsf{parts}(I) = \emptyset$. We have to show

$$\underline{M} \subseteq \mathcal{C}(\mathsf{S}_C).$$

There are four message rules.

First, consider the case $(H, M) \in or_1$; then:

$$\begin{aligned} M &= A \to B : \ N, A, B, \{N_a, N, A, B\}_{\mathsf{shr}(A)} \\ C' &= \{N_a, N_b, K\} \ddagger \{A, B\} \in H \end{aligned}$$

Notice that $N \notin \mathsf{S}_C$, because N is unused, and $A, B \notin \mathsf{S}_C$ because they are agents. Now consider the encrypted term.

Case $A \in C_\alpha$**.** Then $\mathsf{shr}(A) \in \mathsf{S}_C$; so $\underline{M} \notin \mathcal{C}(\mathsf{S}_C)$.

Case $A \notin C_\alpha$**.** Then $C' \neq C$, so $N_a \notin \mathsf{S}_C$. Hence, $\underline{M} \notin \mathcal{C}(\mathsf{S}_C)$.

Second, in case $(H, M) \in or_2$,

$$\begin{aligned} M &= B \to \mathsf{Srv} : \ N, A, B, X, \{N_b, N, A, B\}_{\mathsf{shr}(B)} \\ M_1 &= A' \to B : \ N, A, B, X \in H \end{aligned}$$

where $N_b \in C'_\sigma$ and $C'_\alpha = \{A, B\}$. Since $M_1 \in H$ and H is I-occult, $\mathsf{analz}(\underline{H} \cup I) \subseteq \mathcal{C}(\mathsf{S}_C)$. But the unencrypted terms $N, A, B, X \in \mathsf{analz}(\underline{M_1}) \subseteq \mathsf{analz}(\underline{H} \cup I)$ so $N, A, B, X \in \mathcal{C}(\mathsf{S}_C)$. The encrypted term is also in the coideal, using the same arguments as for or_1.

Third, in case $(H, M) \in or_3$,

$$\begin{aligned} M &= \mathsf{Srv} \to B : \ N, \{Na, K\}_{\mathsf{shr}(A)}, \{N_b, K\}_{\mathsf{shr}(B)} \\ M_2 &= B' \to \mathsf{Srv} : \ N, A, B, \{N_a, N, A, B\}_{\mathsf{shr}(A)}, \{N_b, N, A, B\}_{\mathsf{shr}(B)} \in H \end{aligned}$$

and there exists a spell $C' \in H$ such that $K \in C'_\sigma$ and $C'_\alpha = \{A, B\}$.

We know $N \in \mathcal{C}(\mathsf{S}_C)$ because it came from M_2. The first encrypted term of M is in the coideal if $A \in C_\alpha$. Otherwise, assume $A \notin C_\alpha$. We know $C \neq C'$ and we must consider the components N_a, K. $K \notin C_\sigma$, so $K \in \mathcal{C}(\mathsf{S}_C)$. As for N_a, it comes from a term in M_2 encrypted with $\mathsf{shr}(A)$, so $N_a \in \mathcal{C}(\mathsf{S}_C)$ because H is I-occult.

The same argument can be used for the second encrypted term and N_b.

The fourth case $(H, M) \in or_4$ is trivial because the message fields have been copied from a received message in H, which is I-occult. ■

6 Example: The Needham-Schroeder Public-Key Protocol

The Needham-Schroeder public-key protocol is a more challenging example, which to our knowledge has not been verified by hand before. The original protocol was found to be flawed by Lowe, who suggested a change in one message that made it secure, as far as he could tell from a model-checking analysis [4]. We demonstrate the applicability of the secrecy theorem (Theorem 1) by proving that Lowe's corrected version of the protocol, which we refer to as NSL, is secure.

$$
\begin{aligned}
a_1 &= A \to B : \{N_a,\ A\}_{\mathsf{pub}(B)} \\
b_2 &= B \to A : \{N_a,\ N_b,\ B\}_{\mathsf{pub}(A)} \\
a_3 &= A \to B : \{N_b\}_{\mathsf{pub}(B)}
\end{aligned}
$$

Figure 2: The Needham-Schroeder-Lowe Protocol

The informal description of the NSL protocol is in Figure 2. Here is the trace relation version.

Definition 11 (NSL Protocol)
The protocol NSL is defined as the union of the binary relations g_0, a_1, b_2, and a_3.

$$
\begin{aligned}
g_0(H,C) &= (\exists A,B,N_a,N_b) \\
&\quad N_a, N_b \in \mathsf{unused}(H) \\
&\quad \wedge\ C = \{N_a, N_b\} \ddagger \{A,B\} \\
a_1(H,M) &= (\exists A,B,C,N_a) \\
&\quad C \in H \wedge\ N_a \in C_\sigma \wedge\ C_\alpha = \{A,B\} \\
&\quad \wedge\ M = A \to B : \{N_a,\ A\}_{\mathsf{pub}(B)} \\
b_2(H,M) &= (\exists A,B,A',C,N_a,N_b) \\
&\quad C \in H \wedge\ N_b \in C_\sigma \wedge\ C_\alpha = \{A,B\} \\
&\quad \wedge\ A' \to B : \{N_a,\ A\}_{\mathsf{pub}(B)} \in H \\
&\quad \wedge\ M = B \to A : \{N_a,\ N_b,\ B\}_{\mathsf{pub}(A)} \\
a_3(H,M) &= (\exists A,B,B',N_a,N_b) \\
&\quad A \to B : \{N_a,\ A\}_{\mathsf{pub}(B)} \in H \\
&\quad \wedge\ B' \to A : \{N_a,\ N_b,\ B\}_{\mathsf{pub}(A)} \in H \\
&\quad \wedge\ M = A \to B : \{N_b\}_{\mathsf{pub}(B)}
\end{aligned}
$$

It is immediate from the definition of NSL that secrets in spells are unused in the prior trace, thus NSL is a protocol in the sense of Definition 4.

Theorem 3 (NSL Protection)
The NSL protocol is event-occult.

Proof. Let P be the NSL protocol and choose I. Let $(H, M) \in P$, where $H \in \mathcal{U}_I(P)$ such that H is I-occult. Let $C \in H$ such that $I \subseteq \mathcal{C}(\mathsf{S}_C)$ and $C_\sigma \cap \mathsf{parts}(I) = \emptyset$. Since H is I-occult, $\mathsf{analz}(\underline{H} \cup I) \subseteq \mathcal{C}(\mathsf{S}_C)$. We have to show that $\underline{M} \subseteq \mathcal{C}(\mathsf{S}_C)$.

There are three message rules.

Case 1. $a_1(H, M)$.

$$M \;=\; A' \to B : \{N_a, A\}_{\mathsf{pub}(B)}$$

and $C' \in H$ such that $N_a \in C'_\sigma$ and $C'_\alpha = \{A, B\}$. If $B \in C_\alpha$ then $\mathsf{shr}(B) \in \mathsf{S}_C$ and the encrypted term is in the coideal. Otherwise, assume $B \notin C_\alpha$. Then $C \neq C'$ and $N_a \notin C_\sigma$. This fact, together with $A \notin \mathcal{I}(\mathsf{S}_C)$ yields $\underline{M} \notin \mathcal{I}(\mathsf{S}_C)$.

Case 2. $b_2(H, M)$.

$$M = B \to A : \{N_a, N_b, B\}_{\mathsf{pub}(A)}$$

and there must exist

$$M_1 = A' \to B : \{N_a, A\}_{\mathsf{pub}(B)} \in H$$

and $C' \in H$ such that $N_b \in C'_\sigma$ and $C'_\alpha = \{A, B\}$.

If $A \in C_\alpha$ then $\underline{M} \notin \mathcal{I}(\mathsf{S}_C)$ and we are done. Suppose $A \notin C_\alpha$. Then we must show that N_a, N_b, and $B \in \mathcal{C}(\mathsf{S}_C)$. N_b is handled like N_a in the first message and B is an agent. It is also trivial for N_a if $B \notin C_\alpha$ because N_a is then exposed in M_1 and H is I-occult.

We must show that $N_a \notin \mathsf{S}_C$ if we assume that $B \in C_\alpha$. Find the earliest occurrence of the subterm $M_1 = \{N_a, A\}_{\mathsf{pub}(B)}$. That is, there is a message M' whose content has M_1 as a subterm, and M_1 is not a part of the prior trace H'. Also, $M_1 \notin \mathsf{parts}(I)$ unless $N_a \in \mathsf{parts}(I)$, in which case $Na \notin \mathsf{S}_C$ by choice of C.

M' might be either faked or honest. If M' is faked, $M_1 \in \mathsf{parts}(\mathsf{fake}(\underline{H'} \cup I)) = \mathsf{parts}(\underline{H'} \cup I) \cup \mathsf{fake}(\underline{H'} \cup I)$ so that we must have $M_1 \in \mathsf{fake}(\underline{H'} \cup I)$. Since $M_1 \notin \mathsf{parts}(\underline{H'} \cup I)$ it must have been synthesized, meaning $[N_a, A] \in \mathsf{fake}(\underline{H'} \cup I) \subseteq \mathcal{C}(\mathsf{S}_C)$, so $N_a \notin \mathsf{S}_C$.

If M' is honest, inspection of the rules and the message component types shows that $\underline{M'} = M_1$ and $a_1(H', M')$ holds. But the analysis of rule a_1 has already been covered in the first case.

Case 3. $a_3(H, M)$. If the receiving agent B of M is in the cabal, then the content is not in the ideal. Thus, assume that B is not in the cabal. From the definition of a_3,

$$M_1 \;=\; A \to B : \{N_a, A\}_{\mathsf{pub}(B)} \in H$$

$$\begin{aligned} M_2 &= B' \to A : \{N_a, N_b, B\}_{\mathsf{pub}(A)} \in H \\ M &= A \to B : \{N_b\}_{\mathsf{pub}(B)} \end{aligned}$$

and one has to show $N_b \notin \mathsf{S}_C$.

If M_2 is honest, then there exists a prefix H' of H such that $b_2(H', M_2)$ and there exists C' with $N_b \in C'_\sigma$ and $C'_\alpha = \{A, B\}$. But $B \notin C_\alpha$, so $C \neq C'$ and $N_b \notin \mathsf{S}_C$.

Note that this step fails if the sender of M_2 does not occur in the encryption field, since then we could not say (in the rule) that $B \in C'_\alpha$. This is the difference between NSL and the original protocol.

If M_2 is faked, find the earliest message M' containing M_2 as a part, where the prior trace is H'. By choice of M', $M_2 \notin \mathsf{parts}(\underline{H'})$. Also, $M_2 \notin \mathsf{parts}(I)$, otherwise $N_b \in \mathsf{parts}(I)$, and we have assumed $\mathsf{parts}(I) \cap \mathsf{S}_C = \emptyset$, so $N_b \notin \mathsf{S}_C$ and we would be done. So $M_2 \notin \mathsf{parts}(\underline{H'} \cup I)$.

If M' is faked, we have

$$M_2 \in \mathsf{parts}(\mathsf{fake}(\underline{H'} \cup I)) = \mathsf{parts}(\underline{H'} \cup I) \cup \mathsf{fake}(\underline{H'} \cup I)$$

so

$$M_2 \in \mathsf{fake}(\underline{H'} \cup I) = \mathsf{synth}(\mathsf{analz}(\underline{H'} \cup I)).$$

Since $M_2 \notin \mathsf{parts}(\underline{H'} \cup I)$ it must be that M_2 has been synthesized, so

$$N_a, N_b, B \in \mathsf{fake}(\underline{H'} \cup I).$$

But $\mathsf{fake}(\underline{H'} \cup I) \subseteq \mathcal{C}(\underline{H'} \cup I)$ since H is I-occult, implying that $N_b \notin \mathsf{S}_C$.

If M' is honest, inspection of the protocol rules shows that $M' = M_2$ and $b_2(H, M_2)$, and this case was covered previously. ■

7 Conclusions

Our secrecy theorem separates protocol-dependent and protocol-independent aspects of secrecy proofs. The protocol-dependent part is to show the "event-occult" property, which only asks whether honest messages compromise secrets, given strong assumptions about the preservation of secrecy in the prior message history.

The secrets to be protected are defined in an explicit, uniform way by introducing "spell" events into the protocol. Spell events generate the short-term secrets for a particular "cabal," the set of agents sharing the new secrets. Secrets are shown to be protected even

when the long-term secrets of other agents, or the short-term secrets in other protocol runs (with other spells) are compromised.

The security of a protocol can be subverted even when the secrets it generates are protected. To take a simple example, consider the single-message key-distribution protocol:

$$A \to B : \ A, \{K\}_{\mathsf{pub}(B)}.$$

We can show that the session key K is kept secret. However, B would be foolish to believe that K came from A and use it to encrypt information to be shared only with A.

There are two ways to avoid this kind of problem. One is to conduct a separate authentication proof, and attempt to establish that the key received by B was actually sent by A, and is fresh. If *both* the secrecy proof and the authentication proof succeed (and the second will fail in this example), the protocol would be shown secure. While most authors who have developed analysis techniques for secrecy have extended those techniques to perform authentication proofs as well, we should consider that there are some very appealing authentication logic techniques designed for this purpose [1, 3]. Their only drawback is that they cannot show secrecy properties. It would be ideal to use them in a context where secrecy has already been shown.

If the ultimate objective is really to show secrecy, not for the session key *per se,* but for some text encrypted with it, then there is another way to focus on the correct goal: include the use of the key in the protocol. To do this, add one or more statements to the protocol specifi cation. In the example above, we could add the message:

$$B \to A : \ \{N\}_K$$

where N is a new secret. The augmented protocol is not event-occult.

The closure results on the coideal have turned out to be a useful addition to the arsenal of proof techniques, enabling interesting examples to be shown secure. Protocol proofs are still complex enough so that we feel proof-checking and automation to be valuable for the sake of assurance, and we believe that the same techniques that simplify manual proofs will also be helpful in organizing machine-assisted proofs.

References

[1] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.

[2] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In *Formal Methods and Security Protocols*, Federated Logic Conference, 1999.

[3] L. Gong, R. Needham, and R. Yahalom. Reasoning about belief in cryptographic protocols. In *IEEE Symposium on Research in Security and Privacy*, pages 234–248. IEEE Computer Society, 1990.

[4] G. Lowe. Breaking and fi xing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of TACAS*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, 1996.

[5] J. Meseguer and J. Goguen. Initiality, induction, and computability. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge University Press, 1982.

[6] D. Otway and O. Rees. Effi cient and timely mutual authentication. *ACM Operating System Review*, 21(1):8–10, 1987.

[7] L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1):85–128, 1998.

[8] S. Schneider. Verifying authentication protocols in CSP. *IEEE Transactions on Software Engineering*, 24(9):741–758, September 1998.

[9] J. Thayer, J. Herzog, and J. Guttman. Honest ideals on strand spaces. In *11th IEEE Computer Security Foundations Workshop*, pages 66–78. IEEE Computer Society, 1998.

Local Secrecy for State-Based Models*

Presented at FMCS'2000

Jon Millen and Harald Rueß
SRI International
Menlo Park, CA 94025, USA
{millen,ruess}@csl.sri.com

Abstract

Proofs of secrecy invariants for cryptographic protocols can be facilitated by separating the protocol-dependent part from the protocol-independent part. Our Secrecy theorem encapsulates the use of induction so that the discharge of protocol-specific proof obligations is reduced to first-order reasoning. The theorem has been proved and applied in the PVS environment with supporting protocol representation theories based on a state-transition model. This technique has been successfully applies to both standard benchmark examples and to parts of the verification of the Enclave group management system.

1 Introduction

Cryptographic protocols are used to achieve goals like authentication and key distribution in a hostile internet environment. Formal methods can be used to verify the adequacy of the design of such protocols. Security goals are often formalized as invariants. In this paper we concentrate on proving secrecy invariants, which are important both for their own sake and to support authentication goals.

The main emphasis of this paper lies in the description of our PVS [5] formalization of the secrecy theorem published in [3]. This theorem reduces secrecy proofs for protocols to first-order reasoning; in particular, discharging these proof obligations does not require any inductions. The trick is to confine the inductions to general, protocol-independent lemmas, so that the protocol-specific part of the proof is minimized. Moreover, secrecy protocols

*This work was funded by DARPA through AFRL contract F30602-98-C-0258 and by DARPA through Rome Lab contract F30602-96-C-0291.

are modularized in the sense that there are separate verification conditions for each protocol rule.

The secrecy theorem in [3] was based on Paulson's trace model. Here we reformulate this theorem to also work on a state-based model which is more compatible with the one propagated by Mitchell et al. in [1]. We illustrate the encoding of specific protocols in this model using the Otway-Rees protocol [4]. We do not, however, go into details of proofs, since they are mostly straightforward adaptations of the ones stated in [3].

In order to formulate our results, we borrow the notion of ideals on strand spaces [6], and we show how this concept is useful in a state model context for stating and proving secrecy invariants. We show how the complement of an ideal, which we call a *coideal*, serves as a catalyst to apply Paulson's calculus-like set operators. Our protocol model is also unusual in that message events are interspersed with "spell" events that generate the short-term secrets in a session and specify which principals are supposed to share them.

Besides proving secrecy results of standard benchmark protocols like the Otway-Rees and the Needham-Schroeder (public key) protocols, our methods have been applied successfully[1] in the process of verifying the group management services of Enclaves [2].

2 The Modeling Approach

Our modeling approach is fairly close to the MSR idea in [1], although the details of the notation are different. A protocol is a rule for placing messages and updating local states in a global set of current events. Encrypted message fields are represented symbolically by terms indicating the key and plaintext field.

2.1 Fields

The modeling task begins by defining the *primitive* data types that may occur as message fields: agents, keys, and nonces. (In another context we might use "principal" instead of "agent.") These sets are assumed to be disjoint, and they are all subtypes (subsets) of the field type $\mathcal{F}$. They are modeled as abstract datatypes in PVS.

An agent is either an 'ordinary' user, a dedicated server `Srv`, or the supposedly malicious `Spy`. Each agent `A` has some long-term keys: a public key `Pub(A)`, a corresponding private key `Prv(A)`, and a symmetric key `Shr(A)`.

Message fields are divided into primitive and compound fields. The primitive fields containing agents, nonces, and keys are constructed as `Agent(A)`, `Nonce(N)`, and `Key(K)`.

[1] Private communication: B. Dutertre, SRI International.

(The PVS conversion mechanism is used to suppress these injections in the sequel.) Compound fields are constructed by concatenation or encryption. The concatenation of `X` and `Y` is the term `X ++ Y`. or encryption `Encr(K, X)`. The encryption of `X` using the key `K` is `Encr(K,X)`, regardless of the type of key. The possible message fields are elements of the datatype `field`.

Agents and compound fields are never designated as secret by policy, though some compound fields may have to be protected to maintain the secrecy of some of their components. Thus, we define basic fields as nonces and keys, which are the kinds of primitive fields that may be designated as secret according to policy. The PVS definition of the membership predicate `basic?` is shown below. PVS fragments are displayed in this paper within boxes.

```
basic?: set[field] = union(Nonce?, Key?)
```

As a notational convention, variables `A`, `B` and variants always stand for agents; `K` and variants always stand for keys; and `N` and variants are always nonces. `X`, `Y` and `Z` are arbitrary fields.

Each key `K` has an inverse.

```
inv(K): key =
   CASES K OF
    Pub(A): Prv(A), Prv(A): Pub(A), Shr(A): Shr(A), Ssk(A): Ssk(A)
   ENDCASES
```

Thus, both `Shr(A)` and `Ssk(A)` are symmetric. The special agent `Server` is assumed to hold the symmetric (and thus, shared) key `Shr(A)` of any agent `A`.

2.2 Events

There are three kinds of events: messages, spells, and state events.

```
event: DATATYPE
BEGIN
  Msg(Cont: field): Msg?
  Cast(Secrets: set[(basic?)], Cabal: set[agent]): Spell?
  State(Role: nat, Label: nat, Memory: field): State?
END event
```

Messages are essentially Paulson's Says events, and the content of a message event is a field. We do not need to refer to the sender and receiver of a message. A *spell* generates

certain session-specific primitive fields and designates them as secret. A spell is an event `Cast(S, C)`, where `S` is a set of short-term basic fields called the *book*, and `C`, the so-called *cabal*, is a set of agents who are permitted to share the secrets in `S`.

As a notational convention, we use `E` (and variants) to denote events, while `M` is a message and `C` is a spell.

A *global state* is simply a collection of events. Notationally, variants of `H` are global states. We shall see later that states reachable by a protocol contain messages in transit and local states of agents participating in the protocol.

```
global: TYPE = set[event]
```

We extend the notion of a content to global states in the natural way. Spells and state events do not contribute to the content. Similarly, the secrets of a state are obtained as the basic fields of the secrets of its cast events.

```
sees(H)(X): boolean = EXISTS (M: (Msg?)): member(M,H) & Cont(M) = X

secrets(H)(X): boolean = EXISTS (C: (Spell?)):
                   member(C,H) & basic?(X) & member(X,Secrets(C))
```

2.3 Inductive Relations

The fundamental operations on sets `S` of message fields, as introduced by Paulson, are `Parts(S)`, `Analz(S)`, and `Synth(S)`.

Briefly, `Parts(S)` is the set of all subfields of fields in the set `S`, including components of concatenations and the plaintext of encryptions (but not the keys). Note that if `member(X, Parts({Y}))`, then `X` is a subterm of `Y`, in the sense of [6], written `X <= Y`. The subterm relation is a partial order.

`Analz(S)` is the subset of `Parts(S)` consisting of only those subfields that are accessible to an attacker. These include components of concatenations, and the plaintext of those encryptions where the inverse key is in `Analz(S)`.

```
Analz(S)(X): INDUCTIVE bool =
   S(X)
 OR (EXISTS Y: Analz(S)(X ++ Y))
 OR (EXISTS Y: Analz(S)(Y ++ X))
 OR (EXISTS K: Analz(S)(Encr(K, X)) AND Analz(S)(inv(K)))
```

The intruder in our model synthesizes faked messages from analyzable parts of a set of available fields. This motivates the definition of `fake(S)`.

```
Fake(S): set[field] = Synth(Analz(S))

Fake_Parts: LEMMA Parts(Fake(S)) = union(Parts(S), Fake(S))
```

3 Ideals and Coideals

If the spy ever obtains some secret field X, it can transmit X as the content of a message. Thus, our secrecy policy is that if the message with content X occurs in some trace, then `NOT member(X,S)`, where S is a set of basic secrets.

The invariant that we will actually prove is that `NOT member(X, Ideal(S))`, where `Ideal(S)` is the *ideal* generated by S: the smallest set of fields that includes S and which is closed under concatenation with any fields and under encryption with keys whose inverses are not in S. `ideal(S` is the k-ideal $I_k[S]$ from [6] where k is the set of keys whose inverses are not in S.

With our choice of k, the ideal is defined as follows:

```
Ideal(S)(X): INDUCTIVE boolean =
   S(X)
 OR (EXISTS Y, Z: X = Y ++ Z & (Ideal(S)(Y) OR Ideal(S)(Z)))
 OR (EXISTS Y, K: X = Encr(K, Y) & Ideal(S)(Y) & NOT S(inv(K)))
```

Under the assumption that any term not in the ideal may be already compromised, it is necessary to protect this whole ideal, because compromising any element of the ideal effectively compromises some element of S. It turns out that protecting this ideal is also sufficient.

The complement of and ideal, which we call a *coideal,* is denoted by `Coideal(S)`. This defines the set of fields that are *public* with respect to the basic secrets S, i.e., fields whose release would not compromise any secrets in S.

The property that makes the notion of "coideal" worth defining is that coideals are closed under attacker analysis, thereby implying that protection of the ideal is sufficient.

```
Analz_Closure: LEMMA Analz(Coideal(S)) = Coideal(S)

Synth_Closure: LEMMA subset?(S,(basic?)) =>
 Synth(Coideal(S)) = Coideal(S)
```

4 Protocols and Secrecy

A protocol specifies which messages or spells can be added to a global state. A secret in a spell book must be *unused* in the prior state, in the sense that it is not a part of any message content and it has not occurred as a secret in a prior spell.

```
unused(H: global)(X: field): boolean =
     basic?(X) & NOT(Parts(sees(H))(X)) & NOT(secrets(H)(X))
```

A protocol rule is a triple consisting of a pre- and a post set of events and a set of nonces. Intuitively, such a rule is applicable in some global state H if the pre events are a subset of H and if the nonces in the rule are unused in H. A rule fires by deleting the pre events from the state and adding the post events.

```
rule: TYPE =
    [# Pre: set[event], Nonces: set[(basic?)], Post: set[event] #]
```

There are several local conditions on protocol rules. First, there is at most one spell in the post, and a cast and a message event may not occur simultaneously in the post. Second, all secrets of casts in the post must be subset of the rule nonces. Third, regularity states that whenever a longterm key K is neither in the parts of the content or the memory of the pre then it is also not in the parts of the content or the memory of the post.

```
single_spell(post: set[event]): boolean =
  FORALL (C, C1: (Cast?), E: (Event?)):
    (member(C, post) & member(C1, post) => C = C1)
  & (member(C, post) & member(E, post) => NOT Msg?(E))

fresh(Ns: set[(basic?)], post: set[event]): boolean =
  FORALL (C: (Cast?)): member(C,post) => subset?(Secrets(C),Ns)

regular(pre, tau1): boolean =
  FORALL(K: longterm):
    (NOT(Parts(sees(pre))(K)) & NOT(Parts(memory(pre))(K)))
       => (NOT(Parts(sees(post))(K)) & NOT(Parts(memory(post))(K)))
```

It is usually straightforward to check that rules of a specific protocol obey these conditions. Usually, we (mis)use the PVS prover to automatically check these static conditions.

Rules that satisfies the conditions above are collected in the type `protocol`.

```
protrule(rl: rule): boolean =
   single_spell(Post(rl))
   & fresh(Nonces(rl),Post(rl))
   & regular(Pre(rl),Post(rl))

protocol: TYPE = set[(protrule)]
```

A protocol `P` and a given set of initial knowledge `I` (of the spy), a *global* `I`-*extension* is a binary relation of states. This relation determines a transition system. An extension is either `honest`, i.e. it corresponds to a move by a player following the rules, or it is `faked` by the spy. As usually, the spy is reduced to add only messages with a content that can be inferred from the content of the current state and the initial knowledge.

```
honest(P: protocol)(H, H1): boolean =
  EXISTS(rl: (P)): subset?(Nonces(rl), unused(H))
 & subset?(Pre(rl), H)
 & H1 = union(Post(rl), difference(H, Prestates(rl)))

fake(I: set[field])(H, H1): boolean =
  EXISTS(X: (Fake(union(sees(H), I)))): H1 = add(Msg(X), H)

global_extension(P: protocol, I: set[field])(H, H1): boolean =
  honest(P)(H, H1) OR fake(I)(H, H1)
```

We need some further concepts before stating our secrecy theorem. The *basic secrets* associated with a spell include not only the elements of the spell book but also the long-term secrets of the agents in the cabal.

```
ltk(C: (Cast?))(X: field): boolean =
   Key?(X)
 & longterm(Val(X))
 & EXISTS(A: agent): Q(A)(Val(X)) & Cabal(C)(A)

basic_secrets(C)(X: field): boolean =
   basic?(X) AND (Secrets(C)(X) OR ltk(C)(X))
```

A spell is *compatible* with an initial knowledge set that does not compromise its associated basic secrets, or mention the short-term secrets in its book.

```
compatible(I: set[field])(C: (Cast?)): boolean =
  disjoint?(basic_secrets(C), Parts(I))
```

The set of reachable states `H` is defined in the usual way using a least fixed-point definition.

```
reachable(P, I)(H): INDUCTIVE boolean =
   empty?(H) OR (EXISTS (G: global): reachable(P, I)(G)
   & global_extension(P, I)(G, H))
```

A protocol is secure with respect to its secrecy policy and the spy's initial knowledge `I` if every reachable state it generates is secret-secure. This property, for traces, was called "discreet" in [3].

```
secret_secure(I: set[field])(H: global): boolean =
  FORALL C: compatible(I)(C) & H(C)
     => subset?(sees(H), Coideal(basic_secrets(C)))
```

The secrecy proof for a protocol has a protocol-independent part and a protocol-dependent part. The protocol-dependent part is expressed by the *occult*ness property defined below. It says that if the prior state is secret-secure, the next message event generated by the protocol does not compromise a secret. This has to be proved individually for each protocol. This protocol property was called "discreet" in [3].

```
occult(P: protocol): boolean =
  FORALL (I: set[field], H: global, C: (Cast?), rp: (protrule)):
       reachable(P, I)(H)
     & secret_secure(I)(H)
     & compatible(I)(C)
     & H(C)
     & subset?(Pre(rp), H)
     & P(rp)
     => subset?(sees(Post(rp)),Coideal(basic_secrets(C)))
```

The protocol-independent part of a secrecy proof is the Secrecy theorem. It only has to be proved once.

```
secrecy: THEOREM
  occult(P) => subset?(reachable(P, I), secret_secure(I))
```

The proof of this theorem is along the lines of the proof in [3] for proving a secrecy theorem for trace models, but now the induction is on the length of protocol extensions (see Definition of reachability).

Notice that these are strictly secrecy results, and show only that the secrets generated in a particular run of the protocol are not compromised. Most authors of protocol proofs have noted that the security objectives of a protocol may be undermined in other ways than by compromising secrets, usually due to some failure of authentication. Possible combinations of secrecy and authentication are discussed in [3].

5 Example: The Otway-Rees Protocol

The goal of the Otway-Rees protocol is to mutually authenticate an initiator and responder and to distribute a session key generated by the server. One session consists of the four messages in Figure 1. We prove that none of the secrets N_a, N_b, or K are disclosed.

$$
\begin{aligned}
or_1 &= A \to B:\ N, A, B, \{N_a, N, A, B\}_{\mathsf{shr}(A)} \\
or_2 &= B \to \mathsf{Srv}:\ N, A, B, \{N_a, N, A, B\}_{\mathsf{shr}(A)}, \{N_b, N, A, B\}_{\mathsf{shr}(B)} \\
or_3 &= \mathsf{Srv} \to B:\ N, \{N_a, K\}_{\mathsf{shr}(A)}, \{N_b, K\}_{\mathsf{shr}(B)} \\
or_4 &= B \to A:\ N, \{N_a, K\}_{\mathsf{shr}(A)}
\end{aligned}
$$

Figure 1: The Otway-Rees Protocol

The informal rules in Figure 1 are easily, albeit somewhat tediously, encoded in the trace model. Here we only state a selection of the formalization of the Otway-Rees protocol rules.

The spell rule `spl1` generates the nonce `Na` as needed for the first protocol step. Note that the server need not be mentioned in the cabal.

```
spl1(A, B: agent, Na: nonce): (protrule) =
    (# Pre    := emptyset,
       Nonces := singleton(Na),
       Post   := singleton(Cast(add(Na,emptyset),
                                add(A, add(B, emptyset))))
     #)
```

The type constraint `(protrule)` causes the PVS type checker to generate verification conditions corresponding to the conditions on protocol rules. These and all the other verification conditions are easily discharged using the PVS prover.

Sending and receiving is split into two parts. The first step in the Otway-Rees protocol, for example, is transcribed as follows.

```
snd1(A, B: agent, N, Na: nonce): (protrule) =
    (# Pre    := add(State(roleA, 0, A ++ B ++ Srv),
                  add(Cast(add(Na, emptyset),
                           add(A, add(B, emptyset))), emptyset)),
       Nonces := add(N, emptyset),
       Post   := add(State(roleA, 1, A ++ B ++ Srv ++ Na),
                   add(Msg(N ++ A ++ B ++ Encr(Shr(A),
                           Na ++ N ++ A ++ B)), emptyset))
     #)

rcv1(A, B: agent, N, Na: nonce): (protrule) =
    (# Pre    := add(State(roleB, 0, B ++ Srv),
                     singleton(Msg(N ++ A ++ B
                                   ++ Encr(Shr(A), Na ++ N ++ A ++ B)))),
       Nonces := emptyset,
       Post   := singleton(State(roleB, 1, B ++ Srv ++ N ++ A))
     #)
```

Rules that introduce nonces (to be kept secret) take them from a prior spell with the expected cabal. When an agent uses a secret from a spell book, the agent does not see any of the other secrets in the same spellbook, though it might know about them from prior messages.

In general, a sequence of states generated by these rules interleaves the behavior of as many agents as we wish, and any number of concurrent or sequential sessions of the same agents. Altogether, the Otway-Rees protocol is formalized as follows.

```
otway_rees: protocol =
   { r: (protrule) |
             EXISTS A, B, N, Na, Nb, K:
                    r = init(A, B)
                OR r = spl1(A, B, Na)
                OR r = snd1(A, B, N, Na)
                OR r = rcv1(A, B, N, Na)
                OR ...  }
```

The secrecy theorem states that it suffices to show `occult(otway_rees)`. In a first step, using skolemization and split rules in order to show occultness for reach rule separately. For the lemma below occultness follows trivially for most protocol rules.

```
sufficient_for_occultness: LEMMA
    disjoint?(Msg?, Post(rp)) => occult(singleton(rp))
```

It remains to prove occultness for four rules in the Otway-Rees protocol. In the case of the `snd1` rule, for example one has to prove.

```
{-1} subset?(sees(H), Coideal(basic_secrets(C)))
{-2} reachable(OR, I)(H)
{-3} H(C)
{-4} H(State(roleA, 0, A ++ B ++ Srv))
{-5} H(Cast(add(Nonce(Na), emptyset), add(A, add(B, emptyset))))
|----------
{1}  Coideal(basic_secrets(C))
                  (N ++ A ++ B ++ Encr(Shr(A), Na ++ N ++ A ++ B))
```

Currently, we still prove these kinds of verification conditions in an interactive way (typically around 20-40 interactions per rule), but the repetitive patterns in these proofs suggest higher-level proof strategies.

6 Conclusions

Our secrecy theorem separates protocol-dependent and protocol-independent aspects of secrecy proofs. The protocol-dependent part is to show the occultness property, which only asks whether honest messages compromise secrets, given strong assumptions about the preservation of secrecy in the prior message history.

The secrets to be protected are defined in an explicit, uniform way by introducing "spell" events into the protocol. Spell events generate the short-term secrets for a particular "cabal", the set of agents sharing the new secrets. Secrets are shown to be protected even when the long-term secrets of other agents, or the short-term secrets in other protocol runs (with other spells) are compromised.

The closure results on the coideal have turned out to be a useful addition to the arsenal of proof techniques, enabling interesting examples to be shown secure. Protocol proofs are still complex enough so that we feel proof-checking and automation to be valuable for the sake of assurance, and we believe that the same techniques that simplify manual proofs will also be helpful in organizing machine-assisted proofs.

Currently, we are developing high-level PVS strategies for automatically discharging most verification conditions for typical protocol rules. In these strategies we try to capture the repetitive patterns that have been showing up in hand and mechanized interactive proofs. It is our hope that, using these strategies, we can prove secrecy results about realistic protocols in a "fairly" automatic way. Also, we have developed a translator from the CAPSL protocol specification language to a corresponding PVS protocol model. In this way, PVS is used as a backend for cryptographic protocol analysis.

References

[1] I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *12th IEEE Computer Security Foundations Workshop*, pages 55–69. IEEE Computer Society, 1999.

[2] L. Gong. Enclaves: Enabling Secure Collaboration over the Internet. *IEEE Journal of Selected Areas in Communications*, 15(3):567–575, April 1997.

[3] J. Millen and H. Rueß. Protocol-independent secrecy. In *2000 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2000.

[4] D. Otway and O. Rees. Efficient and timely mutual authentication. *ACM Operating System Review*, 21(1):8–10, 1987.

[5] S. Owre, J. Rushby, N. Shankar, and F. 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.

[6] J. Thayer, J. Herzog, and J. Guttman. Honest ideals on strand spaces. In *11th IEEE Computer Security Foundations Workshop*, pages 66–78. IEEE Computer Society, 1998.

Proving Secrecy Is Easy Enough*

Presented at CSFW 2001

Véronique Cortier
Laboratoire Spécification et Vérification
Ecole Normale Supérieure de Cachan
61, Avenue du Président Wilson, 94230 Cachan, France
cortier@lsv.ens-cachan.fr

Jon Millen and Harald Rueß
SRI International, Computer Science Laboratory
333 Ravenswood Ave, Menlo Park, CA 94035, USA
{millen,ruess}@csl.sri.com

September 25, 2002

Abstract

We develop a systematic proof procedure for establishing secrecy results for cryptographic protocols. Part of the procedure is to reduce messages to simplified constituents, and its core is a search procedure for establishing secrecy results. This procedure is sound but incomplete in that it may fail to establish secrecy for some secure protocols. However, it is amenable to mechanization, and it also has a convenient visual representation. We demonstrate the utility of our procedure with secrecy proofs for standard benchmarks such as the Yahalom protocol.

1 Introduction

Cryptographic protocols are used to achieve goals like authentication and key distribution in a possibly hostile environment. These protocols are notoriously difficult to design and test, and serious flaws have been found in many protocols. Consequently there has been a growing interest in applying formal methods for validating cryptographic protocols. In particular, standard program verification

*This work was funded by DARPA through the Air Force Research Laboratory Contract F30602-98-C-0258 and by DARPA through Rome Lab contract F30602-96-C-0291.

techniques such as model checking, theorem proving, or invariant generation have been found to be essential tools; a recent overview has been given by Meadows [9].

A popular choice is to use model checking procedures for debugging purposes by searching for attacks. These techniques, however, are not directly applicable for verification, since search spaces usually can not be explored exhaustively. In contrast, approaches based on theorem proving techniques aim at mathematical proofs of the desired protocol properties [3, 6, 15, 17]. We review the techniques that are most closely related to our work. Paulson [15] uses an interactive theorem prover to prove invariance properties by proving that they are *inductive*, i.e. they are preserved by the execution of each and every protocol rule. Domain-specific tactics are crucial for mechanizing the process of proof construction, but verifications still require considerable effort and insight into the workings of the protocol under consideration. Cohen's [3] approach is much more automatic. He constructs a first-order invariant from a protocol description, and uses first-order reasoning for establishing safety properties. Cohen's approach is amenable to both hand proofs and automation. Indeed, he applies his method to verify the large majority of the benchmark protocols in the Clark and Jacob survey [2] with only a small amount of user intervention.

In contrast to the work by Paulson and Cohen we do not consider safety properties in general, but we restrict ourselves to the specific case of proving secrecy invariants of cryptographic protocols; that is, our main interest is in proving that secrets are not accidentally revealed to unauthorized agents. Proving secrecy invariants for cryptographic protocols has often been found to be the hardest task in analyzing a protocol [15]. Indeed, secrecy has been shown to be undecidable even under very weak assumptions on the protocol [4], while specialized logics for establishing authentication are usually decidable [11].

Our proof technique is to perform inductive proofs, as advocated by Paulson [15]. To help express secrecy goals, we make use of the "spell" events introduced in [10]. However, this paper does not use the trace model as in [10] or [15], but a new state-transition model similar to the MSR model proposed by Mitchell el al [1]. We have found that the Secrecy Theorem in [10] could be adapted to work just as well in this context. The current model and our use of PVS to perform inductive proofs with this model were presented at a workshop that did not have a published proceedings [16]. We have used this approach mainly for proving secrecy of standard benchmark protocols such as the Otway-Rees and the Needham-Schroeder protocol. Dutertre et al [5] used our techniques for verifying the group management services of Enclaves [7].

The starting point for this paper is the observation that secrecy proofs based on the decomposition of the Secrecy Theorem follow a standard pattern that is amenable to mechanization, and which also has a convenient visual representa-

tion. Part of the procedure is to reduce messages to simplified constituents called *branches*. The core is a search procedure for establishing secrecy results. This procedure is sound but incomplete in that it may fail to establish secrecy for some secure protocols.

The paper is structured as follows. In Section 2 we review a state-based model for modeling cryptographic protocols, and we state a suitable security policy together with a corresponding secrecy theorem as introduced in [16]. This theorem reduces secrecy proofs to local proof obligations on protocol transitions; these obligations are called *occultness* conditions. In Section 3 we develop a characterization of the occultness notion, which is used to initialize our proof procedure. Then, in Section 4 we describe a search procedure for establishing secrecy results. Moreover, Section 4 contains a soundness result for this search procedure, and we sketch a convenient graphical representation for occultness proofs. Section 5 includes some case studies drawn from the Clark-Jacob survey [2] such as the Yahalom and the Kao Chow repeated authentification protocol. We also demonstrate a proof of non-occultness from a failed proof attempt. Section 6 contains some concluding remarks.

2 Background

We give an overview of state-based encodings of protocols, a security policy based on the notion of coideals, and a secrecy theorem for generating local verification conditions. More detailed descriptions can be found in [10, 16].

Message Fields. The set $Fields$ of message fields is made up of primitive and compound fields. The primitive fields are those of types $Agent$, Key, and $Nonce$. Keys and nonces form the set $Basic$; these basic fields are the only types of fields that may be designated as secret, as a protocol policy goal. Compound fields are constructed by concatenation $[X, Y]$ (often written without brackets) or encryption $\{X\}_K$.

As a notational convention, variables A, B and variants always stand for agents; K and variants always stand for keys; and N and variants are always nonces. The reserved subscript "s" identifies a set, so N_s is a set of nonces.

Each agent A has some long-term keys: a public key $\mathsf{pub}(A)$, a corresponding private key $\mathsf{prv}(A)$, and a symmetric key $\mathsf{shr}(A)$, which is shared between A and a designated server agent Srv. Each key K has an inverse key K^{-1}; in particular, $\mathsf{pub}(A)^{-1} = \mathsf{prv}(A)$, $\mathsf{prv}(A)^{-1} = \mathsf{pub}(A)$, while $\mathsf{shr}(A)^{-1} = \mathsf{shr}(A)$, as is the case with any symmetric keys. Keys generated during a protocol session are always symmetric keys.

Events and Global States. There are three kinds of events: *message*, *spell*, and *state* events. A message event is simply a field representing the content of the message. A spell event $C = S \ddagger L \in Spells$, generates the *book*, or session-specific set of basic secrets $Book(C) = S$, which are shared among the set $Cabal(C) = L$ of agents, the *cabal*.

A state event is of the form $Q = \mathsf{A}_n(X) \in States$ where A is a role name, n is a natural number that represents the step of the protocol, and $X = Mem(Q)$ is a concatenated field that represents the memory held by the state. We also write $Mem(H) = \{Mem(Q) | Q \in H \cap States\}$ for any event set H. As a notational convention, we use Q (and variants) to denote state events, while M is a message event, and C is a spell event. The set of *basic secrets* of a spell consists of its book plus the long-term keys of its cabal:

$$Sec(C) = Book(C) \cup ltk(Cabal(C))$$

The long-term keys are those generated by $\mathsf{pub}(.)$, $\mathsf{prv}(.)$, and $\mathsf{shr}(.)$.

A *global state* is a set (not a multiset) of events. Notationally, variants of H are global states or event sets. The *content* of a global state is its set of messages, written:

$$Cont(H) \stackrel{\text{def}}{=} H \cap Fields$$

Similarly, the secrets of a global state are obtained from its spell events.

$$Sec(H) \stackrel{\text{def}}{=} \bigcup \{Sec(C) | C \in H\}$$

A basic field X is *unused* in H if it is neither a part of a field in the content nor a secret of H.

$$unused(H) \stackrel{\text{def}}{=} \{X \in Basic \mid X \notin \mathsf{parts}(Cont(H)), X \notin Sec(H)\}$$

Inductive Relations. $\mathsf{parts}(S)$ is the set of all subfields of fields in the set of fields S, including components of concatenations and the plaintext of encryptions (but not the keys). $\mathsf{analz}(S)$ is the subset of $\mathsf{parts}(S)$ consisting of only those subfields that are accessible to an attacker. These include components of concatenations, and the plaintext of those encryptions where the inverse key is in $\mathsf{analz}(S)$. More precisely, $\mathsf{analz}(S)$ is the smallest superset of S such that $X \in \mathsf{analz}(S)$ and $Y \in \mathsf{analz}(S)$ if $[X, Y] \in \mathsf{analz}(S)$, and $X \in \mathsf{analz}(S)$ if $\{X\}_K \in \mathsf{analz}(S)$ and $K^{-1} \in \mathsf{analz}(S)$. Finally, $\mathsf{synth}(S)$ is the set of fields constructible from S by concatenation and encryption using fields and keys in S. It is defined to be the smallest superset of S such that $[X, Y] \in \mathsf{synth}(S)$ if $X \in \mathsf{synth}(S)$ and $Y \in \mathsf{synth}(S)$,

$$\emptyset \quad \xrightarrow{\{N_a, N_b, K\}} \left\{ \begin{array}{l} \{N_a, N_b, K\} \ddagger \{A, B\}, \\ \mathsf{A}_1(A, B, Srv), \\ \mathsf{B}_1(B, Srv), \\ \mathsf{Srv}_1(Srv) \end{array} \right\} \tag{0}$$

$$\left\{ \begin{array}{l} \{N_a, \_\} \ddagger \{A, B\} \\ \mathsf{A}_1(A, B, Srv) \end{array} \right\} \xrightarrow{\{N\}} \left\{ \begin{array}{l} \mathsf{A}_2(A, B, Srv, N_a), \\ [N, A, \{N_a, N, B\}_{\mathsf{shr}(A)}] \end{array} \right\} \tag{1}$$

$$\left\{ \begin{array}{l} [N, A, X], \\ \mathsf{B}_1(B, Srv), \\ \{N_b, \_\} \ddagger \{A, B\} \end{array} \right\} \xrightarrow{\emptyset} \left\{ \begin{array}{l} \mathsf{B}_2(B, Srv, A, N_b), \\ [N, B, A, X, \{N_b, N, A\}_{\mathsf{shr}(B)}] \end{array} \right\} \tag{2}$$

$$\left\{ \begin{array}{l} M, \\ \mathsf{Srv}_1(Srv), \\ \{K, \_\} \ddagger \{A, B\} \end{array} \right\} \xrightarrow{\emptyset} \left\{ \begin{array}{l} \mathsf{Srv}_2(Srv), \\ [N, \{N_a, K\}_{\mathsf{shr}(A)}, \{N_b, K\}_{\mathsf{shr}(B)}] \end{array} \right\} \tag{3}$$

where $M \stackrel{\text{def}}{=} [N, B, A, \{N_a, N, B\}_{\mathsf{shr}(A)}, \{N_b, N, A\}_{\mathsf{shr}(B)}]$

Figure 1: Encoding of part of Otway-Rees protocol.

and $\{X\}_K \in \mathsf{synth}(S)$ if $X \in \mathsf{synth}(S)$ and $K \in \mathsf{synth}(S)$. The intruder in our model synthesizes faked messages from analyzable parts of a set of available fields. This motivates the definition $\mathsf{fake}(S) \stackrel{\text{def}}{=} \mathsf{synth}(\mathsf{analz}(S))$.

Ideals and Coideals. An *ideal* $\mathcal{I}(S)$ denotes the set of fields that have to be protected in order not to reveal any secrets in S [18]. It is defined as the smallest superset of S such that $[X, Y] \in \mathcal{I}(S)$ if $X \in \mathcal{I}(S)$ or $Y \in \mathcal{I}(S)$, and $\{X\}_K \in \mathcal{I}(S)$ if $X \in \mathcal{I}(S)$ and $K^{-1} \notin \mathcal{I}(S)$. The complement of an ideal, the coideal, is denoted by $\mathcal{C}(S)$. This defines the set of fields that are public with respect to the basic secrets S, i.e., fields whose release would not compromise any secrets in S. Coideals are interesting because they are closed under attacker analysis; i.e. $\mathsf{fake}(\mathcal{C}(S)) = \mathcal{C}(S)$ for all primitive fields S.

Protocols. A protocol transition t is of the form $Pre(t) \xrightarrow{New(t)} Post(t)$, where $Pre(t)$ and $Post(t)$ are set of events and $New(t)$ is a set of nonces. Such transitions specify a possible global state change in a way to be explained below.

Except for an initialization transition, a transition t shows a state change for one role. It may also produce, in the post, a message or a spell but not both.

A primitive field occurring in post messages or state memory must occur in the messages or state memory of the pre or among the nonces. This condition is called *regularity*, and it implies that no long-term keys are deliberately introduced into a post message. There is also a restriction that secrets in a post spell are all in $New(t)$. (The freshness of nonces in $New(t)$ resides in reachability.)

A *protocol* is simply a set of protocol transitions. A protocol specification is a set of rules, where each rule is a schema defining a set of transitions using terms with free variables. More formally, a transition t is an *instance* of a rule rl iff there exists a ground substitution σ, defined on the variables of rl, such that $t = \sigma(rl)$.

Protocol rules for the first three messages of the familiar Otway-Rees [13] (OR) and Needham-Schroeder-Lowe [8, 12] (NSL) public key protocols can be found in Figures 1 and 2, respectively. Often, as in the Needham-Schroeder specification, we omit the state events for brevity when they are not needed for our purposes.

In both protocols, each session is initiated with a spell to introduce the session-specific secrets and a corresponding cabal. In addition, the Otway-Rees protocol introduces a non-secret nonce N in rule 1.

Global State Transitions. Given a protocol P and a set of initial knowledge I (of the spy), the *global succession* relation transforms a state H to a new state H'. A succession is either *honest*, i.e. it corresponds to an action by an agent following the protocol, or it is *faked* by the spy.

- H' is an *honest* successor of H, denoted by $honest(P)(H, H')$, if there exists an applicable transition t in P such that $H' = (H \backslash (Pre(t) \cap States)) \cup Post(t)$.
- H' is a *fake* successor of H, denoted by $fake(I)(H, H')$, if there exists a field $X \in \mathsf{fake}(Cont(H) \cup I)$ such that $H' = H \cup \{X\}$.

In the honest case, a transition t is *applicable* in H if $Pre(t) \subseteq H$ and $New(t) \subseteq unused(H)$. In the fake case, the spy is restricted to adding only messages that can be inferred from the content of the current state and the initial knowledge. In either case, we write $global(P)(I)(H, H')$. This relation determines a logical transition system with the empty set of events as its initial state. The set of reachable states of this transition system is denoted by $reachable(P, I)$.

Because protocol spell books introduce only unused secrets, it is easy to show that the spell books of different spells are disjoint.

Lemma 1 (Disjoint Book) *If $C, C' \in H \in reachable(P, I)$ then either $C = C'$ or $Book(C)$ and $Book(C')$ are disjoint.*

Secrecy Policy. A spell is *compatible* with an initial knowledge set I that does not mention its associated basic secrets.

$$compatible(I) \stackrel{\text{def}}{=} \{C \mid Sec(C) \cap \mathsf{parts}(I) = \emptyset\}$$

Given the spy's initial knowledge I, a global state H is called *I-discreet* if $Cont(H) \subseteq \mathcal{C}(Sec(C))$ for all I-compatible spells $C \in H$; these states are collected in the set $discreet(I)$. Now, a protocol P is called *discreet* if $discreet(I)$ is an invariant of the transition relation associated with P; i.e. for all I, $reachable(P, I)$ is a subset of $discreet(I)$.

Secrecy Theorem. As in [10], the Secrecy Theorem serves to split the secrecy proof for a protocol into a protocol-independent part and a protocol-dependent part. The protocol-dependent part is expressed by the occultness property. It says that if the prior state is discreet, the next message event generated by the protocol does not compromise a secret.

Some more notation needs to be introduced before defining occultness. A *P-configuration* is a tuple (I, H, C) such that $H \in reachable(P, I)$, $H \in discreet(I)$, $C \in compatible(I)$, and $C \in H$. Now, a protocol P is said to be *occult* if for all *P-configurations* (I, H, C) and for each applicable transition t in P,

$$Cont(Post(t)) \subseteq \mathcal{C}(Sec(C)).$$

The protocol-independent part of a secrecy proof is the Secrecy Theorem.

Theorem 1 (Secrecy Theorem) *A protocol P is discreet iff it is occult.*

This theorem reduces secrecy proofs to proving occultness of individual rules of the protocol. In the case of the Otway-Rees protocol in Figure 1, for example, we are reduced to showing occultness of the rules (1),(2),(3), since occultness holds trivially for rule (0). (The fourth rule is also easy to handle.) For rule 1 of the Otway-Rees protocol we have to prove that for all reachable and I-discreet global states H, and for all I-compatible spells $C \in H$ it follows from the applicability conditions

- $\{N_a, \_\} \ddagger \{A, B\} \in H$,
- $\mathsf{A}_1(A, B, Srv) \in H$, and
- $N \in unused(H)$

$$\emptyset \xrightarrow{\{N_a, N_b\}} \{\{N_a, N_b\} \ddagger \{A, B\}\} \quad (0)$$

$$\{\{N_a, \_\} \ddagger \{A, B\}\} \xrightarrow{\emptyset} \{[\{N_a, A\}_{\mathsf{pub}(B)}]\} \quad (1)$$

$$\left\{ \begin{array}{l} \{N_b, \_\} \ddagger \{A, B\}, \\ [\{N_a, A\}_{\mathsf{pub}(B)}] \end{array} \right\} \xrightarrow{\emptyset} \{[\{N_a, N_b, B\}_{\mathsf{pub}(A)}]\} \quad (2)$$

$$\{[\{N_a, N_b, B\}_{\mathsf{pub}(A)}]\} \xrightarrow{\emptyset} \{[\{N_b\}_{\mathsf{pub}(B)}]\} \quad (3)$$

Figure 2: Encoding of Needham-Schroeder-Lowe protocol.

that $[N, A, \{N_a, N, B\}_{\mathsf{shr}(A)}] \in \mathcal{C}(Sec(C))$. To establish this, we have to check two cases, depending on whether C is the spell in the rule or not. If it is, we note that $\mathsf{shr}(A)$ is in the coideal; in the other case, there is no secret to protect, because the Disjoint Book Lemma implies that N_a is not in $Book(C)$. This case split argument is one of the tasks that are simplified away using the search procedure we will present.

It is undecidable whether or not a given protocol P is occult. Undecidability of protocol security is well known, and has been proved in several different models. See, for example, [4] and its references. A proof for this particular model works by a simple encoding of the reachability problem of Turing machines such that the encoded Turing machine reaches its final state iff the protocol is not occult.

Let T be a Turing Machine, Q the set of states, (q_0 is the initial state and q_f is the final state), Σ the Tape Alphabet, ($\sharp$ is the blank symbol), its transitions are on the form $q_1\, a_1 \rightarrow q_2\, a_2, N$, where $q_1, q_2 \in Q$, $a_1, a_2 \in \Sigma$ and $\mathbb{N} \in \{L, R, S\}$. The interpretation is : if the machine T is in state q_1 and its head points a_1 then T changes to state q_2, replaces a_1 with a_2 and moves the head right (if $N = R$), left (if $N = L$) or stays at the current cell (if $N = S$).

We do a copy Σ' of the Tape Alphabet : a primed letter represents the letter pointed by the head of the Turing machine. We associate a number $n_a \in \mathbb{N}$ to each $a \in \Sigma$, $a' \in \Sigma'$, a number $n_q \in \mathbb{N}$ to each $q \in Q$.

We encode the letters $a \in \Sigma \cup \Sigma'$ and the states $q \in Q$ as following :

$$\overline{a} = \underbrace{N, \cdots, N}_{n_a \text{ times}}, A, \qquad \overline{q} = \underbrace{N, \cdots, N}_{n_q \text{ times}}, A$$

(Actually, the letters and the states are encoded by a specific length of nonces and are separated by the name of an agent A)

We encode the transitions $q_1\, a_1 \to q_2\, a_2, R$ by the rules:

$$\left\{[\{K, \overline{q_1}, X, \overline{a_1'}, \overline{b}, Y\}_{\mathsf{shr}(A)}]\right\} \xrightarrow{\emptyset} \left\{[\{K, \overline{q_2}, X, \overline{a_2}, \overline{b'}, Y\}_{\mathsf{shr}(A)}]\right\}$$
$$\text{for all } b \in \Sigma$$
$$\left\{[\{K, \overline{q_1}, X, \overline{a_1'}\}_{\mathsf{shr}(A)}]\right\} \xrightarrow{\emptyset} \left\{[\{K, \overline{q_2}, X, \overline{a_2}, \overline{\sharp'}\}_{\mathsf{shr}(A)}]\right\}$$

In addition, we have to consider the same rules where respectively X, X and Y, and Y are omitted.

$\mathsf{shr}(A)$ is a private key shared between the server and A but it could be any shared key between 2 agents.

The initialization rule is

$$\emptyset \quad \xrightarrow{\{K,N\}} \quad \left\{K \ddagger A, [\{K, \overline{q_0}, \overline{\sharp'}\}_{\mathsf{shr}(A)}]\right\}$$

and the "final" rule is

$$\left\{[\{K, \overline{q_f}, X\}_{\mathsf{shr}(A)}]\right\} \quad \xrightarrow{\emptyset} \quad \{[K]\}$$

The final state of the Turing machine is reachable iff this protocol is not occult.

Using theorem 1 it follows that it is also undecidable whether or not a given protocol is discreet.

3 Branches

Occultness proofs work by contradiction. In proving occultness of rule 1 of the Otway-Rees protocol in Figure 1, for example, one tries to obtain a contradiction from the assumption $[N, A, \{N_a, N, B\}_{\mathsf{shr}(A)}] \in \mathcal{I}(Sec(C))$. Using the definition of ideals we are reduced to show that each of the cases $N \in \mathcal{I}(Sec(C))$, $A \in \mathcal{I}(Sec(C))$, and $\{N_a, N, B\}_{\mathsf{shr}(A)} \in \mathcal{I}(Sec(C))$ yields a contradiction. The second case yields an immediate contradiction, since agents names are not elements of ideals. Furthermore, using the definition of ideals, the third case can be simplified further to the disjunction $\mathsf{shr}(A) \notin \mathcal{I}(Sec(C))$ or $[N_a, N, B] \in \mathcal{I}(Sec(C))$. In general, a field M is in the coideal generated by $Sec(C)$ iff for each nonce or key B in $\mathsf{parts}(M)$, either B is not in $Sec(C)$ or B is encrypted with at least one key in $Sec(C)$.

This observation suggests that, instead of examining M itself, we examine the basic secrets occurring in it and the keys protecting them. A *branch* is a pair consisting of a basic field and a set of keys. The following recursion computes the branches occurring in a field M.

Definition 1 $\mathsf{bnch}(M)$ *is defined as* $\mathsf{bnch}(M, \emptyset)$*, where*

$$\begin{aligned}
\mathsf{bnch}(N, K_s) &= \{(N, K_s)\} \\
\mathsf{bnch}(K, K_s) &= \{(K, K_s)\} \\
\mathsf{bnch}(A, K_s) &= \emptyset \\
\mathsf{bnch}([M_1, M_2], K_s) &= \mathsf{bnch}(M_1, K_s) \cup \\
&\qquad \mathsf{bnch}(M_2, K_s) \\
\mathsf{bnch}(\{M\}_K, K_s) &= \mathsf{bnch}(M, K_s \cup \{K\})
\end{aligned}$$

Thus,

$$\mathsf{bnch}([N, A, \{N_a, N, B\}_{\mathsf{shr}(A)}]) = \{(N, \emptyset),\ (N_a, \{\mathsf{shr}(A)\}),\ (N, \{\mathsf{shr}(A)\})\}.$$

It turns out that field M is in $\mathcal{C}(S)$ if and only if its branches satisfy a simple condition. The proof of is by induction on the operator depth of M.

Proposition 1 *Let S be a set of basic fields; then:*

$M \in \mathcal{C}(S)$ iff for all $(Y, K_s) \in \mathsf{bnch}(M)$:
$Y \in S \Rightarrow K_s^{-1} \cap S \neq \emptyset$.

Definition 2 *For a protocol P, a branch $b = (Y, K_s)$, E_s a set of events, and N_s a set of nonces, the predicate $occ(P, b)(E_s, N_s)$ is defined to hold iff for all P-configurations (I, H, C) such that*

1. *$E_s \subseteq H$,*
2. *$N_s \subseteq unused(H)$, and*
3. *$Y \in Sec(C)$*

it is the case that $K_s^{-1} \cap Sec(C) \neq \emptyset$. For a transition t we write $occ(P, b)(t)$ instead of $occ(P, b)(Pre(t), New(t))$.

The following characterization of protocol occultness is a straightforward consequence of Proposition 1.

Proposition 2 *A protocol P is occult iff*

1. *for all transitions $t \in P$,*
2. *for all message fields M such that $[M] \in Post(t)$, and*
3. *for all branches $b \in \mathsf{bnch}(M)$*

the predicate $occ(P, b)(Pre(t), New(t))$ holds.

4 A Search Procedure for Establishing Occultness

Now, we describe a search procedure for establishing $occ(P, b)(t)$ for a given branch b and a transition t. This algorithm proceeds by applying some *basic tests* which are sufficient for establishing that the occultness predicate above holds. Whenever these tests fail, a *back step* is performed. Such a step explores every possibility of how certain message fields could have been published on the network.

Lemma 2 (Basic Tests) *Let $b = (Z, K_s)$ be a branch, E_s a set of events, and N_s a set of nonces; then: $occ(P, b)(E_s, N_s)$ holds if one of the following is true.*

1. $Z \in N_s$
2. *There exists a K'_s such that $K'_s \subseteq K_s$ and $(Z, K'_s) \in \mathsf{bnch}(Cont(E_s))$; in this case we write $(Z, K_s) \,\tilde{\in}\, \mathsf{bnch}(Cont(E_s))$.*
3. *There exists a spell $C \in E_s$ such that $Z \in Book(C)$ and $K_s^{-1} \cap Sec(C) \neq \emptyset$; in this case we write $db(E_s, Z, K_s)$.*

Note that we employ the obvious extension of $\mathsf{bnch}(.)$ to sets of fields. The operator $\tilde{\in}$ says that a branch may have more keys than necessary, which is not harmful, since one good key is enough.

Given a P-configuration (I, H, C) such that the requirements listed in Definition 2 hold, Lemma 2 is proved as follows. First, consider the basic test $Z \in N_s$. Since $N_s \subseteq unused(H)$, it follows that $Z \notin Sec(C)$. Thus, $occ(P, b)(E_s, N_s)$ holds. Second, assume $(Z, K_s) \,\tilde{\in}\, \mathsf{bnch}(Cont(E_s))$ and let $K'_s \subseteq K_s$ be such that $(Z, K'_s) \in \mathsf{bnch}(Cont(E_s))$. Since $E_s \subseteq H$ and H is I-*discreet*, it follows that $Cont(E_s) \subseteq \mathcal{C}(Sec(C))$. Consequently, using Lemma 1, $K'^{-1}_s \cap Sec(C) \neq \emptyset$, and thus $K_s{}^{-1} \cap Sec(C) \neq \emptyset$. The third part of Lemma 2 is a consequence of the disjoint book lemma (Lemma 1).

Consider, for example, rule 2 of the Otway-Rees protocol in Figure 1. This rule, denoted by $or2$, contains a message variable X in its pre. Thus, $or2$ denotes an infinite set of transitions, and a uniform proof of the occultness of this family of transitions starts by introducing a symbolic constant X'.

$$(N, \emptyset), (N, \{\mathsf{shr}(B)\}) \,\tilde{\in}\, \mathsf{bnch}(Cont(\{[N, A, X'],\ \{N_b, \_\} \ddagger \{A, B\}\}))\ ,$$

it follows that both $occ(OR, (N, \emptyset))(or2)$ and $occ(OR, (N, \{\mathsf{shr}(B)\}))(or2)$ hold. Furthermore, since the predicate $db(\{[N, A, X'],\ \{N_b, \_\} \ddagger \{A, B\}\}, N_b, \{\mathsf{shr}(B)\})$ holds, it follows that $occ(OR, (N_b, \{\mathsf{shr}(B)\}))(or2)$ holds, too.

The occultness proof of the Otway-Rees protocol uses only basic tests. In general, however, other rules have to be taken into consideration. Consider, for example, the case $(N_a, \{\mathsf{pub}(A)\}) \in \mathsf{bnch}(\{N_a, N_b, B\}_{\mathsf{pub}(A)})$ for proving rule 2 of the Needham-Schroeder-Lowe protocol in Figure 2; this rule is denoted by $nsl2$. None of the basic tests above establishes that $occ(NSL, (N_a, \{\mathsf{pub}(A)\}))(nsl2)$ holds. The purpose of a *back step* is to obtain additional information for applying the basic tests. For each message event M in E_s, two possibilities have to be taken into consideration: either M has been published by an honest agent following the protocol rules or M was injected by the intruder.

Definition 3 (Search) *Let P be a protocol, t be a transition of P, and b be a branch of the form (Z, K_s); then:*

$$
\begin{aligned}
main(P,b)(t) &\stackrel{def}{=} Z \in New(t) \vee \\
&\qquad search(P,b)(Pre(t)) \\
search(P,b)(E_s) &\stackrel{def}{=} b \,\widetilde{\in}\, \mathsf{bnch}(Cont(E_s)) \ \vee \\
&\qquad db(E_s, Z, K_s) \ \vee \ back(P,b)(E_s) \\
back(P,b)(E_s) &\stackrel{def}{=} (\exists M \in E_s : \ Z \in \mathsf{parts}(M)) \\
&\qquad (honest(P,b)(M) \wedge fake(P,b)(M)) \\
honest(P,b)(M) &\stackrel{def}{=} (\forall t' \in P, M \in \mathsf{parts}(Cont(Post(t')))) \\
&\qquad search(P,b)(Pre(t')) \\
fake(P,b)(M) &\stackrel{def}{=} (\forall M_1, \ldots, M_n : [M_1, \ldots, M_n] = M) \\
&\qquad \vee_{M_i} (Z \in \mathsf{parts}(M_i) \wedge search(P,b)(\{M_i\}))
\end{aligned}
$$

These predicates determine a search procedure in the usual way. For example, $P_1 \vee P_2$ is computed non-deterministically: if the computation of P_1 (or P_2) terminates with $true$, then the computation of $P_1 \vee P_2$ terminates with $true$. Using these conventions, Defi nition 3 gives rise to a nondeterministic proof procedure for establishing occultness.

Now we outline the proof of soundness for our procedure. The proof of the main lemma applies induction on the number of back steps in deducing that predicate $search(P, Z, K_s)(E_s)$ holds. A detailed proof can be found in the appendix.

Lemma 3 (Main Lemma) *Let P be a protocol, b be some branch, and E_s a set of of events; then:*

If the predicate $search(P,b)(E_s)$ holds, then $occ(P,b)(E_s,\_)$ holds, too.

Altogether, soundness of the search procedure follows from the Lemmas 2 and 3, and the secrecy theorem (Theorem 1).

Theorem 2 (Soundness) *Let P be a protocol. If $main(P,b)(Pre(t),New(t))$ holds*

- *for all transitions $t \in P$,*
- *for all message events $M \in Post(t)$, and*
- *for all branches $b \in \mathsf{bnch}(M)$,*

then P is discreet.

Our method, however, is not complete. If one of the proof obligations can not be shown to hold, then one may not necessarily conclude that the protocol is not discreet. Moreover, there are occult protocols for which our search procedure does not terminate; such an example can be found in Section 5.

Let us return to proving occultness of the rule $nsl2$; for the branch $b = (N_a, \{\mathsf{pub}(A)\})$ the derivation starts as follows.

$$\begin{aligned}
& main(NSL,b)(nsl2) \\
\Longleftrightarrow\quad & search(NSL,b)(\{\{N_b|_\} \ddagger \{A,B\},) \\
& \qquad\qquad [\{N_a,A\}_{\mathsf{pub}(B)}]\} \\
\Longleftrightarrow\quad & back(NSL,b)(\{\{N_b|_\} \ddagger \{A,B\},) \\
& \qquad\qquad [\{N_a,A\}_{\mathsf{pub}(B)}]\} \\
\Longleftrightarrow\quad & honest(NSL,b)(\{N_a,A\}_{\mathsf{pub}(B)}) \ \wedge \\
& \qquad fake(NSL,b)(\{N_a,A\}_{\mathsf{pub}(B)})
\end{aligned}$$

Since only the first rule of the Needham-Schroeder-Lowe protocol contains a message of the form $\{N_a,A\}_{\mathsf{pub}(B)}$ in its *post*,

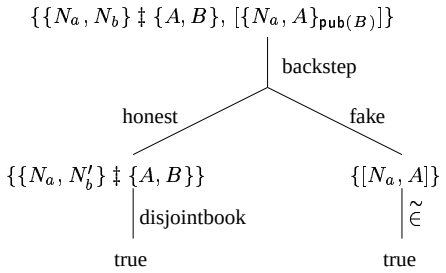
$$\begin{aligned}
& honest(NSL,b)(\{N_a,A\}_{\mathsf{pub}(B)}) \\
\Longleftrightarrow\quad & search(NSL,b)(\{\{N_a|_\} \ddagger \{A,B\}\}) \\
\Longleftrightarrow\quad & true
\end{aligned}$$

This reduces to $true$ because of the disjoint book test.

$$\begin{aligned}
& fake(NSL,b)(\{N_a,A\}_{\mathsf{pub}(B)}) \\
\Longleftrightarrow\quad & search(NSL,b)(N_a,A) \\
\Longleftrightarrow\quad & true
\end{aligned}$$

since $(N_a, \{\mathsf{pub}(A)\}) \stackrel{\sim}{\in} \mathsf{bnch}([N_a, A])$. Consequently, rule $nsl2$ is occult.

Derivations based on the predicates in Definition 3 can be visualized as search trees. These search trees have set of events as nodes, the edges are labeled either with a basic test or with the name of one of the search steps. A leaf is $true$ if one of the basic tests succeeds, and $false$ if all the basic tests fail and if there is no more message in the set of events of the parent node. Branching corresponds to a conjunction, and disjunctions are realized by copying derivation trees. For the rule $nsl2$ and the branch $(N_a, \{\mathsf{pub}(A)\})$, for example, the run of $search$ is visualized as follows.



In general, the search tree generated by the predicates in Definition 3 may be infinitely branching whenever there is an infinite set of protocol transitions. However, the set of honest transitions is usually generated by a finite set of rules on the form $rl = Pre(rl) \longrightarrow Post(rl)$, such that each transition t of the protocol is obtained by a substitution σ, i.e. $Pre(t) = Pre(rl)\sigma$, $New(t) = New(rl)\sigma$, etc ... The remainder of this section is devoted to lifting the results above from transitions to rules. In this way, we obtain occultness proof obligations for rules which possibly contain variables.

The notion of branches has to be extended to include messages fields containing variables X by adding the case $\mathsf{bnch}(X, K_s) = \{(X, K_s)\}$ to Definition 1. Now, the search algorithm in Definition 3 is lifted to this new case of field variables in branches.

Definition 4 *Let P be a protocol, b be a branch, and rl be a rule; then:*

$$main'(P,b)(rl) \stackrel{def}{=} \begin{cases} b \stackrel{\sim}{\in} \mathsf{bnch}(Cont(Pre(rl))) \\ \quad \textit{if } b = (Z, \_) \textit{ and } Z \textit{ is a variable;} \\ main(P,b)(rl) \\ \quad \textit{otherwise.} \end{cases}$$

The soundness of this extension follows from the following fact.

Lemma 4 *If $main'(P,b)(Pre(rl), New(rl))$ holds for all $M \in Post(rl)$, for all $b \in \mathsf{bnch}(M)$, then*

- *for all instances t of rule rl,*
- *for all message events $M' \in Post(t)$,*
- *for all branches $b \in \mathsf{bnch}(M')$*

the predicate $main(P, b)(Pre(t), New(t))$ holds.

Let $b \in \mathsf{bnch}(M')$ such that $M' \in Post(t)$. If $b \in \mathsf{bnch}(M)$, then the predicate $main(P, b)(Pre(t), New(t))$ holds by the definition of $main'(P, b)(Pre(rl), New(rl))$. Otherwise, if $b = (Z, K_s)$ comes from an instantiation σ of a field variable, there exists $X \in \mathsf{parts}(M)$ such that $(X, K_1) \in \mathsf{bnch}(M)$ and $(Z, K_2) \in \mathsf{bnch}(X\sigma)$ with $K_s = K_1 \cup K_2$. Now, $main'(P, (X, K_1))(Pre(rl), New(rl))$ holds, and consequently $(X, K_1) \stackrel{\sim}{\in} \mathsf{bnch}(Cont(Pre(rl)))$, $b \stackrel{\sim}{\in} \mathsf{bnch}(Cont(Pre(rl\sigma)))$, and finally $main(P, b)(Pre(t), New(t))$ hold. This finishes the proof of Lemma 4.

Theorem 3 *Let P be a protocol. If $main'(P, b)(Pre(rl), New(rl))$ holds*

- *for all rules $rl \in P$,*
- *for all message events $M \in Post(rl)$, and*
- *for all branches $b \in \mathsf{bnch}(M)$,*

then P is discreet.

$$\emptyset \xrightarrow{\{N_b, K_{ab}\}} \{\{N_b, K_{ab}\} \ddagger \{A, B\}\} \tag{0}$$

$$\emptyset \xrightarrow{N_a} \{[A, N_a]\} \tag{1}$$

$$\left\{\begin{matrix}\{N_b, K_{ab}\} \ddagger \{A, B\}, \\ [A, N_a]\end{matrix}\right\} \xrightarrow{\emptyset} \{[B, \{A, N_a, N_b\}_{\mathsf{shr}(B)}]\} \tag{2}$$

$$\left\{\begin{matrix}\{N_b, K_{ab}\} \ddagger \{A, B\}, \\ [B, \{A, N_a, N_b\}_{\mathsf{shr}(B)}]\end{matrix}\right\} \xrightarrow{\emptyset} \{[\{B, K_{ab}, N_a, N_b\}_{\mathsf{shr}(A)}, \{A, K_{ab}\}_{\mathsf{shr}(B)}]\} \tag{3}$$

$$\{[\{B, K_{ab}, N_a, N_b\}_{\mathsf{shr}(A)}, X\}] \xrightarrow{\emptyset} \{[X, \{N_b\}_{K_{ab}}]\} \tag{4}$$

Figure 3: Encoding of the Yahalom protocol.

$$\emptyset \xrightarrow{\{K_{ab}\}} \{\{K_{ab}\} \ddagger \{A, B\}\} \quad (0)$$

$$\emptyset \xrightarrow{N_a} \{[A, B, N_a]\} \quad (1)$$

$$\left\{\begin{matrix}\{K_{ab}\} \ddagger \{A, B\}, \\ [A, B, N_a]\end{matrix}\right\} \xrightarrow{\emptyset} \begin{matrix}\{[\{A, B, N_a, K_{ab}\}_{\mathsf{shr}(A)}, \\ \{A, B, N_a, K_{ab}\}_{\mathsf{shr}(B)}]\}\end{matrix} \quad (2)$$

$$\{[X, \{A, B, N_a, K_{ab}\}_{\mathsf{shr}(B)}]\} \xrightarrow{N_b} \{[X, \{N_a\}_{K_{ab}}, N_b]\} \quad (3)$$

$$\begin{matrix}\{[\{A, B, N_a, K_{ab}\}_{\mathsf{shr}(A)}, \\ \{N_a\}_{K_{ab}}, N_b]]\}\end{matrix} \xrightarrow{\emptyset} \{[\{N_b\}_{K_{ab}}]\} \quad (4)$$

Figure 4: Encoding of the Kao Chow Repeated Authentifi cation protocol.

$$\emptyset \xrightarrow{\{N_a\}} \{\{N_a\} \ddagger \{A_1, A_2\}\} \quad (0)$$

$$\{\{N_a\} \ddagger \{A_1, A_2\}\} \xrightarrow{\emptyset} \begin{matrix}\{[A_1, A_2, \cdots, A_n, \\ \{A_1, N_a\}_{\mathsf{shr}(A)}]\}\end{matrix} \quad (1)$$

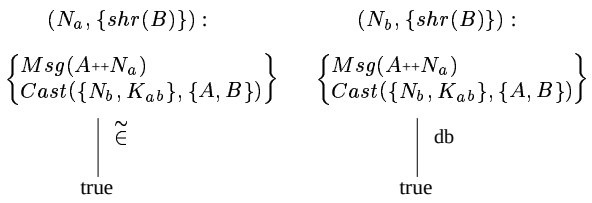
$$\begin{matrix}\{[A_1, A_2, \cdots, A_n, \\ \{A_1, N_a\}_{\mathsf{shr}(A)}]\}\end{matrix} \xrightarrow{\emptyset} \{[\{A_1, A_2, N_a\}_{\mathsf{shr}(A_1)}]\} \quad (2)$$

Figure 5: A protocol which requires at least n back steps for proving occultness.

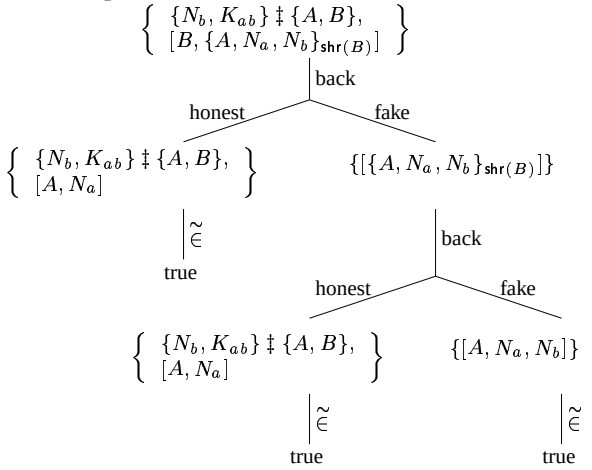
5 Examples

In the previous sections, we have already demonstrated that the Otway-Rees protocol can be proved to be occult using only basic tests. Likewise, the occultness proof of the Needham-Schroeder-Lowe protocol requires at most one back step for each rule and each branch. Here we give an overview of the proof of Yahalom's protocol, which requires up to two back steps for proving occultness. Moreover, we demonstrate the incompleteness of our algorithm with an example of an occult protocol for which the search procedure is non-terminating. Then, we give an example of a protocol that requires at least n back steps in proving occultness. Finally, we use a failed proof attempt of the original Needham-Schroeder protocol to show that it is indeed not occult.

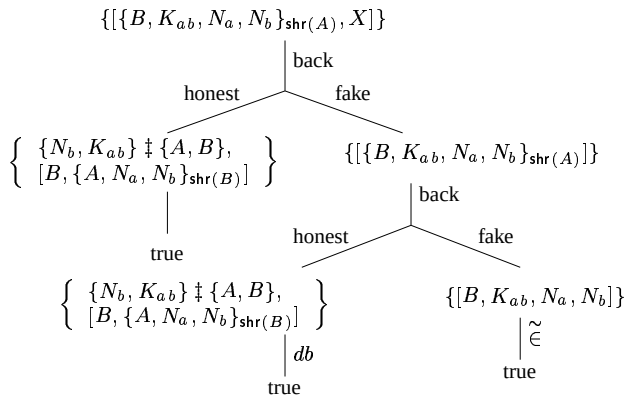
Yahalom Protocol. This protocol has been studied extensively by Paulson [14]. An encoding of the Yahalom protocol (without state events) can be found in Figure 3. Occultness of the initial rule (1) is obvious. For verifying occultness of rule (2) we have to consider the two branches $(N_a, \{\mathsf{shr}(B)\})$, $(N_b, \{\mathsf{shr}(B)\})$ of the single message in the post.



In verifying occultness of rule (3) four branches have to be considered. Occultness for the cases $(N_b, \{\mathsf{shr}(A)\})$, $(K_{ab}, \{\mathsf{shr}(A)\})$, and $(K_{ab}, \{\mathsf{shr}(B)\})$ is established using the disjoint book test, whereas the branch $(N_a, \{\mathsf{shr}(A)\})$ needs two back steps.

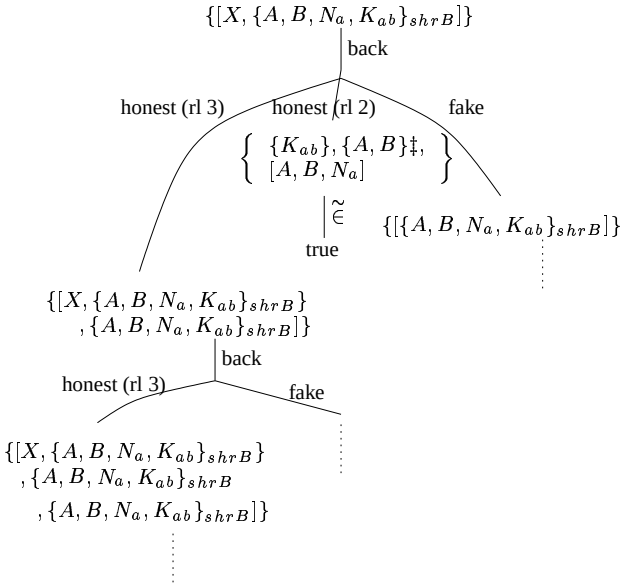


Finally, the branches $(X, \emptyset)$ and $(N_b, \{K_{ab}\})$ have to be considered for establishing occultness of the rule (4). The proof for the $(X, \emptyset)$ branch only needs the basic test "$\tilde{\in}\ \mathsf{bnch}(\ldots)$" and the following proof for the branch $(N_b, \{K_{ab}\})$ requires two back steps.

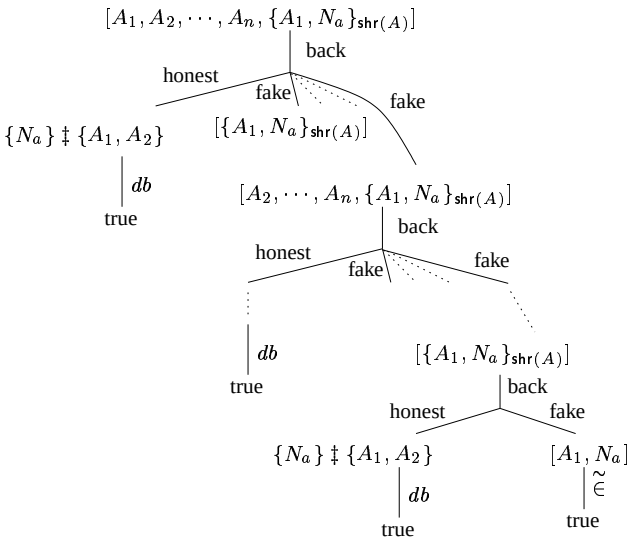


Altogether, the Yahalom protocol is occult.

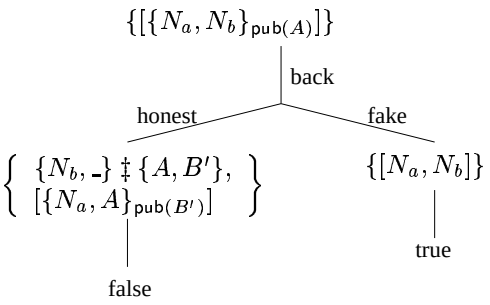
The Kao Chow Repeated Authentification Protocols. An encoding of the Kao Chow repeated authentifi cation protocol can be found in Figure 4. All verifi cation conditions can be proved easily, except for rule three and the branch $(N_a, \{K_{ab}\})$. In this case, the procedure creates an infi nite tree as visualized below. Consequently, our procedure fails to detect occultness of this protocol.



Arbitrary Number of Backsteps. Occultness of the Otway-Rees protocol is proved using only basic tests, the proof of the Needham-Schroeder-Lowe protocol needs at most one back step for verifying each occultness obligation, and the Yahalom is proved using at most two back steps. In general, given a natural number n, there is an occult protocol which requires at least n back steps for proving occultness. Such a family of protocols is given in Figure 4. The proof tree for demonstrating occultness of the rule 2 of this protocol is given as follows; obviously, there is no deduction requiring less back steps.



Failed Proof Attempts. Lowe [8] showed that the original description of the Needham-Schroeder [12] protocol was flawed. The encoding of this protocol is identical to the one in Figure 2 except for the post of rule 2. This post is now assumed to be given by $\{[\{N_a, N_b\}_{\mathsf{pub}(A)}]\}$. Our search procedure terminates with an incomplete proof for this modified rule.



Using this failed proof attempt, we can show that the protocol is indeed not occult. The construction starts at the leaf labelled with $false$. Its parent node contains a cast and is exactly the Pre of one of the rules, say rl, of the protocol. Now, we consider a (partial) run of the protocol where all the rules preceding rl in the protocol description are applied in the given order.

$$\emptyset \xrightarrow{\{N_a, N_b\}} \{\{N_a, N_b\} \ddagger \{A, B'\}\}$$
$$\{\{N_a, N_b\} \ddagger \{A, B'\}\} \xrightarrow{\emptyset} \{[\{N_a, A\}_{\mathsf{pub}(B')}]\}$$

Next, we simulate an attack by following the branch from the $false$ leaf up to its root. The parent node of the $false$ leaf is directly connected with the root by an honest edge.

$$\left\{ \begin{array}{l} \{N_b, \_\} \ddagger \{A, B\}, \\ [\{N_a, A\}_{\mathsf{pub}(B')}] \end{array} \right\} \xrightarrow{\emptyset} \{[\{N_a, N_b\}_{\mathsf{pub}(A)}]\}$$

Having reached the root of the tree, one applies the rule for which our algorithm fails.

$$\{[\{N_a, N_b\}_{\mathsf{pub}(A)}]\} \xrightarrow{\emptyset} \{[\{N_b\}_{\mathsf{pub}(B)}]\}$$

Thus $\{N_b\}_{\mathsf{pub}(B)} \notin \mathcal{C}(Sec(\{N_a, N_b\} \ddagger \{A, B'\}))$, and the protocol is not occult.

6 Discussion

We have developed a procedure for proving the occultness of protocol rules and proved its correctness. If the procedure terminates with $true$, then the argument rule is occult. Moreover, occultness of all rules implies that the protocol is indeed secure. Our procedure follows the informal reasoning steps in [10], mechanizations do not require any user intervention, and there is a visually appealing graphical representation of occultness proofs.

We have tested our proof procedure on selected protocols from the the Clark and Jacob survey [2]. Usually, we can prove occultness using only a small number of search space extensions. The Otway-Rees and the Carlson protocol, for example, are proved to be secure using only basic tests, the Needham-Schroeder protocol needs at most one back step for verifying occultness of each rule and branch, and the Yahalom protocol needs at most two back steps for verifying each occultness conditions. We have also given examples of protocols whose occultness proofs need at least n back steps for an arbitrary natural number.

Much work remains to be done. In order to deal with many protocols used in practice, we have to extend our methods and support protocol features like hashing and timestamps. The algorithm described here is not a semi-decision procedure in the sense that occultness is eventually detected. It may be interesting to investigate subclasses of protocols which only require a bounded number of back steps, and for which our algorithm acts as a decision procedure. Also, we do not yet know under what circumstances a failed proof attempt implies that the protocol is insecure. An advantage of our method seems to be that it permits constructing attacks from failed proof attempts. For example from the failed proof attempt for the original Needham-Schroeder protocol in Section 5 we can construct Lowe's man-in-the-middle attack. We plan to investigate methods for constructing such attacks from failed proof attempts.

References

[1] I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *12th IEEE Computer Security Foundations Workshop*, pages 55–69. IEEE Computer Society, 1999.

[2] J. Clark and J. Jacob. A survey of authentication protocol literature. `http://www.cs.york.ac.uk/~jac/papers/drareviewps.ps`, 1997.

[3] E. Cohen. TAPS: A first-order verifier for cryptographic protocols. In *13th IEEE Computer Security Foundations Workshop*, pages 144–158. IEEE Computer Society, 2000.

[4] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In *Formal Methods and Security Protocols*, Federated Logic Conference, 1999.

[5] B. Dutertre, H. Saïdi, and V. Stavridou. Intrusion-Tolerant Group Management in Enclaves. Accepted for publication at the International Conference on Dependable Systems and Networks (DSN'2001), 2001.

[6] B. Dutertre and S. Schneider. Using a PVS embedding of CSP to verify authentication protocols. In *Theorem Proving in Higher Order Logics, TPHOL's 97*, volume 1275 of *Lecture Notes in Computer Science*, pages 121–136. Springer-Verlag, August 1997.

[7] L. Gong. Enclaves: Enabling Secure Collaboration over the Internet. *IEEE Journal of Selected Areas in Communications*, 15(3):567–575, April 1997.

[8] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of TACAS*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, 1996.

[9] C. Meadows. Invariant generation techniques in cryptographic protocol analysis. In *13th IEEE Computer Security Foundations Workshop*, pages 159–167. IEEE Computer Society, 2000.

[10] J. Millen and H. Rueß. Protocol-independent secrecy. In *2000 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2000.

[11] D. Monniaux. Decision procedures for the analysis of cryptographic protocols by logics of belief. In *12th Computer Security Foundations Workshop*, Mordano, Italy, June 1999. IEEE Computer Society.

[12] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–998, December 1978.

[13] D. Otway and O. Rees. Efficient and timely mutual authentication. *ACM Operating System Review*, 21(1):8–10, 1987.

[14] L. Paulson. Relations between secrets: Two formal analyses of the Yahalom protocol. Technical Report TR432, University of Cambridge, Computer Laboratory, July 1997.

[15] L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1):85–128, 1998.

[16] H. Rueß and J. Millen. Local secrecy for stated-based models. In *Proc. of the Workshop on Formal Methods in Computer Security (FMCS'2000)*, Chicago, IL, 2000.

[17] S. Schneider. Verifying authentication protocols in CSP. *IEEE Transactions on Software Engineering*, 24(9):741–758, September 1998.

[18] J. Thayer, J. Herzog, and J. Guttman. Honest ideals on strand spaces. In *11th IEEE Computer Security Foundations Workshop*, pages 66–78. IEEE Computer Society, 1998.

7 Proof of the Main Lemma

Main Lemma:

> If the predicate $search(P)(Z, K_s)(E_s)$ holds, then $occ(P, Z, K_s)(E_s, -)$ holds, too.

Proof : Instead of proving that $occ(P, Z, K_s)(E_s, N_s)$ holds, we prove the stronger property $occstrong(P, Z, K_s)(E_s)$ defined to hold iff for all P-configurations (I, H, C) such that

1. $Nonstates(E_s) \subseteq H$ and
2. $Z \in Sec(C)$

it is the case that $S \cap Sec(C) \neq \emptyset$,. The set $Nonstates(E_s)$ includes all non-state events in E_s. Obviously, $occstrong(P, Z, K_s)(E_s)$ implies $occ(P, Z, K_s)(E_s, -)$. The proof is by induction on the minimum number of back steps for deriving that $search(P, Z, K_s)(E_s)$ holds.

Initialization. If the derivation of $search(P, Z, K_s)(E_s)$ terminates with *true* and if no back steps has been used, then either $(Z, K_s) \stackrel{\sim}{\in} \mathsf{bnch}(Cont(E_s))$ or $db(E_s, Z, K_s)$ holds. Using basic tests one concludes that $occstrong(P, Z, K_s)(E_s)$ holds in both cases.

Step. Assume that $occstrong(P, Z, K_s)(E_s)$ holds for every $search(P, Z, K_s)(E_s)$ with a derivation that uses less or equal than n of back steps. Furthermore, consider P, Z, K_s, and E_s such that the derivation of $search(P, Z, K_s)(E_s)$ terminates with $true$ and uses $n + 1$ back steps, and assume a P-confi guration (I, H, C) such that $Nonstates(E_s) \subseteq H$ and $Z \in Sec(C)$.

Consequently, the back step terminates with $true$, and there exists a $M \in E_s$ and $Z \in \mathsf{parts}(M)$,such that $honest(P, Z, K_s)(M) \wedge fake(P, Z, K_s)(M) = true$ and the derivation of $honest(P, Z, K_s)(M)$ and $fake(P, Z, K_s)(M)$ uses at most n back steps. Now, M is in E_s, so M is in H. By induction on H there exists two global states H_1, H_2 such that

$$global(P)(I)(H_1, H_2)\ \&\ Nonstates(H_1) \subseteq H\ \&\ M \in \mathsf{parts}(H_2)\ \&\ M \notin \mathsf{parts}(H_1).$$

Apply case analysis depending on whether the global extension is honest or faked.

Case $honest(P)(H_1, H_2)$**:** There exists an applicable transition $t \in P$ such that $H_2 = Post(t) \cup (H_1 \setminus (Pre(t) \cap States))$; thus, $Nonstates(Pre(t)) \subseteq H$ and $M \in \mathsf{parts}(Post(t))$. Since $honest(P, Z, K_s)(M)$ reduces to $true$ and $M \in \mathsf{parts}(Post(t))$, we have $search(P, Z, K_s)(Pre(t))$ holds, and its derivation uses at most n back steps. Thus, $occstrong(P, Z, K_s)(Pre(t))$ holds. Because of the facts $Nonstates(Pre(t)) \in H$ and $Z \in Sec(C)$, it follows that $S \cap Sec(C) = \emptyset$. Consequently, the predicate $occstrong(P, Z, K_s)(E_s)$ holds.

Case $fake(I)(H_1, H_2)$**:** By defi nition of $fake$, $H_2 = H_1 \cup \{M'\}$ where $M' \in \mathsf{fake}(Cont(H) \cup I)$. Since $M \in \mathsf{parts}(H_2)$ and $M \notin \mathsf{parts}(H_1)$, we know that $M \in \mathsf{parts}(\mathsf{fake}(Cont(H) \cup I))$. It is easy to verify that

$$\mathsf{parts}(\mathsf{fake}(Cont(H) \cup I)) = \mathsf{fake}(Cont(H) \cup I) \cup \mathsf{parts}(Cont(H) \cup I)\ .$$

In addition, $M \notin \mathsf{parts}(I)$ (unless $Z \in \mathsf{parts}(I)$, in which case $Z \notin Book(C)$ by choice of C, which contradicts the hypothesis $Z \in Book(C)$) and $M \notin \mathsf{parts}(Cont(H))$, thus $M \notin \mathsf{parts}(Cont(H) \cup I)$. Consequently,

M must have been synthesized, meaning $X \in \mathsf{fake}(Cont(H) \cup I)$ if $M = \{X\}_K$ or there exist M_1, M_2 such that $M = M_1, M_2$ and $M_1, M_2 \in \mathsf{fake}(Cont(H) \cup I)$.

Now, let $H' = H \cup \{M_1, M_2\}$ (respectively $H' = H \cup \{X\}$). It is easy to verify that (I, H', C) is still a P-configuration, and we get $\{M_1, M_2\} \in H'$ (respectively $\{X\} \in H'$) and $Z \in Sec(C)$.

Assume (without loss of generality) that $Z \in \mathsf{parts}(M_1)$. By definition of $fake(I)(H_1, H_2)$, $search(P, Z, K_s)(\{M_1\})$ (respectively $search(P, Z, K_s)(\{X\})$) holds and its derivation uses at most n back steps. Therefore, the predicate $occstrong(P, Z, K_s)(\{M_1\})$ (resp. $occstrong(P, Z, K_s)(\{X\})$) holds. Finally, one concludes that $occstrong(P, Z, K_s)(E_s)$ holds.

■

An Overview of Formal Verification For the Time-Triggered Architecture*

John Rushby
Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025, USA
rushby@csl.sri.com

Abstract

We describe formal verification of some of the key algorithms in the Time-Triggered Architecture (TTA) for real-time safety-critical control applications. Some of these algorithms pose formidable challenges to current techniques and have been formally verified only in simplified form or under restricted fault assumptions. We describe what has been done and what remains to be done and indicate some directions that seem promising for the remaining cases and for increasing the automation that can be applied. We also describe the larger challenges posed by formal verification of the interaction of the constituent algorithms and of their emergent properties.

1 Introduction

The Time-Triggered Architecture (TTA) provides an infrastructure for safety-critical real-time control systems of the kind used in modern cars and airplanes. Concretely, it comprises an interlocking suite of distributed algorithms for functions such as clock synchronization and group membership, and their implementation in the form of TTA controllers, buses, and hubs. The suite of algorithms is known as TTP/C (an adjunct for non safety-critical applications is known as TTP/A) and was originally developed by Kopetz and colleagues at the Technical University of Vienna [28]; its current specification and commercial realization are by TTTech of Vienna [75]. More abstractly, TTA is part of a comprehensive approach to safety-critical real-time system design [25] that centers on time-triggered operation [26] and includes notions such as "temporal firewalls" [24] and "elementary" interfaces [27].

*This research was supported by NASA Langley Research Center under Cooperative Agreement NCC-1-377 with Honeywell Incorporated, by DARPA through the US Air Force Rome Laboratory under Contract F30602-96-C-0291, by the National Science Foundation under Contract CCR-00-86096, and by the NextTTA project of the European Union.

The algorithms of TTA are an exciting target for formal verification because they are individually challenging and they interact in interesting ways. To practitioners and developers of formal verification methods and their tools, these algorithms are excellent test cases—first, to be able to verify them at all, then to be able to verify them with sufficient automation that the techniques used can plausibly be transferred to nonspecialists for use in similar applications. For the developers and users of TTA, formal verification provides valuable assurance for its safety-critical claims, and explication of the assumptions on which these rest. As new versions of TTA and its implementations are developed, there is the additional opportunity to employ formal methods in the design loop.

TTA provides the functionality of a bus: host computers attach to TTA and are able to exchange messages with other hosts; in addition, TTA provides certain services to the hosts (e.g., an indication which other hosts and their interface controllers are participating reliably in network protocols). Because it is used in safety-critical systems, TTA must be fault tolerant: that is, it must continue to provide its services to nonfaulty hosts in the presence of faulty hosts and in the presence of faults in its own components. In addition, the services that it provides to hosts are chosen to ease the design and construction of fault-tolerant applications (e.g., in an automobile brake-by-wire application, each wheel has a brake that is controlled by its own host computer; the services provided by TTA make it fairly simple to arrange a safe distributed algorithm in which each host can adjust the braking force applied to its wheel to compensate for the failure of one of the other brakes or its host).

Serious consideration of fault-tolerant systems requires careful identification of the fault containment units (components that fail independently), fault hypotheses (the kind, arrival rate, and total number of faults to be tolerated), and the type of fault tolerance to be provided (e.g., what constitutes acceptable behavior in the presence of faults: fault masking vs. fail silence, self stabilization, or never-give-up). The basic goal in verifying a fault-tolerant algorithm is to prove

fault hypotheses satisfied *implies* acceptable behavior.

Stochastic or other probabilistic and experimental methods must then establish that the probability of the fault hypotheses being satisfied is sufficiently large to satisfy the mission requirements.

In this short paper, it is not possible to provide much by way of background to the topics adumbrated above, nor to discuss the design choices in TTA, but a suitable introduction is available in a previous paper [54] (and in more detail in [55]). Neither is it possible, within the limitations of this paper, to describe in detail the formal verifications that have already been performed for certain TTA algorithms. Instead, my goal here is to provide an overview of these verifications, and some of their historical antecedents, focusing on the importance of the exact fault hypotheses that are considered for each algorithm and on the ways in which the different algorithms interact. I also indicate techniques that increase the amount

of automation that can be used in these verifications, and suggest approaches that may be useful in tackling some of the challenges that still remain.

2 Clock Synchronization

As its full name indicates, the Time-Triggered Architecture uses the passage of time to schedule its activity and to coordinate its distributed components. A fault tolerant distributed clock synchronization algorithm is therefore one of TTA's fundamental elements.

Host computers attach to TTA through an interface controller that implements the TTP/C protocol. I refer to the combination of a host and its TTA controller as a *node*. Each controller contains an oscillator from which it derives its local notion of time (i.e., a clock). Operation of TTA is driven by a global schedule, so it is important that the local clocks are always in close agreement. Drift in the oscillators causes the various local clocks to drift apart so periodically (several hundred times a second) they must be resynchronized. What makes this difficult is that some of the clocks may be faulty.

The clock synchronization algorithm used in TTA is a modification of the Welch-Lynch (also known as Lundelius-Lynch) algorithm [78], which itself can be understood as a particular case of the abstract algorithm described by Schneider [66]. Schneider's abstract algorithm operates as follows: periodically, the nodes decide that it is time to resynchronize their clocks, each node determines the skews between its own clock and those of other nodes, forms a *fault-tolerant average* of these values, and adjusts its own clock by that amount.

An intuitive explanation for the general approach is the following. After a resynchronization, all the nonfaulty clocks will be close together (this is the definition of synchronization); by the time that they next synchronize, the nonfaulty clocks may have drifted further apart, but the amount of drift is bounded (this is the definition of a good clock); the clocks can be brought back together by setting them to some value close to the middle of their spread. An "ordinary average" (e.g., the mean or median) over all clocks may be affected by wild readings from faulty clocks (which, under a *Byzantine* fault hypothesis, may provide different readings to different observers), so we need a "fault-tolerant average" that is insensitive to a certain number of readings from faulty clocks.

The Welch-Lynch algorithm is characterized by use of the *fault-tolerant midpoint* as its averaging function. If we have n clocks and the maximum number of simultaneous faults to be tolerated is k ($3k < n$), then the fault-tolerant midpoint is the average of the $k+1$'st and $n-k$'th clock skew readings, when these are arranged in order from smallest to largest. If there are at most k faulty clocks, then some reading from a nonfaulty clock must be at least as small as the $k+1$'st reading, and the reading from another nonfaulty clock must be at least as great as the $n-k$'th; hence, the average of these two readings should be close to the middle of the spread of readings from good clocks.

The TTA algorithm is basically the Welch-Lynch algorithm specialized for $k = 1$ (i.e., it tolerates a single fault): that is, clocks are set to the average of the 2nd and $n-1$'st

clock readings (i.e., the second-smallest and second-largest). This algorithm works and tolerates a single arbitrary fault whenever $n \geq 4$. TTA does not use dedicated wires to communicate clock readings among the nodes attached to the network; instead, it exploits the fact that communication is time triggered according to a global schedule. When a node a receives a message from a node b, it notes the reading of its local clock and subtracts a fixed correction term to account for the network delay; the difference between this adjusted clock reading and the time for b's transmission that is indicated in the global schedule yields a's perception of the skew between clocks a and b.

Not all nodes in a TTA system need have accurate oscillators (they are expensive), so TTA's algorithm is modified from Welch-Lynch to use only the clock skews from nodes marked[1] as having accurate oscillators. Analysis and verification of this variant can be adapted straightforwardly from that of the basic algorithm. Unfortunately, TTA adds another complication.

For scalability, an implementation on the Welch-Lynch algorithm should use data structures that are independent of the number of nodes—i.e., it should not be necessary for each node to store the clock difference readings for all (accurate) clocks. Clearly, the second-smallest clock difference reading can be determined with just two registers (one to hold the smallest and another for the second-smallest reading seen so far), and the second-largest can be determined similarly, for a total of four registers per node. If TTA used this approach, verification of its clock synchronization algorithm would follow straightforwardly from that of Welch-Lynch. Instead, for reasons that are not described, TTA does not consider all the accurate clocks when choosing the second-smallest and second-largest, but just four of them.

The four clocks considered for synchronization are chosen as follows. First, TTA is able to tolerate more than a single fault by reconfiguring to exclude nodes that are detected to be faulty. This is accomplished by the group membership algorithm of TTA, which is discussed in the following section.[2] The four clocks considered for synchronization are chosen from the members of the current membership; it is therefore essential that group membership have the property that all nonfaulty nodes have the same members at all times. Next, each node maintains a queue of four clock readings[3]; whenever a message is received from a node that is in the current membership and that has the `SYF` field set, the clock difference reading is pushed on to the receiving node's queue (ejecting the oldest reading in the queue). Finally, when the current slot has the synchronization field (`CS`) set in the `MEDL`, each node runs the synchronization algorithm using the four clock readings stored in its queue.

Formal verification of the TTA algorithm requires more than simply verifying a four-clocks version of the basic Welch-Lynch algorithm: for example, the chosen clocks can

[1] By having the `SYF` field set in the `MEDL` (the global schedule known to all nodes).

[2] A node whose clock loses synchronization will suffer send and/or receive faults and will therefore be detected and excluded by the group membership algorithm.

[3] It is described as a push-down stack in the TTP/C specification [75], but this seems to be an error.

change from one round to the next. However, verification of the basic algorithm provides a foundation for the TTA case.

Formal verification of clock synchronization algorithms has quite a long history, beginning with Rushby and von Henke's verification [60] of the interactive convergence algorithm of Lamport and Melliar Smith [32]; this is similar to the Welch-Lynch algorithm, except that the *egocentric mean* is used as the fault-tolerant average. Shankar [70] formally verified Schneider's abstract algorithm and its instantiation for interactive convergence. This formalization was subsequently improved by Miner (reducing the difficulty of the proof obligations needed to establish the correctness of specific instantiations), who also verified the Welch-Lynch instantiation [38]. All these verifications were undertaken with EHDM [61], a precursor to PVS [41]. The treatment developed by Miner was translated to PVS and generalized (to admit nonaveraging algorithms such as that of Srikanth and Toueg [73] that do not conform to Schneider's treatment) by Schwier and von Henke [69]. This treatment was then extended to the TTA algorithm by Pfeifer, Schwier and von Henke [45]. The TTA algorithm is intended to operate in networks where there are at least four good clocks, and it is able to mask any single fault in this circumstance. Pfeifer, Schwier and von Henke's verification establishes this property. Additional challenges still remain, however.

In keeping with the *never give up* philosophy that is appropriate for safety-critical applications, TTA should remain operational with less than four good clocks, though "the requirement to handle a Byzantine fault is waived" [75, page 85]. It would be valuable to characterize and formally verify the exact fault tolerance achieved in these cases. One approach to achieving this would be to undertake the verification in the context of a "hybrid" fault model such as that introduced for consensus by Thambidurai and Park [74]. In a pure Byzantine fault model, all faults are treated as arbitrary: nothing is assumed about the behavior of faulty components. A hybrid fault model introduces additional, constrained kinds of faults and the verification is extended to examine the behavior of the algorithm concerned under combinations of several faults of different kinds. Thambidurai and Park's model augments the Byzantine or *arbitrary* fault model with *manifest* and *symmetric* faults. A manifest fault is one that is consistently detectable by all nonfaulty nodes; a symmetric fault is unconstrained, except that it appears the same to all nonfaulty nodes. Rushby reinterpreted this fault model for clock synchronization and extended verification of the interactive convergence algorithm to this more elaborate fault model [49]. He showed that the interactive convergence algorithm with n nodes can withstand a arbitrary, s symmetric, and m manifest faults simultaneously, provided $n > 3a + 2s + m$. Thus, a three-clock system using this algorithm can withstand a symmetric fault or two manifest faults.

Rushby also extended this analysis to *link* faults, which can be considered as asymmetric and possibly intermittent manifest faults (i.e., node a may obtain a correct reading of node b's clock while node c obtains a detectably faulty reading). The fault tolerance of the algorithm is then $n > 3a + 2s + m + l$ where l is the maximum, over all pairs of nodes, of the number of nodes that have faulty links to one or other of the pair.

It would be interesting to extend formal verification of the TTA algorithm to this fault model. Not only would this enlarge the analysis to cases where fewer than three good clocks remain, but it could also provide a much simpler way to deal with the peculiarities of the TTA algorithm (i.e., its use of queues of just four clocks). Instead of explicitly modeling properties of the queues, we could, under a fault model that admits link faults, imagine that the queues are larger and contain clock difference readings from the full set of nodes, but that link faults reduce the number of valid readings actually present in each queue to four (this idea was suggested by Holger Pfeifer). A recent paper by Schmid [64] considers link faults for clock synchronization in a very general setting, and establishes bounds on fault tolerance for both the Welch-Lynch and Srikanth-Toueg algorithms and I believe this would be an excellent foundation for a comprehensive verification of the TTA algorithm.

All the formal verifications of clock synchronizations mentioned above are "brute force": they are essentially mechanized reproductions of proofs originally undertaken by hand. The proofs depend heavily on arithmetic reasoning and can be formalized at reasonable cost only with the aid of verification systems that provide effective mechanization for arithmetic, such as PVS. Even these systems, however, typically mechanize only linear arithmetic and require tediously many human-directed proof steps (or numerous intermediate lemmas) to verify the formulas that arise in clock synchronization. The new ICS decision procedures [16] developed for PVS include (incomplete) extensions to nonlinear products and it will be interesting to explore the extent to which such extensions simplify formal verification of clock synchronization algorithms.[4] Even if all the arithmetic reasoning were completely automated, current approaches to formal verification of clock synchronization algorithms still depend heavily on human insight and guidance. The problem is that the synchronization property is not inductive: it must be strengthened by the conjunction of several other properties to achieve a property that is inductive. These additional properties are intricate arithmetic statements whose invention seems to require considerable human insight. It would be interesting to see if modern methods for invariant discovery and strengthening [6, 7, 76] can generate some of these automatically, or if the need for them could be sidestepped using reachability analysis on linear hybrid automata.

All the verifications described above deal with the steady-state case; initial synchronization is quite a different challenge. Note that (re)initialization may be required during operation if the system suffers a massive failure (e.g., due to powerful electromagnetic effects), so it must be fast. The basic idea is that a node that detects no activity on the bus for some time will assume that initialization is required and it will broadcast a wakeup message: nodes that receive the message will synchronize to it. Of course, other nodes may make the same determination at about the same time and may send wakeup messages that collide with others. In these cases, nodes back off for (different) node-specific intervals and try again. However, it is difficult to detect collisions with perfect accuracy and simple algorithms can lead to existence of groups of nodes synchronized within themselves but un-

[4] It is not enough to mechanize real arithmetic on its own; it must be combined with inequalities, integer linear arithmetic, equality over uninterpreted function symbols and several other theories [50].

aware of the existence of the other groups. All of these complications must be addressed in a context where some nodes are faulty and may not be following (indeed, may be actively disrupting) the intended algorithm. The latest version of TTA uses a star topology and the initialization algorithm is being revised to exploit some additional safeguards that the central guardian makes possible [42]. Verification of initialization algorithms is challenging because, as clearly explained in [42], the essential purpose of such an algorithm is to cause a transition between two models of computation: from asynchronous to synchronous. Formal explication of this issue, and verification of the TTA initialization algorithm, are worthwhile endeavors for the future.

3 Transmission Window Timing

Synchronized clocks and a global schedule ensure that nonfaulty nodes broadcast their messages in disjoint time slots: messages sent by nonfaulty nodes are guaranteed not to collide on the bus. A faulty node, however, could broadcast at any time—it could even broadcast constantly (the *babbling* failure mode). This fault is countered by use of a separate fault containment unit called a *guardian* that has independent knowledge of the time and the schedule: a message sent by one node will reach others only if the guardian agrees that it is indeed scheduled for that time.

Now, the sending node, the guardian, and each receiving node have synchronized clocks, but there must be some slack in the time window they assign to each slot so that good messages are not truncated or rejected due to clock skew within the bounds guaranteed by the synchronization algorithm. The design rules used in TTA are the following, where Π is the maximum clock skew between synchronized components.

- The receive window extends from the beginning of the slot to $4\,\Pi$ beyond its allotted duration.
- Transmission begins $2\,\Pi$ units after the beginning of the slot and should last no longer than the allotted duration.
- The bus guardian for a transmitter opens its window Π units after the beginning of the slot and closes it $3\,\Pi$ beyond its allotted duration.

These rules are intended to ensure the following requirements.

Agreement: If any nonfaulty node accepts a transmission, then all nonfaulty nodes do.

Validity: If any nonfaulty node transmits a message, then all nonfaulty nodes will accept the transmission.

Separation: messages sent by nonfaulty nodes or passed by nonfaulty guardians do not arrive before other components have finished the previous slot, nor after they have started the following one.

Formal specification and verification of these properties is a relatively straightforward exercise. Description of a formal treatment using PVS is available as a technical report [57].

4 Group Membership

The clock synchronization algorithm tolerates only a single (arbitrary) fault. Additional faults are tolerated by diagnosing the faulty node and reconfiguring to exclude it. This diagnosis and reconfiguration is performed by the *group membership* algorithm of TTA, which ensures that each TTA node has a record of which nodes are currently participating correctly in the TTP/C protocol. In addition to supporting the internal fault tolerance of TTA, membership information is made available as a service to applications; this supports the construction of relatively simple, but correct, strategies for tolerating faults at the application level. For example, in an automobile brake-by-wire application, the node at each wheel can adjust its braking force to compensate for the failure (as indicated in the membership information) of the node or brake at another wheel. For such strategies to work, it is obviously necessary that the membership information should be reliable, and that the application state of nonmembers should be predictable (e.g., the brake is fully released).

Group membership is a distributed algorithm: each node maintains a private *membership* list, which records all the nodes that it believes to be nonfaulty. Reliability of the membership information is characterized by the following requirements.

Agreement: The membership lists of all nonfaulty nodes are the same.

Validity: The membership lists of all nonfaulty nodes contain all nonfaulty nodes and at most one faulty node (we cannot require immediate removal of faulty nodes because a fault must be manifested before it can be diagnosed).

These requirements can be satisfied only under restricted fault hypotheses. For example, validity cannot be satisfied if new faults arrive too rapidly, and it is provably impossible to diagnose an arbitrary-faulty node with certainty. When unable to maintain accurate membership, the best recourse is to maintain agreement, but sacrifice validity. This weakened requirement is called *clique avoidance*.

Two additional properties also are desirable in a group membership algorithm.

Self-diagnosis: faulty nodes eventually remove themselves from their own membership lists and fail silently (i.e., cease broadcasting).

Reintegration: it should be possible for excluded but recovered nodes to determine the current membership and be readmitted.

TTA operates as a broadcast bus (even though the recent versions are stars topologically); the global schedule executes as a repetitive series of *rounds*, and each node is allocated a broadcast slot in each round. The fault hypothesis of the membership algorithm

is a benign one: faults must arrive two or more rounds apart, and must be symmetric in their manifestations: either *all* or exactly *one* node may fail to receive a broadcast message (the former is called a *send* fault, the latter a *receive* fault). The membership requirements would be relatively easy to satisfy if each node were to attach a copy of its membership list to each message that it broadcasts. Unfortunately, since messages are typically very short, this would use rather a lot of bandwidth (and bandwidth was a precious commodity in early implementations of TTA), so the algorithm must operate with less explicit information and nodes must infer the state and membership of other nodes through indirect means. This operates as follows.

Each active TTA node maintains a membership list of those nodes (including itself) that it believes to be active and operating correctly. Each node listens for messages from other nodes and updates its membership list according to the information that it receives. The time-triggered nature of the protocol means that each node knows when to expect a message from another node, and it can therefore detect the absence of such a message. Each message carries a CRC checksum that encodes information about its sender's *C-State*, which includes its local membership list. To infer the local membership of the sender of a message, receivers must append their estimate of that membership (and other C-state information) to the message and then check whether the calculated CRC matches that sent with the message. It is not feasible (or reliable) to try all possible memberships, so receivers perform the check against just their own local membership, and one or two variants.

Transmission faults are detected as follows: each broadcaster listens for the message from its *first successor* (roughly speaking, this will be the next node to broadcast) to check whether it suffered a transmission fault: this will be indicated by its exclusion from the membership list of the message from its first successor. However, this indication is ambiguous: it could be the result of a transmission fault by the original broadcaster, or of a receive fault by the successor. Nodes use the local membership carried by the message from their *second successor* to resolve this ambiguity: a membership that excludes the original broadcaster but includes the first successor indicates a transmission fault by the original broadcaster, and one that includes the original broadcaster but excludes the first successor indicates a receive fault by the first successor.

Nodes that suffer receive faults could diagnose themselves in a similar way: their local membership lists will differ from those of nonfaulty nodes, so their next broadcast will be rejected by both their successors. However, the algorithm actually performs this diagnosis differently. Each node maintains *accept* and *reject* counters that are initialized to 1 and 0, respectively, following its own broadcast. Incoming messages that indicate a membership matching that of the receiver cause the receiver to increment its accept count; others (i.e., those that indicate a different membership or that are considered invalid for other reasons) cause it to increment its reject count. Before broadcasting, each node compares its accept and reject counts and shuts down unless the former is greater than the latter.

Formal verification of this algorithm is difficult. We wish to prove that agreement and validity are invariants of the algorithm (i.e., they are true of all reachable states), but it is

difficult to do this directly (because it is hard to characterize the reachable states). So, instead, we try to prove a stronger property: namely, that agreement and validity are *inductive* (that is, true of the initial states and preserved by all steps of the algorithm). The general problem with this approach to verification of safety properties of distributed algorithms is that natural statements of the properties of interest are seldom inductive. Instead, it is necessary to strengthen them by conjoining additional properties until they become inductive. The additional properties typically are discovered by examining failed proofs and require human insight.

Before details of the TTA group membership algorithm were known, Katz, Lincoln, and Rushby published a different algorithm for a similar problem, together with an informal proof of its correctness [23] (I will call this the "WDAG" algorithm). A flaw in this algorithm for the special case of three nodes was discovered independently by Shankar and by Creese and Roscoe [12] and considerable effort was expended in attempts to formally verify the corrected version. A suitable method was found by Rushby [53] who used it to formally verify the WDAG algorithm, but used a simplified algorithm (called the "CAV" algorithm) to explicate the method in [53]. The method is based on strengthening a putative safety property into a *disjunction* of "configurations" that can easily be proved to be inductive. Configurations can be constructed systematically and transitions among them have a natural diagrammatic representation that conveys insight into the operation of the algorithm. Pfeifer subsequently used this method to verify validity, agreement, and self-diagnosis for the full TTA membership algorithm [44] (verification of self-diagnosis is not described in the paper).

Although the method just described is systematic, it does require considerable human interaction and insight, so more automatic methods are desirable. All the group membership algorithms mentioned (CAV, WDAG, TTA) are n-process algorithms (so-called *parameterized systems*), so one attractive class of methods seeks to reduce the general case to some fixed configuration (say four processes) of an abstracted algorithm that can be model checked. Creese and Roscoe [12] report an investigation along these lines for the WDAG algorithm. The difficulty in such approaches is that proving that the abstracted algorithm is faithful to the original is often as hard as the direct proof.

An alternative is to *construct* the abstracted algorithm using automated theorem proving so that the result is guaranteed to be sound, but possibly too conservative. These methods are widely used for predicate [62] and data [11] abstraction (both methods are implemented in PVS using a generalization of the technique described in [63]), and have been applied to n-process examples [71]. The precision of an abstraction is determined by the guidance provided to the calculation (e.g., which predicates to abstract on) and by the power of the automated deduction methods that are employed.[5] The logic called WS1S is very attractive in this regard, because it is very expressive (it can represent arithmetic and set operations on integers) and it is decidable [14]. The method implemented in the PAX tool [4, 5] performs

[5] In this context, automated deduction methods are used in a *failure-tolerant* manner, so that if the methods fail to prove a true theorem, the resulting abstraction will be sound, but more conservative than necessary.

automated abstraction of parameterized specifications modeled in WS1S. Application of the tool to the CAV group membership protocol is described on the PAX web page at `http://www.informatik.uni-kiel.de/~kba/pax/examples.html`. The abstraction yields a finite-state system that can be examined by model checking. I conjecture that extension of this method to the TTA algorithm may prove difficult because the counters used in that algorithm add an extra unbounded dimension.

The design of TTA (and particularly of the central guardian) is intended to minimize violations of the benign fault hypothesis of the group membership algorithm. But we cannot guarantee absence of such violations, so the membership algorithm is buttressed by a clique avoidance algorithm (it would better be called a clique elimination algorithm) that sacrifices validity but maintains agreement under weakened fault hypotheses. Clique avoidance is actually a subalgorithm of the membership algorithm: it comprises just the part that manages the accept and reject counters and that causes a node to shut down prior to a broadcast unless its accept count exceeds its reject count at that point. The clique avoidance algorithm can be analyzed either in isolation or, more accurately, in the presence of the rest of the membership algorithm (this is, the part that deals with the first and second successor).

Beyond the benign fault hypothesis lie *asymmetric* faults (where more than one but less than all nodes fail to receive a broadcast correctly), and *multiple* faults, which are those that arrive less than two rounds apart. These hypotheses all concern loss of messages; additional hypotheses include *processor* faults, where nodes fail to follow the algorithm, and *transient* faults, where nodes have their state corrupted (e.g., by high-intensity radiation) but otherwise follow the algorithm correctly.

Bauer and Paulitsch [3] describe the clique avoidance algorithm and give an informal proof that it tolerates a single asymmetric fault. Their analysis includes the effects of the rest of the membership algorithm. Bouajjani and Merceron [8] prove that the clique avoidance algorithm, considered in isolation, tolerates multiple asymmetric faults; they also describe an abstraction for the n-node, k-faults parameterized case that yields a counter automaton. Reachability is decidable for this class of systems, and experiments are reported with two automated verifiers for the $k = 1$ case.

For transient faults, I conjecture that the most appropriate framework for analysis is that of self-stabilization [68]. An algorithm is said to be *self-stabilizing* if it converges to a stable "good" state starting from an arbitrary initial state. The arbitrary initial state can be one caused by an electromagnetic upset (e.g., that changes the values of the accept and reject counters), or by other faults outside the benign fault hypotheses.

An attractive treatment of self-stabilization is provided by the "Detectors and Correctors" theory of Arora and Kulkarni. The full theory [2, 31] is comprehensive and more than is needed for my purposes, so I present a simplified and slightly modified version that adapts the important insights of the original formulation to the problem at hand.

We assume some "base" algorithm M whose purpose is to maintain an invariant S: that is, if the (distributed) system starts in a state satisfying the predicate S, then execution of M will maintain that property. In our case, M is the TTA group membership algorithm, and S

is the conjunction of the agreement and validity properties. M corresponds to what Arora and Kulkarni call the "fault-intolerant" program, but in our context it is actually a fault-tolerant algorithm in its own right. This aspect of the system's operation can be specified by the Hoare formula

$$\{S\}\ M||F\ \{S\}$$

where F is a "fault injector" that characterizes the fault hypothesis of the base algorithm and $M||F$ denotes the concurrent execution of M and F.

Now, a transient fault can take the system to some state not satisfying S, and at this point our hope is that a "corrector" algorithm C will take over and somehow cause the system to converge to a state satisfying S, where the base algorithm can take over again. We can represent this by the following formula

$$C \models \Diamond S$$

where $\Diamond$ is the *eventually* modality of temporal logic.

In our case, C is the TTA clique avoidance algorithm. So far we have treated M and C separately but, as noted previously, they must actually run concurrently, so we really require

$$\{S\}\ C||M||F\ \{S\}$$

and

$$C||M||F \models \Diamond S.$$

The presence of F in the last of these represents the fact that although the disturbance that took the system to an arbitrary state is assumed to have passed when convergence begins, the standard, benign fault hypothesis still applies.

To ensure the first of these formulas, we need that C does not interfere with M—that is, that $C||M$ behaves the same as M (and hence $C||M||F$ behaves the same as $M||F$). A very direct way to ensure this is for C actually to be a subalgorithm of M—for then $C||M$ *is* the same as M. As we have already seen later, this is the case in TTA, where the clique avoidance algorithm is just a part of the membership algorithm.

A slight additional complication is that the corrector may not be able to restore the system to the ideal condition characterized by S, but only to some "safe" approximation to it, characterized by S'. This is the case in TTA, where clique avoidance sacrifices validity. Our formulas therefore become the following.

$$\{S\}\ C||M||F\ \{S\} \tag{1}$$

$$\{S'\}\ C||M||F\ \{S' \vee S\},\ \text{and}\ S \supset S' \tag{2}$$

and

$$C||M||F \models \Diamond S'. \tag{3}$$

The challenge is formally to verify these three formulas. Concretely, (1) is accomplished for TTA by Pfeifer's verification [44] (and potentially, in more automated form, by extensions to the approaches of [4, 8]), (2) should require little more than an adjustment to those proofs, and the hard case is (3). Bouajjani and Merceron's analysis [8] can be seen as establishing

$$C \models \Diamond S'$$

for the restricted case where the arbitrary initial state is one produced by the occurrence of multiple, possibly asymmetric faults in message transmission or reception. The general case must consider the possibility that the initial state is produced by some outside disturbance that sets the counters and flags of the algorithm to arbitrary values (I have formally verified this case for a simplified algorithm), and must also consider the presence of M and F. Formal verification of this general case is an interesting challenge for the future. Kulkarni [30, 31] has formally specified and verified the general detectors and correctors theory in PVS, and this provides a useful framework in which to develop the argument.

A separate topic is to examine the consequences of giving up validity in order to maintain agreement under the clique avoidance algorithm. Under the never give up philosophy, it is reasonable to sacrifice one property rather than lose all coordination when the standard fault hypothesis is violated, but some useful insight may be gained through an attempt to formally characterize the possible behaviors in these cases.

Reintegration has so far been absent from the discussion. A node that diagnoses a problem in its own operation will drop out of the membership, perform diagnostic tests and, if these are satisfactory (indicating that the original fault was a transient event), attempt to reintegrate itself into the running system. This requires that the node first (re)synchronizes its clock to the running system, then acquires the current membership, and then "speaks up" at its next slot in the schedule. There are potential difficulties here: for example, a broadcast by a node a may be missed by a node b whose membership is used to initialize a reintegrating node c; rejection of its message by b and c then causes the good node a to shut down. This scenario is excluded by the requirement that a reintegrating node must correctly receive a certain number of messages before it may broadcast itself. Formal examination of reintegration scenarios is another interesting challenge for the future.

5 Interaction of Clock Synchronization and Group Membership

Previous sections considered clock synchronization and group membership in isolation but noted that, in reality, they interact: synchronization depends on membership to eliminate nodes diagnosed as faulty, while membership depends on synchronization to create the time-triggered round structure on which its operation depends. Mutual dependence of components on the correct operation of each other is generally formalized in terms of assume-guarantee reasoning, first introduced by Chandy and Misra [39] and Jones [22]. The idea is

to show that component X_1 guarantees certain properties P_1 on the assumption that component X_2 delivers certain properties P_2, and *vice versa* for X_2, and then claim that the composition of X_1 and X_2 guarantees P_1 and P_2 unconditionally. This kind of reasoning appears—and indeed is—circular in that X_1 depends on X_2 and *vice versa*. The circularity can lead to unsoundness and there has been much research on the formulation of rules for assume-guarantee reasoning that are both sound and useful. Different rules may be compared according to the kinds of system models and specification they support, the extent to which they lend themselves to mechanized analysis, and the extent to which they are preserved under refinement (i.e., the circumstances under which X_1 can be replaced by an implementation that may do more than X_1).

Closer examination of the circular dependency in TTA reveals that it is not circular if the temporal evolution of the system is taken into consideration: clock synchronization in round t depends on group membership in round $t-1$, which in turn depends on clock synchronization in round $t-2$ and so on. McMillan [37] has introduced an assume-guarantee rule that seems appropriate to this case. McMillan's rule can be expressed as follows, where H is a "helper" property (which can be simply *true*), $\Box$ is the "always" modality of Linear Temporal Logic (LTL), and $p \rhd q$ (" p constrains q") means that if p is always true up to time t, then q holds at time $t+1$ (i.e., p fails before q), where we interpret time as rounds.

$$\frac{\begin{array}{c}\langle H\rangle X_1 \langle P_2 \rhd P_1\rangle \\ \langle H\rangle X_2 \langle P_1 \rhd P_2\rangle\end{array}}{\langle H\rangle \;\; X_1 || X_2 \;\; \langle \Box(P_1 \wedge P_2)\rangle} \tag{4}$$

Notice that $p \rhd q$ can be written as the LTL formula $\neg(p \,\mathsf{U}\, \neg q)$, where U is the LTL "until" operator. This means that the antecedent formulas can be established by LTL model checking if the transition relations for X_1 and X_2 are finite.

I believe the soundness of the circular interaction between the clock synchronization and group membership algorithms of TTA can be formally verified using McMillan's rule. To carry this out, we need to import the proof rule (4) into the verification framework employed—and for this we probably need to embed the semantics of the rule into the specification language concerned. McMillan's presentation of the rule only sketches the argument for its soundness; a more formal treatment is given by Namjoshi and Trefler [40], but it is not easy reading and does not convey the basic intuition. Rushby [56] presents an embedding of LTL in the PVS specification language and formally verifies the soundness of the rule. The specification and proof are surprisingly short and provide a good demonstration of the power and convenience of the PVS language and prover.

Using this foundation to verify the interaction between the clock synchronization and group membership algorithms of TTA remains a challenge for the future. Observe that such an application of assume-guarantee reasoning has rather an unusual character: conventionally, the components in assume-guarantee reasoning are viewed as separate, peer processes, whereas here they are distributed algorithms that form part of a protocol hierarchy (with membership above synchronization).

6 Emergent Properties

Clock synchronization, transmission window timing, and group membership are important properties, but what makes TTA useful are not the individual properties of its constituent algorithms, but the emergent properties that come about through their combination. These emergent properties are understood by the designers and advocates of TTA, but they have not been articulated formally in ways that are fully satisfactory, and I consider this the most important and interesting of the tasks that remain in the formal analysis of TTA.

I consider the three "top level" properties of TTA to be the time-triggered model of computation, support for application-independent fault tolerance, and partitioning. The time-triggered model of computation can be construed narrowly or broadly. Narrowly, it is a variant on the notion of synchronous system [35]: these are distributed computer systems where there are known upper bounds on the time that it takes nonfaulty processors to perform certain operations, and on the time that it takes for a message sent by one nonfaulty processor to be received by another. The existence of these bounds simplifies the development of fault-tolerant systems because nonfaulty processes executing a common algorithm can use the passage of time to predict each others' progress, and the absence of expected messages can be detected. This property contrasts with asynchronous systems, where there are no upper bounds on processing and message delays, and where it is therefore provably impossible to achieve certain forms of consistent knowledge or coordinated action in the presence of even simple faults [9, 17]. Rushby [52] presents a formal verification that a system possessing the synchronization and scheduling mechanisms of TTA can be used to create the abstraction of a synchronous system. An alternative model, closer to TTA in that it does not abstract out the real-time behavior, is that of the language Giotto [19] and it would be interesting to formalize the connection between TTA and Giotto.

More broadly construed, the notion of time-triggered system encompasses a whole philosophy of real-time systems design—notably that espoused by Kopetz [25]. Kopetz' broad conception includes a distinction between *composite* and *elementary* interfaces [27] and the notion of a *temporal firewall* [24].

A time-triggered system does not merely schedule activity within nodes, it also manages the reliable transmission of messages between them. Messages obviously communicate data between nodes (and the processes within them) but they may also, through their presence or absence and through the data that they convey, influence the flow of control within a node or process (or, more generically, a component). An important insight is that one component should not allow another to control its own progress. Suppose, for example, that the guarantees delivered by component X_1 are quite weak, such as, "this buffer may sometimes contain recent data concerning parameter A." Another component X_2 that uses this data must be prepared to operate when recent data about A is unavailable (at least from X_1). It might seem that predictability and simplicity would be enhanced if we were to ensure that the flow of data about A is reliable—perhaps using a protocol involving acknowledgments. But in fact, contrary to this intuition, such a mechanism would greatly

increase the coupling between components and introduce more complicated failure propagations. For example, X_1 could block waiting for an acknowledgment from X_2 that may never come if X_2 has failed, thereby propagating the failure from X_2 to X_1. Kopetz [27] defines interfaces that involve such bidirectional flow of control as composite and argues convincingly that they should be eschewed in favor of elementary interfaces in which control flow is unidirectional.

The need for elementary interfaces leads to protocols for nonblocking asynchronous communication that nonetheless ensure timely transmission and mutual exclusion (i.e., no simultaneous reading and writing of the same buffer). In computer science, these are known as lock- and wait-free atomic register constructions ([1] is a convenient survey, focussing on the work of Lamport, who first introduced the topic), but similar constructions were developed independently in the avionics and real-time communities. The best-known of these is the four-slot protocol of Simpson [72]. Formal analyses of Simpson's protocol have been developed by Clark [10] (using Petri nets), by Rushby [59] (using model checking), and by Henderson and Paynter [18] (using PVS). Hesselink [21] have verified some atomic register constructions from the computer science literature using ACL2.

TTA uses a protocol called NBW (nonblocking write) [29] whose wait-free element was inspired by Simpson's algorithm, and whose lock-free construction is that of Lamport [33]. It would be useful to undertake a formal examination of NBW (which is used in the Communication Network Interface (CNI) that provides communication between hosts and their TTA controllers), particularly since Simpson's algorithm requires atomic control registers, and Rushby's analysis [59] shows that it fails when this (very strong) assumption is violated.

The larger issue of formally characterizing composite and elementary interfaces has not yet been tackled, to my knowledge. It is debatable whether formalization of these notions is best performed as part of a broad treatment of time-triggered systems, or as part of an orthogonal topic concerned with application-independent fault tolerance. *Temporal firewalls*, another element in Kopetz' comprehensive philosophy [24], seem definitely to belong in the treatment of fault tolerance. The standard way to communicate a sensor sample is to package it with a timestamp: then the consuming process can estimate the "freshness" of the sample. But surely the useful lifetime of a sample depends on the accuracy of the original reading and on the dynamics of the parameter being measured—and these factors are better known to the process doing the sensing than to the process that consumes the sample. So, argues Kopetz, it is better to turn the timestamp around, so that it indicates the "must use by" time, rather than the time at which the sample was taken. This is the idea of the temporal firewall, which exists in two variants. A *phase-insensitive* sensor sample is provided with a time and a guarantee that the sampled value is accurate (with respect to a specification published by the process that provides it) until the indicated time. For example, suppose that engine oil temperature may change by at most 1% of its range per second, that its sensor is completely accurate, and that the data is to be guaranteed to 0.5%. Then the sensor sample will be provided with a time 500 ms ahead of the instant when it

was sampled, and the receiver will know that it is safe to use the sampled value until the indicated time. A *phase-sensitive* temporal firewall is used for rapidly changing parameters; in addition to sensor sample and time, it provides the parameters needed to perform state estimation. For example, along with sampled crankshaft angle, it may supply RPM, so that angle may be estimated more accurately at the time of use.

The advantage of temporal firewalls is that they allow some of the downstream processing (e.g., sensor fusion) to become less application dependent. Temporal firewalls are consistent with modern notions of *smart sensors* that co-locate computing resources with the sensor. Such resources allow a sensor to return additional information, including an estimate of the accuracy of its own reading. An attractive way to indicate (confidence in) the accuracy of a sensor reading is to return two values (both packaged in a temporal firewall) indicating the upper and lower 95% (say) confidence interval. If several such intervals are available from redundant sensors, then an interesting question is how best to combine (or *fuse*) them. Marzullo [36] introduces the sensor fusion function $\bigcap_{f,n}(S)$ for this problem; Rushby formally verifies the soundness of this construction (i.e., the fused interval always contains the correct value) [58]. A weakness of Marzullo's function is that it lacks the "Lipschitz Condition": small changes in input sensor readings can sometimes produce large changes in its output. Schmid and Schossmaier [65] have recently introduced an improved fusion function $\mathcal{F}_n^f(S)$ that does satisfy the Lipschitz condition, and is optimal among all such functions. It would be interesting to verify formally the properties of this function.

Principled fault tolerance requires not only that redundant sensor values are fused effectively, but that all redundant consumers agree on exactly the same values; this is the notion of *replica determinism* [46] that provides the foundation for *state machine replication* [67] and other methods for application-independent fault tolerance based on exact-match voting. Replica determinism in its turn depends on *interactively consistent* message passing: that is, message passing in which all nonfaulty recipients obtain the same value [43], even if the sender and some of the intermediaries in the transmission are faulty (this is also known as the problem of *Byzantine Agreement* [34]). It is well known [35] that interactive consistency cannot be achieved in the presence of a single arbitrary fault with less than two rounds of information exchange (one to disseminate the values, and one to cross-check), yet TTA sends each message in only a single broadcast. How can we reconcile this practice with theory? I suggest in [55] that the interaction of message broadcasts with the group membership algorithm (which can be seen as a continuously interleaving two-round algorithm) in TTA achieves a "Draconian consensus" in which agreement is enforced by removal of any members that disagree. It would be interesting to subject this idea to formal examination, and to construct an integrated formal treatment for application-level fault tolerance in TTA similar to those previously developed for classical state machine replication [13, 48].

The final top-level property is the most important for safety-critical applications; it is called *partitioning* and it refers to the requirement that faults in one component of TTA, or in one application supported by TTA, must not propagate to other components and appli-

cations, and must not affect the operation of nonfaulty components and applications, other than through loss of the services provided by the failed elements. It is quite easy to develop a formal statement of partitioning—but only in the absence of the qualification introduced in the final clause of the previous sentence (see [51] for an extended discussion of this topic). In the absence of communication, partitioning is equivalent to isolation and this property has a long history of formal analysis in the security community [47] and has been adapted to include the real-time attributes that are important in embedded systems [79]. In essence, formal statements of isolation state that the behavior perceived by one component is entirely unchanged by the presence or absence of other components. When communication between components is allowed, this simple statement no longer suffices, for if X_1 supplies input to X_2, then absence of X_1 certainly changes the behavior perceived by X_2. What we want to say is that the *only* change perceived by X_2 is that due to the faulty or missing data supplied by X_1 (i.e., X_1 must not be able to interfere with X_2's communication with other components, nor write directly into its memory, and so on). To my knowledge, there is no fully satisfactory formal statement of this interpretation of partitioning.

It is clear that properties of the TTA algorithms and architecture are crucial to partitioning (e.g., clock synchronization, the global schedule, existence of guardians, the single-fault assumption, and transmission window timing are all needed to stop a faulty node violating partitioning by babbling on the bus), and there are strong informal arguments (backed by experiment) that these properties are sufficient [55], but to my knowledge there is as yet no comprehensive formal treatment of this argument.

7 Conclusion

TTA provides several challenging formal verification problems. Those who wish to develop or benchmark new techniques or tools can find good test cases among the algorithms and requirements of TTA. However, I believe that the most interesting and rewarding problems are those that concern the interactions of several algorithms, and it is here that new methods of compositional analysis and verification are most urgently needed. Examples include the interaction between the group membership and clique avoidance algorithms and their joint behavior under various fault hypotheses, the mutual interdependence of clock synchronization and group membership, and the top-level properties that emerge from the collective interaction of all the algorithms and architectural attributes of TTA. Progress on these fronts will not only advance the techniques and tools of formal methods, but will strengthen and deepen ties between the formal methods and embedded systems communities, and make a valuable contribution to assurance for the safety-critical systems that are increasingly part of our daily lives.

Acknowledgments

Günther Bauer of TU Vienna provided helpful comments and corrections for a previous version of this paper.

References

Papers on formal methods and automated verifi cation by SRI authors can generally be located by visiting home pages or doing a search from `http://www.csl.sri.com/programs/formalmethods`.

[1] James H. Anderson. Lamport on mutual exclusion: 27 years of planting seeds. In *20th ACM Symposium on Principles of Distributed Computing*, pages 3–12, Association for Computing Machinery, Newport, RI, August 2001.

[2] Anish Arora and Sandeep S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. In *18th International Conference on Distributed Computing Systems*, pages 436–443, IEEE Computer Society, Amsterdam, The Netherlands, 1998.

[3] Günther Bauer and Michael Paulitsch. An investigation of membership and clique avoidance in TTP/C. In *19th Symposium on Reliable Distributed Systems*, Nuremberg, Germany, October 2000.

[4] Kai Baukus, Saddek Bensalem, Yassine Lakhnech, and Karsten Stahl. Abstracting WS1S systems to verify parameterized networks. In Susanne Graf and Michael Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, pages 188–203, Berlin, Germany, March 2000.

[5] Kai Baukus, Yassine Lakhnech, and Karsten Stahl. Verifying universal properties of parameterized networks. In Matthai Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Volume 1926 of Springer-Verlag *Lecture Notes in Computer Science*, pages 291–303, Pune, India, September 2000.

[6] Saddek Bensalem, Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, and Yassine Lakhnech. A transformational approach for generating non-linear invariants. In Jens Palsberg, editor, *Seventh International Static Analysis Symposium (SAS'00)*, Volume 1824 of Springer-Verlag *Lecture Notes in Computer Science*, pages 58–74, Santa Barbara CA, June 2000.

[7] Saddek Bensalem and Yassine Lakhnech. Automatic generation of invariants. *Formal Methods in Systems Design*, 15(1):75–92, July 1999.

[8] Ahmed Bouajjani and Agathe Merceron. Parametric verification of a group membership algorithm. In Werner Damm and Ernst-Rüdiger Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Volume 2469 of Springer-Verlag *Lecture Notes in Computer Science*, pages 311–330, Oldenburg, Germany, November 2002.

[9] Tushar D. Chandra, Vassos Hadzilacos, Sam Toueg, and Bernadette Charron-Bost. On the impossibility of group membership. In *Fifteenth ACM Symposium on Principles of Distributed Computing*, pages 322–330, Association for Computing Machinery, Philadelphia, PA, May 1996.

[10] Ian G. Clark. *A Unifi ed Approach to the Study of Asynchronous Communication Mechanisms in Real Time Systems*. PhD thesis, King's College, London University, May 2000.

[11] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering*, pages 439–448, IEEE Computer Society, Limerick, Ireland, June 2000.

[12] S. J. Creese and A. W. Roscoe. TTP: A case study in combining induction and data independence. Technical Report PRG-TR-1-99, Oxford University Computing Laboratory, Oxford, England, 1999.

[13] Ben L. Di Vito and Ricky W. Butler. Formal techniques for synchronized fault-tolerant systems. In C. E. Landwehr, B. Randell, and L. Simoncini, editors, *Dependable Computing for Critical Applications—3*. Volume 8 of Springer-Verlag, Vienna, Austria *Dependable Computing and Fault-Tolerant Systems*, pages 163–188, September 1992.

[14] Jacob Elgaard, Nils Klarlund, and Anders Möller. Mona 1.x: New techniques for WS1S and WS2S. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verifi cation, CAV '98*, Volume 1427 of Springer-Verlag *Lecture Notes in Computer Science*, pages 516–520, Vancouver, Canada, June 1998.

[15] E. A. Emerson and A. P. Sistla, editors. *Computer-Aided Verifi cation, CAV '2000*, Volume 1855 of Springer-Verlag *Lecture Notes in Computer Science*, Chicago, IL, July 2000.

[16] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonization and Solving. In G. Berry, H. Comon, and A. Finkel, editors, *Computer-Aided Verifi cation, CAV '2001*, Volume 2102 of Springer-Verlag *Lecture Notes in Computer Science*, pages 246–249, Paris, France, July 2001.

[17] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[18] N. Henderson and S. E. Paynter. The formal classification and verification of Simpson's 4-slot asynchronous communication mechanism. In Peter Lindsay, editor, *FME 2002: Formal Methods–Getting IT Right*, pages 350–369, Copenhagen, Denmark, July 2002.

[19] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: a time-triggered language for embedded programming. In Henzinger and Kirsch [20], pages 166–184.

[20] Tom Henzinger and Christoph Kirsch, editors. *EMSOFT 2001: Proceedings of the First Workshop on Embedded Software*, Volume 2211 of Springer-Verlag *Lecture Notes in Computer Science*, Lake Tahoe, CA, October 2001.

[21] Wim H. Hesselink. An assertional criterion for atomicity. *Acta Informatica*, 28(5):343–366, 2002.

[22] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM TOPLAS*, 5(4):596–619, 1983.

[23] Shmuel Katz, Pat Lincoln, and John Rushby. Low-overhead time-triggered group membership. In Marios Mavronicolas and Philippas Tsigas, editors, *11th International Workshop on Distributed Algorithms (WDAG '97)*, Volume 1320 of Springer-Verlag *Lecture Notes in Computer Science*, pages 155–169, Saarbrücken Germany, September 1997.

[24] Herman Kopetz and R. Nossal. Temporal firewalls in large distributed real-time systems. In *6th IEEE Workshop on Future Trends in Distributed Computing*, pages 310–315, IEEE Computer Society, Tunis, Tunisia, October 1997.

[25] Hermann Kopetz. *Real-Time Systems: Design Princples for Distributed Embedded Applications*. The Kluwer International Series in Engineering and Computer Science. Kluwer, Dordrecht, The Netherlands, 1997.

[26] Hermann Kopetz. The time-triggered model of computation. In *Real Time Systems Symposium*, IEEE Computer Society, Madrid, Spain, December 1998.

[27] Hermann Kopetz. Elementary versus composite interfaces in distributed real-time systems. In *The Fourth International Symposium on Autonomous Decentralized Systems*, IEEE Computer Society, Tokyo, Japan, March 1999.

[28] Hermann Kopetz and Günter Grünsteidl. TTP—a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, January 1994.

[29] Hermann Kopetz and Johannes Reisinger. The non-blocking write protocol NBW: A solution to a real-time synchronization problem. In *Real Time Systems Symposium*, pages 131–137, IEEE Computer Society, Raleigh-Durham, NC, December 1993.

[30] Sandeep Kulkarni, John Rushby, and N. Shankar. A case study in component-based mechanical verification of fault-tolerant programs. In *ICDCS Workshop on Self-Stabilizing Systems*, pages 33–40, IEEE Computer Society, Austin, TX, June 1999.

[31] Sandeep S. Kulkarni. *Component-Based Design of Fault Tolerance*. PhD thesis, The Ohio State University, Columbus, OH, 1999.

[32] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[33] Leslie Lamport. Concurrent reading and writing. *Association for Computing Machinery*, 20(11):806–811, November 1977.

[34] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[35] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, San Francisco, CA, 1996.

[36] Keith Marzullo. Tolerating failures of continuous-valued sensors. *ACM Transactions on Computer Systems*, 8(4):284–304, November 1990.

[37] K. L. McMillan. Circular compositional reasoning about liveness. In Laurence Pierre and Thomas Kropf, editors, *Advances in Hardware Design and Verifi cation: IFIP WG10.5 International Conference on Correct Hardware Design and Verifi cation Methods (CHARME '99)*, Volume 1703 of Springer-Verlag *Lecture Notes in Computer Science*, pages 342–345, Bad Herrenalb, Germany, September 1999.

[38] Paul S. Miner. Verification of fault-tolerant clock synchronization systems. NASA Technical Paper 3349, NASA Langley Research Center, Hampton, VA, November 1993.

[39] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, July 1981.

[40] Kedar S. Namjoshi and Richard J. Trefler. On the completeness of compositional reasoning. In Emerson and Sistla [15], pages 139–153.

[41] 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.

[42] Michael Paulitsch and Wilfried Steiner. The transition from asynchronous to synchronous system operation: An approach for distributed fault-tolerant systems. In *The 22nd International Conference on Distributed Computing Systems (ICDCS '02)*, pages 329–336, IEEE Computer Society, Vienna, Austria, July 2002.

[43] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[44] Holger Pfeifer. Formal verification of the TTA group membership algorithm. In Tommaso Bolognesi and Diego Latella, editors, *Formal Description Techniques and Protocol Specification, Testing and Verification FORTE XIII/PSTV XX 2000*, pages 3–18, Pisa, Italy, October 2000.

[45] Holger Pfeifer, Detlef Schwier, and Friedrich W. von Henke. Formal verification for time-triggered clock synchronization. In Weinstock and Rushby [77], pages 207–226.

[46] Stefan Poledna. *Fault-Tolerant Systems: The Problem of Replica Determinism*. The Kluwer International Series in Engineering and Computer Science. Kluwer, Dordrecht, The Netherlands, 1996.

[47] John Rushby. The design and verification of secure systems. In *Eighth ACM Symposium on Operating System Principles*, pages 12–21, Asilomar, CA, December 1981. (ACM *Operating Systems Review*, Vol. 15, No. 5).

[48] John Rushby. A fault-masking and transient-recovery model for digital flight-control systems. In Jan Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Kluwer International Series in Engineering and Computer Science, chapter 5, pages 109–136. Kluwer, Boston, Dordecht, London, 1993.

[49] John Rushby. A formally verified algorithm for clock synchronization under a hybrid fault model. In *Thirteenth ACM Symposium on Principles of Distributed Computing*, pages 304–313, Association for Computing Machinery, Los Angeles, CA, August 1994. Also available as NASA Contractor Report 198289.

[50] John Rushby. Automated deduction and formal methods. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, Volume 1102 of Springer-Verlag *Lecture Notes in Computer Science*, pages 169–183, New Brunswick, NJ, July/August 1996.

[51] John Rushby. Partitioning for avionics architectures: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999. Available at `http://www.csl.sri.com/~rushby/abstracts/partitioning`, and `http://techreports.larc.nasa.gov/ltrs/PDF/1999/cr/NASA-99-cr209347.pdf`; also issued by the FAA.

[52] John Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660, September/October 1999.

[53] John Rushby. Verification diagrams revisited: Disjunctive invariants for easy verification. In Emerson and Sistla [15], pages 508–520.

[54] John Rushby. Bus architectures for safety-critical embedded systems. In Henzinger and Kirsch [20], pages 306–323.

[55] John Rushby. A comparison of bus architectures for safety-critical embedded systems. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, September 2001. Available at `http://www.csl.sri.com/~rushby/abstracts/buscompare`.

[56] John Rushby. Formal verification of McMillan's compositional assume-guarantee rule. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, September 2001.

[57] John Rushby. Formal verification of transmission window timing for the time-triggered architecture. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, March 2001.

[58] John Rushby. Formal verification of Marzullo's sensor fusion interval. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, January 2002.

[59] John Rushby. Model checking Simpson's four-slot fully asynchronous communication mechanism. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, July 2002.

[60] John Rushby and Friedrich von Henke. Formal verification of algorithms for critical systems. *IEEE Transactions on Software Engineering*, 19(1):13–23, January 1993.

[61] John Rushby, Friedrich von Henke, and Sam Owre. An introduction to formal specification and verification using EHDM. Technical Report SRI-CSL-91-2, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1991.

[62] Hassen Saïdi and Susanne Graf. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer-Aided Verifi cation, CAV '97*, Volume 1254 of Springer-Verlag *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997.

[63] Hassen Saïdi and N. Shankar. Abstract and model check while you prove. In Nicolas Halbwachs and Doron Peled, editors, *Computer-Aided Verifi cation, CAV '99*, Volume 1633 of Springer-Verlag *Lecture Notes in Computer Science*, pages 443–454, Trento, Italy, July 1999.

[64] Ulrich Schmid. How to model link failures: A perception-based fault model. In *The International Conference on Dependable Systems and Networks*, pages 57–66, IEEE Computer Society, Goteborg, Sweden, July 2001.

[65] Ulrich Schmid and Klaus Schossmaier. How to reconcile fault-tolerant interval intersection with the Lipschitz condition. *Distributed Computing*, 14(2):101–111, May 2001.

[66] Fred B. Schneider. Understanding protocols for Byzantine clock synchronization. Technical Report 87-859, Department of Computer Science, Cornell University, Ithaca, NY, August 1987.

[67] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[68] Marco Schneider. Self stabilization. *ACM Computing Surveys*, 25(1):45–67, March 1993.

[69] D. Schwier and F. von Henke. Mechanical verification of clock synchronization algorithms. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Volume 1486 of Springer-Verlag *Lecture Notes in Computer Science*, pages 262–271, Lyngby, Denmark, September 1998.

[70] Natarajan Shankar. Mechanical verification of a generalized protocol for Byzantine fault-tolerant clock synchronization. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Volume 571 of Springer-Verlag *Lecture Notes in Computer Science*, pages 217–236, Nijmegen, The Netherlands, January 1992.

[71] Natarajan Shankar. Combining theorem proving and model checking through symbolic analysis. In *CONCUR 2000: Concurrency Theory*, pages 1–16, State College, PA, August 2000. Available at `ftp://ftp.csl.sri.com/pub/users/shankar/concur2000.ps.gz`.

[72] H. R. Simpson. Four-slot fully asynchronous communication mechanism. *IEE Proceedings, Part E: Computers and Digital Techniques*, 137(1):17–30, January 1990.

[73] T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.

[74] Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *7th Symposium on Reliable Distributed Systems*, pages 93–100, IEEE Computer Society, Columbus, OH, October 1988.

[75] *Specifi cation of the TTP/C Protocol (version 0.6p0504)*. Time-Triggered Technology TTTech Computertechnik AG, Vienna, Austria, May 2001.

[76] Ashish Tiwari, Harald Rueß, Hassen Saïdi, and N. Shankar. A technique for invariant generation. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001*, Volume 2031 of Springer-Verlag *Lecture Notes in Computer Science*, pages 113–127, Genova, Italy, April 2001.

[77] Charles B. Weinstock and John Rushby, editors. *Dependable Computing for Critical Applications—7*, Volume 12 of IEEE Computer Society *Dependable Computing and Fault Tolerant Systems*, San Jose, CA, January 1999.

[78] J. Lundelius Welch and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, April 1988.

[79] Matthew M. Wilding, David S. Hardin, and David A. Greve. Invariant performance: A statement of task isolation useful for embedded application integration. In Weinstock and Rushby [77], pages 287–300.