FINITE-STATE ANALYSIS OF SECURITY PROTOCOLS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Vitaly Shmatikov
May 2000

Copyright by Vitaly Shmatikov 2000
All Rights Reserved

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

John C. Mitchell
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

David L. Dill

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Dan Boneh

Approved for the University Committee on Graduate Studies:

Abstract

Security protocols are notoriously difficult to design and debug. Even if the cryptographic primitives underlying a protocol are secure, unexpected interactions between parts of the protocol or several instances of the same protocol can lead to catastrophic security breaches. Since protocol attacks tend to be very subtle and hard to catch during the design and analysis process, some computer assistance is desirable.

The main contribution of this thesis is to demonstrate how fully automatic finite-state techniques can be used to analyze a wide variety of security protocols. While the general methodology of finite-state analysis, also referred to as model-checking, is well established in protocol verification, its application to security protocols is a recent area of research. Therefore, we present several case studies in which security protocols are simplified and formally modeled as finite-state systems. Automatic exhaustive state search is then performed. The search either discovers a state in which protocol correctness conditions are violated - the sequence of steps leading to such a state from the start state represents a successful attack on the protocol, - or proves the protocol correct subject to the limitations of the model.

In our first study, we analyze SSL 3.0, a real-world Internet security protocol that is widely deployed and used by millions of computer users. In the second study, we focus on fair exchange. Unlike the secrecy and authentication protocols that have traditionally been the subject of formal security analysis, fair exchange protocols are designed to guarantee properties such as fairness and accountability. Modeling these properties, and using automatic tools to discover attacks are important steps towards establishing finite-state analysis as a valid component of the protocol design process.

The formal adversary model used in the case studies is relatively simple, treating the underlying cryptography as a perfect "black box." Nevertheless, the model is surprisingly effective for discovering previously unknown weaknesses and attacks on published security protocols.

All analyses described in this thesis were performed using a general-purpose finite-state tool called Murφ. To alleviate the state-space explosion problem, we developed several state reduction techniques that exploit fundamental properties of security protocols to reduce the size of the state graph that has to be explored by several orders of magnitude. These optimizations make analysis

of large protocols feasible, and establish Murφ as a viable protocol analysis tool.

Acknowledgements

This thesis would not have been possible without support and guidance of my advisor, John Mitchell. I owe him a debt of gratitude for helping me choose security protocol analysis as my thesis topic, providing advice and encouragement along the way, sponsoring my trips to conferences, paying for whatever computer equipment struck my fancy, and, last but not the least, putting up with long periods of idleness, procrastination, and inactivity when my energy was dissipated on other pursuits.

The Hertz Foundation generously paid for 5 of my 6 years at Stanford, supporting what I consider - from a graduate student perspective - a positively luxurious lifestyle and asking for virtually nothing in return.

I am deeply grateful to Uli Stern who was my principal collaborator for the first two years of my research on security protocol analysis. He introduced me to the field of protocol verification, sparked many ideas, and made an important contribution to some of the results presented in this thesis.

I owe special thanks to the members of my reading committee, David Dill and Dan Boneh, for graciously agreeing to review this thesis, and their helpful comments.

Amit Patel and Steve Freund performed the unimaginable feat of tolerating me as an officemate for many years in spite of my taking over the better half of the office and subjecting them to hours of phone conversations in a language they could not understand.

I am grateful that I had the honor to work with Viviana Bono and Amit on topics in programming language theory of which they knew much more than I did. I learned a lot from them. Many friends and colleagues at Stanford made it a place that I will sorely miss. Of these, Leonid Litvak and Dominic Hughes deserve a special mention.

Finally, none of this would have happened were it not for my parents who kept as close a track of my progress at Stanford as was humanly possible from 6,000 miles away. They were a sure source of support when my confidence flagged. My PhD is their achievement to a much larger degree than they realize.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background and motivation

A communication *protocol* is a set of rules. It defines the format of data transmitted between two or more *agents* - typically, computers or other electronic devices. As public computer networks such as the Internet become an increasingly important medium of information exchange, protocols have to operate in an environment where not every agent is trusted, and some of the agents may have a malicious intent. The main purpose of a *security protocol* is to define a set of rules for transmitting information in a way that prevents a malicious agent(s) - who may be actively involved in communication or passively observing network traffic - from inflicting damage upon honest protocol participants. Typically, a security protocol relies on one or more *cryptographic functions* such as encryption, digital signature scheme, etc. to achieve its purpose.

Protection of information is a broad concept. Depending on the situation, it may involve *secrecy* - roughly, ensuring that a particular piece of data cannot be accessed by an unauthorized agent. In a network where the identity of the protocol counterparty cannot be established directly, *authentication* is an important task. If the goal of the protocol is to enable two parties who do not necessarily trust each other to exchange items of value, mutual *fairness* must be guaranteed. A security protocol may be required to provide *accountability*. In this case, the protocol must furnish every participant with sufficient evidence so that a misbehaving party can be identified and proved guilty. A number of other properties have been proposed by protocol designers and users.

Determining whether a protocol indeed guarantees a particular security property is a difficult task. Humans are fallible. There are known cases of protocols where errors were discovered 20 years after the protocol had been unveiled to public review [36]. Not all of the protocol errors are due to flawed cryptography. Even if the cryptographic primitives underlying a protocol are secure, unexpected interactions between parts of the protocol or several instances of the same protocol can lead to catastrophic security breaches. Therefore, some form of computer assistance in finding attacks on protocols and proving protocols correct is desirable.

An extensive body of formal methods has been developed for verification of hardware and software systems. Applied to a security protocol, they can be used to establish whether the protocol guarantees the security properties intended by its designers. Formal methods are not a replacement for human analysis. The class of attacks they are capable of finding is necessarily limited. Therefore, "proofs" of correctness produced by means of formal analysis should always be viewed with caution. Nevertheless, application of formal methods to security protocol analysis is useful insofar as it helps discover errors in protocol design at an early stage and prevents deployment of flawed protocols.

This thesis focuses on a particular flavor of formal analysis: *finite-state analysis*, also referred to as model-checking. After the protocol is represented as an abstract finite-state system and the protocol's security guarantees are expressed as state invariants, the finite-state tool automatically explores the entire state graph reachable from the initial state. If there exists a reachable state in which an invariant is violated, the path from the initial state to the violating state represents an attack on the protocol.

For large protocols, manual verification of all possible interactions between protocol participants is impossible. Finite-state tools provide a useful service to the protocol analyst by automatically enumerating all protocol interactions and verifying that the desired properties hold in every case. This approach ideally complements more advanced techniques that rely on detailed human analysis.

To represent a protocol as an abstract finite-state model, it is necessary to clearly specify all message sequences and security properties. This translation from an informal specification published as an RFC or academic paper into a simple, precise model can be very valuable. Many protocol errors result from a misunderstanding of the cryptographic and other assumptions underlying the protocol. Creation of a finite-state model for the protocol requires careful specification of all assumptions and thus helps discover such errors. Examples can be found in the case studies below. The resulting abstract model also has an independent value since it can be used by other protocol analysis techniques.

Finite-state analysis is conceptually simple and fully automatic, and can be employed even by protocol designers who are not formal methods experts. The class of attacks that can be found is, however, limited. The protocol model analyzed by the finite-state tool must treat cryptography at an abstract level, typically ignoring the number-theoretic properties of the underlying cryptographic functions. Because the state graph must be finite, the model must impose a finite bound on the number of protocol instances, principals, and applications of cryptographic functions, potentially missing some attacks.

The purpose of this thesis is to demonstrate that finite-state techniques can be successfully used to analyze a wide variety of security protocols, uncovering nontrivial insights and, in some cases, attacks that have been overlooked by human analysts. Therefore, case studies form the core of the thesis. We start by giving an overview of Murφ, the finite-state analysis tool used in the rest of the thesis, in Chapter 2. A case study of the SSL 3.0 protocol, the de facto standard for secure Internet communications, is presented in Chapter 3. In Chapter 4, we describe state reduction techniques and Murφ optimizations that make analysis of large protocols feasible. Chapter 5 describes application of finite-state analysis to fair exchange protocols, using two online contract signing protocols as case studies. Finally, conclusions appear in Chapter 6.

The key contributions presented in this thesis have previously appeared in a series of conference and workshop publications [46, 54, 52, 53]. The basic methodology of using Murφ for finite-state analysis of security analysis was first developed by Mitchell, Mitchell, and Stern [45].

1.2 Related work

Formal methods have been extensively used to analyze key exchange and authentication protocols. In this section, we give a brief overview of the field, focusing on the methods that are most similar to the one used in this thesis. Comparative studies of different approaches can be found in [32] and [42].

1.2.1 Finite-state analysis

FDR

The Failure Divergences Refinement Checker (FDR) [39, 49], is a general-purpose model checker for Communicating Sequential Processes (CSP) [27]. To analyze a protocol with FDR, it is necessary to model protocol participants as CSP processes, add the adversary model, and use FDR to test whether the implementation of the protocol refines the specification by exhaustively searching through all states reachable by the processes. The general methodology of security protocol analysis with FDR was developed by Lowe, Roscoe, et al. [50, 37, 51] and led to some early successes in discovering attacks on published protocols, such as Lowe's attack on the Needham-Schroeder Public-Key authentication protocol [36, 37]. Lowe also developed a tool called Casper [38] for automatic construction of the adversary model from the high-level protocol specification. FDR is fully automatic, but suffers from the usual drawbacks of finite-state analysis such as the need to impose finite limits on system parameters and exponential state space explosion.

Brutus

Brutus is a special-purpose model checker developed by Clarke, Jha, and Marrero [41, 11]. It was designed especially for security protocol analysis. Unlike general-purpose model checkers such as FDR, Brutus explicitly maintains the knowledge corresponding to the principals and the adversary. By representing this knowledge as a set of atomic facts and rewrite rules, Brutus can implicitly represent an infinite set of facts.

Interrogator

Interrogator [44] is a special-purpose model checker for security protocols developed by Jonathan Millen. Starting from an insecure state, it uses backward search to look for a path from the initial state. If a path is found, it corresponds to a successful attack on the protool. If the search fails, however, nothing can be concluded about the security of the protocol.

1.2.2 Hybrid approaches

Athena

Athena [55], developed by Dawn Song, represents protocols using the Strand Space Model [21] of Thayer Fabrega et al. Protocol properties are specified as logic formulas, and an automatic proof search is performed using both model checking and theorem proving techniques. If a counterexample is found, an attack on the protocol can be constructed. Because it represents protocols using the Strand Space Model, Athena can verify an arbitrary number of concurrent protocol instances, and the state space to be searched is typically smaller than for pure finite-state analysis tools. However, there is no guarantee that proof search will terminate.

NRL Protocol Analyzer

The NRL Protocol Analyzer [43] by Meadows is a special-purpose verification tool for security protocols. It uses a combination of model checking and theorem proving. NRL Protocol Analyzer starts from an insecure goal state and searches the state space backwards to prove that the goal state in unreachable. Theorem-proving techniques are used to reduce the size of the state space to be searched. Protocol Analyzer can prove protocol properties for an arbitrary number of protocol instances, but analysis is not fully automatic, and there is no guarantee of termination.

Bolignano

Bolignano [6, 7] proposed a verification technique that combines model-checking with abstraction to enable fully automatic analysis of security protocols. First, an abstract model of the protocol is constructed and protocol properties are translated into properties of the abstract model. Then, exhaustive search is performed on the finite state space of the abstract model to prove that properies hold in every reachable state. The advantage of this approach is that infinite state spaces can be collapsed into the finite-state space of the abstract model, and security properties of the protocols can thus be proved in the general case, without limitations on the number of instances, etc. The drawback is that a safe abstraction may be difficult to construct for a given protocol property.

1.2.3 Theorem proving

Isabelle

Paulson [48] applied Isabelle, a general-purpose theorem prover for higher-order logics, to verify properties of security protocols by induction on protocol traces. While analysis can be performed for an arbitrary number of instances, there is no guarantee of termination, and it may be difficult to reconstruct the attack on the protocol from a failed proof.

Logics of authentication

Belief logics such as BAN [9] and GNY [25] have been proposed for reasoning about properties of security protocols. Proofs are typically constructed by hand, and are error-prone. Kindred and Wing developed a technique for automatic theory generation for belief logics [33].

Chapter 2

Overview of the Murφ Verifier

The general methodology for modeling security protocols with the Murφ finite-state analysis tool was developed by Mitchell, Mitchell, and Stern [45]. In this chapter, we outline the basic approach to protocol analysis with Murφ.

2.1 The Murφ verification system

Murφ [16] is a protocol or, more generally, finite-state machine verification tool. It has been successfully applied to several industrial protocols, especially in the domains of multiprocessor cache coherence protocols and multiprocessor memory models [17, 57, 62]. The purpose of finite-state analysis, commonly called "model checking," is to exhaustively search all execution sequences. While this process often reveals errors, failure to find errors does not imply that the protocol is completely correct, because the Murφ model may simplify certain details and is inherently limited to configurations involving a small number of, say, clients and servers.

To use Murφ for verification, one has to model the protocol in the Murφ language and augment this model with a specification of the desired properties. The Murφ system automatically checks, by explicit state enumeration, if all reachable states of the model satisfy the given specification. For the state enumeration, either breadth-first or depth-first search can be selected. Reached states are stored in a hash table to avoid redundant work when a state is revisited. The memory available for this hash table typically determines the largest tractable problem.

The Murφ language is a simple high-level language for describing nondeterministic finite-state machines. Many features of the language are familiar from conventional programming languages. The main features not found in a "typical" high-level language are described in the following paragraphs.

The *state* of the model consists of the values of all global variables. In a *startstate* statement, initial values are assigned to global variables. The transition from one state to another is performed by *rules*. Each rule has a Boolean condition and an action, which is a program segment that is executed atomically. The action may be executed if the condition is true (*i.e.*, the rule is enabled) and typically changes global variables, yielding a new state. Most Murφ models are nondeterministic since states typically allow execution of more than one rule.

Murφ has no explicit notion of *processes*. Nevertheless a process can be implicitly modeled by a set of related rules. The *parallel composition* of two processes in Murφ is simply done by using the union of the rules of the two processes. Each process can take any number of steps (actions) between the steps of the other. The resulting computational model is that of *asynchronous, interleaving* concurrency. Parallel processes communicate via shared variables; there are no special language constructs for communication.

The Murφ language supports *scalable* models. In a scalable model, one is able to change the size of the model by simply changing constant declarations. When developing protocols, one typically starts with a small protocol configuration. Once this configuration is correct, one gradually increases the protocol size to the largest value that still allows verification to complete. In many cases, an error in the general (possibly infinite state) protocol will also show up in a downscaled (finite-state) version of the protocol. Murφ can only guarantee correctness of the downscaled version of the protocol, but not correctness of the general protocol. The Murφ verifier supports automatic *symmetry* reduction of models by special language constructs [29].

The desired properties of a protocol can be specified in Murφ by invariants, which are Boolean conditions that have to be true in every reachable state. If a state is reached in which some invariant is violated, Murφ prints an error trace – a sequence of states from the start state to the state exhibiting the problem.

2.2 The methodology

In outline, we have analyzed protocols using the following sequence of steps:

1. *Formulate the protocol.* This generally involves simplifying the protocol by identifying the key steps and primitives. The Murφ formulation of a protocol, however, is more detailed than the high-level descriptions often seen in the literature, since one has to decide exactly which messages will be accepted by each participant in the protocol. Since Murφ communication is based on shared variables, it is also necessary to define an explicit message format as a Murφ type.

2. *Add an adversary to the system.* We generally assume that the adversary (or intruder) can masquerade as an honest participant in the system, capable of initiating communication with a truly honest participant, for example. We also assume that the network is under control of the adversary and allow the adversary the following actions:

- overhear every message, remember all parts of each message, and decrypt ciphertext when it has the key,
- intercept (delete) messages,
- generate messages using any combination of initial knowledge about the system and parts of overheard messages.

Although it is simplest to formulate an adversary that nondeterministically chooses between all possible actions at every step of the protocol, it is more efficient to reduce the choices to those that actually have a chance of affecting other participants.

3. *State the desired correctness condition.* A typical correctness criterion includes, *e.g.*, that no secret information can be learned by the intruder. Examples of correctness conditions can be found in the case studies below.

4. *Run the protocol* for some specific choice of system size parameters. If Murφ finds a reachable state in which an invariant is violated, it outputs the sequence of rules leading to it from the start state. This sequence effectively describes the attack.

2.3 Intruder model

The intruder model described above is limited in its capabilities and does not have all the power that a real-life intruder may have. In the following, we discuss examples of these limitations.

No cryptanalysis. Our intruder ignores both computational and number-theoretic properties of cryptographic functions. As a result, it cannot perform any cryptanalysis whatsoever. If it has the proper key, it can read an encrypted message (or forge a signature). Otherwise, the only action it can perform is to store the message for a later replay. We do not model any cryptographic attacks such as brute-force key search (with a related notion of computational time required to attack the encryption) or attacks relying on the mathematical properties of cryptographic functions.

No probabilities. Murφ has no notion of probability. Therefore, we do not model "propagation" of attack probabilities through our finite-state system (*e.g.*, how the probabilities of breaking the encryption, forging the signature, etc. accumulate as the protocol progresses). We also ignore, *e.g.*, that the intruder may learn some probabilistic information about the participants' keys by observing multiple runs of the protocol.

No partial information. Keys, nonces, etc. are treated as atomic entities in our model. Our intruder cannot break such data into separate bits. It also cannot perform an attack that results in the partial recovery of a secret (*e.g.*, half of the secret bits).

We will refer to this intruder model as the Dolev-Yao intruder, following [18]. A detailed study of the model can be found in [19].

In spite of the above limitations, we believe that Murφ is a useful tool for analyzing security protocols. It considers the protocol at a high level and helps discover a certain class of errors that do not involve attacks on cryptographic functions employed in the protocol. For example, Murφ is useful for discovering "authentication" bugs, where the assumptions about key ownership, source of messages, etc. are implicit in the protocol but never verified as part of the message exchange. Also, Murφ models can successfully discover attacks on plaintext information (such as version rollback attacks in SSL) and implicit assumptions about message sequence in the protocol (such as unacknowledged receipt of *Finished* messages in SSL – see Chapter 3). Murφ discovered a previously unknown attack on the Garay-Jakobsson-MacKenzie abuse-free contract signing protocol, and a weakness in the Asokan-Shoup-Waidner optimistic contract signing protocol (see Chapter 5).

2.4 Efficiency

Murφ implements a rich set of methods for increasing the size of the protocols that can be verified – consisting of both several techniques to reduce the number of reachable states [28] and several techniques to perform the state space search more efficiently, reducing runtime and memory requirements [56], including symmetry reduction [29], hash compaction [58], reversible rules [30], and repetition constructors [31]. In addition, there is a parallel version of the Murφ verifier [59].

Our experience with using Murφ for security protocol analysis has led us to design and implement several optimization techniques that are specific to security protocols and improve the tool's performance significantly when analyzing a typical security protocol. A detailed description of these techniques and the corresponding soundness proofs can be found in section 4 below.

Chapter 3

Case Study: SSL 3.0

In an effort to understand the difficulties involved in applying the approach described in Chapter 2 to larger and more complex protocols, we use Murφ to analyze the SSL 3.0 handshake protocol.

3.1 Overview

The SSL 3.0 protocol is the de facto standard for secure Internet communication. Analyzing the protocol is a challenge, since it has more steps and greater complexity than the other security protocols analyzed using automatic finite-state enumeration. In addition to demonstrating that finite-state analysis is feasible for protocols of this complexity, our study also points to several anomalies in SSL 3.0. However, we have not demonstrated the possibility of compromising sensitive data in any implementation of the protocol.

In the process of analyzing SSL 3.0, we have developed a "rational reconstruction" of the protocol. More specifically, after initially attempting to familiarize ourselves with the handshake protocol, we found that we could not easily identify the purpose of each part of certain messages. Therefore, we set out to use our analysis tool to identify, for each message field, an attack that could arise if that field were omitted from the protocol. Arranging the simplified protocols in the order of increasing complexity, we obtain an incremental presentation of SSL. Beginning with a simple, intuitive, and insecure exchange of the required data, we progressively introduce signatures, hashed data, and additional messages, culminating in a close approximation of the actual SSL 3.0 handshake protocol.

In addition to allowing us to understand the protocol more fully in a relatively short period of time, this incremental reconstruction also provides some evidence for the "completeness" of our analysis. Specifically, Murφ exhaustively tests all possible interleavings of protocol and intruder actions, making sure that a set of correctness conditions is satisfied in all cases. It is easy for such analysis to be "incomplete" by not checking all of the correctness conditions intended by the protocol designers or users. In developing our incremental reconstruction of SSL 3.0, we were forced to confirm the importance of each part of each message. In addition, since no formal or high-level description of SSL 3.0 was available, we believe that the description of SSL 3.0 that we extracted from the Internet Draft [23] may be of interest.

Our analysis covers both the standard handshake protocol used to initiate a secure session and the shorter protocol used to resume a session [23, Section 5.5]. Murφ analysis uncovered a weak form of version rollback attack (see Section 3.3.9) that can cause a version 3.0 client and a version 3.0 server to commit to SSL 2.0 when the protocol is resumed. Another attack on the resumption protocol (described in Sections 3.3.8 and 3.3.9) is possible in SSL implementations that strictly follow the Internet Draft [23] and allow the participants to send application data without waiting for an acknowledgment of their *Finished* messages. Finally, an attack on cryptographic preferences (see Section 3.3.6) succeeds if the participants support weak encryption algorithms which can be broken in real time. Apart from these three anomalies, we were not able to uncover any errors in our final protocol. Since SSL 3.0 was designed to be backward-compatible, we also implemented and checked a full model for SSL 2.0 as part of the SSL 3.0 project. In the process, Murφ uncovered the major problems with SSL 2.0 that motivated the design of SSL 3.0.

Our Murφ analysis of SSL is based on the assumption that cryptographic functions cannot be broken. For this and other reasons (discussed below), we cannot claim that we found all attacks on SSL. But our analysis has been efficient in helping discover an important class of attacks.

The two prior analyses of SSL 3.0 that we are aware of are an informal assessment carried out by Wagner and Schneier [61] and a formal analysis by Dietrich using a form of belief logic [15]. (We read the Wagner and Schneier study before carrying out our analysis, but did not become aware of the Dietrich study until after we had completed the bulk of our work.) Wagner and Schneier comment on the possibility of anomalies associated with resumption, which led us to concentrate our later efforts on this area. It is not clear to us at the time of this writing whether we found any resumption anomalies that were not known to these investigators. However, in email comments resulting from circulation of an earlier document [60], we learned that while our second anomaly was not noticed by Wagner and Schneier, it was later reported to them by Michael Wiener. Neither anomaly seems to have turned up in the logic-based study of Dietrich [15].

3.2 The SSL 3.0 handshake protocol

The primary goal of the SSL 3.0 handshake protocol is to establish secret keys that "provide privacy and reliability between two communicating applications" [23]. Henceforth, we call the communicating applications the client (C) and the server (S). The basic approach taken by SSL is to have C generate a fresh random number (the *secret* or *shared secret*) and deliver it to S in a secure manner. The secret is then used to compute a so-called *master secret* (or *negotiated cipher*), from which, in turn, the keys that protect and authenticate subsequent communication between C and S are computed. While the SSL *handshake protocol* governs the secret key computation, the SSL *record layer protocol* governs the subsequent secure communication between C and S.

As part of the handshake protocol, C and S exchange their respective *cryptographic preferences*, which are used to select a mutually acceptable set of algorithms for encrypting and signing handshake messages. In our analysis, we assume for simplicity that RSA is used for both encryption and signatures, and cryptographic preferences only indicate the desired lengths of keys. In addition, SSL 3.0 is designed to be backward-compatible so that a 3.0 server can communicate with a 2.0 client and vice versa. Therefore, the parties also exchange their respective version numbers.

The basic handshake protocol consists of three messages. With the *ClientHello* message, the client starts the protocol and transmits its version number and cryptographic preferences to the server. The server replies with the *ServerHello* message, also transmitting its version number and cryptographic preferences. Upon receipt of this message, the client generates the shared secret and sends it securely to the server in the *secret exchange message*.

Since we were not aware of any formal definition of SSL 3.0, we based our model of the handshake protocol on the Internet Draft [23]. The Draft does not include a precise list of requirements that must be satisfied by the communication channel created after the handshake protocol completes. Based on our interpretation of the informal discussion in Sections 1 and 5.5 of the Internet Draft, we believe that the resulting channel can be considered "secure" if and only if the following properties hold:

- Let $Secret_C$ be the number that C considers the shared secret, and $Secret_S$ the number that S considers the shared secret. Then $Secret_C$ and $Secret_S$ must be identical.

- The secret shared between C and S is not in intruder's database of known message components.

- The parties agree on each other's identity and protocol completion status. Suppose that the last message of the handshake protocol is from S to C. Then C should reach the state ($Done_C$) in which it is ready to start communicating with S using the negotiated cipher only if S is already in the state ($Done_S$) in which it is ready to start communicating with C using the negotiated cipher. Conversely, S should reach the state $Done_S$ only if C is in the state in which it is waiting for the last message of the handshake protocol.

- The cryptographic algorithms selected by the parties for encryption and authentication of handshake messages are the strongest ones that are supported by both C and S. We model this by requiring that the cryptosuite stored by S as C's cryptographic preferences is identical to the one actually sent by C, and vice versa.

- The parties have a consistent opinion about each other's version, *i.e.*, it is never the case that an SSL 3.0 client and a 3.0 server are communicating using the SSL 2.0 protocol.

We propose that any violation of the foregoing invariants that goes undetected by the legitimate participants constitutes a successful attack on the protocol.

SSL 3.0 supports *protocol resumption*. In the initial run of the protocol, C and S establish a shared secret by going through the full protocol and computing secret keys that protect subsequent communication. SSL 3.0 allows the parties to resume their connection at a later time without repeating the full protocol. If the *ClientHello* message sent by C to S includes the identifier of an SSL session that is still active according to S's internal state, S assumes that C wants to resume a previous session. No new secret is exchanged in this case, but the master secret and the keys derived from it are recomputed using new nonces. (See Section 3.3.8 for an explanation of how nonces are used in the protocol to prevent replay attacks, and Appendix B to see how the master secret is computed from the nonces and shared secret.) Our Murφ model supports protocol resumption.

Finally, it should be noted that whenever one of the parties detects an inconsistency in the messages it receives, or any of the protocol steps fails in transmission, the protocol is aborted and the parties revert to their initial state. This implies that SSL is susceptible by design to some forms of "denial of service" attacks: an intruder can simply send an arbitrary message to a client or server engaged in the handshake protocol, forcing protocol failure.

3.3 Modeling SSL 3.0

We start our incremental analysis with the simplest and most intuitive version of the protocol and give an attack found by Murφ. We then add a little piece of SSL 3.0 that foils the attack, and let Murφ discover an attack on the augmented protocol. We continue this iterative process until no more attacks can be found. The final protocol closely resembles SSL 3.0, with some simplifications that result from our assumption of perfect cryptography (see below).

3.3.1 Notation

The following notation will be used throughout this chapter.

Ver_i	SSL version number of party i
$Suite_i$	Cryptographic preferences of party i
N_i	Random nonce generated by party i
$Secret_i$	Random secret generated by party i
K_i^+	Public encryption key of party i
V_i	Public verification key of party i
$\mathrm{Sig}_i\{\ldots\}$	Signed by party i
$\{\ldots\}_{K_i^+}$	Encrypted by public key K_i^+
$Messages$	All messages up to this point
$\langle I \rangle$	Message is intercepted by the intruder

3.3.2 Assumptions about cryptography

In general, our model assumes perfect cryptography. The following list explains what this assumption implies for all cryptographic functions used in SSL.

Opaque encryption. Encryption is assumed to be opaque. If a message has the form $\{x\}_{K_i^+}$, only party i can learn x. (This is only true *iff* the private key K_i^- is not available to any party except i. This is a safe assumption, given that no participants in the SSL handshake protocol are ever required to send their private key over the network.) The intruder may, however, store the entire encrypted message and replay it later without learning x. The structure of the encrypted message is inaccessible to the intruder, *i.e.*, it cannot split the encrypted message into parts and insert them into other encrypted messages.

Unforgeable signatures. Signatures are assumed to be unforgeable. Messages of the form $\mathrm{Sig}_i\{x\}$ can only be generated by the party i. Anyone who possesses i's verification key V_i is able to verify that the message was indeed signed by i. We assume that signatures do not encrypt. Therefore, x can be learned by anyone.

Hashes. Hashes are assumed to be preimage resistant and 2nd-preimage resistant: given a message of the form $\mathrm{Hash}\{x\}$, it is not computationally feasible to discover x, nor find any x' such that $\mathrm{Hash}\{x'\} = \mathrm{Hash}\{x\}$. It is therefore assumed that a participant can determine whether $x = x'$ by comparing $\mathrm{Hash}\{x\}$ to $\mathrm{Hash}\{x'\}$.

Trusted certificate authority. There exists a trusted certificate authority (CA). All parties are assumed to possess CA's verification key V_{CA}, and are thus able to verify messages signed by CA. Every party i is assumed to possess CA-signed certificates for its own public keys: $\mathrm{Sig}_{CA}\{i, K_i^+\}$ (certifying that public encryption key K_i^+ indeed belongs to i) and $\mathrm{Sig}_{CA}\{i, V_i\}$ (certifying that public verification key V_i indeed belongs to i).

3.3.3 Protocol A

Basic protocol (A)

The first step of the basic protocol consists of C sending information about its identity, SSL version number, and cryptographic preferences (aka *cryptosuite*) to S. Upon receipt of C's *Hello* message, S sends back its version, cryptosuite (S selects one set of algorithms from the preference list submitted by C), and its public encryption key. C then generates a random secret and sends it to S, encrypted by S's public key.

Notice that the first *Hello* message (that from C to S) contains the identity of C. There is no way for S to know who initiated the protocol unless this information is contained in the message itself (perhaps implicitly in the network packet header).

$$
\begin{array}{ll}
C \rightarrow S & C,\ Ver_C,\ Suite_C \\
S \rightarrow C & Ver_S,\ Suite_S,\ K_S^+ \\
C \rightarrow S & \{Secret_C\}_{K_S^+}
\end{array}
$$

$\langle$Change to negotiated cipher$\rangle$

Attack on A

Protocol A does not explicitly (and securely) associate the server's name with its public encryption key. This allows the intruder to insert its own key into the server's *Hello* message. The client then encrypts the generated secret with the intruder's key, enabling the intruder to read the message and learn the secret.

$$
\begin{array}{ll}
C \rightarrow S & C,\ Ver_C,\ Suite_C \\
S \rightarrow C\langle I\rangle & Ver_S,\ Suite_S,\ K_S^+ \\
I \rightarrow C & Ver_S,\ Suite_S,\ \underline{K_I^+} \\
C \rightarrow S\langle I\rangle & \{Secret_C\}_{K_I^+} \\
I \rightarrow S & \{Secret_C\}_{K_S^+}
\end{array}
$$

⟨Change to negotiated cipher⟩

3.3.4 Protocol B

A + server authentication

To fix the bug in Protocol A, we add verification of the public key. The server now sends its public key K_S^+ in a certificate signed by the certificate authority. As described before, the certificate has the following form: $\mathrm{Sig}_{CA}\{S, K_S^+\}$.

We assume that signatures are unforgeable. Therefore, the intruder will not be able to generate $\mathrm{Sig}_{CA}\{S, K_I^+\}$. The intruder may send the certificate for its own public key $\mathrm{Sig}_{CA}\{I, K_I^+\}$, but the client will reject it since it expects S's name in the certificate. Finally, the intruder may generate $\mathrm{Sig}_I\{S, K_I^+\}$, but the client expects a message signed by CA, and will try to verify it using CA's verification key. Verification will fail since the message is not signed by CA, and the client will abort the protocol. Notice that SSL's usage of certificates to verify the server's public key depends on the trusted certificate authority assumption (see Section 3.3.2 above).

$$
\begin{array}{ll}
C \rightarrow S & C,\ Ver_C,\ Suite_C \\
S \rightarrow C & Ver_S,\ Suite_S,\ \mathrm{Sig}_{CA}\{S, K_S^+\} \\
C \rightarrow S & \{Secret_C\}_{K_S^+}
\end{array}
$$

⟨Change to negotiated cipher⟩

Attack on B

Protocol B includes no verification of the client's identity. This allows the intruder to impersonate the client by generating protocol messages and pretending they originate from C. In particular, the intruder is able to send its own secret to the server, which the latter will use to compute the master secret and the derived keys.

$$\underline{I} \rightarrow S \qquad C,\ Ver_C,\ Suite_C$$

$$S \rightarrow C\langle I \rangle \qquad Ver_S,\ Suite_S,\ \mathrm{Sig}_{CA}\{S, K_S^+\}$$

$$I \rightarrow S \qquad \{\underline{Secret_I}\}_{K_S^+}$$

⟨Change to negotiated cipher⟩

3.3.5 Protocol C

B + client authentication

To fix the bug in Protocol B, the server has to verify that the secret it received was indeed generated by the party whose identity was specified in the first *Hello* message. For this purpose, SSL employs client signatures.

The client sends to the server its verification key in the CA-signed certificate $\mathrm{Sig}_{CA}\{C, V_C\}$. In addition, immediately after sending its secret encrypted with the server's public key, the client signs the hash of the secret $\mathrm{Sig}_C\{\mathrm{Hash}\,(Secret_C)\}$ and sends it to the server. Hashing the secret is necessary so that the intruder will not be able to learn the secret even if it intercepts the message. Since the server can learn the secret by decrypting the client key exchange message, it is able to compute the hash of the secret and compare it with the one sent by the client.

Notice that the server can be assured that V_C is indeed C's verification key since the intruder cannot insert its own key in the CA-signed certificate $\mathrm{Sig}_{CA}\{C, V_C\}$ assuming that signatures are unforgeable. Therefore, the server will always use V_C to verify messages ostensibly signed by the client, and all messages of the form $\mathrm{Sig}_I\{\ldots\}$ will be rejected. Even if the intruder were able to generate the message $\mathrm{Sig}_C\{\mathrm{Hash}\,(Secret_I)\}$, the attack will be detected when the server computes $\mathrm{Hash}\,(Secret_C)$ and discovers that it is different from $\mathrm{Hash}\,(Secret_I)$.

Instead of signing the hashed secret, the client can sign the secret directly and send it to the server encrypted by the server's public key. The SSL definition, however, does not include encryption in this step [23, Section 5.6.8]. We used hashing instead of encryption as well since we intend our incremental reconstruction of SSL to follow the definition as closely as possible. One of the anonymous reviewers of the conference version of this paper suggested that hashing is used instead of encryption so that the encrypted part of the message (*i.e.*, a secret as opposed to a signed secret) fits within the modulus size of the server's encryption function.

$$
\begin{array}{ll}
C \rightarrow S & C,\ Ver_C,\ Suite_C \\[1em]
S \rightarrow C & Ver_S,\ Suite_S,\ \mathrm{Sig}_{CA}\{S, K_S^+\} \\[1em]
C \rightarrow S & \mathrm{Sig}_{CA}\{C, V_C\},\ \{Secret_C\}_{K_S^+}, \\
 & \mathrm{Sig}_C\{\mathrm{Hash}\,(Secret_C)\}
\end{array}
$$

⟨Change to negotiated cipher⟩

Attack on C

Even though the intruder can modify neither keys, nor shared secret in Protocol *C*, it is able to attack the plaintext information transmitted in the *Hello* messages. This includes the parties' version numbers and cryptographic preferences.

By modifying version numbers, the intruder can convince an SSL 3.0 client that it is communicating with a 2.0 server, and a 3.0 server that it is communicating with a 2.0 client. This will cause the parties to communicate using SSL 2.0, giving the intruder an opportunity to exploit any of the known weaknesses of SSL 2.0.

By modifying the parties' cryptographic preferences, the intruder can force them into selecting a weaker encryption and/or signing algorithm than they normally would. This may make it easier for the intruder to decrypt the client's secret exchange message, or to forge the client's signature.

$$
\begin{array}{ll}
C \rightarrow S\langle I\rangle & C,\ Ver_C,\ Suite_C \\[1em]
I \rightarrow S & C,\ \underline{Ver_I},\ \underline{Suite_I} \\[1em]
S \rightarrow C\langle I\rangle & Ver_I,\ Suite_S,\ \mathrm{Sig}_{CA}\{S, K_S^+\} \\[1em]
I \rightarrow C & Ver_I,\ \underline{Suite_I},\ \mathrm{Sig}_{CA}\{S, K_S^+\} \\[1em]
C \rightarrow S & \mathrm{Sig}_{CA}\{C, V_C\},\ \{Secret_C\}_{K_S^+}, \\
 & \mathrm{Sig}_C\{\mathrm{Hash}\,(Secret_C)\}
\end{array}
$$

⟨Change to negotiated cipher⟩

3.3.6 Protocol D

C + post-handshake verification of plaintext

The parties can prevent attacks on plaintext by repeating the exchange of versions and cryptographic preferences once the handshake protocol is complete; the additional messages will be called *verification messages*. Since the intruder cannot learn the shared secret, it cannot compute the master secret and the derived keys and thus cannot interfere with the parties' communication after they switch to the negotiated cipher.

Suppose the intruder altered the cryptographic preferences in the client's *Hello* message. When the client sends its version and cryptosuite to the server under the just negotiated encryption, the intruder cannot change them. The server will detect the discrepancy and abort the protocol. This is also true for the server's version and cryptosuite.

$$
\begin{array}{ll}
C \to S & C,\ Ver_C,\ Suite_C \\[1em]
S \to C & Ver_S,\ Suite_S,\ \mathrm{Sig}_{CA}\{S, K_S^+\} \\[1em]
C \to S & \mathrm{Sig}_{CA}\{C, V_C\},\ \{Secret_C\}_{K_S^+}, \\
 & \mathrm{Sig}_C\{\mathrm{Hash}\,(Secret_C)\} \\[1em]
\multicolumn{2}{l}{\langle \textsf{Change to negotiated cipher} \rangle} \\[1em]
S \to C & \{\mathrm{Hash}\,(Ver_C, Suite_C,\ Ver_S, \\
 & \qquad Suite_S)\}_{\mathrm{Master}(Secret_C)} \\[1em]
C \to S & \{\mathrm{Hash}\,(Ver_C, Suite_C,\ Ver_S, \\
 & \qquad Suite_S)\}_{\mathrm{Master}(Secret_C)}
\end{array}
$$

The above protocol is secure against attacks on version numbers and cryptographic preferences except in the following circumstances:

1. If an attack on version number in the first *Hello* message causes the parties to switch to a different protocol such as SSL 2.0, they will not exchange verification messages and the attack will not be detected. See Section 3.3.9 for further discussion of anomalies related to the version rollback attack.

2. By changing cryptosuites in the *Hello* messages, the intruder may force the parties to use a very weak public-key encryption algorithm that can be broken in real time (*i.e.*, while the current

run of the handshake protocol is in progress). If the intruder can break the encrypted message containing the client's secret, it can compute the master secret and the derived keys and will thus be able to forge post-handshake verification messages. The only defense against this kind of attack is to prohibit SSL implementations from using weak cryptographic algorithms in the handshake protocol even if hello messages from the protocol counterparty indicate preference for such algorithms.

Attack on D

In Protocol D, the parties verify only plaintext information after the handshake negotiation is complete. Since the intruder cannot forge signatures, invert hash functions, or break encryption without the correct private key, it can neither learn the client's secret, nor substitute its own. It may appear that D provides complete security for the communication channel between C and S. However, Murφ discovered an attack on client's identity that succeeds even if all cryptographic algorithms are perfect.

Intruder I intercepts C's hello message to server S, and initiates the handshake protocol with S *under its own name*. All messages sent by S are re-transmitted to C, while most of C's messages, including the post-handshake verification messages, are re-transmitted to S. (See the protocol run below for details. Re-transmission of C's verification message is required to change the sender identifier, which is not shown explicitly below.) As a result, both C and S will complete the handshake protocol successfully, but C will be convinced that it is talking to S, while S will be convinced that it is talking to I.

Notice that I does not have access to the secret shared between C and S. Therefore, it will not be able to generate or decrypt encrypted messages after the protocol is complete, and will only be able to re-transmit C's messages. However, the server will believe that the messages are coming from I, whereas in fact they were sent by C.

This kind of attack, while somewhat unusual in that it explicitly reveals the intruder's identity, may prove harmful for a number of reasons. For example, it deprives C of the possibility to claim later that it communicated with S, since S will not be able to support C's claims (S may not even know about C's existence). If S is a pay server providing some kind of online service in exchange for anonymous "electronic coins" such as eCash [20], I may be able to receive service from S using C's cash. Recall, however, that I can only receive the service if it is not encrypted, which might be the case for large volumes of data.

$C \to S\langle I\rangle$	$C,\ \mathit{Ver}_C,\ \mathit{Suite}_C$
$I \to S$	$\underline{I},\ \mathit{Ver}_C,\ \mathit{Suite}_C$
$S \to I$	$\mathit{Ver}_S,\ \mathit{Suite}_S,\ \mathrm{Sig}_{CA}\{S, K_S^+\}$
$I \to C$	$\mathit{Ver}_S,\ \mathit{Suite}_S,\ \mathrm{Sig}_{CA}\{S, K_S^+\}$
$C \to S\langle I\rangle$	$\mathrm{Sig}_{CA}\{C, V_C\},\ \{\mathit{Secret}_C\}_{K_S^+},$ $\mathrm{Sig}_C\{\mathrm{Hash}\,(\mathit{Secret}_C)\}$
$I \to S$	$\underline{\mathrm{Sig}_{CA}\{I, V_I\}},\ \{\mathit{Secret}_C\}_{K_S^+},$ $\underline{\mathrm{Sig}_I}\{\mathrm{Hash}\,(\mathit{Secret}_C)\}$

⟨Change to negotiated cipher⟩

$S \to I$	$\{\mathrm{Hash}\,(\mathit{Ver}_C, \mathit{Suite}_C, \mathit{Ver}_S, \mathit{Suite}_S)\}_{\mathrm{Master}(\mathit{Secret}_C)}$
$I \to C$	$\{\mathrm{Hash}\,(\mathit{Ver}_C, \mathit{Suite}_C, \mathit{Ver}_S, \mathit{Suite}_S)\}_{\mathrm{Master}(\mathit{Secret}_C)}$
$C \to S\langle I\rangle$	$\{\mathrm{Hash}\,(\mathit{Ver}_C, \mathit{Suite}_C, \mathit{Ver}_S, \mathit{Suite}_S)\}_{\mathrm{Master}(\mathit{Secret}_C)}$
$I \to S$	$\{\mathrm{Hash}\,(\mathit{Ver}_C, \mathit{Suite}_C, \mathit{Ver}_S, \mathit{Suite}_S)\}_{\mathrm{Master}(\mathit{Secret}_C)}$

3.3.7 Protocol E

D + post-handshake verification of all messages

To fix the bug in Protocol D, the parties verify *all* of their communication after the handshake is complete. Now the intruder may not re-transmit C's messages to S, because C's *Hello* message contained C, while the *Hello* message received by the server contained I. The discrepancy will be detected in post-handshake verification.

$$C \rightarrow S \qquad C,\ Ver_C,\ Suite_C$$

$$S \rightarrow C \qquad Ver_S,\ Suite_S,\ \mathrm{Sig}_{CA}\{S, K_S^+\}$$

$$C \rightarrow S \qquad \mathrm{Sig}_{CA}\{C, V_C\},\ \{Secret_C\}_{K_S^+},\ \mathrm{Sig}_C\{\mathrm{Hash}\,(Secret_C)\}$$

⟨Change to negotiated cipher⟩

$$S \rightarrow C \qquad \{\mathrm{Hash}\,(Messages)\}_{\mathrm{Master}(Secret_C)}$$

$$C \rightarrow S \qquad \{\mathrm{Hash}\,(Messages)\}_{\mathrm{Master}(Secret_C)}$$

Attack on E

I observes a run of the protocol and records all of C's messages. Some time later, I initiates a new run of the protocol, ostensibly from C to S, and replays recorded C's messages in response to messages from S. Even though I is unable to read the recorded messages, it manages to convince S that the latter is talking to C, even though C did not initiate the protocol.

$$C \rightarrow S \qquad C,\ Ver_C,\ Suite_C$$

$$S \rightarrow C \qquad Ver_S,\ Suite_S,\ \mathrm{Sig}_{CA}\{S, K_S^+\}$$

$$C \rightarrow S \qquad \mathrm{Sig}_{CA}\{C, V_C\},\ \{Secret_C\}_{K_S^+},\ \mathrm{Sig}_C\{\mathrm{Hash}\,(Secret_C)\}$$

⟨Change to negotiated cipher⟩

$$S \rightarrow C \qquad \{\mathrm{Hash}\,(Messages)\}_{\mathrm{Master}(Secret_C)}$$

$$C \rightarrow S \qquad \{\mathrm{Hash}\,(Messages)\}_{\mathrm{Master}(Secret_C)}$$

Next run of the protocol ...

$I \to S$	$C,\ Ver_C,\ Suite_C$
$S \to C\langle I \rangle$	$Ver_S,\ Suite_S,\ \mathrm{Sig}_{CA}\{S, K_S^+\}$
$I \to S$	$\mathrm{Sig}_{CA}\{C, V_C\},\ \{Secret_C\}_{K_S^+},$ $\mathrm{Sig}_C\{\mathrm{Hash}\,(Secret_C)\}$

⟨Change to negotiated cipher⟩

$S \to C\langle I \rangle$	$\{\mathrm{Hash}\,(Messages)\}_{\mathrm{Master}(Secret_C)}$
$I \to S$	$\{\mathrm{Hash}\,(Messages)\}_{\mathrm{Master}(Secret_C)}$

3.3.8 Protocol F

E + nonces

By adding random nonces to each run of the protocol, SSL 3.0 ensures that there are always some differences between independent runs of the protocol. The intruder is thus unable to replay verification messages from one run in another run.

$C \to S$	$C,\ Ver_C,\ Suite_C,\ N_C$
$S \to C$	$Ver_S,\ Suite_S,\ N_S,\ \mathrm{Sig}_{CA}\{S, K_S^+\}$
$C \to S$	$\mathrm{Sig}_{CA}\{C, V_C\},\ \{Secret_C\}_{K_S^+},$ $\mathrm{Sig}_C\{\mathrm{Hash}\,(Secret_C)\}$

⟨Change to negotiated cipher⟩

$S \to C$	$\{\mathrm{Hash}\,(Messages)\}_{\mathrm{Master}(Secret_C)}$
$C \to S$	$\{\mathrm{Hash}\,(Messages)\}_{\mathrm{Master}(Secret_C)}$

Attack on F

The exact semantics of the verification messages exchanged after switching to the negotiated cipher (*i.e.*, *Finished* messages in the SSL terminology) is somewhat unclear. Section 5.6.9 of [23] states: "No acknowledgment of the finished message is required; parties may begin sending encrypted data immediately after sending the finished message. Recipients of finished messages must verify that the contents are correct." The straightforward implementation of this definition led Murφ to discover the following attack on Protocol F:

1. I modifies the *Hello* messages, changing the legitimate parties' cryptosuites so as to force them into choosing a weak public-key encryption algorithm for the secret exchange.

2. I records the weakly encrypted $Secret_C$ as it is being transmitted from C to S.

3. After C and S switch to the negotiated cipher, I delays their verification messages indefinitely, preventing them from discovering the attack on cryptosuites and gaining extra time to crack the public-key encryption algorithm and learn $Secret_C$.

4. Once the secret is learned, I is able to compute the keys and forge verification messages.

Since we did not model weak encryption that can be broken by the intruder, we also did not model the last step of the attack explicitly. Instead, if the model reached the state after the third step, the attack was considered successful.

Note that in the actual SSL 3.0 protocol $Secret_C$ is not used directly as the symmetric key between C and S. It serves as one of the inputs to a hash function that computes the actual symmetric key. Therefore, even if the intruder is able to figure out the symmetric key, this will not necessarily compromise the shared secret $Secret_C$.

To obtain $Secret_C$, the intruder has to force the parties into choosing weak public-key encryption for the *secret exchange* message, and then break the chosen encryption algorithm in real-time. This attack can only succeed if both parties support cryptosuites with very weak public-key encryption (*e.g.*, with a very short RSA key). We are not aware of any existing SSL implementations for which this is the case.

3.3.9 Protocol Z (final)

To prevent the attack on Protocol F, it is sufficient to require that the parties consider the protocol incomplete until they each receive the correct verification message. Murφ did not discover any bugs in the model implemented according to this semantics.

Alternatively, yet another piece of SSL can be added to Protocol F. If the client sends the server a hash of all messages *before* switching to the negotiated cipher, the server will be able to detect an attack on its cryptosuite earlier.

$C \rightarrow S$	$C,\ \mathit{Ver}_C,\ \mathit{Suite}_C,\ N_C$
$S \rightarrow C$	$\mathit{Ver}_S,\ \mathit{Suite}_S,\ N_S,\ \mathrm{Sig}_{CA}\{S, K_S^+\}$
$C \rightarrow S$	$\mathrm{Sig}_{CA}\{C, V_C\},\ \{\mathit{Secret}_C\}_{K_S^+},$ $\mathrm{Sig}_C\{\mathrm{Hash}\,(\mathit{Messages})\}$

⟨Change to negotiated cipher⟩

$S \rightarrow C$	$\{\mathrm{Hash}\,(\mathit{Messages})\}_{\mathrm{Master}(\mathit{Secret}_C)}$
$C \rightarrow S$	$\{\mathrm{Hash}\,(\mathit{Messages})\}_{\mathrm{Master}(\mathit{Secret}_C)}$

Murφ was used to model Protocol Z with 2 clients, 1 intruder, 1 server, no more than 2 simultaneous open sessions per server, and no more than 1 resumption per session. No new bugs were discovered. However, Murφ found two anomalies in the protocol employed to resume a session.

Protocol Z with resumption: cryptosuite attack

Adding the extra verification message suffices for the full handshake protocol but not for the resumption protocol. When a session is resumed, the parties switch to the negotiated cipher immediately after exchanging *Hello* messages. Therefore, the intruder can alter cryptographic preferences in the *Hello* messages and then delay the parties' *Finished* messages indefinitely, preventing them from detecting the attack. It appears that this attack does not jeopardize the security of SSL 3.0 in practice, since no secret is exchanged in the resumption protocol. In fact, it is not clear to us if the cryptosuites in the *Hello* messages are used at all in the resumption protocol.

Protocol Z with resumption: version rollback attack

In our model of Protocol Z, the participants switch to SSL 2.0 if a version number in the *Hello* messages is different from 3.0. (Since the Internet Draft for SSL 2.0 has expired and is not publicly available at the moment, we included a specification of SSL 2.0 in Appendix A.)

The *Finished* messages in SSL 2.0 do not include version numbers or cryptosuites, therefore Protocol Z is susceptible to the attack on cryptographic preferences described in Section 3.3.5. In the following example, it is assumed for simplicity that client authentication is not used. Also, SSL 2.0 *Hello* messages have a slightly different format than SSL 3.0 *Hello* messages and do not contain explicit version information. To simplify presentation, we assume that the intruder converts a 3.0 *Hello* message into a 2.0 *Hello* message simply by changing the version number.

$C \to S\langle I \rangle$	$C,\ 3.0,\ \mathit{Suite}_C,\ N_C$
$I \to S$	$C,\ \underline{2.0},\ \underline{\mathit{Suite}_I},\ N_C$
$S \to C\langle I \rangle$	$2.0,\ \mathit{Suite}_S,\ N_S,\ \mathrm{Sig}_{CA}\{S, K_S^+\}$
$I \to C$	$2.0,\ \underline{\mathit{Suite}_I},\ N_S,\ \mathrm{Sig}_{CA}\{S, K_S^+\}$
$C \to S$	$\{\mathit{Secret}_C\}_{K_S^+}$
⟨Change to negotiated cipher⟩	
$C \to S$	$\{N_S\}_{\mathrm{Master}(\mathit{Secret}_C)}$
$S \to C$	$\{N_C\}_{\mathrm{Master}(\mathit{Secret}_C)}$

To prevent the version rollback attack, SSL 3.0 clients add their version number to the secret they send to the server. When the server receives a secret with 3.0 embedded in it from a 2.0 client, it can determine that there has been an attack on the client's *Hello* message in which the client's true version number (3.0) was rolled back to 2.0.

However, this defense does not work in the case of session *resumption*. Murφ discovered a version rollback attack on the resumption protocol. The attack succeeds since in the resumption protocol, the client does not send a secret to the server, and the intruder's alteration of version numbers in the *Hello* messages goes undetected.

Strictly speaking, this attack is not a violation of the specification [23], since the latter implicitly allows an SSL session that was established using the 3.0 protocol to be resumed using the 2.0 protocol. However, this attack makes implementations of SSL 3.0 potentially vulnerable to SSL 2.0 weaknesses. Wagner and Schneier [61] reach a similar conclusion in their informal analysis of SSL 3.0.

3.3.10 Protocol Z vs. SSL 3.0

Figure 3.1 shows the definition of the SSL 3.0 handshake protocol according to [23]. When several messages from the same party follow each other in the original definition, they have been collapsed into a single protocol step (*e.g.*, *Certificate*, *ClientKeyExchange*, and *CertificateVerify* were joined into *ClientVerify*). The underlined pieces of SSL 3.0 are not in Protocol *Z*.

`ClientHello` $C \rightarrow S$ $C,\ Ver_C,\ Suite_C,\ N_C$

`ServerHello` $S \rightarrow C$ $Ver_S,\ Suite_S,\ N_S,\ \mathrm{Sig}_{CA}\{S, K_S^+\}$

`ClientVerify` $C \rightarrow S$

$$\begin{aligned}
&\mathrm{Sig}_{CA}\{C, V_C\},\\
&\{Ver_C, Secret_C\}_{K_S^+},\\
&\mathrm{Sig}_C\{\,\mathrm{Hash}\,(\mathrm{Master}(N_C,\ N_S,\ Secret_C)+\\
&\qquad\qquad Pad_2 +\\
&\qquad\qquad \mathrm{Hash}\,(Messages + C\ +\\
&\qquad\qquad\qquad \mathrm{Master}(N_C,\ N_S,\ Secret_C)\ +\\
&\qquad\qquad\qquad Pad_1))\,\}
\end{aligned}$$

⟨Change to negotiated cipher⟩

`ServerFinished` $S \rightarrow C$

$$\begin{aligned}
&\{\,\mathrm{Hash}\,(\mathrm{Master}(N_C,\ N_S,\ Secret_C) + Pad_2 +\\
&\qquad\qquad \mathrm{Hash}\,(Messages\ + S\ +\\
&\qquad\qquad\qquad \mathrm{Master}(N_C,\ N_S,\ Secret_C)\ +\\
&\qquad\qquad\qquad Pad_1))\,\}_{\mathrm{Master}(N_C,\ N_S,\ Secret_C)}
\end{aligned}$$

`ClientFinished` $C \rightarrow S$

$$\begin{aligned}
&\{\,\mathrm{Hash}\,(\mathrm{Master}(N_C,\ N_S,\ Secret_C) + Pad_2 +\\
&\qquad\qquad \mathrm{Hash}\,(Messages\ + C\ +\\
&\qquad\qquad\qquad \mathrm{Master}(N_C,\ N_S,\ Secret_C)\ +\\
&\qquad\qquad\qquad Pad_1))\,\}_{\mathrm{Master}(N_C,\ N_S,\ Secret_C)}
\end{aligned}$$

Figure 3.1: The SSL 3.0 handshake protocol

Assuming that the cryptographic functions are perfect, the underlined pieces can be removed from the SSL 3.0 handshake protocol without jeopardizing its security. However, they do serve a useful purpose by strengthening cryptography and making brute-force attacks on the protocol less feasible.

For example, recall that the shared secret is not used directly as the symmetric key between C and S. Instead, it is used as input to a (pseudorandom) function that computes the actual shared secret. Therefore, breaking the symmetric cipher will not necessarily compromise the shared secret as it would require inverting two hash functions. To obtain the shared secret, the intruder would have to break public-key encryption in the *ClientKeyExchange* message.

The construction of the keyed hash in *ClientVerify*, *ServerFinished*, and *ClientFinished* messages as $\text{Hash}\,(K, Pad_2, \text{Hash}\,(K, Pad_1, text))$ follows the HMAC method proposed by Krawczyk et al. [35], who proved that adding a secret key to the function makes it significantly more secure even if the actual hash function is relatively weak.

In general, we would like to emphasize that SSL 3.0 contains many security measures that are designed to protect the protocol against cryptographic attacks. Since we modeled an idealized protocol in Murφ under the perfect cryptography assumption, we found SSL 3.0 secure even without these features.

3.4 Conclusions

Our first case study shows that the finite-state enumeration tool Murφ can be successfully applied to complex security protocols like SSL. The analysis uncovered some anomalies in SSL 3.0. (None of these anomalies, however, poses a direct threat to the security of SSL 3.0.) Of these anomalies at least one had slipped through expert human analysis, confirming the usefulness of computer assistance in protocol design.

Chapter 4

Murφ Optimizations

In this chapter, we describe two state reduction techniques and a method for faster evaluation of parameterized rule conditions. These techniques increase efficiency of finite-state analysis, and make analysis of large security protocols feasible.

4.1 Overview

Finite-state analysis tools such as Murφ exhaustively enumerate all reachable states of the model, checking for each state whether it satisfies the desired correctness criteria. The main problem in this approach is the very large number of reachable states for most protocols. In this chapter, we describe two techniques that reduce the number of reachable states and hence allow the analysis of larger protocols. We prove the techniques sound, *i.e.*, we show that each protocol error that would have been discovered in the original state graph will still be discovered in the reduced state graph. The techniques are based on certain protocol properties that we have identified as characteristic of security protocols. We have implemented both techniques in the Murφ verification system and evaluated them on the SSL [23] and Kerberos [34] protocols.

The first technique is to let the intruder always intercept messages sent by honest participants (instead of making such interception optional). This technique has resulted in a very large reduction in both the number of reachable states and execution time. While this technique has been used by several researchers [37, 6], it has neither been proved sound, nor has its importance been demonstrated.

The second technique prevents the intruder from sending messages to honest participants in states where at least one of the honest participants is able to send a message. Intuitively, the technique makes the intruder more powerful since the intruder maximally increases its knowledge before forging and sending messages to honest participants; hence the analysis of the reduced state graph should not miss any attacks on the protocol. This technique typically saved a factor of two or more in the number of reachable states as well as execution time. It is interesting to note that that this technique is more powerful than partial-order techniques exploiting the independence of the honest participants.

In addition to the two state reduction techniques, we also describe a technique that reduces the execution time of Murφ, but not the number of reachable states. The technique is based on the following observations:

The intruder model employed in Murφ is highly nondeterministic and thus gives rise to a large number of state transition rules. In every reachable state, the enabling conditions of all rules are evaluated. Evaluation can be sped up by partitioning the rules into sets with identical enabling conditions and evaluating the condition only once for each set. This technique typically increased the overall speed of Murφ by a factor of four.

4.2 Properties of security protocols

In this section, we identify several characteristic properties of security protocols that we will use to develop state reduction techniques. These properties characterize every security protocol we have encountered so far, including, *e.g.*, Kerberos [34], SSL [23], and Needham-Schroeder [47]. The properties are quite simple, yet recognizing them is useful both for better understanding of the protocols and for making finite-state analysis as efficient as possible within the basic framework of the Dolev-Yao intruder model.

4.2.1 Protocol invariants are monotonic

The invariants used to specify correctness of security protocols are typically of the forms "Intruder does not know X" or "If honest participant A reaches state S_1, then honest participant B must be in state S_2." Assume that an invariant of this form is invalid for a given state. Then increasing the intruder's knowledge set (which is part of the state) will not make the violated invariant valid. Hence we will call the protocol invariant *monotonic* with respect to the intruder's *knowledge set.* All protocol invariants we have encountered to date have been monotonic.

The implication of this property for state reduction is that we can safely rearrange the reachable state graph, possibly eliminating some states, as long as we can guarantee that for every state in the old graph, there exists a state in the new graph in which the state of all honest protocol participants is the same while the intruder's knowledge set is the same or larger. Because protocol invariants are monotonic, such state reductions are *sound* in the sense that any invariant that would have been violated in the old state graph will still be violated in the new graph.

We also assume that protocol invariants are defined in terms of the intruder's knowledge set and the states of the honest protocol participants. The invariants should not depend on the state of the network. (Since the network is assumed to be controlled by the intruder, invariants that depend on the state of the network can be rewritten to depend on the intruder's knowledge set.)

4.2.2 Intruder controls the network

It is traditionally assumed in security protocol analysis that the intruder exercises full control over the network, including the option to intercept any message. Instead of giving the intruder the option to intercept messages, we will assume that the intruder always intercepts. We model interception in Murφ by having the intruder remove the message from the "network" and store it in its database. The intruder can then replay the message to the intended recipient, or forge a similar-looking message.

Intuitively, the assumption that every message gets intercepted will not weaken the intruder and should hence be sound. In Section 4.4.1 below, this assumption is used to cut the transitions leading to redundant states, differing only in the contents of the intruder's database. The only state left is the one in which the database contains all information observable from the exchange of protocol messages up to that moment.

4.2.3 Honest protocol participants are independent

Honest protocol participants are fully independent from each other. The only means of communication between the participants is by sending messages on the network, which is assumed to be fully controlled by the intruder. Sending a message to another participant is thus equivalent to simply handing it to the intruder, hoping that the latter will not be able to extract any useful information from it and will replay it intact to the intended recipient.

As a consequence, an honest protocol participant has no way of knowing for sure what the current state of other participants is, since all information about the rest of the world arrives to him through a network fully controlled by the intruder. Actions of each honest participant (*i.e.*, sending and receiving of messages) thus depend only on its own local state and not on the global state that comprises the states of all participants plus that of the intruder. In our formal representation of security protocols as state graphs, we will rely on this property to make all transition rules for honest participants local (see Section 4.3.2 below). We do not consider protocols with out-of-band communication as they are beyond the scope of our research with Murφ.

4.3 Protocols as state graphs

In this section, we define a formalism for describing finite-state machines associated with security protocols.

4.3.1 States

The global state of the system is represented by a vector:

$$s = [s_1, \ldots, s_N, e]$$

where N is the number of honest protocol participants, s_i is the local state of the protocol participant i, and e is the state of the intruder.

Instead of modeling the global network, we model a separate 1-cell local network for each honest protocol participant. All messages intended for that participant are deposited in its local network as described in Section 4.3.2 below. The local state of an honest participant i is a pair

$$s_i = \langle v_i, m_i \rangle$$

where v_i is the vector of current values of i's local state variables, and m_i is the message currently in i's local network. It is possible that $m_i = \varepsilon$, representing the empty network.

The state of the intruder is simply the set of messages that the intruder has intercepted so far (assume that its initial knowledge is represented as an intercepted message also):

$$e = \{m_{e_1}, \ldots, m_{e_M}\}$$

The intruder's knowledge is obviously not limited to the intercepted messages. The intruder can split them into components, decrypt and encrypt fields, assemble new messages, etc. However, the full knowledge set can always be synthesized from the intercepted messages, since they are the only source of information available to the intruder. Therefore, our chosen representation for the intruder's state is sufficient to represent the intruder's knowledge. When necessary, we will refer to the set of messages that can be synthesized from the set of intercepted messages as synth(e). (Since operations like encryption and pairing can be applied infinitely many times in the synthesis, the intruder's full knowledge is generally infinite. In practice, one can extract finite limits on the numbers of times encryption and pairing have to be applied from the protocol definition, making the intruder's knowledge finite.)

4.3.2 Rules for honest participants

All transition rules between states have the following form in our formalism:

$$r_k^{(i|e)} = \text{if } c_k(s_i|e) \text{ then } s \;\rightarrow\; s'$$

where $c_k(s_i|e)$ is the condition of the rule (it depends on the local state s_i in case of an honest protocol participant, and the knowledge set e in case of the intruder), s is the original global state, s' is the global state obtained as the result of rule application.

We can assume without loss of generality that every transition rule for an honest protocol participant consists of reading a non-empty message off the local network, changing the local variables, and sending a non-empty message to another participant. If necessary, the protocol can be rewritten so as to avoid "hidden" transitions that change the state of a participant without visible activity on the network. The initial transition for each participant can be triggered by a special message deposited into its local network in the start state of the system, and the last transition can be rewritten so that it deposits another special message on the network. This ensures that every transition reads and writes into the network. Also, the 1-cell capacity restriction on the local network is not essential, since we will eventually assume that every message is intercepted by the intruder immediately after it has been sent.

The transition rules for an honest protocol participant i are represented as follows:

$$r_k^{(i)} = \text{if } \ c_k(v_i, m_i) \text{ then } [\ldots\langle v_i, m_i\rangle \ldots \langle v_j, \varepsilon\rangle \ldots] \ \rightarrow \ [\ldots\langle v_i', \varepsilon\rangle \ldots \langle v_j, m_j\rangle \ldots]$$

Informally, if condition c_k evaluates to `true` given i's local state $s_i = \langle v_i, m_i\rangle$, then i reads message m_i off its local network, executes some code changing its local state variables from v_i to v_i', and sends message m_j to participant j by depositing it in j's local network. Note that the rule is local — its condition depends only on i's local state. We assume that honest participants are deterministic. In any state, there is no more than one rule enabled for each participant. However, it is possible that rules for several participants are enabled in the same state, resulting in nondeterminism.

4.3.3 Rules for the intruder

The transition rules for the intruder are global. The first set of rules describes the intruder intercepting a message intended for an honest participant:

$$r_{-_i}^{(e)} = \text{if origin}(m) \neq e \text{ then } [\ldots\langle v_i, m\rangle \ldots e] \ \rightarrow \ [\ldots\langle v_i, \varepsilon\rangle \ldots e \cup \{m\}]$$

The intruder first checks the origin of the message on i's local network, since we do not want the intruder to remove its own messages. If the message was generated by an honest participant, it is removed from the network and added to the intruder's database. Note that the local variables of the honest participant are not affected by this action.

The second set of intruder rules describes the intruder generating a message and sending it to an honest participant whose local network is empty:

$$r_{+_i}^{(e)} = \text{if } \texttt{true} \text{ then } [\ldots\langle v_i, \varepsilon\rangle \ldots e] \ \rightarrow \ [\ldots\langle v_i, m\rangle \ldots e] \qquad m \in \text{synth}(e)$$

The intruder creates a new message (either by replaying an intercepted message $m \in e$, or by synthesizing m from the messages stored in e) and sends it to participant i by depositing it in i's local network. Clearly, the intruder rules are nondeterministic, as there may be several intruder rules enabled in the same state.

There are no other transition rules in the system.

4.3.4 State graph

The finite-state machine associated with the protocol is a directed graph $\{S, T, s_0, Q\}$. The vertices S are all possible states of the protocol. The directed edges T are pairs of states labeled by rules such that $r_k^{(i|e)} : \langle s, s' \rangle \in T$ *iff* the transition rule $r_k^{(i|e)}$ is enabled in state s (*i.e.*, its condition evaluates to `true`) and transforms s into state s'. The finite-state machine is nondeterministic. If several rules are enabled in state s, there will be several edges leaving the corresponding graph vertex. We will refer to the subgraph reachable from vertex s as $R(s)$.

$s_0 \in S$ is the start state of the protocol.

Q is the set of protocol invariants. An invariant is a function from states to boolean values $q : S \rightarrow \texttt{true}|\texttt{false}$ such that $q(s) = \texttt{false}$ if the invariant q is violated in s, otherwise $q(s) = \texttt{true}$. Since protocol invariants do not depend on the state of the network, the value of q in any state does not depend on the m_i values representing the current contents of the participants' local networks.

We say that state $s = [s_1 \dots s_N, e]$ is *subsumed* by another state $s' = [s'_1 \dots s'_N, e']$ *iff* $\forall i \; s_i = s'_i$ and $e \subseteq e'$. Informally, s' subsumes s if the states of all honest participants are the same in s' as in s while the intruder's knowledge set is the same or larger. In the rest of this chapter, subsumption will be denoted as $s \preceq s'$.

Note that if $s \preceq s'$, then all rules enabled in s are enabled in s', too. The reverse is not true since the intruder's knowledge set is larger in s' and some rules may be enabled in s' but not in s. Therefore, $R(s)$ (set of states reachable from s) is isomorphic to a subgraph of $R(s')$. Thus, for any t reachable from s, there exists a t' reachable from s' such that $t \preceq t'$. We will refer to this fact as *inheritance of subsumption*.

Protocol invariants are monotonic with respect to the intruder's knowledge set (increasing intruder's knowledge does not repair any violated invariants). Therefore, if $s \preceq s'$, then $q(s) = \texttt{false}$ implies that $q(s') = \texttt{false}$. We will refer to this fact as *monotonicity of invariance*.

4.3.5 Soundness of state reduction

A finite-state analyzer such as Murφ verifies the protocol by traversing the state graph starting from state s_0. For every state s it reaches, Murφ verifies that all invariants are valid, *i.e.*, $\forall q \; q(s) = \texttt{true}$. Violation of any invariant signals an error in the protocol.

A state reduction technique transforms the original state graph $\{S, T, s_0, Q\}$ into a new graph $\{S', T', s_0, Q\}$. We claim that the technique is *sound* if all errors that would have been discovered in the original graph will still be discovered in the new graph. More formally, if the old graph contains a reachable state in which one of the invariants was violated, then the new graph should contain a reachable state in which the *same* invariant is violated.

$$\exists s \in R(s_0)\ \ \exists q \in Q \ \text{ s.t. } \ q(s) = \texttt{false} \qquad \text{implies} \qquad \exists s' \in R'(s_0) \ \text{ s.t. } \ q(s') = \texttt{false}$$

In the soundness proofs for the state reduction techniques below, we will demonstrate that the state graph contains two vertices s and s' such that $s \preceq s'$. By monotonicity of invariance, all invariants that are violated in s are also violated in s'. Moreover, by inheritance of subsumption, for any t reachable from s, there exists a t' reachable from s' such that all invariants violated in t are also violated in t'.

Suppose that we find a vertex s^* in the state graph such that there are edges leading from s^* to both s and s'. We cut the edge from s^* to s. The subgraph rooted in s may become unreachable, reducing the number of states to be explored. However, we do not "lose" any protocol errors by eliminating these states. Every eliminated state in which an invariant is violated has a counterpart reachable from s' in which the same invariant is violated.

4.4 State reduction techniques

In this section, we describe the state reduction techniques and prove them sound.

4.4.1 Intruder always intercepts

Consider a state in which one of the rules for honest participants is enabled:

$$s_1 = [\ldots \langle v_i, m_i \rangle \ldots \langle v_j, \varepsilon \rangle \ldots e]$$

Suppose that state s_1 is such that condition $c_k(v_i, m_i)$ evaluates to $\texttt{true}$. Then rule $r_k^{(i)}$ (participant i sends message m_j to participant j) is enabled in s_1. Suppose that message m_i is not known to the intruder, *i.e.*, $m_i \notin e$. We intend to reduce the number of states to be explored by having the intruder always intercept message m_i.

Fig. 4.1 shows the subgraph rooted in s_1, where

$$\begin{aligned} s_2 &= [\ldots \langle v_i', \varepsilon \rangle \ldots \langle v_j, m_j \rangle \ldots e] \\ s_3 &= [\ldots \langle v_i, \varepsilon \rangle \ldots \langle v_j, \varepsilon \rangle \ldots e \cup \{m_i\}] \\ s_4 &= [\ldots \langle v_i, m_i \rangle \ldots \langle v_j, \varepsilon \rangle \ldots e \cup \{m_i\}] \\ s_5 &= [\ldots \langle v_i', \varepsilon \rangle \ldots \langle v_j, m_j \rangle \ldots e \cup \{m_i\}] \end{aligned}$$

Each transition in Fig. 4.1 is labeled with the corresponding rule: rule $r_k^{(i)}$ is enabled both in s_1

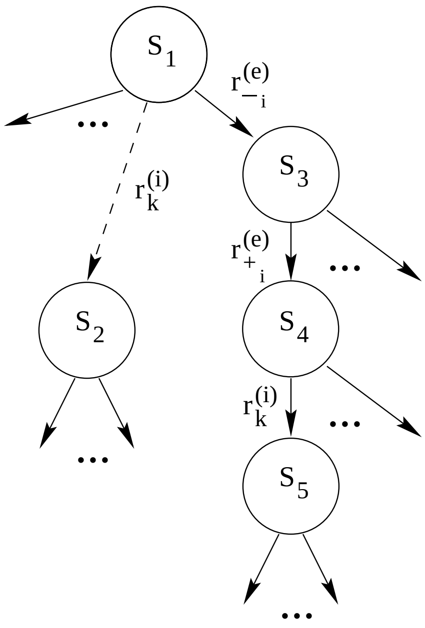

Figure 4.1: Reduction: "Intruder always intercepts"

and s_4; rule $r^{(e)}_{-_i}$ (intruder removes a message from i's local network and stores it in the database) is enabled in s_1; rule $r^{(e)}_{+_i}$ (intruder deposits a message from its database into i's local network) is enabled in s_3. There may be additional rules enabled in states $s_1, \ldots, s_5$, but we will limit our attention to the subgraph shown in the figure.

We now observe that $s_2 \preceq s_5$ (the states of all honest participants are the same but the intruder's knowledge set is larger in s_5). Therefore, we can remove the graph edge leading from s_1 to s_2. Any invariant violation that can be discovered by analyzing $R(s_2)$ will be discovered by analyzing $R(s_5)$.

This state reduction technique effectively eliminates direct communication between participants. Every message sent on the network is intercepted by the intruder and added to the intruder's database. There is no need to consider states in which the database is "incomplete" (s_2 in the example above) since they are subsumed by the states in which the database is as complete as possible (s_5 in the example above), containing all messages exchanged on the network so far.

We can simplify the state transition rules slightly by assuming that rule $r^{(i)}_k$ deposits the generated message m_j directly into the intruder's database (recall that the original rule deposited the message into j's local network).

$$\hat{r}^{(i)}_k = \text{if } c_k(v_i, m_i) \text{ then } [\ldots\langle v_i, m_i\rangle\ldots\langle v_j, \varepsilon\rangle\ldots e] \;\rightarrow\; [\ldots\langle v'_i, \varepsilon\rangle\ldots\langle v_j, \varepsilon\rangle\ldots e \cup \{m_j\}]$$

This simplification also eliminates the need for $r^{(e)}_{-_i}$ rules. Another consequence is that $r^{(e)}_{+_i}$ are the only rules that can deposit a message into an honest participant's local network.

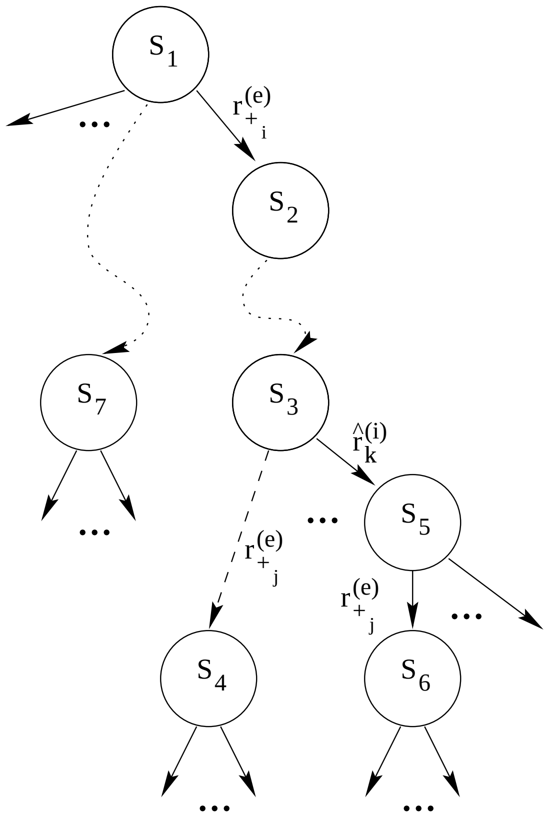

Figure 4.2: Reduction: "Intruder does not send if honest participant can send"

4.4.2 Intruder does not send if honest participant can send

Fig. 4.2 shows a fragment of the state graph in which

$$
\begin{aligned}
s_1 &= [\ldots\langle v_i, \varepsilon\rangle \ldots \langle \ldots, \ldots\rangle \ldots e] \\
s_2 &= [\ldots\langle v_i, m_i\rangle \ldots \langle \ldots, \ldots\rangle \ldots e] \\
s_3 &= [\ldots\langle v_i, m_i\rangle \ldots \langle v_j, \varepsilon\rangle \ldots e] \\
s_4 &= [\ldots\langle v_i, m_i\rangle \ldots \langle v_j, m_j\rangle \ldots e] \\
s_5 &= [\ldots\langle v'_i, \varepsilon\rangle \ldots \langle v_j, \varepsilon\rangle \ldots e \cup \{m'_j\}] \\
s_6 &= [\ldots\langle v'_i, \varepsilon\rangle \ldots \langle v_j, m_j\rangle \ldots e \cup \{m'_j\}] \\
s_7 &= [\ldots\langle v_i, \varepsilon\rangle \ldots \langle v_j, m_j\rangle \ldots e] \\
& \quad \text{where } m_i, m_j \in \text{synth}(e),\ m'_j \notin e
\end{aligned}
$$

In state s_3, at least two rules are enabled: $r^{(e)}_{+_j}$ corresponds to the intruder depositing m_j into j's network, and $\hat{r}^{(i)}_k$ corresponds to the intruder waiting for participant i to send its message first. We intend to demonstrate that the number of states to be explored can be reduced by considering only the latter case. Formally, the edge from s_3 to s_4 can be cut from the state graph since any violation of protocol invariants that can be discovered in $R(s_4)$ will be discovered either in $R(s_6)$, or in $R(s_7)$,

Suppose that there exists a state $t_E \in R(s_4)$ such that $q(t_E) = \texttt{false}$ for some invariant $q \in Q$. Consider the sequence of state transitions that leads from s_4 to t_E. Each transition is labeled by the corresponding rule:

$$s_4 \xrightarrow{r_0} t_0 \xrightarrow{r_1} t_1 \rightarrow \ldots \xrightarrow{r_E} t_E$$

We will consider two cases.

Case 1. Suppose $\exists L \in 0..E\ r_L = \hat{r}_k^{(i)}$, and none of the rules r_l for $l \in 0..L-1$ involve participant i. In other words, rule $\hat{r}_k^{(i)}$ is executed at some point between s_4 and t_E.

Consider that state s_6 is identical to s_4 except for the state of participant i, which does not matter for rules r_l, and the intruder's knowledge set, which is strictly larger in s_6. Therefore, the rule sequence r_l is enabled in s_6, leading to a state $\tilde{t}_{L-1}$. To complete the proof for this case, observe that t_L is equivalent to $\tilde{t}_{L-1}$.

Case 2. Suppose that $\forall l \in 0..E\ r_l \neq \hat{r}_k^{(i)}$, *i.e.*, none of rules leading to the erroneous state involve participant i reading message m_i off the network and sending message m'_j to participant j. The only rule enabled for i is $\hat{r}_k^{(i)}$, and it cannot become disabled as long as the local state of i does not change. Therefore, none of the rules r_l involve participant i at all, and s_i is the same in t_E as in s_4. To prove soundness for this case, we will first demonstrate that if the state graph includes state s_3, it must also include state s_7, and then we will show that if an invariant violation can be discovered in $R(s_4)$, it can also be discovered in $R(s_7)$.

To prove that if the state graph contains s_3, it must also contain s_7, consider the sequence of rules that leads from the start state s_0 to s_3. Since participant i's local network contains m_i in it, the last rule in the sequence that involves i must be $r_{+_i}^{(e)}$, since only the intruder can deposit a message into an honest participant's local network. In Fig. 4.2, s_1 represents the state to which rule $r_{+_i}^{(e)}$ was applied, and s_2 represents the resulting state. Note that none of the rules leading from s_2 to s_3 involve i, since $r_{+_i}^{(e)}$ was the *last* such rule in the sequence. Therefore, the same rules are enabled in state s_1 that differs from s_2 only in the contents of i's local network. Applying the rules, we obtain the state $\tilde{s}$ that differs from s_3 only in the contents of i's local network, namely, i's network is empty in $\tilde{s}$ but contains m_i in s_3. Therefore, $\tilde{s} = s_7$.

To complete the soundness proof, observe that state s_7 is identical to s_4 except for the state of participant i. Since none of the rules r_l leading to t_E involve participant i, the same sequence of rules is enabled in s_7, leading to a state $\tilde{t}_E$ such that the only difference between $\tilde{t}_E$ and t_E is the state of participant i. More precisely, i's local network is empty in $\tilde{t}_E$ but contains m_i in t_E. This implies that $q(\tilde{t}_E) = \texttt{false}$, since invariants do not depend on the state of the network. The proof is complete.

The described state reduction technique makes sure that the intruder never sends a message if there is an honest participant who is ready to send a message, too (*i.e.*, one of the rules for honest participants is enabled). The intuitive reason for this is that the intruder's knowledge set should be as complete as possible before the intruder synthesizes a message. By waiting until the honest participant sends its message and intercepting it, the intruder potentially increases its knowledge set.

To ensure that the intruder sends messages only when no one else can, we can modify the condition for rule $r_{+_i}^{(e)}$ so that it is enabled only in the states of the following form:

$$s = [\langle v_1, \varepsilon\rangle, \ldots, \langle v_N, \varepsilon\rangle, e]$$

An alternative implementation of this state reduction technique is to assign a lower priority to the intruder rules than to the rules for honest participants (see Section 4.5.1 below).

Remark. A related state reduction technique exploits the independence of honest protocol participants to fix a partial order and eliminate multiple interleavings of message sends. Since all participants send their messages to the intruder, the order in which the messages are deposited into the intruder's database does not matter.

However, there is no need to impose a partial order on the honest participants if the technique described in this section is implemented. To see why this is the case, suppose we start with an empty network. After the intruder sends a message to the first participant, the rules for the intruder become disabled until the first participant replies by depositing its message into the intruder's database. Only then can the intruder send a message to another participant. Hence there are no states in which more than one participant is ready to send a message.

4.5 Implementation issues

In this section, we discuss how state reduction techniques can be implemented by assigning priorities to transition rules. We also describe another Murφ optimization that does not rely on state reduction.

4.5.1 Rule priorities

To support the state reduction techniques described above, we extended Murφ language with *rule priorities.* In the extended language, a model implementor can assign an integer priority to every rule in the system. If several rules are enabled in a particular state, Murφ will only explore the subgraphs corresponding to rules with the highest priority. User-specified rule priorities help Murφ recognize which of the rules are associated with honest participants and which are intruder rules.

The technique from Section 4.4.1 (intruder always intercepts) can be implemented by assigning the highest priority to the "Intruder intercepts" rule. Then, whenever a new message appears on the network, it is immediately intercepted by the intruder and stored in its database.

The technique from Section 4.4.2 (intruder does not send if an honest participant can) can be implemented by assigning the lowest priority to the rules for intruder sending a message. The only other rules in the system are those associated with honest participants, and each of those rules sends a message to the network. Therefore, as long as there exists an honest participant who is ready to send a message, the intruder will not send messages but will intercept those from honest participants, increasing its knowledge set.

4.5.2 Parameterized rule conditions

After an honest participant has sent its message, the system reaches a state in which the only enabled rules are those representing the intruder sending messages to honest participants. Typically, the structure of the messages to be sent is known from the protocol specification. However, the mechanical intruder employed in Murφ cannot determine on its own what values have to be assigned to message fields so that the resulting message is accepted by the recipient and potentially leads to a successful attack. Therefore, the intruder will try all possible combinations of field values. For example, the following rule represents the intruder forging a *Finished* message in SSL 3.0 (*Finished* messages contain a record of all information previously sent in the protocol):

```
-- Intruder generates a ServerFinished message with
-- a forged handshake log

ruleset d: ClientId do
  choose  n1: int.nonces do
  choose  n2: int.nonces do
  choose  secretKey: int.secretKeys do
  ruleset sender: ServerId do
    rule "Intruder generates ServerFinished (forged log)"

      cli[d].state = M_SERVER_FINISHED

    ==>

      -- Forge a message with the above parameters
      ...
end
```

In the above rule, the intruder nondeterministically chooses (using the `ruleset` and `choose` constructs) a recipient `d` from among the known SSL clients, two nonces `n1` and `n2` from its database of nonces, a `secretKey` from its database of keys, and the ostensible `sender` of the message from among the SSL servers, and finally forges and sends the message if the recipient is in a state in which it is ready to receive the message. (The intruder is assumed to know the states of all honest protocol participants since the states can be inferred from the protocol definition and the observed message exchange.)

The above Murφ code defines a rule for each possible assignment to the parameters `d`, `n1`, `n2`, `secretKey`, and `sender`. The number of defined rules is thus equal to the product of the cardinalities of the sets from which the parameter values are drawn. Clearly, if the intruder databases are sufficiently large and there are many choices for each of the parameters, the number of defined rules is large.

In each state reached during the state space search, Murφ checks for each rule whether it is enabled or not. Many rules, however, share the same condition. In the example above, all rules generated for the same value of `d` have the same condition, regardless of the values of the other parameters. Thus, it is sufficient to evaluate the condition just once for each possible assignment to `d`.

To exploit this idea, we modified the Murφ compiler so that it separates the parameters for each `rule` statement into two sets. The *Dep* set contains all parameters that are mentioned in the rule condition (`d` in the example above). The *Indep* set contains the parameters *not* mentioned in the rule condition (`n1`, `n2`, `secretKey`, and `sender` in the example above).

As before, a separate rule is defined for every possible assignment to the parameters from *Dep* $\cup$ *Indep*. The set of defined rules is partitioned into subsets so that each subset corresponds one-to-one to a particular assignment to the parameters from *Dep*. Therefore, each subset contains the rules corresponding to all possible assignments to the parameters from *Indep* given a particular assignment to the parameters from *Dep*.

Instead of evaluating every rule condition in each state, the Murφ verifier was altered to only evaluate the condition for one (arbitrarily chosen) rule from each rule subset. The result of evaluation is the same for all rules in the subset since they differ only in the values of the parameters from *Indep*, and the conditions do not depend on those parameters. Therefore, if the condition evaluates to `true`, all rules in the subset are executed. If the condition evaluates to `false`, the verifier immediately moves on to the next subset corresponding to the next assignment to the parameters from *Dep*.

In our experience with security protocols, *Dep* is usually much smaller than *Indep*. The rules with the largest number of parameters are those representing the intruder's sending a message to an honest participant. The parameters of such rules contain values (chosen from the intruder's database) for the message fields of the forged message, while the rule conditions depend only on the current state of the recipient and not on the contents of the intruder's database. In fact, the intruder's knowledge should not have any influence on whether an honest participant is ready to receive a particular message or not. Therefore, not evaluating rule conditions for every possible assignment to the rule parameters has proved very profitable in our Murφ analyses.

4.5.3 Preliminary results

We used the state reduction techniques described in this chapter to reduce the size of the finite-state model for SSL 3.0, which is the largest security protocol analyzed with Murφ to date. The first state reduction technique (intruder always intercepts) reduced the number of states by a factor of 20. Although spectacular, a reduction of this magnitude was to be expected. In the original model, every message could be intercepted by the intruder or delivered directly from sender to recipient. Intuitively, a sequence of N messages resulted in 2^N states, since each of the N messages could be intercepted by the intruder or not.

We profiled the resulting model and determined that most of the execution time was spent evaluating rule conditions due to the very large number of generated rules. Optimizing rule condition evaluation as described in Section 4.5.2 reduced execution time by a factor of 3.7 with the same number of states.

Finally, we implemented the second state reduction technique (intruder does not send if an honest participant can), which resulted in further 43% reduction in the number of states and a 40% reduction in the execution time.

We also evaluated our techniques on the Kerberos protocol, starting with the efficient condition evaluation, and then adding the two state reduction techniques. Table 4.1 shows the numbers of reachable states and execution times dependent on the numbers of clients and servers in the protocol. As in the case of SSL, the biggest savings result from the first state reduction technique. Note that the savings resulting from the second state reduction technique increase with increasing numbers of clients and servers, as one would expect.

Table 4.1: Numbers of reachable states and execution times dependent on the model parameters in the Kerberos protocol

clients	servers		previous scheme	efficient conditions	always intercept	intruder send low priority
1	2	states	14 317	14 317	232	175
		time	191.8s	68.9s	2.0s	1.6s
2	2	states			541	193
		time			4.4s	2.2s
3	3	states			856	195
		time			12.3s	3.6s

4.6 Conclusions

We described two state reduction techniques that exploit characteristic properties of security protocols. These techniques reduce both the number of reachable states and execution time of finite-state analysis. In addition, we described a method for minimizing the time required to evaluate parameterized rule conditions, further reducing total execution time.

The techniques described in this chapter have proved very useful for analyzing large security protocols in Murφ. While the first state reduction technique (intruder always intercepts) has been employed in other finite-state analysis tools, the second state reduction technique (intruder does not send if an honest participant can) is novel and is expected to prove useful beyond the Murφ community.

Future research includes extending the Murφ verifier so that it can automatically recognize subsumption relations between states and remove subsumed states from the state queue. This technique is independent from the second state reduction technique described in this chapter. We expect that combining the two techniques will result in larger state reductions than those achieved by either of the two techniques on its own. In addition, we plan to exploit the fact that, when the second state reduction technique is implemented, every message send from the intruder is immediately followed by a corresponding reply from the honest participant. This allows, for example, to combine send-reply pairs into a single intruder rule, which should result in a significant reduction in the number of reachable states.

Chapter 5

Case Study: Contract Signing

With continuing growth of electronic commerce on the Internet, the issues of trust and fairness are becoming increasingly important. In this chapter, we demonstrate how the finite-state analysis methods developed for secrecy and authentication protocols such as SSL 3.0 can be successfully employed to discover weaknesses in fair exchange protocols. We use two protocols for online contract signing as representative case studies.

5.1 Introduction

Contracts are an important part of business. If two parties wish to sign a contract, but do not share other motives, then each may refuse to sign until the other has demonstrated their commitment to the contract. While simultaneous commitment can be achieved by sitting at a table and signing identical paper copies together, distributed contract signing over a network is inherently asymmetric: someone has to send the first message. In one contemporary style of contract-signing protocol, two rounds of communication are used. In the first round, each party declares their willingness to be bound by the contract. In the second, they each send some remaining data needed to complete the contract. If a trusted third party is able to enforce the contract based on partial completion of the protocol, then it is possible to conduct distributed contract signing so that various symmetric correctness conditions are satisfied. In optimistic contract signing, the third party is only needed in case of a dispute. Otherwise, the protocol can be completed without involving the third party.

The most basic correctness condition for contract signing is called fairness. A contract signing protocol is *fair* if, after completion of the protocol, either both parties have a signed contract or neither does. Another property is called *accountability*: if any party cheats by not following the steps required by the protocol, the resulting network messages will unambiguously show which party has cheated. Accountability is particularly important for the trusted third party, since it has the ability to resolve or abort a contract.

A more complex condition, introduced in [24], has been called *abuse-freeness*. This condition is intended to guarantee that neither party has a specific kind of advantage over the other during the execution of the protocol. To illustrate by example, suppose Alice offers to sell her house to Bob and Bob signs a contract for a certain price. If Alice holds the contract without signing, she may be able to use the contract to convince another buyer to pay more than Bob. Meanwhile, Bob has committed his financial resources to the incomplete transaction and cannot enter into competing deals. In this scenario, Alice obtains evidence she can use to convince another buyer that she alone can decide whether to complete the contract or reject it. This kind of asymmetry can be prevented in physical simultaneous transactions, but it is difficult to prevent abuse in distributed protocols.

While formal methods have been extensively used to analyze the security properties of key exchange and authentication protocols, less attention has been paid to other kinds of protocols, such as fair exchange. In [26], Heintze et al. used the FDR model checker to verify NetBill [13] and Digicash [10] protocols. The correctness conditions they establish are different in character from the ones we consider here.

In this chapter, we describe an automated analysis of two optimistic contract signing protocols. The first protocol was proposed by Asokan, Shoup, and Waidner [1] (we shall refer to it as the ASW protocol). The ASW protocol uses standard cryptography and special forms of contract to guarantee fairness and accountability of trusted third party. The second protocol was proposed by Garay, Jakobsson, and MacKenzie [24] (we shall refer to it as the GJM protocol). It relies on a cryptographic construct called *private contract signature* to guarantee abuse-freeness in addition to fairness and third party accountability.

Using Murφ, we verify fairness properties claimed for the protocols and uncover several weaknesses. In case of the ASW protocol, a malicious protocol participant is able to obtain a valid contract while the honest participant, even if resorts to the help of the trusted third party, can only obtain a replacement contract which is inconsistent with the one possessed by the malicious participant. The same weakness also allows the intruder to stage a replay attack.

Our finite-state analysis of the GJM protocol reveals an attack that leads to the loss of abuse-freeness and third party accountability. Specifically, the contract initiator, O, using a weak form of passive assistance (or information leak) from the third party, is able to choose whether to reveal a completed contract or accept an abort token provided by the third party. Furthermore, if O chooses to reveal its completed contract, and the discrepancy with R's abort token is observed, it is not possible to determine whether the third party participated in the inconsistency or not.

Although the sequences of actions demonstrating these weaknesses are relatively short and easy to follow, the analysis is subtle in several respects. First, both sequences involve interaction between the optimistic two-party transaction normally used to sign a contract, the abort protocol used by one party to time out and stop the protocol, and the resolve protocol used to request enforcement by the third party. As a result of the complexity of interactions between these three subprotocols, we did not suspect any problems until our analysis tool uncovered violations of one of our correctness conditions. Only then, after examining the traces provided, were we able to isolate specific aspects of the protocols that allow the attacks.

In both cases, we suggest simple changes to the protocols that prevent the attacks. For the GJM protocol, the same repair was also proposed by the authors of the protocol after we described the attack [40]. The repaired protocols appear to be correct; Murφ analysis does not suggest any errors. We also show that some assumptions about the communication channels can be relaxed without violating fairness or other intended properties of the protocols.

There is some subtlety in the way that the basic protocol requirements, fairness, abuse-freeness, and accountability, are specified. In examining fairness, for example, we realized that an *abort* message from the third party does not mean that no participant will receive a contract. This is inherent in optimistic two-party protocols: after the protocol has finished without involving the trusted third party, one of the parties can ask the third party to abort the protocol. Another subtlety surrounds abuse-freeness, which is an assertion about choices at intermediate states in the execution of the protocol. Abuse-freeness is not a property that can be determined by examining individual traces of protocol execution independently. Since Murφ is a trace-based tool, we had to devise some extension of the protocol environment, involving an outside party who issues *sign* and *abort* challenges, in order to automatically verify the states in which one participant has the power to determine the eventual outcome of the protocol.

Given the results of the present study, it is interesting to compare the ASW protocol with the GJM protocol, taking into account that the former was not designed to be abuse-free. The ASW protocol may be preferable to the original GJM protocol, since the latter can only be used if the trusted third party and associated communication channels are completely trustworthy due to problems with third party accountability. The repaired GJM protocol appears to guarantee accountability, and may be chosen over the ASW protocol assuming that private contract signatures are available. By contrast, the ASW protocol relies only on standard cryptographic constructs. Analysis of the comparative advantages of cryptographic signature schemes is beyond the scope of this study.

The remainder of this chapter is structured as follows. Section 5.2 provides background on fair exchange protocols, section 5.3 describes the ASW protocol, section 5.4 presents our modeling assumptions, results of our analysis, and suggested repairs. Section 5.5 describes the GJM protocol, which is then analyzed in section 5.6. Section 5.7 compares the two protocols, and brief concluding remarks appear in section 5.8.

5.2 Fair exchange

Intuitively, a protocol is *fair* if no protocol participant can gain an advantage over other participants by misbehaving. For example, a protocol in which two parties exchange one item for another is fair if it ensures that at the end of the exchange, either each party receives the item it expects, or neither receives any information about the other's item [1]. Fair exchange protocols are used for online payment systems, in which a payment is exchanged for an item of value [13], contract signing, in which parties exchange commitments to a contractual text [5, 1, 24], certified electronic mail [3, 63, 14], and other purposes. There are several varieties of fair exchange protocols.

Gradual exchange protocols [5, 8] work by having the parties release their items in small installments, thus ensuring that at any given moment the amount of information received by each side is approximately the same. The drawback of this approach is that a large number of communication steps between the parties is required. Gradual exchange is also problematic if the items to be exchanged have "threshold" value (either the item is valuable, or it is not).

Another category of fair exchange protocols is based on the notion of a *trusted third party* [13, 63, 14]. The trusted third party supervises communication between the protocol participants and ensures that no participant receives the item it wants before releasing its own item. Variations of this approach include fair exchange protocols with a semi-trusted third party [22]. The main drawback of the third party solution is that the third party may become the communication bottleneck if it has to be involved in all instances of the protocol in order to guarantee fairness. The protocol may also need to impose demands on the communication channels, *e.g.*, by requiring that all messages are eventually delivered to their intended recipients.

Recently, several protocols have been proposed for *optimistic* fair exchange [1, 4, 24]. While the third party T may need to be trusted by all parties to the exchange, T needs to act only if one of the parties misbehaves or there is a communication failure. This may ease the communication bottleneck associated with T, making fair exchange more practical for realistic applications.

The contract signing protocol of Garay, Jakobsson, and MacKenzie [24] extends the concept of fairness by introducing the notion of *abuse-freeness*. Informally, a protocol is abuse-free if neither participant can prove to an outside party that it has the power to abort the protocol or successfully complete contract negotiation. In financial applications, the ability to prove that one can resolve or abort a particular contract negotiation may be as important as the actual signing of a contract, making abuse-freeness critical for fair exchange protocols to be deployed in the financial arena.

Fair exchange protocols present a considerable challenge to formal analysis techniques. Fairness invariants are difficult to express precisely based on an informal protocol specification. Since fair exchange protocols are designed to protect honest participants from being cheated by a misbehaving counterpart, it is necessary to model malicious or corrupt protocol participants in addition to the standard intruder. A detailed discussion of how corrupt participants are formally modeled in Murφ can be found in section 5.4.1 below. Fair exchange protocols with the trusted third party typically provide a guarantee of third party verifiability or accountability, promising that any loss of fairness resulting from the third party's corruption can be traced and proved to an outside arbiter. These guarantees are difficult to understand and formalize for automatic verification.

5.3 Asokan-Shoup-Waidner protocol

In this section, we describe the optimistic contract signing protocol by Asokan, Shoup, and Waidner [1] (the ASW protocol). We start by giving a high-level description of the objectives of the protocol and the assumptions under which it operates. We then explain the protocol steps in detail and formalize the correctness conditions posed by the protocol designers. The notation has been changed from the original paper to facilitate explanation.

A common point of confusion about this protocol is the notion of "contract." In general, one might expect a contract to be a pair of digital signatures of an agreed upon text, one signature from each party negotiating the contract. In the ASW protocol, normal termination without use of the third party will produce a contract that contains two digital signatures and additional data generated in the run of the protocol. However, the contracts produced by the third party are not necessarily of this form. In order to understand the ASW protocol, it is important to keep in mind not only the steps of each subprotocol (discussed below), but also the forms of contract that the protocol designers have established for the protocol.

5.3.1 Objectives

The ASW protocol is designed to enable two parties, called O (originator) and R (responder), to obtain each other's commitment on a previously agreed contractual text. The protocol is asynchronous. As the exchange subprotocol progresses, either participant may contact the trusted third party T. The third party may decide, on the basis of communication it has received, whether to issue a replacement contract or an abort token. Abort tokens are *not* a proof that the exchange has been canceled, as explained below.

5.3.2 Assumptions

The protocol uses conventional, universally-verifiable digital signatures and a hash function. We write $\text{S-Sig}_i(\ldots)$ for a message signed by party i and assume that all protocol participants have the ability verify signatures produced by any party. We also assume that there exists a collision-resistant one-way hash function, Hash().

Prior to executing the protocol, the parties are assumed to agree on each other's identity, the identity of the trusted third party T, and the contractual text. It is also assumed that every protocol participant knows everybody else's signature verification key, which is typically the public key. This implies that the protocol must be preceded by the "handshake" phase in which a key exchange and/or authentication protocol is executed to establish the shared initial knowledge. Since it is not necessary for the handshake protocol to guarantee fairness, we do not consider it as part of this study.

The original paper [1] is self-contradictory in its description of the assumptions about the communication channels between protocol participants. It first states that the communication channels between any two protocol participants are assumed to be *confidential*, *i.e.*, eavesdroppers will not be able to determine the contents of messages traveling through these channels. This can be achieved by encrypting all messages with the intended recipient's public key. Later, however, the paper states that no assumptions are made about the communication channel between O and R. In [1], it is also assumed that the channels between each participant and the trusted third party T are *resilient*, *i.e.*, any message deposited into the channel will eventually be delivered to its intended recipient. However, there are no time guarantees: the intruder can succeed in delaying messages by an arbitrary, but finite amount of time. In section 5.4 below, we analyze the protocol under various assumptions about the quality of communication channels.

Implicit in the protocol specification is the assumption that the trusted third party T must maintain a permanent database with the status of every protocol run that it has ever been asked to abort or resolve. (Each run can be identified by the first message me_1 — see below.) Abort and resolve requests are processed by T on the first-come, first-served basis. Therefore, in order to ensure fairness, T must always be able to determine whether a particular instance of the protocol has been aborted or resolved already.

5.3.3 Protocol

The ASW protocol consists of three interdependent subprotocols: *exchange*, *abort*, and *resolve*. The parties (O and R) generally start the exchange by following the *exchange* subprotocol. If both O and R are honest and there is no interference from the network, each obtains a valid contract upon the completion of the *exchange* subprotocol. The originator O also has the option of requesting the trusted third party T to abort an exchange that O has initiated. To do so, O executes the *abort* subprotocol with T. Finally, both O and R may each request that T resolve an exchange that has not been completed. After receiving the initial message of the *exchange* protocol, they may do so by executing the *resolve* subprotocol with T.

At the end of the protocol, each party is guaranteed to end up with a valid contract or an abort token. As described briefly above, the protocol definition in [1] provides two forms of contract:

$$\begin{array}{ll} \{me_1, N_O, me_2, N_R\} & \text{(standard contract)} \\ \text{S-Sig}_T\{me_1, me_2\} & \text{(replacement contract)} \end{array}$$

where me_1, me_2, N_O, N_R are defined below. Note that the protocol definition does *not* consider a signed contractual text by itself a valid contract.

Abort tokens have the following form:

$$\text{S-Sig}_T\{aborted, ma_1\}$$

where ma_1 is defined below.

An abort token should *not* be interpreted as a proof that the exchange has been canceled. The protocol does not prevent a dishonest O from obtaining an abort token after signing the contract with R. (In this case, O may have both the abort token and the contract, while R only has the contract). The protocol is designed, however, to prevent one party from receiving *only* the abort token in any situation where the other can receive a valid contract.

Exchange subprotocol. As mentioned earlier, it is assumed that prior to initiating the exchange, the two parties agree on the contractual text ($text$) and the identity of the trusted third party T. They are also assumed to know each other's public verification key. Specifically, O knows the key V_R that can be used to verify messages signed by R, and R knows V_O.

When there is no delay or blockage of network messages and neither party tries to cheat the other, O and R may create a contract by the following steps:

$$\begin{array}{llcl}
O \to R & me_1 & = & \text{S-Sig}_O\{V_O, V_R, T, text, \text{Hash}(N_O)\} \\
R \to O & me_2 & = & \text{S-Sig}_R\{me_1, \text{Hash}(N_R)\} \\
O \to R & me_3 & = & N_O \\
R \to O & me_4 & = & N_R
\end{array}$$

In the first step of the subprotocol, O commits to the contractual text by hashing a random number N_O, and signing a message that contains both $\text{Hash}(N_O)$ and $text$. N_O is called the *contract authenticator*. While O does not actually reveal the value of the contract authenticator to the recipient of message me_1, O is committed to it. As in a standard commitment protocol, we assume that the hash function is 2nd-preimage resistant: it is not computationally feasible for O to find a different number N'_O such that $\text{Hash}(N'_O) = \text{Hash}(N_O)$.

In the second step, R replies with its own commitment. Finally, O and R exchange the actual contract authenticators. At the end of the exchange, both O and R obtain a standard contract of the form $\{me_1, N_O, me_2, N_R\}$.

Abort subprotocol. The initiator O may attempt to abort the exchange. An honest O may do this if a reply from R is not received within a reasonable amount of time. To abort, O sends an abort request to the trusted third party T by signing the first message me_1 of the exchange together with *aborted*. The exact format of *aborted* is not specified in [1]; we assume that it is some predefined bit string.

Here are the steps of the *abort* subprotocol, with further description of T's action below:

$$
\begin{array}{llcl}
O \to T & ma_1 & = & \text{S-Sig}_O\{\mathit{aborted}, me_1\} \\
T \to O & ma_2 & = & \text{Has } me_1 \text{ been resolved already?} \\
 & & & \quad \text{Yes}: \ \text{S-Sig}_T\{me_1, me_2\} \\
 & & & \quad \text{No}: \ \ \text{S-Sig}_T\{\mathit{aborted}, ma_1\} \\
 & & & \qquad\qquad \mathit{aborted} := \text{true}
\end{array}
$$

When T receives an abort request, T checks its permanent database of past actions to decide how to proceed. If T has not previously been requested to resolve this instance of the protocol, T marks me_1 as aborted in its permanent database and sends an abort token to O. If me_1 is already marked as resolved, this means that T has previously resolved this exchange in response to an earlier request (as described below). T must have obtained both me_1 and me_2. Therefore, in response to O's abort request, T creates a replacement contract S-Sig$_T\{me_1, me_2\}$ and sends it to O.

Since T stores the result of aborting (indicated by *aborted* := true) in its permanent database, an abort token is effectively a promise by T that it will not resolve this instance of the protocol in the future. As mentioned above, an abort token is *not* a proof that the exchange has been aborted, as the parties can complete contract signing without involving T if they follow the *exchange* subprotocol.

It is useful to bear in mind that while an honest O may send an abort request to T if it does not receive me_2 within a reasonable time, there is no guarantee that O will be able to abort. If the exchange has been already resolved by someone who knows both me_1 and me_2, T will not grant the abort request and will send O a replacement contract instead — even if O has not received me_2. Note also that even though R is not allowed to send abort requests to T, this does not put R at a disadvantage since it has the option of simply ignoring all messages from O.

Resolve subprotocol. Either party may request that T resolve the exchange. In order to do so, the party must possess both me_1 and me_2. Therefore, R can send a resolve request at any time after receiving me_1, and O can do so at any time after receiving me_2. When T receives a resolve request, it checks whether me_1 is already marked as aborted. If it is, T replies with the abort token, otherwise it marks me_1 as resolved and generates a replacement contract by counter-signing the resolve request.

Below, we show the resolve protocol between R and T. The protocol between O and T is symmetric.

$$
\begin{array}{llcl}
R \to T & mr_1 & = & \{me_1, me_2\} \\
T \to R & mr_2 & = & \text{Has } me_1 \text{ been aborted already?} \\
 & & & \quad \text{Yes}: \ \text{S-Sig}_T\{\mathit{aborted}, ma_1\} \\
 & & & \quad \text{No}: \ \ \text{S-Sig}_T\{me_1, me_2\} \\
 & & & \qquad\qquad \mathit{resolved} := \text{true}
\end{array}
$$

Although the generated contract has a different form than the contract produced by the *exchange* subprotocol, the protocol design assumes that in any transaction requiring a contract, either form would be accepted as binding. In other words, the protocol designers consider the definition of contract to be part of the protocol specification and choose to use two forms of valid contract in their protocol.

The first request received by T determines the permanent status of the protocol. After T resolves or aborts the protocol for the first time, it should send identical replies in response to all future requests. If the first request to reach T is an abort request from O, T's response to all requests will be the abort token. If the first request to reach T is a resolve request from O or R, T's response to all requests will be the replacement contract. This leads to an implicit race condition which is not, however, a violation of fairness requirements as defined in section 5.3.4.

5.3.4 Correctness conditions

The designers claim that the ASW protocol has the following properties:

Claim 1.

If the communication channel between O and R is resilient, the protocol satisfies the following requirement:

Effectiveness. If both parties behave correctly and do not want to abandon the exchange, then when the protocol has completed, each has the other's commitment and authenticator, *i.e.*, O has $\text{Hash}(N_R)$ and N_R, while R has $\text{Hash}(N_O)$ and N_O.

Claim 2.

If the communication channels between O and T, and R and T are resilient, the optimistic contract signing protocol satisfies the following requirements:

Strong fairness. When the protocol has completed, either both O and R have valid contracts, or neither one does.

Timeliness. At the beginning of the exchange, every participant can be sure that the protocol will be completed within finite time. At completion, the state of the exchange will either be final, or, in the words of the protocol designers, any changes to it will not "degrade the level of fairness" achieved by the participant so far. For example, if a party has not been cheated at the end of the protocol, it cannot be cheated later on.

Non-repudiability. After an effective exchange (see above), each participant P will be able to prove the origin of the valid contract it has received, and prove that P's protocol counterparty has received P's authenticator or a valid replacement contract from T.

Third party accountability. If the trusted third party T can be forced to eventually send a valid reply to every request, then any participant who is cheated as a result of T's misbehavior will be able to prove that T misbehaved in an external dispute.

5.4 Analysis of the ASW protocol

In order to search for protocol errors, we implemented the *exchange*, *abort*, and *resolve* subprotocols in the Murφ language. The protocol was combined with the standard intruder model described in section 2.2. Most of the correctness conditions of section 5.3.4 were stated as Murφ invariants. During state exploration, Murφ checks that each invariant holds in every reachable state. Conditions such as timeliness and non-repudiability cannot be trivially represented as state invariants, and are discussed informally below.

Our first attempt to analyze the protocol failed because according to the protocol specification, the trusted third party T is always ready to accept abort and resolve requests. Therefore, if one of the parties is strongly corrupt (*i.e.*, the intruder has access to its signing key — see section 5.4.1 below), then in every state of the protocol the intruder can generate a new resolve or, if O is the corrupt party, abort request and send it to T. The trusted third party will then add the request to its database, resulting in a new, larger state. This makes the state space of the protocol infinite. The only solution is to arbitrarily limit the number of times the intruder can generate a request to T in the course of one instance of the protocol. This restriction is not necessary if there are no corrupt parties, since there is only a finite number of frivolous requests that can be computed by the intruder. However, Murφ analysis is slowed down considerably if in every state there is an enabled rule allowing the intruder to send a request to T. This section describes the results of our analysis with the intruder limited to no more than 2 requests to T per protocol instance.

We discuss the modeling of corrupt protocol participants in section 5.4.1. The subsequent subsections discuss the analysis of each protocol correctness condition in turn. Finally, we suggest repairs to the protocol in section 5.4.6.

5.4.1 Modeling corrupt participants

Fair exchange protocols must protect an honest participant from being cheated by a malicious counterpart. Therefore, analysis of a fair exchange protocol must consider the possibility of one or more participants becoming corrupt and cooperating with the intruder.

There are several ways to model a corrupt protocol participant in Murφ. In the simplest case, the corrupt participant is assumed to share its private key with the intruder, enabling the latter to sign and decrypt messages on its behalf. This is equivalent to the intruder using the corrupt party as an oracle for signing and decrypting messages. We will call such collaboration with the intruder *strong corruption*.

A weaker form of corruption occurs when a protocol participant does not share its key with the intruder, and does not sign any messages it is not supposed to sign in the normal course of the protocol. However, it may be willing to engage the intruder's help in obtaining an unfair advantage in the exchange or contract signing process. This may involve accepting messages from the intruder and lying to an outside party about their source, *e.g.*, by claiming that they arrived from the protocol counterpart or T through the standard communication channels. We will call this *weak corruption*.

A weakly corrupt protocol participant is akin to a fence who is willing to accept hot goods without asking too many questions but will not do anything overtly illegal himself. A contract signing protocol that does not protect an honest participant from being cheated by a weakly corrupt counterpart defeats its own purpose and is largely useless. In the real world, it is impossible to be sure that an untrusted agent is not weakly corrupt, *i.e.*, that it is not acting in collusion with the intruder who has control over the public network on which the contract is negotiated.

The weakest form of corruption is the case when a participant, perhaps unintentionally, gives the intruder an ability to monitor (but not to modify or re-schedule) all incoming network traffic. This kind of corruption does not require that the corrupt party has a malicious intent. All the intruder needs is an oversight in network protection. For example, careless disposal of incoming messages may enable the intruder to root through the garbage and read all discarded messages. We will call this form of corruption *accidental corruption*.

5.4.2 Fairness

We started the analysis by verifying the strong fairness property of the protocol (see section 5.3.4). As a reminder, strong fairness guarantees that when the protocol has completed, either both protocol participants have valid contracts, or neither one does.

Confidential channels, one instance of the protocol. First, we analyzed one run, or instance of the protocol under the assumption that all communication channels are confidential. This prevents the intruder from learning anything from the messages as they pass through the network. The only operation the intruder can perform in this setting is to store a message and replay it later.

In fact, the protocol specification [1] says that the protocol provides fairness if the communication channel between O and R, or those between O, R and T, is *resilient*, *i.e.*, any message deposited in the channel will eventually be delivered to the recipient. Resilience is not a safety property and requires special effort to model with Murφ. Adding liveness properties to Murφ (*e.g.*, by means of rules that are enabled only in "final" states where no other rules apply) is a topic of current research.

For the purposes of this study, we made all protocol invariants conditional on the protocol's successful completion. Therefore, in order for an attack on the protocol to succeed, the intruder must deliver messages to their intended recipients so that the latter are convinced that they have successfully completed the protocol. As long as one of the parties is in a state where it's waiting for a message, the protocol is not complete, and there are no fairness guarantees. This approximation of resilience actually *strengthens* the intruder by not requiring it to eventually forward *all* intercepted messages to their intended recipients. Therefore, if Murφ did not find any attack on the protocol in our model, it would not have found any attacks in the model where the channels are both confidential and resilient.

Murφ did not discover any violations of strong fairness by analyzing a single instance of the protocol under the channel confidentiality assumption. It did discover that the intruder can achieve the following:

- Prevent O from aborting the protocol by delaying its abort request to T until R computes me_2, and then submitting me_1 and me_2 (ostensibly from R) to T, thus resolving the protocol. Then O will receive a replacement contract in response to its abort request.
- Force O to submit an abort request to T by intercepting me_2.
- Force R (respectively, O) to submit a resolve request to T by intercepting me_3 (me_4).
- Resolve the protocol directly by submitting a resolve request to T once both me_1 and me_2 have been sent into the network as part of the *exchange* subprotocol.

None of the above, however, is a violation of strong fairness as defined in section 5.3.4.

Confidential channels, two instances of the protocol. After increasing the bound on the number of protocol instances, Murφ discovered the following replay attack:

I observes an instance of the protocol

$$\begin{array}{lllrcl}
O & \rightarrow & R & me_1 & = & \text{S-Sig}_O\{V_O, V_R, T, \mathit{text}, \text{Hash}(N_O)\} \\
R & \rightarrow & O & me_2 & = & \text{S-Sig}_R\{me_1, \text{Hash}(N_R)\} \\
O & \rightarrow & R & me_3 & = & N_O \\
R & \rightarrow & O & me_4 & = & N_R
\end{array}$$

Later ...

$$\begin{array}{lllrcl}
I & \rightarrow & R & me_1 & = & \text{S-Sig}_O\{V_O, V_R, T, \mathit{text}, \text{Hash}(N_O)\} \\
R & \rightarrow & O & me'_2 & = & \text{S-Sig}_R\{me_1, \text{Hash}(N'_R)\} \quad \langle I \text{ intercepts}\rangle \\
I & \rightarrow & R & me_3 & = & N_O \\
R & \rightarrow & O & me'_4 & = & N'_R \quad \langle I \text{ intercepts}\rangle
\end{array}$$

To stage this attack, the intruder must observe an instance of the protocol, recording all messages sent by O. After the protocol completes, the intruder can initiate another instance of the protocol by replaying the recorded me_1. R will respond with a new me'_2, to which the intruder responds with the old me_3. The result of this attack is that the intruder can get R to commit to the text of an old contract with O without O's or T's knowledge.

The protocol as described in [1] contains no protection against this kind of attack. Perhaps this was a conscious decision on the part of the protocol designers who did not intend the protocol to be secure against replay attacks. If the contractual text contains a timestamp, expiration date, or some other information that might help in determining its freshness, R may be able to detect the attack. It can be argued that any well-written contract must contain such information. However, this should be stated explicitly as part of the protocol specification and not left for the protocol user to infer.

The replay attack discovered by Murφ discovered is different from the simpler one in which a malicious R keeps the old contract to which O had previously committed and tries to reuse it. In case of our replay attack, the new contract is *different* from the old one. Recall that a standard contract is the combination of me_1, me_2, and contract authenticators: $\{me_1, me_2, N_O, N_R\}$. Since me_2 is different in the second instance of the protocol, the contract is different. This implies that O cannot even obtain a valid replacement contract by requesting it from the trusted third party since in order to do so, it needs me_2' which it never receives. In fact, O is not even aware that an exchange between R and the intruder has taken place.

The replay attack succeeds even if both O and R are honest. Suppose that O is a retailer who periodically purchases supplies from R online using the contract signing protocol. All purchase contracts are exactly the same, as is often the case in real life, and it is agreed (offline) that all contracts expire immediately upon fulfillment (*i.e.*, R receives the order, fills it, and forgets about it). Then the intruder can use the replay attack to impersonate O and submit a false purchase contract on its behalf, convincing R that O has committed to a new purchase and providing R with a false proof of O's commitment.

Note that there is no need for the intruder to involve the trusted third party in the protocol in order to stage the replay attack. This means that there will be no evidence of the attack such as could have been provided by a resolve request kept by T.

The main weakness of the protocol is the fact that O's message me_3 that contains the contract authenticator is sent in response to R's commitment message me_2 but is not related to it in any way, making it possible for the intruder to replay an old me_3. A small change to the protocol that prevents the attack is described in section 5.4.6.

Standard channels. After repairing the protocol to prevent the replay attack, we performed Murφ analysis without the confidentiality assumption on the channels but still within the constraints of the standard Dolev-Yao intruder model (see section 2.2). Murφ did not discover any new attacks. This can be attributed to the fact that messages $me_{1,2}$, $ma_{1,2}$, and mr_2 are all signed, and mr_1 contains signed messages as its components. Assuming that every protocol participant knows everybody else's correct public key (this is a necessary requirement for the protocol to succeed even in the absence of the intruder), signatures prevents the intruder from modifying messages in transit. Since no signing keys are transmitted as part of the protocol, the intruder cannot gain the ability to sign messages unless one of the parties leaks its key. Therefore, the intruder is just as powerful as in the case of confidential channels.

This result suggests that the channel confidentiality assumption can be relaxed. The protocol ensures fairness even if the channels are controlled by a Dolev-Yao intruder.

Corrupt protocol participant. Finally, we analyzed the protocol under the assumption that one of the participants is corrupt. We modeled this by giving the intruder access to the corrupt party's private information such as its private key and contract authenticator even before the latter is divulged as part of the *exchange* subprotocol. This corresponds to *strong corruption* as described in section 5.4.1. The intruder is effectively indistinguishable from the corrupt participant in this case, thus we can model the protocol under the assumption that the corrupt participant has full control over the network.

Murφ discovered that a corrupt O can obtain *both* an abort token signed by T and a valid contract with R. In fact, O does not require assistance from the intruder to do this. It can simply execute the *exchange* subprotocol with R and then the *abort* subprotocol with T. As a result, R obtains a valid contract with O, while O obtains a contract with R and T's abort token. Since both parties have the contract, this is not a violation of fairness as defined in section 5.3.4, but it also implies that abort tokens may not be accepted as a proof that contract negotiation failed.

Murφ also discovered the following attack, in which a malicious R obtains a contract which is inconsistent with that obtained by O.

$$
\begin{array}{lllrcl}
O & \rightarrow & R & me_1 & = & \text{S-Sig}_O\{V_O, V_R, T, \mathit{text}, \text{Hash}(N_O)\} \\
R & \rightarrow & O & me_2 & = & \text{S-Sig}_R\{me_1, \text{Hash}(N_R)\} \\
 & & & \multicolumn{3}{l}{R \text{ computes new random } N'_R \text{ and}} \\
 & & & \multicolumn{3}{l}{me'_2 = \text{S-Sig}_R\{me_1, \text{Hash}(N'_R)\},} \\
 & & & \multicolumn{3}{l}{\text{but keeps them secret}} \\
O & \rightarrow & R & me_3 & = & N_O \\
 & & & \multicolumn{3}{l}{R \text{ sends nothing}} \\
O & \rightarrow & T & mr_1 & = & \{me_1, me_2\} \\
T & \rightarrow & O & mr_2 & = & \text{S-Sig}_T\{me_1, me_2\}
\end{array}
$$

In this attack, R computes two different responses me_2 and me'_2 to O's initial message me_1 using different random numbers N_R and N'_R. It sends out me_2 and keeps the other secret. After it receives O's contract authenticator N_O, R does not respond at all. It has already obtained a valid standard contract $\{me_1, N_O, me'_2, N'_R\}$. Since O does not receive me_4 from R, it requests trusted third party T to resolve the protocol. T issues a replacement contract by counter-signing me_1 and me_2. However, O's contract is *different* from that possessed by R because it contains the hash of a different random number: N_R rather than N'_R.

Clearly, this is a problem, since each party possesses a valid contract, but the two contracts are inconsistent. Recall that the protocol employs a non-standard definition of contracts (section 5.3.3), according to which a valid contract is *more* than a signed contractual text. Even though the contractual texts in the two contracts are the same, the random numbers and commitments are different, and it is unclear how the contracts should be enforced or interpreted, given that both are valid according to the protocol specification. The original paper [1] does not say anything about how this situation should be handled.

This problem is caused by the same weakness of the protocol that makes the replay attack possible. O's contract authenticator N_O is sent in response to me_2 but is not explicitly linked to it. This enables R to use N_O with a different message me'_2 to form a valid contract without revealing its own commitment to O. More generally, Murφ analysis points to the fact that O's half of the contract contains *no* information that links it to R's half of the contract. The modification of the protocol described in section 5.4.6 prevents this attack, too.

5.4.3 Timeliness

Eventual completion of the protocol is guaranteed by channel resilience. Since we did not fully model resilience, it is possible for the intruder in our model to prevent the protocol from completing, but this attack is trivial.

The concept of "fairness degradation" is not defined in the paper [1] and thus difficult to formalize so that it can be checked by a finite-state analysis tool. Based on our informal understanding of the protocol, if fairness is achieved at the end of the protocol, then it cannot be lost since the protocol provides no means of invalidating a contract. If a party has the valid contract once the protocol completes, then it cannot be cheated regardless of whether the other party has the contract or not. If a party does not have a valid contract, then the other party must not have a contract either (otherwise, there is no fairness). The only remaining question is whether it is possible to reuse information from an instance of the protocol that did not result in a valid contract in another instance that does produce a valid contract (then even if the first instance of the protocol was fair, fairness will be lost in the second instance). Murφ did not find any attacks of this nature.

5.4.4 Non-repudiability

Non-repudiability condition (see section 5.3.4) requires that after an honest party completes the protocol, it must be able to prove the origin of the valid contract it receives. Since the ability to prove something is difficult to formalize, we did not attempt to verify non-repudiability with Murφ. One can use informal reasoning to conclude that since commitment messages are signed and it is assumed that the signature scheme is secure, O's signature on me_1, R's signature on me_2, and T's signature on mr_2 prove the origin of any valid contract, whether it is a standard contract computed by O and R, or a replacement contract issued by T. Non-repudiability of receipt is conditional on fairness: if O has a valid contract at the completion of the protocol, then R must have a valid contract, too (otherwise, there is no fairness). Therefore, R must have received O's contract authenticator or a replacement contract from T. R's non-repudiability of receipt is symmetric. Unfortunately, this sort of reasoning is difficult to verify with a finite-state tool.

5.4.5 Trusted third party accountability

The ASW protocol is not intended to guarantee fairness if the trusted third party T is corrupt. However, *third party accountability* implies that if one of the participants loses fairness as a result of T's misbehavior, it should be able to prove this misbehavior to an independent arbiter.

Accountability only holds if the trusted third party is guaranteed to send a valid response to all requests. Our Murφ model approximates this guarantee by making all protocol invariants, including third party accountability, conditional on the protocol's successful completion. Therefore, in order to succeed, any attack staged by the intruder, possibly acting in collusion with corrupt T, must involve generating a valid response to every request sent by an honest participant. Otherwise, the honest participant will not complete the protocol, and the attack will fail. Also, accountability is only feasible if O is notified when R tries to enforce a contract and vice versa. If a protocol participant does not know that it is being cheated, it cannot go after T to prove its misbehavior.

Before formulating a formal protocol invariant that could be verified with the help with Murφ, we had to determine what it means to be able to prove T's misbehavior. Based on our interpretation of the protocol description in [1], we believe that the cheated protocol participant can prove that T misbehaved if and only if it can produce two documents, both signed by T, that contradict each other. More specifically, the cheated participant must be able to demonstrate an abort token signed by T *and* a replacement contract for the same instance of the protocol, also signed by T. Since T is supposed to process all abort and resolve requests on the first-come, first-served basis and the initial request determines the status of the protocol in perpetuity, it should never be the case that T issues both an abort token and a replacement contract for the same instance of the protocol.

Based on the above interpretation, we believe that third party accountability is violated *if and only if* the following conditions hold (the conditions are formulated assuming that O is the cheated party; the conditions for R are symmetric):

- T is corrupt (modeled by giving the intruder access to T's signing key).
- R has O's contract authenticator.
- O has neither R's contract authenticator, nor a replacement contract signed by T.

If R has a replacement contract signed by T instead of a standard contract with O's contract authenticator, then T is always accountable! Suppose that R tries to enforce its replacement contract. When O goes to T and requests to either abort, or resolve the protocol, T must send O a valid response. If T sends a replacement contract, then there is no fairness violation and O is not cheated since both parties possess the same contract. If T sends an abort token, then O is indeed cheated (since R has a contract and O does not), but O can then prove T's misbehavior by demonstrating its abort token and R's replacement contract, both signed by T.

However, if R has a standard contract with O's contract authenticator, then R's contract is *not* signed by T, and O cannot prove T's misbehavior since it cannot produce two inconsistent documents signed by T. This case satisfies the conditions listed above.

Murφ did not discover any states reachable by the protocol that satisfy the conditions. We conclude that the third party is indeed accountable.

5.4.6 Repairing the protocol

The ASW protocol can be repaired so as to prevent the attacks described in section 5.4.2 by explicitly linking message me_3 with message me_2. This is a standard technique to ensure that an old me_3 cannot be replayed by the intruder in response to a fresh me_2 and that R can obtain a standard contract only with the same contract authenticator that it has sent to O as part of me_2. A similar change must be made to me_4 to prevent a symmetric replay attack.

$$\begin{array}{llcl} O \to R & me_1 & = & \text{S-Sig}_O\{V_O, V_R, T, \mathit{text}, \text{Hash}(N_O)\} \\ R \to O & me_2 & = & \text{S-Sig}_R\{me_1, \text{Hash}(N_R)\} \\ O \to R & me_3 & = & \textbf{S-Sig}_{\mathbf{O}}\{N_O, \mathbf{Hash}(\mathbf{N_R})\} \\ R \to O & me_4 & = & \textbf{S-Sig}_{\mathbf{R}}\{N_R, \mathbf{Hash}(\mathbf{N_O})\} \end{array}$$

5.5 Garay-Jakobsson-MacKenzie protocol

In this section, we describe the abuse-free optimistic contract signing protocol of Garay, Jakobsson, and MacKenzie [24]. The protocol is closely related to the ASW protocol described above. Both involve a 4-step *exchange* subprotocol, and similar *abort* and *resolve* subprotocols. Even though the two protocols have similar structure, the actual contents of the messages differ. Unlike the ASW protocol, the GJM protocol is designed to guarantee abuse-freeness in addition to fairness and third party accountability.

This section follows the pattern of section 5.3. We start by briefly describing the objectives of the GJM protocol, and then explain the properties of *private contract signatures* (PCS), an innovation of Garay, Jakobsson, and MacKenzie used to make contract signing abuse-free. Finally, we describe the protocol steps in detail and formalize the correctness conditions.

5.5.1 Objectives and assumptions

The GJM protocol is designed to enable two parties, O and R, to exchange signatures on a contractual text. It is assumed that prior to executing the protocol, the parties agree on each other's identity, the contractual text, and the identity of the trusted third party T. Every protocol participant is assumed to know the correct signature verification key of the other party and T. As above, we write $\text{S-Sig}_i(m)$ for the result of signing text m with the key of party i. It is also assumed that every participant has a private communication channel with T.

The protocol is asynchronous. As the exchange protocol progresses, either participant may contact the trusted third party T. The third party may decide, on the basis of the communication it received, to either resolve the protocol by issuing the other party's signature, or "abort" the protocol by issuing an abort token. As in the ASW protocol, abort tokens are *not* a proof that the exchange has been canceled. The intruder may schedule messages and insert its own messages in the network, but cannot delay messages sent between participants and T indefinitely.

5.5.2 Private Contract Signatures

The GJM protocol relies on the cryptographic primitive called *private contract signature* (PCS). We write $\text{PCS}_O(m, R, T)$ for party O's private contract signature of text m for party R (known as the *designated verifier*) with respect to third party T. The main properties of PCS are summarized below:

- $\text{PCS}_O(m, R, T)$ can be verified like a conventional signature, *i.e.*, there exists a probabilistic polynomial-time algorithm PCS-Ver such that
  $\text{PCS-Ver}(m, O, R, T, s)$ is *true iff* $s = \text{PCS}_O(m, R, T)$.

- $\text{PCS}_O(m, R, T)$ can be feasibly computed by either O, or R, but nobody else. This is the key property of PCS that distinguishes it from a conventional, universally-verifiable signature, as the latter can only be computed by O. When the designated verifier R receives $s = \text{PCS}_O(m, R, T)$, he will be convinced that s was computed by O, but, unlike O's conventional signature, s cannot be used by R to prove this to an outside party.

- $\text{PCS}_O(m, R, T)$ can be converted into a conventional signature by either O, or T, but nobody else, including R. For the purposes of this study, we focus on the *third-party accountable* version of PCS, in which the converted signatures produced by O and T can be distinguished. We will call them $\text{S-Sig}_O(m)$ and $\text{TP-Sig}_O(m)$, respectively. Unlike PCS, converted signatures are universally verifiable by anybody in possession of the correct signature verification key.

An efficient discrete log-based PCS scheme is presented in [24].

5.5.3 Protocol

The GJM protocol consists of three interdependent subprotocols: *exchange*, *abort*, and *resolve*. Message sequences are identical to those of the ASW protocol (see section 5.3.3). The parties (O

and R) start the exchange by following the *exchange* subprotocol. If both O and R are honest and there is no interference from the network, they obtain each other's signatures as the final steps of the *exchange* subprotocol. The originator O also has the option of requesting the trusted third party T to abort an exchange that O has initiated. To do so, O executes the *abort* subprotocol with T. Finally, both O and R may each request that T resolve an exchange that has not been completed. After receiving the initial message of the *exchange* protocol, they may do so by executing the *resolve* subprotocol with T.

At the end of the protocol, each party is guaranteed to end up with the other party's universally-verifiable signature of the contractual text, or an abort token signed by T and O, of the form $\text{S-Sig}_T(\text{S-Sig}_O(m, O, R, \textit{abort}))$.

Exchange subprotocol. When there is no interference from the network and neither party tries to cheat the other, O and R may exchange signatures by the following steps:

$$\begin{array}{llcl}
O \to R & me_1 & = & \text{PCS}_O(m, R, T) \\
R \to O & me_2 & = & \text{PCS}_R(m, O, T) \\
O \to R & me_3 & = & \text{S-Sig}_O(m) \\
R \to O & me_4 & = & \text{S-Sig}_R(m)
\end{array}$$

In the first step of this subprotocol, O commits to the contractual text m by producing a private contract signature of m with R as the designated verifier. The purpose of PCS is to convince R that O signed m, while depriving R of the possibility to prove this to an outside party. In the second step, R replies with its own PCS of m with O as the designated verifier. Finally, O and R exchange their actual, universally-verifiable signatures of m. At end of the exchange, both O and R obtain a signed contract of the form $\{\text{S-Sig}_O(m), \text{S-Sig}_R(m)\}$.

Abort subprotocol. As in the ASW protocol, O may attempt to abort the exchange if it times out waiting for a reply from R. Here are the steps of the subprotocol:

$$\begin{array}{llcl}
O \to T & ma_1 & = & \text{S-Sig}_O(m, O, R, \textit{abort}) \\
T \to O & ma_2 & = & \text{Has } O \text{ or } R \text{ resolved already?} \\
 & & & \quad \begin{array}{rll} \text{Yes}: & \text{S-Sig}_R(m) & \text{if } R \text{ has resolved, or} \\ & \text{TP-Sig}_R(m) & \text{if } O \text{ has resolved} \\ \text{No}: & \text{S-Sig}_T(ma_1) & \\ & \textit{aborted} := \text{true} & \end{array}
\end{array}$$

When T receives an abort request, T checks its permanent database of past actions to decide how to proceed. If T has not previously been requested to resolve this instance of the protocol, T marks m as aborted in its permanent database and sends an abort token to O. If m is already marked as resolved, this means that T has previously resolved this exchange in response to an earlier request. As a result of the resolution procedure (described below), honest T must have obtained both O's and R's universally-verifiable signatures of m. Therefore, in response to O's abort request, T forwards O either S-Sig$_R(m)$ or TP-Sig$_R(m)$, either of which can serve as a proof that R indeed signed m.

An abort token is a promise by T that it will not resolve this instance of the protocol in the future. It is not a proof that the exchange has been aborted, as the parties can complete contract signing without involving T if they follow the *exchange* subprotocol.

Resolve subprotocol. Either party may request that T resolve the exchange. In order to do so, the party must possess the other party's PCS of the contract (with T as the designated third party), and submit it to T along with its own universally-verifiable signature of the contract. Therefore, R can send a resolve request at any time after receiving me_1, and O can do so at any time after receiving me_2. When T receives a resolve request, it checks whether the contract is already marked as aborted. If it is, T replies with the abort token. If the contract has been resolved by the other party, T replies with that party's signature. Finally, if the contract has been neither aborted, nor resolved by the other party, T converts PCS into a universally-verifiable signature, sends it to the requestor, and stores the requestor's own signature in its private database.

Below, we show the resolve protocol between R and T. The protocol between O and T is symmetric.

$$
\begin{array}{lrcl}
R \to T & mr_1 & = & \mathrm{PCS}_O(m, R, T), \text{S-Sig}_R(m) \\
T \to R & mr_2 & = & \text{Has } O \text{ aborted already?} \\
 & & & \quad \text{Yes}: \ \text{Send S-Sig}_T(\text{S-Sig}_O(m, O, R, \mathit{abort})) \\
 & & & \quad \text{No}: \ \text{Has } O \text{ resolved already?} \\
 & & & \qquad\qquad \text{Yes}: \ \text{Send S-Sig}_O(m) \\
 & & & \qquad\qquad \text{No}: \ \text{Store S-Sig}_R(m) \\
 & & & \qquad\qquad\qquad\quad \text{Convert } \mathrm{PCS}_O(m, R, T) \text{ into} \\
 & & & \qquad\qquad\qquad\qquad\qquad \text{TP-Sig}_O(m) \\
 & & & \qquad\qquad\qquad\quad \text{Send TP-Sig}_O(m) \\
 & & & \qquad\quad \mathit{resolved} := \text{true}
\end{array}
$$

As in the ASW protocol, the first request received by T determines the permanent status of the protocol.

5.5.4 Correctness conditions

The designers claim that the GJM protocol has the following properties:

Completeness. A restricted adversary cannot prevent a set of correct participants from obtaining a valid signature of a contract. The restricted adversary has signing oracles that can be queried on any message except the contractual text m and can arbitrarily schedule messages from participants to T, but cannot delay messages between the correct participants enough to cause any timeouts.

Fairness. The GJM protocol satisfies the following fairness conditions:

- It is impossible for a corrupt participant to obtain a valid contract without allowing the remaining participant to also obtain a valid contract.
- Once an honest participant obtains a cancellation message (*i.e.*, an abort token) from the trusted third party T, it is impossible for any other participant to obtain a valid contract.
- Every honest participant is guaranteed to complete the protocol.

Abuse-freeness. It is impossible for a protocol participant, at any point in the protocol, to be able to prove to an outside party that he has the power to choose between aborting and successfully completing the contract. One of the main contributions of [24] is to introduce the notion of abuse-freeness to electronic contract signing.

Trusted third party accountability. If one of the parties is cheated because of T's misbehavior, the cheated party should be able to prove to an outside arbiter that T misbehaved. It is not specified precisely in [24] what can serve as a proof of misbehavior, but typically such proof consists of two contradictory messages signed by T [2], *e.g.*, an abort token and a converted PCS signature of the same text m. Since the steps of the protocol do not allow T to both abort and resolve the protocol, any PCS conversion performed by T after it aborted the protocol (and vice versa) serves as a proof of T's misbehavior.

There are actually two versions of the GJM protocol, one providing third party accountability and the other not. The difference between the two protocols lies in two versions of PCS. In our analysis, we focus on the case when the PCS scheme provides third-party accountability, *i.e.*, the distributions of S-Sig$_i(m)$ and TP-Sig$_i(m)$ are disjoint, and thus it is possible for the verifier to distinguish whether the signature is a "real" signature of i, or a PCS of i converted by T.

5.6 Analysis of the GJM protocol

We used Murφ to perform finite-state analysis of the GJM protocol. Corrupt protocol participants were modeled as described in section 5.4.1. Since abuse-freeness cannot be trivially represented as a state invariant, we employed a partial verification method described in section 5.6.1 below. The rest of the section describes the analysis of each protocol correctness condition in turn. In section 5.6.6, we suggest repairs to the protocol.

5.6.1 Modeling abuse-freeness

Our approach to verifying whether the protocol is abuse-free consists of two parts. First, we use Murφ to determine whether any protocol participant possesses the power to determine the outcome of the protocol regardless of the actions of the other party, assuming the other party is honest and genuinely interested in signing the contract. This is done by augmenting the system with an additional outside party we call the *Challenger*.

In order to verify whether a participant P has the power at some point in the protocol, we have it send a message to the Challenger asserting its control over the outcome. The Challenger then nondeterministically chooses a desired outcome: abort or successful contract completion. (It is a consequence of fairness that there are only two possible outcomes: either T aborts and no one receives a signed contract or both parties receive a signed contract.)

After receiving the Challenger's request, P has to interact with the honest participant in such a way so as to drive the protocol to the requested outcome. If there exists a trace in which the outcome of the protocol is not consistent with that requested by the Challenger, we conclude that P does not possess the power to determine the outcome. The key idea here is that determining whether P satisfies the Challenger's request is a state invariant and can be verified by Murφ.

The second part of abuse is that a participant P with the power to determine the outcome must be able to prove this to an outside arbiter. However, we have not formulated a straightforward way of verifying properties such as "P can prove something" in Murφ. Therefore, we have only analyzed this part of the protocol by informal means.

Our analysis of abuse-freeness of the original and repaired GJM protocols can be found in sections 5.6.5 and 5.6.6, respectively.

5.6.2 Completeness

To verify the completeness guarantee (section 5.5.4), we used Murφ to analyze the protocol under the assumption that neither protocol participants, nor the trusted third party T are corrupt. We also restricted the intruder by requiring it to forward all messages originating from protocol participants to their intended recipients, and assumed that the channels between the participants and T are completely private, *i.e.*, the intruder cannot eavesdrop on the traffic or introduce new messages into the channels.

Under these restrictions, Murφ failed to find an attack that would prevent the participants from obtaining valid signatures of the contract. Therefore, our analysis confirms that the GJM protocol is indeed complete modulo limitations of the Murφ model (see section 2.3).

5.6.3 Fairness

First, we analyzed the protocol under the assumption that both participants are honest, *i.e.*, neither tries to cheat the other. This also implies that neither participant knowingly cooperates with the intruder. Murφ discovered that the intruder can achieve the following:

- Force O to submit an abort request to T by intercepting me_2.
- Prevent O from aborting the protocol by delaying O's abort request to T until R times out waiting for me_3 and submits a resolve request to T. Then O will receive R's signature in response to its abort request.
- Force R (respectively, O) to submit a resolve request to T by intercepting me_3 (me_4).

Murφ also discovered that O can use the protocol to obtain *both* an abort token signed by T and a valid contract signed by R. To do so, O executes the *exchange* subprotocol with R and then the *abort* subprotocol with T. As a result, R obtains O's signature, while O obtains R's signature and T's abort token.

There is an important difference between the GJM protocol and the ASW protocol. In the latter, the intruder can directly resolve the protocol by submitting a resolve request to T once both me_1 and me_2 have been sent into the network as part of the *exchange* subprotocol. This is impossible in the GJM protocol since resolve requests must include the originating party's signature on the contract which the intruder cannot compute without cooperating with that party.

None of the above is a violation of fairness as defined in section 5.5.4.

In the remainder of this subsection, we focus on the cases when at least one of the protocol participants is malicious or corrupt. For brevity, we omit the discussion of all combinations and concentrate on the most interesting insights about the protocol revealed by our analysis.

Weakly corrupt O, intruder monitors $R \to T$ channel. We analyzed the protocol under the assumption that party O is malicious, *i.e.*, its intention is to cheat R by obtaining R's signature of the contractual text m without releasing its own signature. O is weakly corrupt: it is willing to engage the intruder's help in obtaining R's signature, but will not sign or decrypt messages for the intruder.

The intruder I is assumed to have the ability to eavesdrop on and delay messages sent from R to T, but not to modify or remove them. Below we analyze the protocol under the assumption that the communication channel between R and T is inaccessible to the intruder.

Under these assumptions, Murφ uncovered the following attack:

$$
\begin{array}{lrcl}
O \to R & me_1 & = & \mathrm{PCS}_O(m, R, T) \\
R \to O & me_2 & = & \mathrm{PCS}_R(m, O, T) \\
\multicolumn{4}{l}{\quad I \text{ intercepts } me_2\text{, or } O \text{ receives and discards it}} \\
O \to T & ma_1 & = & \text{S-Sig}_O(m, O, R, \mathit{abort}) \\
R \to T & mr_1 & = & \mathrm{PCS}_O(m, R, T), \text{S-Sig}_R(m) \\
\multicolumn{4}{l}{\quad I \text{ eavesdrops on } mr_1\text{, learns S-Sig}_R(m)\text{, delays } mr_1 \text{ until } T \text{ receives } ma_1} \\
T \to O & ma_2 & = & \text{S-Sig}_T(\text{S-Sig}_O(m, O, R, \mathit{abort})) \\
\multicolumn{4}{l}{\quad I \text{ intercepts } ma_2\text{, or } O \text{ receives and hides it}} \\
T \to R & mr_2 & = & \text{S-Sig}_T(\text{S-Sig}_O(m, O, R, \mathit{abort})) \\
I \to O & & & \text{S-Sig}_R(m, O, T)
\end{array}
$$

As a result, O obtains R's signature of the contract S-Sig$_R(m, O, T)$, while R obtains the abort token from T.

Recall the second fairness condition from section 5.5.4: once a correct participant (R) obtains an abort token from the trusted third party T, it should be impossible for any other participant to obtain a valid contract. Even though Murφ cannot currently be used to verify non-safety properties such as "it is impossible for a participant to obtain a valid contract," this condition can be approximated by the following safety invariant: "it is never the case that the correct participant possesses the abort token, while some other participant possesses a valid contract, if the abort token was received first." Clearly, the above attack violates this invariant.

The first fairness condition is violated as well: the corrupt participant (O) obtained a valid contract without allowing the remaining participant (R) to also obtain a valid contract. The reason for this is that the only information from O that R has in its possession is PCS$_O(m, R, T)$ sent in message me_1. This PCS can be converted into a universally-verifiable signature either by O (who won't do this because it's corrupt), or by T (who won't do this because it has already aborted the protocol, and must send abort tokens in response to all requests). Therefore, R has no means to obtain O's universally-verifiable signature of the contractual text m. This condition, however, is not trivially reduced to a safety invariant and is thus difficult to verify with Murφ.

It is unclear whether this attack is a bona fide violation of fairness. The original paper [24, p. 462] states that if one party shows the abort token, and the other a valid set of signatures S-Sig$_O(m)$, S-Sig$_R(m)$, then the contract must be valid. Indeed, it can be argued that R implicitly agreed to sign the contract by sending its signature to T in message mr_1, even though it received an abort token in response. We do believe that this attack violates abuse-freeness (see section 5.6.5 below).

Weakly corrupt O, accidentally corrupt T. In order to stage the attack described in the previous section, the intruder must be able to access the communication channel between R and T. The original paper [24] specifies that communication between any participant and T is conducted over a private channel. In this case, the intruder will not be able to eavesdrop on message mr_1 sent by R to T in order to resolve the protocol, and will not be able to learn S-Sig$_R(m)$. In fact, even if R and T communicate over a public network, encrypting mr_1 with T's public key will prevent the intruder from splitting it into parts and reusing one of the parts to help O gain an unfair advantage. It is worth noting, however, that the protocol specification in [24] does not require that mr_1 be encrypted.

Now consider the case when the $R \rightarrow T$ channel is secure, but T is *accidentally corrupt*, and I has passive access to all of its incoming communication (see section 5.4.1 for our definition of accidental corruptness). This does not require active cooperation with the intruder on the part of T, just negligence in handling messages it receives from protocol participants. I does not need the ability to split messages into parts, remove them from the network, or even insert its own messages into the network. Having passive access to T's communication with R is sufficient for I to learn S-Sig$_R(m)$ and divulge it to O. Therefore, the attack succeeds in this case.

5.6.4 Trusted third party accountability

Suppose that T is accidentally corrupt and I successfully stages the attack described in section 5.6.3, causing R to lose fairness as a result. Since we are analyzing a TTP-accountable version of the GJM protocol (see section 5.5.4), we would like to verify whether the trusted third party T can be held accountable. The original paper [24] defines a TP-accountable PCS scheme, but does not give a precise definition of TTP accountability. Since the GJM protocol is closely related to ASW protocol, for the purposes of our formal analysis we used the ASW definition of TTP accountability (called "verifiability of trusted third party" in [2, 1]):

"Assuming the third party T can be forced to eventually send a valid reply to every request, the verifiability of trusted third party property requires that if T misbehaves, resulting in the loss of fairness for P, then P can prove the misbehavior of T to an arbiter (or verifier) in an external dispute."

Following the designers of the ASW protocol, we assume that the proof must consist of two inconsistent messages signed by T, *e.g.*, an abort token and a converted PCS. (Recall that in the TTP-accountable version of PCS, TP-Sig$_R(m)$ obtained as a result of T's conversion of PCS$_R(m)$ is distinct from S-Sig$_R(m)$). According to the protocol specification, T must process all requests on the first-come, first-served basis. Therefore, the first request received by T determines the status of the contract in perpetuity, and it should never be the case that T issues an abort token and a converted PCS signature for the same contract.

However, if R loses fairness as a result of T's accidental corruption, it has no means of proving to an outside party that T is corrupt. O is in possession of genuine S-Sig$_R(m)$, *not* a converted PCS.

If O is willing to lie about the source of this signature, then R cannot pin the blame on T. The only message signed by T is the abort token, and in the absence of two inconsistent messages signed by T, it is unclear what R can use as a proof to hold T accountable.

Since abort requests are signed, R can prove that the abort token it received from T was originally generated by O. But protocol specification allows for the case when O obtains a valid signature of R after sending off its abort request. This may happen if, for example, T received R's resolve request before O's abort request, resolved the protocol, and forwarded R's signature in response to O's abort request. O can also claim that it received R's signature directly from R.

At best, R can argue that *either O, or T is lying*: either O is lying that it received R's signature from T in response to its abort request, or T is lying that it received O's abort request before R's resolve request (in the latter case, T would not have sent the abort token in response to R's request). This is a very weak form of accountability - in effect, the cheated party in a 3-party protocol is arguing that one of the other two is lying.

We believe that the difference between the possibilities (O is corrupt, or T is corrupt) is too significant to allow any confusion between the two. The protocol is designed to withstand corrupt participants, so the fact that O is corrupt is fairly trivial. T, on the other hand, plays a crucial role due to its ability to resolve or abort contract signings, and any negligence or dishonesty on the part of T should be immediately detected and, if proved, should lead to revocation of T's authority to function as the trusted third party.

5.6.5 Abuse-freeness

As we mentioned above, it is unclear whether the attack described in section 5.6.3 violates fairness, since R actually signs the contractual text m, implicitly agreeing to the contract. Abuse-freeness, on the other hand, is clearly violated. After receiving S-Sig$_R(m)$ from the intruder and S-Sig$_T$(S-Sig$_O(m, O, R, \mathit{abort})$) from T, O is free to decide whether to enforce the contract using the former, or consider it aborted using the latter. O can present both messages to an outside party, thus proving that it has the power to abort or successfully complete the protocol. Therefore, the GJM protocol is not abuse-free in this case.

The argument about TTP accountability given in section 5.6.4 applies to abuse-freeness as well as to fairness. If R is abused by O as a consequence of T's accidental corruption, R cannot prove to an outside arbiter that T misbehaved.

5.6.6 Repairing the protocol

The basic error in the GJM protocol can be attributed to the fact that data sent in the *resolve* subprotocol are exactly the same as data sent in the *exchange* subprotocol. The GJM protocol can therefore be repaired by replacing the standard signature in each resolve request with PCS. This was independently suggested by the authors of the protocol after we brought the attack described in section 5.6.3 to their attention [40].

In the repaired protocol, resolve requests from R to T will have the following form (requests from O to T are symmetric):

$$mr_1 \quad = \quad \mathrm{PCS}_O(m, R, T),\ \mathbf{PCS_R(m, O, T)}$$

Our analysis of the repaired protocol did not uncover any attacks. Murφ confirmed that R still has the power to determine the outcome of the protocol after receiving the first message from O (see section 5.6.1). However, the only information in R's possession at this point is $\mathrm{PCS}_O(m, R, T)$, and R cannot use it to prove anything to an outside arbiter due to the designated verifier property of PCS (see section 5.5.2). We conclude that the repaired protocol is abuse-free. By contrast, the ASW protocol is not abuse-free. In the ASW protocol, R, too, has the power to determine the outcome after the first message received from O, but since universally-verifiable signatures are used, this power can be proved to an outside arbiter.

Unlike the original protocol, the repaired protocol is TTP-accountable. In the repaired protocol, T never receives universally-verifiable signatures of the contract from either O, or R. Any universally-verifiable signature leaked by corrupt T must be the result of PCS conversion, and its origin can be traced to T if the TTP-accountable version of PCS is used.

Murφ analysis indicates that the private channel assumption for communication between protocol participants and T can be relaxed. Even if the intruder can eavesdrop on messages exchanged with T, the protocol is still fair and abuse-free as long as the channels are *resilient*, *i.e.*, every message is guaranteed to eventually reach its intended recipient. This is significant because this implies that the repaired protocol does not need to operate on top of a secrecy protocol, or use any form of encryption in order to guarantee fairness. The protocol can still be subject to cryptographic attacks on PCS and signature schemes and/or other attacks that could not have been discovered in the Murφ model.

Additional analysis of the repaired protocol has been performed by Satyaki Das using the new-generation Murφ tool that relies on predicate abstractions to analyze infinite state spaces. It did not discover any attacks on an arbitrary number of protocol instances executed by different principals.

5.7 Comparison of the two protocols

The ASW and GJM optimistic contract signing protocols are closely related. Both are designed to guarantee fairness and trusted third party accountability. The GJM protocol is also designed to guarantee abuse-freeness at the cost of relying on a non-standard cryptographic primitive. We used Murφ to analyze both protocols, and believe that our analysis can provide some guidance when choosing between the two protocols and/or deciding which one to deploy.

The notion of a contract in the GJM protocol is very simple and intuitive: a participant's universally-verifiable signature of the contractual text serves as his commitment to the contract. By contrast, contracts in the ASW protocol have a rather complicated form, making understanding and analysis of the protocol difficult. Surprisingly, our Murφ analysis helped us discover an unexpected advantage of ASW contracts.

In the ASW protocol, the trusted third party must sign all contracts it resolves. This is not so in the original GJM protocol: if one party resolved the protocol by appealing to T, then T will simply forward its signature in response to all requests by the other party. The consequence is the failure of TTP accountability, as described in section 5.6.5. A corrupt O can claim with impunity that it received R's signature from R itself or from T even if it actually received it from the intruder. The absence of T's signature on resolved contracts makes it impossible to determine how the contract was obtained.

One of the main advantages of the GJM protocol is its guarantee of abuse-freeness. We believe our Murφ analysis demonstrates that abuse-freeness cannot be guaranteed by the original protocol if the trusted third party T is even accidentally corrupt, let alone if it actively cooperates with the intruder. As we explain in sections 5.6.3 and 5.6.5, T's corruption may lead to O's gaining the ability to prove to an outside party that it is in O's power to abort or successfully complete the contract, but R will be unable to hold T accountable.

It is also worth noting that the ASW protocol can be used with an arbitrary digital signature scheme, while the GJM protocol requires a special type of signatures. Analyzing the computational and cryptographic advantages of different signature schemes is beyond the scope of Murφ. There are also other, relatively minor differences between the protocols. In the ASW protocol, the intruder can force the protocol to be resolved after eavesdropping on the first two messages exchanged in the protocol and forwarding them to T. This is impossible in the original GJM protocol.

Our analysis leads us to conclude that the choice between the two contract signing protocols must depend on the environment in which the protocol is to be deployed. If there exists an absolutely trusted third party which is connected by private, secure communication channels to all potential contract signers, then the original GJM protocol will guarantee fairness and abuse-freeness (of course, there may exist cryptographic and other attacks that lie beyond the scope of the finite-state Murφ model, and would not have been uncovered by our analysis).

If there is a possibility that the trusted third party may be corrupt or negligent, or that the channel between one of the participants and the third party could be monitored by the intruder, then the original GJM protocol may be subject to an attack that will lead to the loss of abuse-freeness. The absence of TTP accountability in this case presents a potentially serious problem. Our Murφ analysis did not discover any attacks on the ASW protocol that compromise TTP accountability, so the ASW protocol may be a better fit. Again, it is important to keep in mind that Murφ is only capable of finding a limited class of attacks.

Repairs suggested in section 5.6.6 appear to restore the GJM protocol's guarantee of accountability. Therefore, the repaired GJM protocol may be deployed in any environment where private contract signatures are available.

5.8 Conclusions

This study shows how a finite-state analysis tool can be used to study contract signing protocols and discover potential attacks and weaknesses. Our main results are the discovery of a weakness in the Asokan-Shoup-Waidner protocol and an error in the Garay-Jakobsson-MacKenzie protocol. In both cases, a relatively simple change to one or two messages produces a correct contract signing protocol. In addition, our Murφ-based analysis indicates that private channel assumptions can be relaxed.

In order to carry out our automated analysis of abuse-freeness, we needed to augment the system with an outside observer called the Challenger. The role of the Challenger is to nondeterministically challenge one party to demonstrate that this party has control over the outcome of the protocol. This method may be useful for verifying control-related properties of other protocols.

Fair exchange protocols are a new area of application for formal methods, and specification of protocol guarantees in the form suitable for automatic verification is still a challenge, especially in the case of such non-trivial properties as trusted third party accountability and abuse-freeness. We do believe that as online fair exchange and contract signing protocols gain increasing acceptance and a correspondingly high level of assurance is expected from them, formal techniques such as finite-state analysis will prove a useful tool for uncovering interesting insights and non-obvious attacks.

Chapter 6

Conclusions

This thesis demonstrates that finite-state analysis is a useful, practical tool for security protocol designer. After presenting the general methodology for modeling security protocols as finite-state systems and protocol security guarantees as state invariants, we used the general-purpose Murφ verifier to perform analysis of several security protocols, ranging from SSL 3.0 - a fairly conventional, albeit complex, secrecy and authentication protocol - to optimistic contract signing protocols that aim to provide subtle guarantees of mutual fairness, accountability, and abuse-freeness.

Murφ is fully automatic and has a procedural-style input language that can be used without formal methods expertise. Eventual termination of the verification procedure is guaranteed, and, if Murφ discovers a violation of a protocol correctness condition, it automatically reconstructs the steps needed to stage a successful attack. Optimizations described in this thesis enable analysis of large real-world protocols and alleviate the state space explosion problem. Thus, Murφ enjoys the advantage of being user-friendly and not as forbidding to the casual protocol analyst as some of the more advanced formal tools.

The main disadvantage of the Murφ approach to protocol analysis is that the failure to find an attack does not constitute a proof of correctness. Since Murφ can only handle finite-state systems, a bound must be imposed on the number of protocol instances, applications of cryptographic functions, etc. Also, capabilities of the adversary are limited to formal message manipulation that treats the underlying cryptographic primitives as perfect "black boxes." Therefore, Murφ should be viewed primarily as a tool for finding a certain class of attacks and not as a verifier that will prove the protocol "correct".

The proof of the pudding is in the eating. A formal analysis tool is only useful insofar as it can discover previously unknown attacks on protocols, or reveal non-trivial protocol properties that have been overlooked by the designers. As the case studies presented in this thesis demonstrate, Murφ succeeded at its task for every protocol to which it was applied.

For the SSL 3.0 handshake protocol, we created a formal model of the protocol from an Internet draft, and performed a "rational reconstruction" to establish the security role played by each message component. This resulted in a better understanding of the protocol. In addition to discovering all known attacks on the previous version of SSL, Murφ also found a version rollback anomaly in the resumption subprotocol that had not been previously published.

For contract signing protocols, we formalized protocol specifications and expressed fairness and accountability properties as state invariants amenable to formal verification. In the case of the Asokan-Shoup-Waidner protocol, Murφ discovered a weakness that enables the intruder to stage a replay attack or produce inconsistent versions of a contract. For the Garay-Jakobsson-MacKenzie protocol, Murφ discovered an attack in which a weak form of cooperation between the trusted third party and a malicious participant leads to the loss of abuse-freeness without third party accountability. None of the attacks had been known prior to our analysis. In both cases, we propose repairs to the protocol, and analyze the repaired version.

To improve the efficiency of state space exploration by Murφ, we invented several state reduction techniques and proved them sound. These techniques as well as the underlying notion of state subsumption are applicable to any model checker of security protocols, and therefore have significance beyond the scope of this work.

Dozens of security protocols are being deployed on communication networks every year. By establishing finite-state analysis as a practical design and verification tool, we hope that this thesis will help improve the quality of protocols released for public use, and make the Internet a safer place.

Appendix A

SSL 2.0

This appendix outlines the SSL 2.0 protocol. In the protocol description below, *SessionId* is a number that identifies a particular session. When the server starts a new session with the client, it assigns it a fresh *SessionId*. When the client wants to resume a previous session, it includes its *SessionId* in the *Hello* message, and the server returns *SessionIdHit* which is the same session number with the "session found" bit set.

A.1 New session

`ClientHello`	$C \to S$	$C,\ Suite_C,\ N_C,$
`ServerHello`	$S \to C$	$Suite_S,\ N_S,\ \mathrm{sign}_{CA}\{S, K_S^+\}$
`ClientMasterKey`	$C \to S$	$\{Secret_C\}_{K_S^+}$
⟨Change to negotiated cipher⟩		
`ClientFinish`	$C \to S$	$\{N_S\}_{\mathrm{Master}(Secret_C)}$
`ServerVerify`	$S \to C$	$\{N_C\}_{\mathrm{Master}(Secret_C)}$
`ServerFinish`	$S \to C$	$\{SessionId\}_{\mathrm{Master}(Secret_C)}$

Figure A.1: SSL 2.0 basic protocol

Figure A.1 shows the basic SSL 2.0 protocol. Notice that this protocol does not protect plaintext transmitted in the *Hello* messages, making the protocol vulnerable to version rollback and cryptographic preferences attacks described in Section 3.3.5 above.

A description of other weaknesses in SSL 2.0 can be found in `SSL-Talk` FAQ [12].

A.2 Resumed session

Figure A.2 shows the SSL 2.0 resumption protocol.

`ClientHello`	$C \to S$	$C,\ Suite_C,\ N_C,\ SessionId$
`ServerHello`	$S \to C$	$N_S,\ SessionIdHit$
⟨Change to negotiated cipher⟩		
`ClientFinish`	$C \to S$	$\{N_S\}_{\text{Master}(Secret_C)}$
`ServerVerify`	$S \to C$	$\{N_C\}_{\text{Master}(Secret_C)}$
`ServerFinish`	$S \to C$	$\{SessionId\}_{\text{Master}(Secret_C)}$

Figure A.2: SSL 2.0 resumption protocol

A.3 Resumed session with client authentication

Figure A.3 shows the SSL 2.0 resumption protocol with authentication. *AuthType* is the means of authentication desired by the server, N'_S is the server's challenge, *Certificate type* is the type of the certificate provided by the client, *Client certificate* is the actual certificate (*e.g.*, a CA-signed certificate $\text{sign}_{CA}\{C, V_C\}$ for the client's verification key), and *Response data* is the data that authenticates the client (*e.g.*, signed challenge $\text{sign}_C\{N'_S\}$).

`ClientHello`	$C \to S$	$C,\ Suite_C,\ N_C,\ SessionId$
`ServerHello`	$S \to C$	$N_S,\ SessionIdHit$
⟨Change to negotiated cipher⟩		
`ClientFinish`	$C \to S$	$\{N_S\}_{\text{Master}(Secret_C)}$
`ServerVerify`	$S \to C$	$\{N_C\}_{\text{Master}(Secret_C)}$
`RequestCertificate`	$S \to C$	$\{AuthType,\ N'_S\}_{\text{Master}(Secret_C)}$
`ClientCertificate`	$C \to S$	$\{Certificate\ type,\ Client\ certificate,$ $Response\ data\}_{\text{Master}(Secret_C)}$
`ServerFinish`	$S \to C$	$\{SessionId\}_{\text{Master}(Secret_C)}$

Figure A.3: SSL 2.0 resumption protocol with authentication

Appendix B

SSL 3.0: master secret computation

The SSL 3.0 master secret is computed using

$$\begin{aligned}
&\mathrm{Master}(N_C, N_S, Secret_C) = \\
&\quad \mathrm{MD5}\left(Secret_C + \mathrm{SHA}\left(\text{'}A\text{'} + K\right)\right) \ + \\
&\quad \mathrm{MD5}\left(Secret_C + \mathrm{SHA}\left(\text{'}BB\text{'} + K\right)\right) \ + \\
&\quad \mathrm{MD5}\left(Secret_C + \mathrm{SHA}\left(\text{'}CCC\text{'} + K\right)\right) \ ,
\end{aligned}$$

where $K = Secret_C + N_C + N_S$.

Bibliography

[1] N. Asokan, V. Shoup, and M. Waidner. Asynchronous protocols for optimistic fair exchange. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 86–99, 1998.

[2] N. Asokan, V. Shoup, and M. Waidner. Fair exchange of digital signatures. Technical Report RZ2973, IBM Research Report. Extended abstract in Eurocrypt '98, 1998.

[3] A. Bahreman and J. D. Tygar. Certified electronic mail. In *Proc. Internet Society Symposium on Network and Distributed Systems Security*, pages 3–19, 1994.

[4] Feng Bao, R. H. Deng, and Wenbo Mao. Efficient and practical fair exchange protocols with off-line TTP. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 77–85, 1998.

[5] M. Ben-Or, O. Goldreich, S. Micali, and R. L. Rivest. A fair protocol for signing contracts. *IEEE Transactions on Information Theory*, 36(1):40–46, 1990.

[6] D. Bolignano. Towards a mechanization of cryptographic protocol verification. In *Proc. 9th International Conference on Computer Aided Verification*, pages 131–142, 1997.

[7] D. Bolignano. Integrating proof-based and model-checking techniques for the formal verification of cryptographic protocols. In *Proc. 10th International Conference on Computer Aided Verification*, pages 77–87, 1998.

[8] E. F. Brickell, D. Chaum, I. B. Damgard, and J. van de Graaf. Gradual and verifiable release of a secret. In *Proc. Advances in Cryptology – Crypto '87*, pages 156–166, 1987.

[9] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. Technical Report 39, DEC Systems Research Center, 1989.

[10] D. Chaum, A. Fiat, and M. Naor. Untraceable electronic cash. In *Proc. Advances in Cryptology – Crypto '88*, pages 319–327, 1988.

[11] E. M. Clarke, S. Jha, and W. Marrero. Using state space exploration and a natural deduction style message derivation engine to verify security protocols. In *Proc. IFIP Working Conference on Programming Concepts and Methods (PROCOMET)*, 1998.

[12] Consensus Development Corporation. Secure Sockets Layer discussion list FAQ, `http://www.consensus.com/security/ssl-talkfaq.html`, September 3, 1997.

[13] B. Cox, J. D. Tygar, and M. Sirbu. NetBill security and transaction protocol. In *Proc. 1st USENIX Workshop on Electronic Commerce*, pages 77–88, 1995.

[14] R. H. Deng, Li Gong, A. A. Lazar, and Weiguo Wang. Practical protocols for certified electronic mail. *J. Network and Systems Management*, 4(3):279–297, 1996.

[15] S. Dietrich. *A Formal Analysis of the Secure Sockets Layer Protocol.* PhD thesis, Adelphi University, 1997.

[16] D. Dill. The Murφ verification system. In *Proc. 8th International Conference on Computer Aided Verification*, pages 390–393, 1996.

[17] D. L. Dill, S. Park, and A. G. Nowatzyk. Formal specification of abstract memory models. In *Symposium on Research on Integrated Systems*, pages 38–52, 1993.

[18] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[19] N. A. Durgin and J. C. Mitchell. Analysis of security protocols. In M. Broy and R. Steinbruggen, editors, *Calculational System Design*, pages 369–395. IOS Press, 1999.

[20] `http://www.ecash.net`.

[21] F. J. Thayer Fabrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Why is a security protocol correct? In *Proc. IEEE Symposium on Research in Security and Privacy*, 1998.

[22] M. Franklin and M. Reiter. Fair exchange with a semi-trusted third party. In *Proc. 4th ACM Conference on Computer and Communications Security*, pages 1–6, 1997.

[23] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL protocol version 3.0. `draft-ietf-tls-ssl-version3-00.txt`, November 18, 1996.

[24] J. A. Garay, M. Jakobsson, and P. MacKenzie. Abuse-free optimistic contract signing. In *Proc. Advances in Cryptology – Crypto '99*, pages 449–466, 1999.

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

[26] N. Heintze, J. D. Tygar, J. M. Wing, and H.-C. Wong. Model checking electronic commerce protocols. In *Proc. 2nd USENIX Workshop on Electronic Commerce*, pages 147–164, 1996.

[27] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[28] C. N. Ip. *State Reduction Methods for Automatic Formal Verification.* PhD thesis, Stanford University, 1996.

[29] C. N. Ip and D. L. Dill. Better verification through symmetry. In *Proc. 11th International Conference on Computer Hardware Description Languages and their Applications*, pages 97–111, 1993.

[30] C. N. Ip and D. L. Dill. State reduction using reversible rules. In *Proc. 33rd Design Automation Conference*, pages 564–567, 1996.

[31] C. N. Ip and D. L. Dill. Verifying systems with replicated components in Murφ. In *Proc. 8th International Conference on Computer Aided Verification*, pages 147–158, 1996.

[32] R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *J. Cryptology*, 7(2):79–130, 1994.

[33] D. Kindred and J. M. Wing. Fast, automatic checking of security protocols. In *Proc. 2nd USENIX Workshop on Electronic Commerce*, pages 147–164, 1996.

[34] J. T. Kohl and B. C. Neuman. The Kerberos network authentication service (version 5). Internet Request For Comments RFC-1510, September 1993.

[35] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. Internet Request For Comments RFC-2104, February 1997.

[36] G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1995.

[37] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using CSP and FDR. In *Proc. 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166. Springer-Verlag, 1996.

[38] G. Lowe. Casper: A compiler for the analysis of security protocols. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 18–30, 1997.

[39] Formal Systems (Europe) Ltd. Failures Divergences Refinement - user manual and tutorial. Version 1.3, 1993.

[40] P. MacKenzie. Email communication, September 23, 1999.

[41] W. Marrero, E. M. Clarke, and S. Jha. Model checking for security protocols. Technical Report CMU-SCS-97-139, Carnegie Mellon University, May 1997.

[42] C. Meadows. Analyzing the Needham-Schroeder public-key protocol: A comparison of two approaches. In *Proc. European Symposium On Research In Computer Security*, pages 365–384. Springer-Verlag, 1996.

[43] C. Meadows. The NRL Protocol Analyzer: An overview. *J. Logic Programming*, 26(2):113–131, 1996.

[44] J. Millen. The Interrogator model. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 251–260, 1995.

[45] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murφ. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 141–151, 1997.

[46] J. C. Mitchell, V. Shmatikov, and U. Stern. Finite-state analysis of SSL 3.0. In *Proc. 7th USENIX Security Symposium*, pages 201–215, 1998.

[47] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–9, 1978.

[48] L. Paulson. The inductive approach to verifying cryptographic protocols. *J. Computer Security*, 6:85–128, 1998.

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

[50] A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *Proc. 8th IEEE Computer Security Foundations Workshop*, pages 98–107, 1995.

[51] S. Schneider. Security properties and CSP. In *Proc. IEEE Symposium on Research in Security and Privacy*, 1996.

[52] V. Shmatikov and J. C. Mitchell. Analysis of a fair exchange protocol. In *Proc. Internet Society Symposium on Network and Distributed Systems Security*, pages 119–128, 2000.

[53] V. Shmatikov and J. C. Mitchell. Analysis of abuse-free contract signing. In *Proc. 4th Annual Conference on Financial Cryptography (to appear)*, 2000.

[54] V. Shmatikov and U. Stern. Efficient finite-state analysis for large security protocols. In *Proc. 11th IEEE Computer Security Foundations Workshop*, pages 106–115, 1998.

[55] Dawn Song. Athena: An automatic checker for security protocol analysis. In *Proc. 12th IEEE Computer Security Foundations Workshop*, 1999.

[56] U. Stern. *Algorithmic Techniques in Verification by Explicit State Enumeration.* PhD thesis, Technical University of Munich, 1997.

[57] U. Stern and D. L. Dill. Automatic verification of the SCI cache coherence protocol. In *Proc. Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 21–34, 1995.

[58] U. Stern and D. L. Dill. A new scheme for memory-efficient probabilistic verification. In *Proc. Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification*, pages 333–348, 1996.

[59] U. Stern and D. L. Dill. Parallelizing the Murφ verifier. In *Proc. 9th International Conference on Computer Aided Verification*, pages 256–267, 1997.

[60] D. Wagner. Email communication, August 23, 1997.

[61] D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. In *Proc. 2nd USENIX Workshop on Electronic Commerce*, 1996. Revised version of November 19, 1996 available from `http://www.cs.berkeley.edu/~daw/ssl3.0.ps`.

[62] L. Yang, D. Gao, J. Mostoufi, R. Joshi, and P. Loewenstein. System design methodology of UltraSPARC™-I. In *Proc. 32nd Design Automation Conference*, pages 7–12, 1995.

[63] J. Zhou and D. Gollmann. A fair non-repudiation protocol. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 55–61, 1996.