Finite-State Analysis of Two Contract Signing Protocols

Vitaly Shmatikov [1,2] John C. Mitchell [2]

Computer Science Department, Stanford University, Stanford, CA 94305, U.S.A.

Abstract

Optimistic contract signing protocols allow two parties to commit to a previously agreed upon contract, relying on a third party to abort or confirm the contract if needed. These protocols are relatively subtle, since there may be interactions between the subprotocols used for normal signing without the third party, aborting the protocol through the third party, or requesting confirmation from the third party. With the help of Murφ, a finite-state verification tool, we analyze two related contract signing protocols: the optimistic contract signing protocol of Asokan, Shoup, and Waidner, and the abuse-free contract signing protocol of Garay, Jakobsson, and MacKenzie. For the first protocol, we discover that a malicious participant can produce inconsistent versions of the contract or mount a replay attack. For the second protocol, we discover that negligence or corruption of the trusted third party may allow abuse or unfairness. In this case, contrary to the intent of the protocol, the cheated party is not able to hold the third party accountable. We present and analyze modifications to the protocols that avoid these problems and discuss the basic challenges involved in formal analysis of fair exchange protocols.

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 its commitment to the contract. While simultaneous

Email addresses: shmat@research.bell-labs.com (Vitaly Shmatikov), mitchell@cs.stanford.edu (John C. Mitchell).

[1] Current address: Bell Labs Research Silicon Valley, Lucent Technologies, 3180 Porter Drive, Palo Alto, CA 94304, U.S.A.

[2] Supported in part by DoD MURI "Semantic Consistency in Information Exchange," ONR Grant N00014-97-1-0505, and NSF CCR-9629754.

commitment to contracts printed on paper can be achieved by sitting at a table and signing identical paper copies, distributed contract signing over a network is inherently asymmetric: someone has to send the first message. In one contemporary style of contract signing protocols, 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 satisfy the definition of *signed contract* established by the protocol. 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 desirable for the trusted third party, since the third party has the ability to resolve or abort a contract.

A more complex correctness condition, introduced in [GJM99], 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 agrees to sell her house to Bob and Bob signs a contract containing a certain price. If Alice holds the contract without signing, she can show the contract to a competing buyer Carol and convince Carol that she needs to pay more than Bob if she wants the house. This is disadvantageous for Bob, since information he provides to Alice as part of signing an agreed contract is used to Alice's advantage. In this scenario, abuse is possible only if Bob's signature on the contract is in a form that Carol can read and understand; this is necessary since Alice must convince Carol that she has the choice to accept Bob's offer or turn Bob down and sign a new contract with Carol. This kind of abuse can be prevented in physical simultaneous transactions, but it is difficult to prevent in distributed protocols since a sequential protocol has the potential to bind one party before the other.

In this paper, we describe the results of automated analysis of two optimistic contract signing protocols. The first protocol was proposed by Asokan, Shoup, and Waidner [ASW98]; 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 the trusted third party. The second protocol was proposed by Garay, Jakobsson, and MacKenzie [GJM99]; 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 a finite-state enumeration tool called Murφ, we verify correctness properties claimed for the protocols and uncover several weaknesses. The process of using Murφ involves formulating the protocol and potential attacks in a simplified programming language and allowing Murφ to exhaustively try all possible attacks on all possible protocol runs (within certain bounds on the number of protocol instances considered in a run). Since the number of states explored is enormous, the only output of Murφ that bears repeating in a technical paper are the specific runs that lead to failure of one or more correctness conditions. For those interested in the form of the input language, the general method for analyzing security protocols, or the methods employed in the implementation of Murφ, additional details may be found in [Dil96,Mur,MMS97]. Since the most complicated issue in carrying out a Murφ analysis of a contract signing protocol are the high-level decisions on how to formalize the protocol, how to account for dishonest participants or an external attacker, and how to formulate the correctness conditions, the paper focuses on these issues and the results of the analysis.

Our finite-state analysis of the ASW protocol shows that a malicious protocol participant is able to obtain a valid contract while the honest participant, even by requesting help from the trusted third party, can only obtain a replacement contract which is inconsistent with the one possessed by the malicious participant. The same protocol 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 number of possible 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. More generally, the power of finite-state analysis lies in the ability to consider all possible interleavings of actions, including possible runs that are counterintuitive or not expected by designers of the protocol. Although we do not wish to fault the authors for an honest mistake, this point is supported by review of the GJM correctness proof. The proof printed in [GJM99] proceeds by considering a number of cases but overlooks the specific sequence of actions uncovered by our exhaustive brute-force analysis. While humans generally prefer to reason using general concepts that seem to cover all of the relevant phenomena, finite-state analysis examines all runs methodically and exhaustively, often uncovering problems that could otherwise go undetected.

For both protocols, we suggest simple changes 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 [Mac99]. The repaired protocols appear to be correct, at least within the accuracy of our model; 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. For example an *abort* message from the trusted third party does not mean that no participant will receive a contract. Abort simply means that the third party will not subsequently confirm the contract. This is inherent in optimistic two-party protocols: after the protocol has finished without involving the third party, one of the parties can ask the third party to abort the protocol and the third party, having received no previous messages, is engineered to oblige. 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.

Formal methods have been used to analyze the security properties of key exchange and authentication protocols [KMM94,Ros95,Mea96b,Bol97,Pau98]. In particular, finite-state analysis has been successfully applied to protocols such as Needham-Schroeder [Low96,Mea96a,MCJ97], Kerberos [MMS97], SSL [MSS98], and others. However, less attention has been paid to other kinds of protocols, such as fair exchange. In [HTWW96], Heintze et al. used the FDR model checker to verify NetBill [CTS95] and a digital cash protocol inspired by Digicash [Cha85,CFN88]. The correctness conditions they establish are different in character from the ones we consider here. Schneider [Sch98] analyzed the CSP [Ros97,Sch96] model of the Zhou-Gollmann non-repudiation protocol [ZG96], using process failures to express fairness as a liveness property. In our approach, we model fairness by a set of state invariants as described in section 5.1, and verify a wider set of protocol properties.

The remainder of this paper is structured as follows: section 2 provides background on fair exchange protocols and the formal tool we used for our analysis, section 3 describes the ASW protocol, and section 4 specifies the correctness conditions for optimistic contract signing protocols. Section 5 deals with the issues involved in formal modeling of fair exchange. Results of the analysis and suggested repairs to the ASW protocol are presented in section 6. Section 7 describes the GJM protocol, which is then analyzed in section 8. Brief concluding remarks appear in section 9.

Preliminary accounts of the protocol analyses presented in this paper were previously published in conference proceedings [SM00a,SM00b].

2 Background

2.1 Overview of Murφ

Murφ [Dil96] is a finite-state machine verification tool. Originally developed for hardware verification, Murφ has been successfully used for analyzing security protocols [MMS97,MSS98,SS98]. The Murφ input language is a simple high-level language for describing nondeterministic finite-state machines. The input model consists of the description of variables that define the state of the system and a set of guarded rules that represent actions. While there is no explicit notion of process, a process can be implicitly modeled by a set of related rules. Communication between processes is modeled by shared variables. The Murφ system automatically checks, by explicit state enumeration, if every reachable state of the model satisfies a given set of invariants.

To analyze a security protocol in Murφ, it is necessary to combine the finite-state model of the protocol expressed in the Murφ language with the *intruder model*, specify the start state of the protocol, and formally state protocol invariants as boolean conditions that must be true in every state reachable from the start state. The intruder model typically consists of a set of variables that contain the intruder's knowledge and a set of actions that the intruder may take. We use a very simple, mechanical intruder model. The intruder is assumed to have full control over the public network and allowed to take the following actions: (1) overhear every message, decrypt encrypted messages if it has the key, store parts of message in its internal database, (2) intercept messages and remove them from the network, (3) generate messages using any combination of its initial knowledge, parts of overheard messages, known keys, and other data available to it. If at any moment there are several possible actions that the intruder can take, one is chosen nondeterministically. The Murφ system will analyze all states that are reachable via any interleaving of enabled actions.

Limitations of this approach include the fact that our intruder model has no notion of partial information or probability. It cannot perform cryptanalysis or statistical tests of the network traffic, and it follows the "black box" cryptography model: an encrypted message can be read only if the decrypting key is known, otherwise its contents are assumed to be invisible to the intruder (who is still capable of storing the message and replaying it later in a different context).

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. If Murφ fails to find an invariant violation, this is not a proof that the protocol is correct. Since Murφ can only consider finite models, the number of protocol instances under analysis must be bounded. Therefore, if an attack on the protocol requires a certain number of protocol instances and the size of the analyzed model is less than the threshold, the attack will not be discovered. Also, certain kinds of attacks, in particular attacks involving cryptanalysis, are beyond the scope of the model. In the rest of the paper, we refer to this intruder model as the Dolev-Yao intruder, following [DY83]. Some general discussion and complexity results regarding the model can be found in [CDL+99,DM99].

A new-generation Murφ, currently under development at Stanford, uses the predicate abstraction method to model check infinite state spaces. It was used by Satyaki Das to analyze multiple instances of the GJM protocol. This is discussed in section 8.4 below.

2.2 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 [ASW98]. There exists a large body of literature on fair exchange protocols. Applications include online payment systems, in which a payment is exchanged for an item of value, *e.g.* [CTS95], contract signing, in which parties exchange commitments to a contractual text, such as [BOGMR90,ASW98,GJM99], certified electronic mail, for example [BT94,ZG96,DGLW96], and other purposes. There are several varieties of fair exchange protocols.

Gradual exchange protocols [BOGMR90,Dam95] work by having the parties release their secrets in small installments, thus ensuring that at any given moment no party has a significant advantage. 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*, for example [CTS95,ZG96,DGLW96]. 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 [FR97]. 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 [ASW98,BDM98,GJM99]. 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 [GJM99] 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.

3 Asokan-Shoup-Waidner Protocol

In this section, we describe the optimistic contract signing protocol by Asokan, Shoup, and Waidner [ASW98] (the ASW protocol). We start by giving a high-level description of the objectives of the protocol and the assumptions under which it operates, and then explain the protocol steps in detail. The notation has been changed from the original paper to facilitate explanation.

It is worth noting that, intuitively, 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. This is not the case 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.

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.

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 to 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 [ASW98] 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. 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 6 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.

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 [ASW98] provides two forms of contract:

$$\{me_1, N_O, me_2, N_R\} \qquad \text{(standard contract)}$$

$$\text{S-Sig}_T\{me_1, me_2\} \qquad \text{(replacement contract)}$$

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:

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

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:

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

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 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*. We assume that *aborted* is some predefined bit string.

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

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

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.

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

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. Note, however, that the specification does not indicate how two contracts that have the same form but contain contradictory information (*e.g.*, different nonces) should be treated.

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 4.

4 Correctness conditions

In this section, we specify the correctness conditions that must be satisfied by an optimistic contract signing protocol. To simplify the presentation, we combine the guarantees for ASW [ASW98] and GJM [GJM99] protocols into one set of properties. Even though the two papers employ slightly different terminology, the fairness properties they consider are essentially identical, with the exception of abuse-freeness. Both protocols claim that the properties will be satisfied assuming that the communication channels are resilient, *i.e.*, messages can be arbitrarily delayed and/or scheduled, but will be eventually delivered to their intended recipients.

Fairness. Fairness for protocol participants involves several conditions:

- It is impossible for a corrupt participant to obtain a valid contract without allowing the remaining participant to also obtain a valid contract. Another way of stating this [ASW98] is that when the protocol has completed, either both O and R have valid contracts, or neither one does.
- Once an honest participant obtains a cancelation 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.

Trusted 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. It is not specified precisely what can serve as a proof of misbehavior, but typically such proof consists of two contradictory messages signed by T [ASW97], *e.g.*, an abort token and a replacement contract.

Abuse-freeness. The ASW protocol [ASW98] is *not* designed to guarantee this property, but we mention it here for the sake of uniformity. A protocol is abuse-free if 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 [GJM99] is to introduce the notion of abuse-freeness to electronic contract signing.

There are other properties supported by the protocols, such as timeliness and non-repudiability, but we did not model them formally and omit them for the purposes of this analysis.

5 Formal modeling

Section 2.1 described our general approach to protocol analysis, which involves modeling protocol participants and the adversary as non-deterministic finite-state machines. This allows violations of correctness conditions to be detected by fully automated, exhaustive search of the resulting state space. In this section, we focus on the modeling issues arising in the application of finite-state analysis to fair exchange protocols.

Fair exchange protocols present several challenges. To begin with, fairness invariants may be difficult to express precisely, given only an informal protocol specification. Second, 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 network-based intruder. Third, 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 proven to an outside arbiter. These guarantees are difficult to understand and formalize for automated verification. There are also challenges associated with abuse-freeness.

5.1 Modeling fairness

Murφ, the finite-state verification tool we used to perform the analyses described in this paper, can only search for violations of state invariants. In general, liveness properties such as fairness cannot be expressed in the Murφ language. This leads some researchers to prefer more expressive tools. For instance, in [Sch98] fairness properties of the Zhou-Gollmann non-repudiation protocol [ZG96] are expressed as predicates on process *failures*. However, the fairness conditions defined in section 4 can be expressed as safety properties.

To formulate fairness as a safety property, we only verify fairness in the states in which an honest participant cannot perform any action according to the protocol specification. We will call such states *participant-terminal*. Here is a (slightly simplified) sample of Murφ code illustrating how fairness for the originator O in the ASW protocol (see section 3.3) is formally expressed as a Murφ state invariant:

```
--   If R has O's nonce or a valid replacement contract,
--   then O must have R's nonce or a valid replacement contract
rule "O has been cheated"
      -- Fairness is not guaranteed if T is corrupt
      !badT[O.trustedParty]) &
      -- O finished protocol
      O.state = M_DONE &
      (
      -- R has O's nonce
      R.otherNonce = O.ownNonce |
      -- ... or a valid replacement contract
      R.replacement.me1_commit.n1 = O.ownNonce
      ) &
      (
```

```
      -- O has the wrong nonce
      O.otherNonce  = O.otherCommit.n1 &
      O.otherNonce != R.ownNonce) |
      -- O does not have a replacement contract
      isundefined(O.replacement)) |
      -- O's replacement contract is wrong
      O.replacement.me2_commit.n1 != R.ownNonce))
      )
    ==>
    begin
      error "R has O's nonce or a valid replacement contract,
             but O doesn't have R's"
    end;
  end;
end;
```

The rule signals an error if R possesses O's valid contract, but O does not have R's contract. Note that the rule is conditional on O's being in state `M_DONE`, *i.e.*, it only fires if O has completed its execution sequence in the protocol.

5.2 Soundness

Monotonicity. The soundness of our formulation of fairness, explained in section 5.1 above, follows from the fact that fairness for these protocols is *monotonic*. In other words, if a run of a protocol ceases to be fair at some point, then it remains unfair for the remainder of the run. More specifically, if the protocol reaches a state S_E in which one party has a contract, and the other has no means of obtaining one, then this condition will hold true in *all* subsequent states no matter what actions the cheated party undertakes. If this were not true and there existed a sequence of actions that brings the protocol to a state in which the cheated party obtains a contract, then fairness would *not* have been violated in state S_E, and we arrive at a contradiction.

Therefore, we need not be concerned with discovering fairness violations in the intermediate states where they must be expressed as liveness properties (*e.g.*, "O has no means of obtaining a valid contract", or "no contract will be available to O in the future"). Every such violation is preserved in all subsequent states, some of which are participant-terminal. Therefore, we can simply let the protocol run its course, and search for fairness violations only in the participant-terminal states, where they can be expressed as safety properties (*e.g.*, "R has a contract, but O doesn't").

To complete the argument that our formulation is correct, we need to show that every protocol instance will reach a participant-terminal state even if fairness has been violated in an intermediate state. The argument relies on the following three properties:

Finiteness. All execution sequences of the protocol are finite.

Progress of honest participants. For each honest protocol participant P and each state S_i, we can define $S_i|P$ to be the partial state consisting only of P's private state variables. $S_i|P$ can be thought of as P's view of what is happening in the protocol. Obviously, it is possible that $S_i|P = S_j|P$ for some i and j, *i.e.*, different states may project to the same partial state as far as P is concerned.

For each honest participant P, the protocol specification provides a complete ordering of $S_i|P$. Each time P sends or receives a message, it progresses to the next state, eventually reaching the end of its execution sequence. Once P leaves a state, it cannot reach it again. For example, in the ASW protocol (section 3.3), O starts in the partial state where it is ready to send message me_1, then progresses to the state where it has dispatched me_1 and is waiting for me_2, then to the state where it has received me_2, sent me_3, and is waiting for me_4, and finally reaches the participant-terminal state (represented by `O.state = M_DONE` in the code sample above) where it has received me_4 and considers the protocol completed.

Channel resilience. In the protocols discussed in this paper, all communication channels are assumed to be resilient, *i.e.*, it is guaranteed that every message will be eventually delivered. Therefore, we can assume that every honest participant will progress to the end of its execution sequence even if a successful attack on fairness was staged in an intermediate state. In the example above, this means the protocol will reach a state S^* such that $S^*|O$ is considered by O to be a terminal state, *i.e.*, `O.state = M_DONE`. Channel resilience is a fairly realistic assumption for real-world networks. In fact, if channels are non-resilient, then the intruder can defeat any protocol by simply intercepting messages and preventing them from reaching their intended recipients.

The modeling technique presented in this section extends the applicability of fully automated finite-state analysis to a large class of fairness properties while keeping the formalism conceptually simple and restricted to safety properties. There is an efficiency cost. If we could discover fairness violations immediately in the states where they first occur instead of delaying the discovery until the cheated participant reaches the end of its execution sequence, we would not need to generate all the state subspaces rooted in the violating states. The exact number of states that are generated unnecessarily cannot be estimated in general as it depends on the particular protocol and the nature of the attack that leads to the loss of fairness. Effectively, the tradeoff is between running a simple, robust, well-understood finite-state analysis tool such as Murφ for a bit longer, and modeling the entire protocol in a richer formalism that allows direct expression of liveness properties.

5.3 Modeling trusted third party accountability

Accountability only holds if the trusted third party is guaranteed to send a valid response to all requests. Also, O must be notified whenever 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 of Murφ, it is necessary to determine what it means to be able to prove T's misbehavior. Based on our interpretation of the protocol description in [ASW98], 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 (see section 5.5).
- 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.

These conditions can be modeled by simple state invariants. Therefore, trusted third party accountability can be verified with the help of Murφ.

5.4 Modeling abuse-freeness

Unlike basic fairness, abuse-freeness is *not* a monotonic property, and cannot be easily converted into a safety guarantee. Both fairness and abuse-freeness are assertions about future executions. Fairness violations, however, are *existential* properties of traces and can thus be found by analyzing each trace separately (*e.g.*, "there exists a trace such that O has R's valid contract, but R cannot obtain O's valid contract by following the protocol specification"). Therefore, we can convert fairness conditions into safety properties to be evaluated in terminal states as described in section 5.1, and delay the discovery of violations until the trace has completed.

Violations of abuse-freeness, on the other hand, are *universal* properties of traces ("for all possible traces, one of the parties can a) determine the outcome of the protocol, and b) prove this ability to an outside party"). They only occur in intermediate states, before a particular execution sequence has been chosen by the participants, and cannot be discovered by analyzing the terminal state of an individual trace.

Our partial approach to verifying whether abuse-freeness is violated in the protocol consists of two stages. 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 violation 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 8.3 and 8.4, respectively.

5.5 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. In our formal model, we keep the intruder and the malicious protocol participant separate. This enables us to consider several conceptual levels of corruption. Distinguishing between them is useful for analyzing real-world implications of the protocols.

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*. We do not consider the case when a party divulges its private key to everybody since it can be represented by sending all messages encrypted with that key in plaintext.

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*.

6 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 2.1. The correctness conditions of section 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 we have not verified them formally.

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 5.5), 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 only present analysis of fairness and omit that of trusted third party accountability since we did not discover any interesting behaviors in which accountability is violated.

6.1 Fairness

As a reminder, fairness guarantees that when the protocol has completed, either both protocol participants have valid contracts, or neither one does (see section 4).

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. Murφ did not discover any violations of fairness. 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 delaying me_2.
- Force R (respectively, O) to submit a resolve request to T by delaying 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 fairness as defined in section 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

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

Later ...

$$
\begin{array}{ll}
I \rightarrow R & me_1 = \text{S-Sig}_O\{V_O, V_R, T, \textit{text}, \text{hash}(N_O)\} \\
R \rightarrow O & me_2' = \text{S-Sig}_R\{me_1, \text{hash}(N_R')\} \quad \langle I \text{ intercepts}\rangle \\
I \rightarrow R & me_3 = N_O \\
R \rightarrow O & me_4' = N_R' \quad \langle I \text{ intercepts}\rangle
\end{array}
$$

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 [ASW98] 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 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 repair to the protocol that prevents the attack is described in section 6.2.

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 2.1). 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.

Malicious protocol participant. Finally, we analyzed the protocol under the assumption that one of the participants is malicious. Murφ discovered the following attack, in which a malicious R obtains a contract which is inconsistent with that obtained by O.

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

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.

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 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 [ASW98] 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 6.2 prevents this attack, too.

6.2 *Repairing the protocol*

The ASW protocol can be repaired so as to prevent the attacks described in section 6.1 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.

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

The security of the repaired protocol against replay attacks is strengthened by the fact that cryptographically secure signature schemes are necessarily indeterministic, therefore, even if the same text is signed in two separate instances of the protocol, the resulting messages will be different.

7 Garay-Jakobsson-MacKenzie Protocol

In this section, we describe the abuse-free optimistic contract signing protocol of Garay, Jakobsson, and MacKenzie [GJM99]. 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 3. We start by briefly describing the objectives of the GJM protocol, explain the properties of *private contract signatures* (PCS), an innovation of Garay, Jakobsson, and MacKenzie used to make contract signing abuse-free, and describe the protocol steps in detail.

7.1 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 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.

7.2 Private Contract Signatures

The GJM protocol relies on the cryptographic primitive called *private contract signature* (PCS). We write $\mathrm{PCS}_O(m, R, T)$ 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:

- $\mathrm{PCS}_O(m, R, T)$ can be verified like a conventional signature, *i.e.*, there exists a probabilistic polynomial-time algorithm PCS-Ver such that PCS-Ver(m, O, R, T, s) is *true iff* $s = \mathrm{PCS}_O(m, R, T)$.
- $\mathrm{PCS}_O(m, R, T)$ 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 $s = \mathrm{PCS}_O(m, R, T)$, 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.
- $\mathrm{PCS}_O(m, R, T)$ 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 S-Sig$_O(m)$ and T-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 [GJM99].

7.3 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 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:

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

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:

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

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 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 T-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.

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

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

7.4 Correctness conditions

Correctness conditions for the GJM protocol are effectively the same as those for the ASW protocol, and can be found in section 4 above. In addition to fairness, the GJM protocol is designed to be abuse-free.

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 T-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. 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) can be traced to T and serve as a proof of T's misbehavior.

8 Analysis of the GJM Protocol

We used Murφ to perform finite-state analysis of the GJM protocol. Since abuse-freeness cannot be trivially represented as a state invariant, we employed a partial verification method described in section 5.4. Corrupt protocol participants were modeled as described in section 5.5. In the rest of this section, we analyze protocol correctness conditions, and suggest repairs to the protocol.

8.1 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 the same execution sequences as for the ASW protocol (see section 6.1), none of which is a violation of fairness as defined in section 4.

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.

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:

$$
\begin{array}{lll}
O \to R & me_1 = & \mathrm{PCS}_O(m, R, T) \\
R \to O & me_2 = & \mathrm{PCS}_R(m, O, T) \\
\multicolumn{3}{l}{\quad I \text{ intercepts } me_2\text{, or } O \text{ receives and discards it}} \\
O \to T & ma_1 = & \text{S-Sig}_O(m, O, R, \mathit{abort}) \\
R \to T & mr_1 = & \mathrm{PCS}_O(m, R, T), \text{S-Sig}_R(m) \\
\multicolumn{3}{l}{\quad I \text{ eavesdrops on } mr_1\text{, learns S-Sig}_R(m),} \\
\multicolumn{3}{l}{\quad \text{delays } mr_1 \text{ until } T \text{ receives } ma_1} \\
T \to O & ma_2 = & \text{S-Sig}_T(\text{S-Sig}_O(m, O, R, \mathit{abort})) \\
\multicolumn{3}{l}{\quad I \text{ intercepts } ma_2\text{, or } O \text{ receives and hides it}} \\
T \to R & mr_2 = & \text{S-Sig}_T(\text{S-Sig}_O(m, O, R, \mathit{abort})) \\
I \to O & & \text{S-Sig}_R(m)
\end{array}
$$

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

Recall the second fairness condition from section 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. 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.

Since the fairness conditions from section 4 do not hold, we believe this attack is a bona fide violation of fairness. Even though it can be argued that R implicitly agreed to sign the contract by sending its signature to T in message

mr_1, the fact that it received an abort token in response should guarantee that a contract has not be signed. Note the asymmetry between R and O here — since R may not initiate the *abort* subprotocol itself, the abort token serves as a proof to R that the protocol has been canceled by O, assuming T is honest. We also believe that this attack violates abuse-freeness (see section 8.3 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 [GJM99] 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-$\text{Sig}_R(m)$. 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 [GJM99] does not require that mr_1 be encrypted.

Now consider the case when the $R \to T$ channel is secure, but T is *accidentally corrupt*, and I has passive access to all of its incoming communication (see section 5.5 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. I only needs to make sure that T receives O's abort request ma_1 before it receives R's resolve request mr_1. Scheduling the network to achieve this is not difficult if malicious O sends ma_1 to T *prior* to sending me_1 to R, while honest R receives me_1, generates me_2 and then times out waiting for me_3 before sending mr_1 to T. Having passive access to T's communication with R is sufficient for I to learn S-$\text{Sig}_R(m)$ and divulge it to O. Therefore, the attack succeeds in this case.

8.2 Trusted third party accountability

Suppose that T is accidentally corrupt and I successfully stages the attack described in section 8.1, causing R to lose fairness as a result. Since we are analyzing a TTP-accountable version of the GJM protocol (see section 7.4), we would like to verify whether the trusted third party T can be held accountable, *i.e.*, if R loses fairness because of T's misbehavior, it must be able to prove the misbehavior to an arbiter or verifier in an external dispute.

For the purposes of our formal analysis, we followed the designers of the ASW protocol [ASW98,ASW97] and assumed 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, T-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.

8.3 Abuse-freeness

We claim that the attack described in section 8.1 violates abuse-freeness of the protocol by giving the originator O both the power to determine the outcome of the protocol and the ability to prove this power to an outside arbiter A.

Suppose O uses A as a signature server — instead of signing messages to R with its own private key, it asks A to do the signing with A's private key to which O has no access. This way A can keep track of all signed messages generated by O in the protocol. If key certificates are used, then A can initiate the *exchange* subprotocol with R under its own name, and O, using an intruder eavesdropping on the network and/or an accidentally corrupt T, will demonstrate that it can drive the protocol to the desired conclusion. In either case, A is able to maintain the list of all signatures generated by the originator of the exchange.

O obtains both $\text{S-Sig}_R(m)$ and $\text{S-Sig}_T(\text{S-Sig}_O(m, O, R, \mathit{abort}))$ by proceeding as in section 8.1. At this point, O is free to decide whether to enforce the contract using the former, or consider it aborted using the latter. Thus it has the power to determine the outcome of the protocol. It can prove this power by presenting both messages to A. Note that in this case an abort token *will* convince A that O has the power to abort the protocol since A keeps track of all signatures generated by O, and knows that at no point in the protocol did O generate its signature on the contract, $\text{S-Sig}_O(m)$. This is different from the case, considered in section 3.3, where O initiates the *abort* subprotocol with T after releasing its signature, and, therefore, O's possession of the abort token cannot serve as a proof that O is capable of aborting the protocol.

The argument about TTP accountability given in section 8.2 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.

8.4 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 8.1 to their attention [Mac99].

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

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

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 5.4). However, the only information in R's possession at this point is $\text{PCS}_O(m, R, T)$, and R cannot use it to prove anything to an outside arbiter due to the designated verifier property of PCS (see section 7.2). 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.

9 Conclusions

This study shows how a finite-state analysis tool can be used to study contract signing protocols and discover potential attacks and weaknesses. The main insights into the protocols are a weakness in the Asokan-Shoup-Waidner protocol and an error in the Garay-Jakobsson-MacKenzie protocol. In both cases, we give simple changes to one or two messages that eliminate these problems. Whatever usefulness the protocols might have in their original form, we believe that these small changes are clear improvements in the protocol that will be useful to anyone who wishes to use these or related protocols in practice. In addition, our Murφ-based analysis clarifies the private-channel assumptions needed to guarantee protocol correctness.

Several modeling challenges arose in this study, since the properties involved in contract signing protocols are different and more subtle or more complex than the correctness conditions for common secrecy and authentication protocols studied extensively in the literature. The correctness properties for these protocols are fairness, third-party accountability, and abuse-freeness. While fairness and accountability can be determined from the traces of terminating protocol runs, abuse-freeness is not a trace-based property. The reason is that abuse relies on intermediate states where one party has the power to unilaterally determine whether or not a contract is produced. This is precisely the kind of property that has motivated researchers in concurrency theory to develop partial order models of concurrent systems [Hoa85,Mil95]. Since Murφ is a trace-based tool, we can only formalize and check an approximation to abuse-freeness. More specifically, we conjectured the states of protocol execution that allow one party power over the other and then verified our conjecture by Murφ analysis of a modified protocol environment. In this modified environment, an outside observer called the Challenger nondeterministically challenges one party to demonstrate power over the outcome of the protocol. If that party succeeds in meeting every challenge, then that party must have the power to determine the outcome. Moreover, verification that every challenge is met is a trace-based property. We believe that this method for verifying non-trace-based properties using trace-based tools 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 automated 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 useful for uncovering interesting insights and non-obvious attacks.

References

[ASW97] N. Asokan, V. Shoup, and M. Waidner. Asynchronous protocols for optimistic fair exchange. Technical Report RZ2976, IBM Research Report, November 1997.

[ASW98] 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.

[BDM98] 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.

[BOGMR90] 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.

[Bol97] D. Bolignano. Towards a mechanization of cryptographic protocol verification. In *Proc. 9th International Conference on Computer Aided Verification*, pages 131–142, 1997.

[BT94] A. Bahreman and J. D. Tygar. Certified electronic mail. In *Proc. Internet Society Symposium on Network and Distributed Systems Security*, pages 3–19, 1994.

[CDL+99] I. Cervesato, N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *Proc. 12th IEEE Computer Security Foundations Workshop*, pages 55–69, 1999.

[CFN88] D. Chaum, A. Fiat, and M. Naor. Untraceable electronic cash. In *Proc. Advances in Cryptology – Crypto '88*, pages 319–327, 1988.

[Cha85] D. Chaum. Security without identification: Transaction systems to make big brother obsolete. *Communications of the ACM*, 28(10):1030–1044, 1985.

[CTS95] 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.

[Dam95] I. B. Damgard. Practical and provably secure release of a secret and exchange of signatures. *J. Cryptology*, 8(4):201–222, 1995.

[DGLW96] 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.

[Dil96] D. Dill. The Murφ verification system. In *Proc. 8th International Conference on Computer Aided Verification*, pages 390–393, 1996.

[DM99] 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.

[DY83] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[FR97] 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.

[GJM99] J. A. Garay, M. Jakobsson, and P. MacKenzie. Abuse-free optimistic contract signing. In *Proc. Advances in Cryptology – Crypto '99*, pages 449–466, 1999.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[HTWW96] N. Heintze, J. D. Tygar, J. M. Wing, and H.-C. Wong. Model checking electronic commerce protocols. In *Proc. USENIX 1996 Workshop on Electronic Commerce*, pages 147–164, 1996.

[KMM94] R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *J. Cryptology*, 7(2):79–130, 1994.

[Low96] 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, 1996.

[Mac99] P. MacKenzie. Email communication, September 23, 1999.

[MCJ97] W. Marrero, E. M. Clarke, and S. Jha. Model checking for security protocols. Technical Report CMU-SCS-97-139, Carnegie Mellon University, May 1997.

[Mea96a] 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, 1996.

[Mea96b] C. Meadows. The NRL Protocol Analyzer: An overview. *J. Logic Programming*, 26(2):113–131, 1996.

[Mil95] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1995.

[MMS97] 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.

[MSS98] 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.

[Mur] `http://verify.stanford.edu/dill/murphi.html`.

[Pau98] L. Paulson. The inductive approach to verifying cryptographic protocols. *J. Computer Security*, 6:85–128, 1998.

[Ros95] 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.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1997.

[Sch96] S. Schneider. Security properties and CSP. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 174–187, 1996.

[Sch98] S. Schneider. Formal analysis of a non-repudiation protocol. In *Proc. 11th IEEE Computer Security Foundations Workshop*, pages 64–55, 1998.

[SM00a] 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.

[SM00b] V. Shmatikov and J. C. Mitchell. Analysis of abuse-free contract signing. In *Proc. 4th Annual Conference on Financial Cryptography*, 2000.

[SS98] 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.

[ZG96] J. Zhou and D. Gollmann. A fair non-repudiation protocol. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 55–61, 1996.