BRUTUS: A Model Checker for Security Protocols

Wilfredo Marrero

December 2001

CMU-CS-01-170

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Edmund M. Clarke, Chair
Catherine Meadows, Naval Research Labs
Doug Tygar
Jeannette Wing

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

This research was sponsored by the Maryland Procurement Office (MPO) under Agreement No. MDA90499C5020, the National Science Foundation (NSF) under Grant Nos. CCR-9803774 and CCR-9217549, and by SRC. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the MPO, NSF, SRC, the U.S. government or any other entity.

Keywords: formal verification, authentication, electronic commerce, model checking, partial order

Abstract

As more resources are added to computer networks, and as more vendors look to the world wide web as a viable marketplace, the importance of being able to restrict access and to ensure some kind of acceptable behavior, even in the presence of malicious adversaries, becomes paramount. Many researchers have proposed the use of security protocols to provide these security guarantees. In this thesis, I describe a method of verifying these protocols using a special purpose *model checker*, BRUTUS, which performs an exhaustive state space search of a protocol model. This tool also includes a natural deduction style derivation engine which models the capabilities of an adversary trying to attack the protocol. Since the models are necessarily abstractions, one cannot *prove* a protocol correct. However, the tool is extremely useful as a debugger. I have used this tool to analyze fifteen different security protocols, and have found the previously reported attacks for them.

The common limitation for model checking is the *state explosion problem*. This is particularly true of models in BRUTUS because they are composed of multiple components that are executing concurrently. The traces of the system are defined by an interleaved execution. For this reason, I implemented two well known state reduction techniques in BRUTUS. The first technique exploits the *symmetry* due to replicated components. The second reduction technique is called the *partial order reduction*. This technique exploits the fact that the relative order of certain pairs of actions is immaterial to the overall correctness of the model. This means that it is not always necessary to explore all possible interleavings of actions when performing the analysis. It is also of interest that in the case of security protocol verification, the partial order reduction technique can be generalized so that an even greater reduction is achieved. This thesis describes how these reductions are implemented in BRUTUS, how they improve the efficiency of the model checker, and how they apply to model checking of security protocols in general.

Acknowledgments

I would like to thank my thesis advisor, Ed Clarke, for his continual support and direction. Ed was a constant source of new ideas. It was Ed who originally suggested that I investigate the possibility of applying model checking to security. With his direction, I pushed my original rough ideas in this area into the work presented here. His insistence on a more general logic than what I presented at my thesis proposal has resulted in much better model checker.

In addition, I would like to thank two people with whom I consistently discussed the ideas of this thesis. Somesh Jha showed a great deal of interest in my work from the outset. We have co-authored a number of papers in this area and Somesh always seemed to be able to grasp the essence of an idea even when I couldn't communicate it very well. Carsten Schürmann also deserves a great deal of thanks. Although not a model checking expert, Carsten seemed to be able to grasp the higher level ideas and to suggest solutions to difficulties at a higher, more general level. Carsten's expertise in ML also served me very well when it came time to implement Brutus.

I would also like to thank the other members of my thesis committee. Catherine Meadows, Doug Tygar, and Jeannette Wing were very patient and very giving of their time. Their feedback on the original thesis draft was invaluable and has significantly increased the quality of this work.

There are also a number of people who helped out with this thesis through discussions and through reading previous versions of this work. I would like to thank Sergey Berezin and Marius Minea as well as Patricia, Karl, and Kathryn Schimmel.

Finally, I would like to thank all those "behind the scenes" people without whose love, support, and encouragement this would not have been possible. I would like to thank my parents, Wilfredo and Maria as well as my brother Noel. I also had a number of adoptive families in Pittsburgh: the Schimmels, the Deelys, and the Oratorians. They kept me sane and never let me forget the big picture. And finally, I would like to thank my wife, Anne. I know it was not easy when it seemed like my thesis took over our lives, but you were always there for me. Thank you so much.

Contents

Chapter 1

Introduction

Initially, security for computers was provided by their physical isolation. Unauthorized access to these machines was prevented by restricting physical access. The importance of sharing computing resources led to systems where users had to authenticate themselves, usually by providing a name/password pair. This was sufficient if the user needed to be physically at the console or was connected to the machine across a secure link. However, the efficiency to be gained by sharing data and computing resources has led to computer networks, in which the communication channels cannot always be trusted. In this case, authentication information such as the name/password pairs could be intercepted and even replayed to gain unauthorized access. When such networks were local to a certain user community and isolated from the rest of the world, many were willing to take this risk and to place their trust in the community. However, in order to be able to share information with those outside the community, this isolation would have to be removed. The benefits to be had by such sharing have been enormous, and the gains are demonstrated by the growth of such entities as the Internet and the world wide web. Now, very few, if any, guarantees can be made about the communication links.

Numerous protocols have been proposed that claim to solve many of the security issues by taking advantage of cryptography. The correctness of these protocols is paramount, especially when we consider the size of the networks involved and the desire of users to place confidential information and to allow for monetary transactions to take place across these networks. However, errors in these protocols can be extremely subtle and hard to find. In this thesis, I describe a new model checking tool, BRUTUS, that

I have successfully applied to analyze and verify these kinds of protocols.

1.1 Formal Methods

Engineering and technology have led us to the building of extremely complex systems. In many areas, these systems have grown so complex that it is extremely difficult, if not impossible, to guarantee that even the design of the system is correct. One might argue that nowhere is this more true than in the area of computer science. For this reason, numerous researchers have turned to mathematical models, abstractions, and formal reasoning to try to get a handle on how these systems behave and to verify whether or not they are correct. These methods have the added advantage that one can often automate the tedious task of analyzing and verifying the model.

Formal methods research seems to fall into two distinct camps. The first is *theorem proving*. In this approach, a set of axioms describes the system being analyzed. The desired properties of the system are specified as a set of theorems that need to be proven. One then uses the axioms describing the system and the inference rules of the logic to try to prove the desired theorems. The goal is to automate the actual theorem proving process as much as possible. The automatic generation of axioms describing the system and theorems stating the requirements is desirable as well, but hard to do. The logic and inference system used can be as general as first order logic, or it can be tailored to the particular problem domain as in the case of the BAN Logic [7]. Of the many tools for automated deduction, PVS [27, 59] and Isabelle [63, 64] stand out because of their generality and widespread use. They have also been used in the area of security protocol verification.

The second formal approach is called *model checking*. As opposed to theorem proving, where the system is described by a set of axioms, in model checking, an explicit model of the system is given. In the world of first order logic, this would be equivalent to explicitly providing the interpretation of all the constants, functions, and relations. The desired properties are again specified as logic formulas, but instead of proving theorems, we need to verify that the given model satisfies the formulas.

Because the user needs to model the system instead of providing axioms about the system, a model of computation is required. Kripke structures (finite state transition systems where the states are labeled with atomic propositions) are a popular model to use since they provide a general way of describing the operational behavior of a system. The actual behavior of the system is given by all the possible executions or *traces* of the model. Since the semantics of the model is given in terms of its traces, the specification language or logic needs to be able to refer to events that occur in a trace. It is not surprising, then, that specifications are typically given either as another more abstract transition system, or in terms of a temporal logic formula that is satisfied exactly by the set of correct traces. For a good introduction to temporal logic, see [20].

Like theorem proving, the goal is to automate this verification process as much as possible. This is somewhat easier for model checking because the verification can be reduced to state space exploration which is highly automatable. One of the first efficient temporal logic model checking procedures can be found in [8]. Much of the time since then has been spent combating what is known as the *state space explosion problem*. A simplified description of this problem is the statement that the size of the model of a concurrent system grows exponentially with respect to the number of components in the model. In other words, if it takes k states to model a system with a single component, it takes on the order of k^n states to model a system with n components. One of the biggest advances in combating state explosion is the use of a symbolic representation of the transition system and of the states explored [42]. Unfortunately, this advance has not proved very helpful in the area of security protocol verification. However, two other techniques, symmetry reduction and partial order reduction, have proven useful and are described in this thesis. For a good reference for model checking, including state reduction techniques, see [10].

Because security protocols can be thought of as concurrently executing finite state systems, they are amenable to these kinds of formal methods. In fact, both theorem proving and model checking approaches have been used to analyze security protocols. For this reason, it is beneficial to consider the merits of both. Because model checkers work with a concrete and finite model, they are really more useful in providing negative information. One of the biggest advantages of using a model checker is its counter-example generation capability. This counter-example is invaluable when it comes time to debug the design. One of the biggest disadvantages of model checking is that exhaustive state space exploration requires the models to be finite. Although abstraction techniques exist that can be used to map many infinite models to finite models, in practice, this is still a limitation. In contrast, theorem provers provide positive information and can handle infinite systems quite easily. When a proof is successful, one knows that *any* system satisfying the axioms (including an infinite system) also satisfies the specification. The drawback is that the procedure may not terminate and often there is little feedback when a proof fails.

This thesis deals with BRUTUS, a model checker for security protocols and, as such, does not really discuss theorem proving methods in any detail. The emphasis will be on how BRUTUS works and on how to use BRUTUS. When appropriate, BRUTUS is compared to other verification approaches.

1.2 BRUTUS

Since BRUTUS is a model checker for security protocols, the natural question is, "Why another model checker?" To better answer this question, it is necessary to know a little of the history of how state space exploration techniques have been applied to the problem of security protocol verification. There seem to be two approaches: new special purpose tools and pre-existing general purpose model checkers.

Initial state exploration tools such as the Interrogator [51] and the NRL Protocol Analyzer [46] are special purpose tools that perform what could be considered a backward state exploration. In other words, one gives an insecure state and the tool searches backward trying to find a state that leads to the insecure state and that also satisfies the initial configuration. However, these tools deal with models that are not necessarily finite and so they have some of the properties of theorem provers. Namely, they can require user interaction during the verification process, and they may not terminate.

Later model checking efforts seem to rely on existing general purpose tools. Two well known tools used for this purpose are Murϕ [56] and FDR [37]. Although these tools are meant to be general purpose, they are somewhat weak when it comes to modeling an adversary that is trying to subvert the protocol. The model needs to somehow keep track of what messages are known and not known to the adversary. In the straightforward Kripke structure model, each such message would correspond to an atomic proposition that would be true if and only if the adversary knows that message. Since the number of messages is infinite, the models could themselves be potentially infinite. The only alternative is to arbitrarily restrict the model to keep track of only a finite subset of all possible messages. The question then becomes which messages to include and which to exclude from the model. In essence, the burden is on the user to come up with a finite state machine description of the adversary. This process is all the more complicated if one wants to consider possible interactions between *different* protocols since it then becomes even less clear what messages might be important.

The BRUTUS model checker is an attempt to separate out the adversary from the model and to encode it as a set of rewrite rules that can be applied to messages overheard during the execution of the protocol (or possibly during the execution of a different protocol). While one still must restrict how many of these rewrite rules are applied, one need not specify ahead of time which messages the model needs to keep track of. In essence then, BRUTUS has two orthogonal components. One component is a state exploration component that actually performs the search. The other is the message derivation engine that models the adversary's capabilities. As seen in Figure 1.1, these components interact. The set of possible next states is influenced by which messages the adversary can construct and send out. In turn, the set of messages the adversary can generate is influenced by what messages other agents have sent during the execution of the protocol.

Since I want to analyze a variety of protocols, I developed a specification logic which is powerful enough to describe a variety of security properties. In this logic one can specify not only what the adversary should not know, but also what other agents must or should not know. This allows one to specify non-repudiation as well as secrecy. The past time operator allows the specification of electronic commerce properties like guaranteed payment for goods and absence of unauthorized payments. Even a weak form of anonymity can be specified using this logic.

Finally, I wanted to see how far state space reduction techniques could be pushed when model checking security protocols. While some general purpose model checkers already support reduction techniques (Murϕ, for instance, exploits symmetry), to the best of my knowledge no one has investigated how far these techniques could be pushed when dealing specifically with security protocols. Indeed, I have identified and exploited some

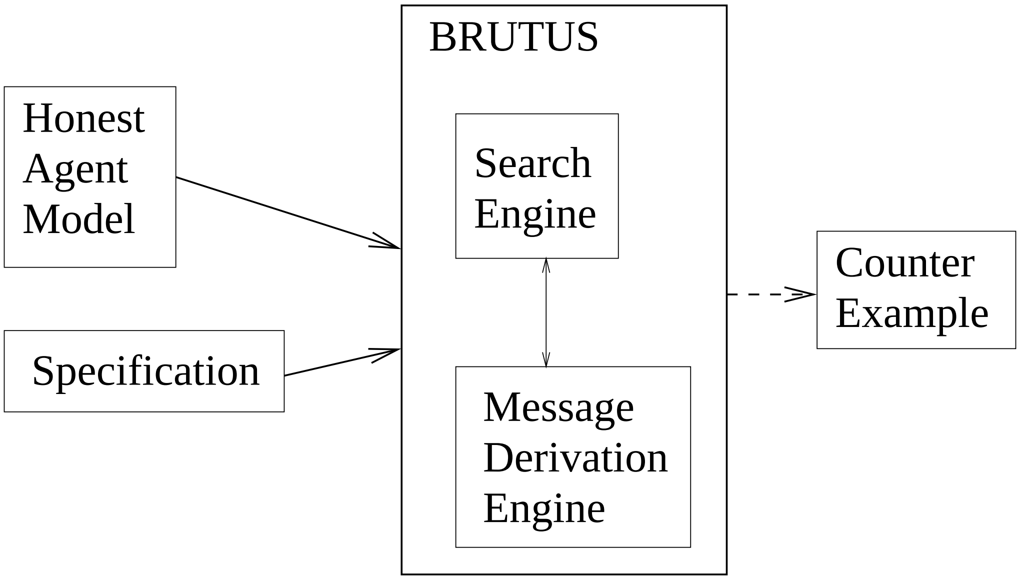

Figure 1.1: Brutus model checker

symmetries that seem to be inherent to and specific to the domain of security protocol verification. I have also generalized the idea of partial order reductions so that traces no longer need to be equivalent in order to perform the reduction. I show how the correctness of traces that are "less likely to be true" guarantees the correctness of other traces that are "more likely to be true." Since this relation between traces is coarser than the one typically used in partial order reduction, more traces are related which translates to fewer traces being explored.

1.3 Thesis Contributions

This research centered around the design, implementation, and use of a security protocol verifier. The overall goal was to create a practical and theoretically sound model checker. Along the way, I looked for ways to make the system more general and for ways to make the system more efficient. I also looked to prove rigorously the correctness of these ideas, and when possible, to generalize the theoretical results. This work resulted in a number of contributions, including the following:

- *A prototype model checker for security protocols* – Many authentication, key exchange, and electronic commerce protocols can be modeled and analyzed.

- *A temporal logic of knowledge* – Brutus uses a temporal logic which can refer to a principal's knowledge, a principal's variables, and a principal's actions. It is powerful enough to express many security properties including authentication, secrecy, privacy, non-repudiation, and authorization.

- *A suite of analyzed protocols* – Brutus was used to model and analyze over fifteen different protocols, three of which were electronic commerce protocols. All previously reported flaws were found.

- *A normalization theorem for the derivations of messages* – I prove that in the adversary model used by most research in this area, all messages that can be generated by the adversary can be generated by first applying only reduction or shrinking rules and then applying only introduction or expanding rules. This proof leads naturally to a very efficient algorithm for determining if a particular message can be generated by the adversary. However, this result depends on a restriction on encryption to use only atomic keys, which is consistent with the the restriction so commonly assumed in the literature.

- *The identification of common symmetries* – This thesis also identifies a number of symmetries that are common to all security protocol models. Unlike more general theorem provers where these symmetries must be detected, the structure of security protocols ensures that certain symmetries always exist. Brutus exploits these symmetries to reduce the state space to be searched. While this technique is certainly not general enough to detect all possible symmetries, the fact that one need not search for them means that the expensive computation of the orbit relation (the equivalence relation induced by the symmetry) can be avoided.

- *A generalization of partial order reductions* – Until now, partial order reductions have been used to identify and to remove from consideration traces that are equal up to the permutation of independent actions. In order to guarantee correctness, these reductions have insisted that the actions also be invisible (i.e., that they cannot affect the value of the specification formula). This restriction has meant that two traces must agree on the specification before one of them could be ignored. I generalize this technique so that a trace can be ignored if one can guarantee that there is always another trace whose correctness implies the correctness of the disregarded trace. In other words, we can throw out possibly faulty traces as long as we do not discard *all* faulty traces. By weakening the restriction on the traces that can be ignored, I increase the number of traces that are thrown out, thereby increasing the amount of reduction attained. This has lead to a significant reduction in the size of the state space explored in Brutus. It is my hope that this generalization will also prove useful in other verification domains.

Chapter 2

Security

The use of cryptography can be traced very far back in history. According to Kahn [32], the first printed book on cryptology dates from the early sixteenth century [77]. Historically, the majority of cryptographic effort was spent on trying to keep messages secret, whether it was battle information, diplomatic information, or even personal information. With the explosion of the world wide web, suddenly cryptography is being used to provide or facilitate digital signatures, digital cash, electronic commerce, electronic voting, and personal identification as well as secure communication. Since the original intent of cryptography was just to ensure secrecy, often extra machinery (in the form of protocols) is necessary to ensure the correctness of these new applications. In this chapter I will introduce some basic building blocks of security and how they are put together in the form of cryptographic protocols. I also discuss how BRUTUS models these constructs. For a more thorough overview of these security topics, see [50, 71].

2.1 Security Building Blocks

Security protocols are built up from more than just cryptographic algorithms, although certainly these cryptographic algorithms are at the heart of the protocols. Cryptography by itself can really only guarantee secrecy and integrity of messages. Cryptographic protocols require that messages also be fresh and authentic. In what follows, I discuss the basic building blocks used to guarantee these properties and the abstractions that I used

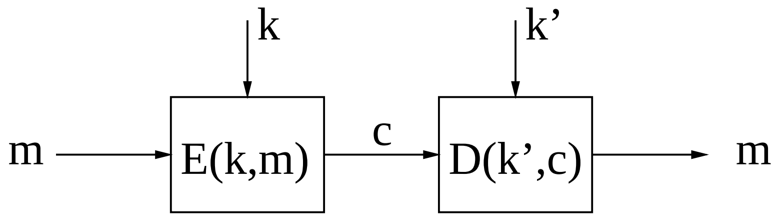

Figure 2.1: Encryption and decryption

when modeling them in BRUTUS.

2.1.1 Ensuring Secrecy and Integrity

In security protocols, secrecy and integrity of messages are provided by encryption. For our purposes, encryption involves just a handful of concepts. First, there is the *plaintext* to be encrypted. The encryption is performed via an encryption algorithm. This algorithm is a function. For every possible plaintext there is one and only one ciphertext. In addition to the plaintext, the encryption function has a second input called the *encryption key*. *Different* keys will yield *different* ciphertexts when applied to the *same* plaintext. The purpose of encryption is to keep the meaning of the plaintext secret while allowing a select group of people (possibly a single person) to recover the original plaintext. This is accomplished via a *decryption* function. The decryption function also requires a key. Notice that since the intent is to keep the plaintext secret, the ciphertext should have the property that it is "difficult" to arrive at the original plaintext (or the decryption key) from just having the ciphertext (and possibly the encryption key). I leave the definition of difficult imprecise although ideally, "difficult" = "impossible." Also note that encryption and decryption are inverse functions; therefore, they each must be one-to-one. If encryption were not one-to-one, then decrypting a ciphertext would yield multiple plaintexts and so the original message could be ambiguous.[1] A diagram of these relationships can be seen in Figure 2.1 where m is the plaintext, c is the ciphertext, k is the encryption key and k' is the decryption key.

All forms of encryption share these components, although there can

[1]Some cryptosystems (Rabin public-key encryption for example) use encryption functions that are not one-to-one. However, these schemes rely on the fact that with extremely high probability only one of the preimages is sensible.

be slight variations. As we shall see, sometimes the encryption key and decryption key are identical and sometimes they are different. Often the encryption function and decryption function are identical as well, and it is only in how they are used that determines when we are encrypting and when we are decrypting.

Symmetric-key cryptographic algorithms are by far the most commonly used. In symmetric-key cryptography, there is only one key which serves as both the encryption key and the decryption key. This is not a surprising restriction since it seems counterintuitive to believe that one can decrypt something without knowing how it was encrypted in the first place.

The main advantage of symmetric-key cryptography is that generally it is computationally inexpensive, especially when compared to public-key cryptography discussed below. This means that it is feasible to encrypt the large amounts of data used in a protocol. One of the largest drawbacks to symmetric-key cryptography is the need to establish a different key for each pair of persons who wish to communicate confidentially.

In public-key cryptography, the keys used for encryption and decryption are different. Furthermore, there is no "easy" way to arrive at the decryption key from knowledge of the encryption key and of ciphertext encrypted with that key. Public-key cryptography gets its name from the fact that one of the keys (the encryption key) is made public. The other key is kept private. This way, anyone can communicate confidentially with the owner of the private key by encrypting with the published public key. Only the owner of the private key can decrypt the message.

The use of two different keys provides public-key cryptography with a number of advantages over symmetric-key cryptography. First, only one key pair is needed per individual instead of one key per pair of individuals. In other words the number of keys needed is linear in the number of principals instead of quadratic. Also, the use of two keys means that one can be publicly distributed, thus removing the need to securely exchange a key before commencing communication. Finally, it allows *digital signatures*. This is accomplished by having the owner of the private key encrypt a plaintext with the private key. Now anyone can verify the signature by decrypting with the public key. Furthermore, the signature can be attributed to the sole owner of the private key since only that individual could have created the correct ciphertext (signature).

Encryption and decryption are modeled in Brutus as black boxes.

In other words, the only way to generate a particular ciphertext is by applying the encryption algorithm to the appropriate plaintext with the appropriate key. Similarly, the only way to get any information out of a ciphertext is by applying the decryption algorithm to the ciphertext with the appropriate decryption key. This simplification is called the *perfect encryption assumption*. In real-life of course, this is not the case. A ciphertext could be generated by guessing, although the probability that the corresponding plaintext is meaningful would be extremely low. Also, cryptanalysis could be applied to a ciphertext to get partial if not complete information about the plaintext, without knowledge of the decryption key. In addition, certain cryptographic algorithms may have additional mathematical properties that are not accounted for in this simplified model which could conceivably lead to an attack on the protocol. For example, the Data Encryption Standard (DES), the most commonly used symmetric encryption algorithm, has the following property [50]:

$$c = E_k(m) \Rightarrow \overline{c} = E_{\overline{k}}(\overline{m})$$

where $\overline{x}$ is the bitwise compliment of x. This is by no means the only way in which our black box assumption of cryptography is broken. The RSA cryptosystem [69, 70], one of the more popular public-key cryptosystems, has the following property:

$$E_k(i \times m) = E_k(i) \times E_k(m)$$

for any integer i. And, *any* block cipher (including DES), has the following property when the length of message m_1 is a multiple of the block size of cryptosystem:

$$E_k(m_1 \cdot m_2) = E_k(m_1) \cdot E_k(m_2).$$

Such properties are not currently modeled in Brutus; therefore, it is possible to miss certain attacks. And while there is nothing in the framework preventing us from adding rules to model how an adversary might exploit these properties, doing so would seriously affect the efficiency of the message derivation engine that models the adversary's capabilities.

2.1.2 Hashes

Hash functions are probably more well known for their use in the hash table data structure than for their use in security protocols. For use in hash tables, it is desirable that a hash function h have the following properties as described in [50]:

compression - h maps an input of arbitrary finite length to an output of fixed length.

ease of computation - h should be easy to compute.

collision resistance - It is computationally infeasible to find a pair of inputs (x and x') that hash to the same value (such that $h(x) = h(x')$).

second preimage resistance - Given an x, it is computationally infeasible to find an x' such that $h(x) = h(x')$.

For use in security protocols, we may also desire the following property:

preimage resistance - It is computationally infeasible to compute an x such that $h(x) = y$ when only y is known.

Hash functions without the preimage resistance property are used most often for their compression property. In other words, protocol designers use the hash of a particular message to stand for the message itself. This is desirable because the hash of the message is typically much smaller than the original message. Since certain cryptographic operations are computationally expensive, designers often apply these operations only after hashing to reduce the cost of encryption. This use is justified because of the collision resistance and the second preimage resistance properties. In other words, the recipient of the hash can check that the hash is correct by computing the hash from the original message. The recipient can also rest assured that the sender will not claim the hash is of something else because it is infeasible for the sender to find a different message that hashes to the same value.

Hashing is modeled in BRUTUS via encryption with a publicly known symmetric key called "HASH". Because of the black box assumptions about cryptography, encryption satisfies the collision resistance and the second preimage resistance properties. In other words, encrypting a message m with the key "HASH" ensures that the only way to generate the hash of m is by encrypting m with "HASH". Since our level of abstraction on messages ignores message length and ease of computation, the compression and the ease of computation properties are not modeled in BRUTUS.

In addition, it is often desirable that a hash function not be invertible. These kinds of hash functions are called *one-way hash functions* and satisfy the preimage resistance property defined above. These one-way hash functions are also modeled using encryption. The only new property we need to satisfy is preimage resistance. To do this, we model one-way hashing via a universally known public key called "ONE-WAY HASH". The corresponding private key is not known by anyone. Thus anyone can hash a message using the public key "ONE-WAY HASH", but no one can invert the hash because no one knows the corresponding private key (decryption key).

2.1.3 Random Numbers

Random numbers are intimately tied to security. In fact, there is an equivalence between random numbers and cryptography. When encrypting a plaintext string of bits, the resulting ciphertext should look like a random string of bits in the sense the ciphertext does not give any information about the plaintext. To an attacker, then, all plaintexts are "equally likely" to have generated the ciphertext in question. Conversely, given a truly random string of bits of sufficient length, a perfect encryption function results from applying the *xor* function to the plaintext string and the random string. Note how the encrypting string (also known as a one-time pad) must be random to ensure that an attacker has no advantage when trying to guess the encryption string or equivalently, the original plaintext string. If it were not random, the attacker might be able to eliminate certain encryption strings from consideration and thus gain an advantage.

The other way that random numbers are used is as *nonces*. In the definition given by van Oorschot [50], a nonce is a value used no more than once for the same purpose. Typically, it is used to prevent the replay of an old or stale message. Usually, the term nonce is used to refer to a random or pseudo-random number used for this purpose. The idea is as follows:

1. The verifier generates a random number and sends it to the other party.

2. The other party *cryptographically binds* that random number to the reply message.

Theoretically, neither step necessarily requires encryption. However, in step two there must be a way of permanently associating the random number to the reply message in such a way that neither can be modified. In practice, this is accomplished by concatenating the number to the message and encrypting the result. Since the number is random, the verifier (i.e., the party that generated the random number) can be assured that any message containing that number must have been created *after* the random number was published. The adversary cannot replay an old message because it cannot bind the newly generated random number to an old message.

Brutus does not actually generate random numbers when simulating protocol execution. However, this is not a limitation. The properties of nonces that are used by cryptographic protocols are:

1. They are unique.
2. They are unpredictable.
3. They are known only to the generator, until disclosed.

All of these are modeled in Brutus by giving each protocol instance a unique nonce term for every nonce it is supposed to generate during the execution of the protocol. Notice that *when* a nonce is generated is not important, but only when it is disclosed. Its disclosure is explicitly modeled when a message containing the nonce is sent by the principal. The fact that the principal is modeled as having already generated the nonce is immaterial. The unpredictability property is trivially satisfied in our model because there is no notion of "guessing." The only way to generate a particular message is via standard message operations (encryption, concatenation, etc.) on *known* messages. So the model captures the necessary properties of nonces. On the other hand, Brutus cannot model poor nonce generators.

2.1.4 Timestamps

The freshness of messages can also be ensured through the use of timestamps. Instead of checking if an incoming message contains a recently generated random number, the receiver of the message can check the timestamp against a local clock. Typically, the value of the timestamp is subtracted from the current local time. This difference is then required to be within some *acceptance window* in order for the message to be accepted as fresh. In addition, the value of the timestamp may be required to be unique with respect to the sender of the message in order to prevent message replay. A second solution to prevent message replay is to use a sufficiently small acceptance window that would prevent a replayed message from possibly arriving "on time."

The use of timestamps can have certain advantages over the use of nonces. The use of nonces requires an extra message to communicate the initial random number to be used in the protocol. The party sending the random number must also maintain short-term state information about the random number so that it can compare with the next message received. It is this kind of state information that can become a liability with respect to denial of service attacks. By starting an arbitrary number of protocol sessions, an attacker can require a principal to generate and record an arbitrary number of random numbers. Applying formal methods to uncover these kinds of attacks is difficult, and only recently has there been any success in this direction [48].

The main drawback to the use of timestamps is the necessity of maintaining secure and synchronized clocks. The clocks must be secure so that an adversary can neither turn the clock back in order to get a principal to accept a stale message nor turn the clock forward in order to construct a message that can be accepted later. These security properties are difficult to guarantee in a distributed environment. Additionally, the values of the clocks must also be sufficiently close in order to prevent valid messages from being rejected because they fall outside the acceptance window. Keeping clocks synchronized is itself a difficult problem. And, as Menezes, van Oorschot and Vanstone point out, the clock synchronization necessary to make timestamps work requires network protocols that must be secure and could potentially depend on timestamps themselves, thus creating a circular dependency [50].

Using model checking to analyze protocols that use timestamps can be difficult because in general, model checking for timed systems can be computationally expensive. Various models and logics for timed systems have been proposed, including both discrete time models and continuous time models. The relative merits have been debated [2, 23] and their relative expressiveness has been compared. Although continuous time is more general (there are properties that are not preserved when moving from a continuous time model to a discrete time model), there are large classes of properties that are preserved when a model is discretized [23]. So far, it seems that only a discrete time model has been used to model check security protocols [38]. BRUTUS does not model time, so currently, a protocol using timestamps must be converted into one that uses nonces before it can be analyzed. Modeling time would be an obvious (although non-trivial) extension to BRUTUS. Much work has also been done on partial order reductions for timed systems [55, 60] and hopefully this work could be used directly for model checking security protocols with timestamps.

2.2 Protocol Examples

I now provide some simple examples of how the building blocks from the previous section can be used to construct security protocols. In particular, I concentrate on authentication, key exchange, and electronic commerce, since these are the kinds of protocols analyzed so far in my work. Again, I point out strengths, weaknesses, and areas for future work with respect to how these protocols can be modeled and analyzed using BRUTUS.

2.2.1 Authentication

The goal of an authentication protocol is to assure one party of the identity of the second party participating in the protocol. In other words, the major goal is to prevent impersonation. It should be the case that no party Ω can play the role of A and cause B to complete a protocol execution and accept Ω as being A. This property must hold even when Ω has previously or is concurrently executing the protocol with A or B, as well as when Ω observes protocol executions between A and B.

The most common way of establishing one's identity in the world of computer security is by demonstrating knowledge of a particular *secret*. The obvious (although not necessarily the best) way to do this is by simply stating what the secret is. This is basically the notion of a password. A and B share knowledge of a password. If this password is secret (known only to A and B), then when some party provides that password to B, B can be assured that the other party is A. This procedure can be modeled as a one message protocol:

$$A \rightarrow B : pswd_A$$

This is probably one of the simplest kinds of authentication and is easily modeled in Brutus by having the password be a unique message that is known only to A and B. This solution obviously does not work in an environment where messages can be overheard; Brutus easily finds this flaw. A naive solution might be to keep the password secret by encrypting it with key k before transmission.

$$A \rightarrow B : \{pswd_A\}_k$$

Since Brutus also models encryption, this modification is easily analyzed as well. The verification finds no violation of secrecy (the password is never divulged), but the authentication property is still not satisfied. In this case, an eavesdropper can replay the encrypted password and successfully impersonate one of the parties without ever learning the actual password:

$$\Omega(A) \rightarrow B : \{pswd_A\}_k$$

Because of the real danger of such replay attacks, most authentication protocols are based on challenge-response identification. In this scheme, one party proves its knowledge of a secret without actually revealing the secret. For example, if party A wishes to authenticate itself to party B, party A would request a challenge from party B. This challenge is a time-variant parameter such as a nonce. Party A then provides a response that depends on both the challenge and the secret. Party B can then verify that the response is correct. Eavesdropping this exchange should not provide an outsider with any useful information since any future authentication session would use a different challenge. A common response is to encrypt the challenge (either with a shared key or with a private key). The other party can then verify the response by decrypting it to get the challenge back. This protocol then takes the following form:

$$\begin{array}{rcl} B \rightarrow A & : & N_B \\ A \rightarrow B & : & \{N_B\}_k \end{array}$$

The uniqueness of the challenge is vital and the obvious replay attack is again found by Brutus if challenges are allowed to repeat (if party B is allowed to use the nonce N_B again). However, since encryption is treated as a black box, BRUTUS cannot find attacks that rely on partial information gained by observing challenge-response pairs (called a *known plaintext attack*) nor attacks that exploit some other weakness in the cryptosystem itself.

2.2.2 Key Exchange

If the authentication is being done across a network, often the two parties would like to exchange a key with which to secure future communication. The easiest way to do this is by simply including a new session key at some point during an authentication protocol. Unlike the simple authentication protocols presented above, *mutual* authentication is now required. In other words, it is not enough for B to know it is talking to A. Party A should also have some assurance that it is talking to B. An example of such a key exchange protocol is one proposed by Needham and Schroeder [58]. This protocol makes use of a trusted third party, S, who generates the session key for A and B. We describe this protocol below and how we model it in BRUTUS:

1. First, the initiator, A, generates a nonce, N_A, and sends it along with its own identity and B's identity to the server S.

$$A \rightarrow S : A, B, N_A$$

2. Next, the server, S, generates a session key K. It encrypts the pair (K, A) with a key shared with B. It then sends this along with the key K, B's name, and A's nonce N_A to A, all encrypted with a key shared with A.

$$S \rightarrow A : \{N_A, B, K, \{K, A\}_{K_B}\}_{K_A}$$

3. The initiator, A, now decrypts this message, verifies it nonce N_A, recovers the session key K, and forwards the encrypted pair $\{K, A\}_{K_B}$ to B.

$$A \rightarrow B : \{K, A\}_{K_B}$$

4. The responder, B, now decrypts this message and recovers the session key K. It now generates its own nonce, N_B, encrypts it with the session key, and sends it back to A.

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

5. The initiator, A, now decrypts the message and recovers the nonce N_B. The initiator then replies back to B by encrypting the value $(N_B - 1)$ with the session key K.

$$A \to B : \{(N_B - 1)\}_K$$

At this point A and B now share a key K with which to communicate privately.

This protocol is also easily modeled in BRUTUS. While the key generation is not directly modeled, it is simulated by ensuring that each different server session uses a different key in the same way that unique nonces are modeled. Also, while arithmetic operations are not directly modeled, the message $(N_B - 1)$ is replaced with the concatenation of N_B with the number 1. This message still has the same property that it is distinct from but based on the original nonce N_B. One can verify that the secrecy of the key K is maintained and that the authentication properties are guaranteed.

However, the protocol does have a weakness. Old session keys are valuable to the adversary. If an adversary Ω is able to acquire an old session key K that was used with B, it can get B to accept it again with the following sequence of messages which starts by replaying an overheard message number three.

3. $\Omega(A) \to B : \{K, A\}_{K_B}$
4. $B \to \Omega(A) : \{N'_B\}_K$
5. $\Omega(A) \to B : \{(N'_B - 1)\}_K$

Since the adversary has acquired the old session key K, it can use it to decrypt $\{N'_B\}_K$ and answer B's new nonce challenge N'_B with $\{(N'_B - 1)\}_K$.

Since BRUTUS does not model possible compromise of old keys directly, one must directly give the adversary knowledge of a previous key exchange session along with the compromised session key in order to model this possibility and reveal the flaw. In other words, if the model has the server using a fresh session key K, the adversary must be given knowledge of a previous session key K' along with the messages involved in that previous session, including that necessary third message $\{K', A\}_{K_B}$.

2.2.3 Electronic Commerce

Secure protocols have also been developed for use in electronic commerce. In addition to the privacy and authentication concerns raised in key exchange, electronic commerce protocols require additional properties such as the atomicity of transactions. Another added difficulty is the fact that electronic commerce protocol designers usually cannot assume that the parties involved in the transaction are honest. Unlike participants in a key exchange protocol, the participants in an electronic commerce protocol have something to gain by trying to subvert the protocol. In particular, the protocol must protect against modification of the terms of the transaction. Often, it is also desirable to have proof of the details of the transaction that can be presented to an outside party if necessary. This is accomplished through the use of the same basic cryptographic building blocks discussed in section 2.1.

As in the case of authentication and key exchange, privacy and protection against replay is often secured through the use of encryption and nonces or timestamps. However, how does a protocol protect the integrity of the details of the transaction? Often, these details are many. These details can include information about the parties involved, a description of the goods or services, the price paid, the customer's authorization, and delivery information. To use encryption on all this information can be costly, so a hash is usually applied to this information before encrypting. As we have already seen, hashing is modeled quite easily in Brutus using encryption with a special key.

The other cryptographic primitive needed for electronic commerce is the digital signature. This is necessary in order to provide an "authorization" mechanism for the protocol. In other words, the customer can authorize the purchase by electronically signing the purchase order. This will also provide proof of authorization to the merchant because anyone can get access to the public key to verify the signature. Since digital signatures can be implemented using public-key cryptography, Brutus can handle this quite easily as well.

Although Brutus has all the constructs necessary to model and to analyze electronic commerce protocols, the *state explosion problem* is a major limitation. Even these simple models have too many reachable states that must be explored during an exhaustive state space search. While electronic commerce protocols tend to have more participants (at least 3), the blowup in the size of the state space comes instead from the complexity of the messages. The size and complexity of the messages used in electronic commerce protocols is much greater than the size of the messages used in authentication and key exchange. For example, the 1KP protocol analyzed in chapter 7 requires 6 messages, 2 of which have more than 20 fields. The authentication protocol described earlier has one message with 7 fields while the other messages have only 2 or 3 fields. Because the verification requires that the adversary try all possible messages that match the structure of the messages in the protocol, the number of messages tried and thus the size of the state space is bound to grow exponentially in the length of the messages. The situation is analogous to comparing the number of strings of length 7 to the number of strings of length 20. (The number of distinct strings grows exponentially in the length of the string just as the number of distinct messages grows exponentially in the length of the message). Later chapters explore ways of trying to cut down the size of the state space.

Chapter 3

Formal System

The formal system I use to analyze security protocols is described in this chapter. Because I am using model checking, the formal system will consist of two parts. The first is an operational model of the system. This model will implicitly describe the possible behaviors of the system. This model will consist of independently executing processes that communicate via messages. The second component is a logic or language in which to express the requirements of the system. Generally, this language will be at a higher level of abstraction in the sense that it does not say *how* a certain relationship between states and/or actions is enforced, as long as the relationship holds for all traces of the model. The implementation of these components in BRUTUS is discussed below.

3.1 The Model

Like other model checkers, BRUTUS requires an operational description of the behavior of agents participating in a protocol. This section begins with a description of the messages involved in a protocol model and how they are constructed. A mathematical model for the agents is then presented. Finally, there is a discussion of the different actions allowed during the execution of a protocol and how they change the state of the system.

3.1.1 Messages

Typically, the messages exchanged during the run of a protocol are constructed from smaller sub-messages using concatenation and encryption. The smallest such sub-messages (i.e., ones which contain no sub-messages themselves) are called *atomic messages*. There are four kinds of *atomic messages*.

- *Data* messages play no role in how the protocol works but are intended to be communicated between principals. The set of all data messages is $\mathcal{D}$.

- *Principal names* are used to refer to the participants in a protocol. The set of all honest principal names is $\mathcal{P}$. To this set we add the unique name Ω for the adversary. The set $\mathcal{P}_\Omega$ refers to the set of all principal names, including the adversary.

- *Nonces* can be thought of as randomly generated numbers. The intuition is that no one can predict the value of a nonce; therefore, any message containing a nonce can be assumed to have been generated after the nonce was generated. (It is not an "old" message.) The set of nonces is $\mathcal{N}$.

- *Keys* are used to encrypt messages. The set of all keys is $\mathcal{K}$. In the formal model, a function is used to form keys. This allows me to associate public keys and session keys with principals. The three functions are:

$$\begin{array}{rcl} \mathit{pubkey} & : & \mathcal{P}_\Omega \rightarrow \mathcal{K} \\ \mathit{privkey} & : & \mathcal{P}_\Omega \rightarrow \mathcal{K} \\ \mathit{symkey} & : & \mathcal{P}_\Omega^+ \rightarrow \mathcal{K} \end{array}$$

  So $\mathit{pubkey}(A)$ is A's public key and $\mathit{privkey}(B)$ is B's private key. The function *symkey* can take any nonempty sequence of principal names as an argument. These principal names are meant to denote the principals who are to share the key. For example $\mathit{symkey}(AS)$ might be a symmetric key shared between A and a server S while $\mathit{symkey}(AB)$ might be a session key intended to be used by A and B. In the literature, notation of the form K_{AS} is used for a symmetric key and notation of the form K_B^{-1} is used for a private key. For the sake of readability, this notation is preferred here over the more formal function described above.

Note that keys have the property that every key k has an inverse k^{-1} such that for all messages m, $\{\{m\}_k\}_{k^{-1}} = m$. For public key cryptography, public keys and private keys are inverses while for symmetric key cryptography, every key is its own inverse.

$$\begin{aligned}(pubkey(X))^{-1} &= privkey(X)\\ (privkey(X))^{-1} &= pubkey(X)\\ (symkey(X))^{-1} &= symkey(X)\end{aligned}$$

Let $\mathcal{A}$ denote the space of *atomic messages.* Then

$$\mathcal{A} = \mathcal{D} \cup \mathcal{P}_\Omega \cup \mathcal{N} \cup \mathcal{K}.$$

The set of all messages $\mathcal{M}$ over some set of atomic messages $\mathcal{A}$ is defined inductively as follows:

- If $a \in \mathcal{A}$ then $a \in \mathcal{M}$. (Any *atomic message* is a message.)
- If $m_1 \in \mathcal{M}$ and $m_2 \in \mathcal{M}$ then $m_1 \cdot m_2 \in \mathcal{M}$. (Two messages can be paired together to form a new message.)
- If $m \in \mathcal{M}$ and key $k \in \mathcal{K}$ then $\{m\}_k \in \mathcal{M}$. (A message m can be encrypted with key k to form a new message.)

The notion of messages can also be generalized to *message templates.* A message template can be thought of as a message containing zero or more message variables. $\mathcal{V}$ will denote the set of message variables. To extend messages to message templates the following rule is added to the inductive definition of messages:

- If $v \in \mathcal{V}$ is a message variable, then $v \in \mathcal{M}$.

Because keys have inverses, a message of the form $\{\{m\}_k\}_{k^{-1}}$ is always rewritten as (or simplified to) m. It is also important to note that the following *perfect encryption assumption* is necessary. The only way to generate $\{m\}_k$ is from m and k. In other words, for all messages m, m_1, and m_2 and keys k and k'

1. $\{m\}_k \neq m_1 \cdot m_2$

2. $\{m\}_k = \{m'\}_{k'}$ implies $m = m'$ and $k = k'$

New messages are constructed from already known or existing messages by encryption, decryption, pairing (concatenation), and projection. This message generation is modeled using a derivation relation "$\vdash$" which captures how a message m can be derived from some initial set of information I.

1. If $m \in I$ then $I \vdash m$.

2. If $I \vdash m_1$ and $I \vdash m_2$ then $I \vdash m_1 \cdot m_2$. (pairing)

3. If $I \vdash m_1 \cdot m_2$ then $I \vdash m_1$ and $I \vdash m_2$. (projection)

4. If $I \vdash m$ and $I \vdash k$ for key k, then $I \vdash \{m\}_k$. (encryption)

5. If $I \vdash \{m\}_k$ and $I \vdash k^{-1}$ then $I \vdash m$. (decryption)

We will use the the notation $\overline{I}$ to denote the *closure* of the set I under the rules given above. In other words $m \in \overline{I}$ if and only if $I \vdash m$.

These rules define the most commonly used derivability relation in the literature. The rules stem from an intruder model proposed by Dolev and Yao [19].

When trying to subvert a protocol, an adversary can only use messages it can derive from some initial set of information and from overheard messages using these rules. For example, if I_0 is some finite set of messages overheard by the adversary and possibly some initially known messages, then $\overline{I_0}$ represents the set of all messages known to the adversary. The adversary is allowed to send any message in $\overline{I_0}$ to any honest agent in an attempt to subvert the protocol. More rules could be added as proposed by Mitchell [56], but at the cost of a much more complex system.

In general, $\overline{I}$ is infinite, but researchers have taken advantage of the fact that one need not actually compute $\overline{I}$. It suffices to check $m \in \overline{I}$ for some finite number of messages m. However, checking if $m \in \overline{I}$ must still be decidable. I discuss this topic in greater detail in Chapter 4.

3.1.2 Instances

A protocol is modeled as the asynchronous composition of a set of named communicating processes, which model the honest agents and the adversary. The model should also include an insecure and unreliable communication medium, in which a principal has no guarantees about the origin of a message, and where an adversary is free to eavesdrop on all communications and free to interfere by introducing fake messages. These properties of the communication medium are modeled by requiring that all communications go through the adversary. In other words, all sent messages are intercepted by the adversary, and all messages received by honest agents are actually sent by the adversary. In addition, the adversary is allowed to create and send new messages from the information it gains by eavesdropping. The specific case where a message arrives safely at its destination without being overheard is not explicitly modeled. However, this behavior is simulated by the case where the adversary intercepts the message and forwards it unchanged. Although the adversary has now overheard the message, this simulation is "safe" in the sense that any attack that could arise from an execution where the adversary does not overhear a message will also arise from a similar execution where the adversary does overhear the message.

In order to make the model finite, a bound is placed on the number of times a principal may attempt to execute the protocol. Each such attempt will be called a *session.* Each session will be modeled as a principal *instantiating* some role in the protocol (i.e., initiator or responder). For this reason, the formal model of an individual session is called an *instance.* Each instance is a separate copy or instantiation of a principal and consists of a single execution of the sequence of actions that make up that agent's role in the protocol, along with all the variable bindings and knowledge acquired during the execution. A principal can have multiple instances, but each instance is executed only once. Combining these instances with a single instance of the adversary results in the model for the entire protocol.

Definition 3.1.1 *Each* instance *of an honest principal is modeled as a 4-tuple* $\mathcal{I}_i = \langle H_i, B_i, I_i, P_i \rangle$ *where:*

- H_i *is the instance ID. This ID is unique for each instance. The notation will be abused by referring to the instance* $\mathcal{I}_i$ *by its ID* H_i *(or* H_Ω*).*

- $B_i \subseteq \mathcal{V} \times \mathcal{M}$ *is a set of variable bindings for this session. It is a partial map from some subset of the variables appearing in the session to the set of messages.* B *always contains at least one binding,* (pr, n) *where* $\mathrm{pr} \in \mathcal{V}$ *is a special variable meant to hold the name of the principal on whose behalf this instance is executing and* $n \in \mathcal{P}$ *is the name of this principal. The initial set of bindings,* B_i^0*, consists of the binding for* pr*, along with the bindings for things like freshly generated nonces and keys. Note that instances that are playing the same roles will have the same bound variables, although their values will differ except for possibly the variable* pr*.*

- $I_i \subseteq \mathcal{M}$ *is the set of messages known by this particular instance. As in the case of bindings, the initial set of known messages is denoted as* I_i^0*. This set is partitioned into a subset consisting of fresh messages* $I_i^\sharp$ *and a set of commonly known messages* $I_i^\flat = I_i^0 - I_i^\sharp$*.* $I_i^\sharp$ *includes messages like newly generated nonces and session keys. In order to be able to parameterize the roles of the protocol by the nonces and keys, freshly generated messages (which will differ even among instances playing the same role) are referred to using variables that are common to the different instances playing the same role. Therefore, the set of freshly generated messages will be equal to the values of the variables bound in* $B_i^0 - \{\mathrm{pr}\}$ *(i.e., the range of the initial bindings except for* pr*). In other words, we have the following equality:*
$$I_i^\sharp = B_i^0(\mathcal{V} - \{\mathrm{pr}\})$$
In particular, then, two instances $\mathcal{I}_i$ *and* $\mathcal{I}_j$ *of the same principal have the same set of initial messages except for freshly generated messages. In other words,* $B_i^0(\mathrm{pr}) = B_j^0(\mathrm{pr})$ *implies* $I_i^\flat = I_j^\flat$*.*

- P_i *is a process description given as a sequence of actions to be performed. These actions include the predefined-defined actions* **<u>send</u>** *and* **<u>receive</u>***, as well as user defined internal actions such as possibly* **commit** *and* **debit** *actions. Note that if two instances* $\mathcal{I}_i$ *and* $\mathcal{I}_j$ *are playing the same role, their process descriptions are identical in the initial state* ($P_i^0 = P_j^0$)*.*

For the sake of uniformity, the adversary is modeled as a special instance, Ω. However, the adversary is not bound to follow the protocol and so it does not make sense to include either a sequence of actions P_Ω or a set of bindings B_Ω for the adversary. Instead, at any time, the adversary can receive any message or it can send any message it can generate from its set of known messages I_Ω. Thus the only component of the adversary instance Ω that is of any interest is the set of known messages I_Ω.

The global model is the asynchronous composition of the models for each instance, including the adversary. Because each instance has a unique instance ID, the state can be represented as a collection (set) of instances instead of an ordered tuple. If there are k instances of the honest agents, the *global state* is

$$\{\Omega, \mathcal{I}_1, \ldots, \mathcal{I}_k\}$$

a collection consisting of the adversary and the k honest instances. Again, I will abuse the notation and refer to the instances by their IDs. In other words, I will refer to the global state as

$$\{H_\Omega, H_1, \ldots, H_k\}$$

where H_Ω is the instance ID of the adversary and each H_i for $1 \leq i \leq k$ is the instance ID of one of the honest agents.

Now that the idea of the state of an instance and the idea of the state of a protocol have been formalized, a formalization is necessary for how the model actually executes. In other words, it is necessary to define how one state transitions into another state. These transitions correspond to actions taken by the individual instances. These actions are described below.

3.1.3 Actions

The actions allowed during the execution of a protocol include the two predefined actions **send** and **receive** , as well as possibly some user defined **internal** actions. The model changes global state as a result of actions taken by the individual components, or instances. More formally, the transition relation is defined as follows:

$$\rightarrow \subseteq \Sigma \times H \times A \times \mathcal{M} \times \Sigma$$

where

- Σ is the set of global states

- H is the set of instance IDs of the honest agents
- A is the set of action names (including **send** and **receive**)
- $\mathcal{M}$ is the set of all possible messages.

The notation $\sigma \stackrel{H_i \cdot a \cdot m}{\longrightarrow} \sigma'$ is used in place of the more cumbersome $(\sigma, H_i, a, m, \sigma') \in \rightarrow$. In the definitions given below, $\Omega = \langle H_\Omega, \emptyset, I_\Omega, \emptyset \rangle$ refers to the adversary while $\mathcal{I}_i = \langle H_i, B_i, I_i, P_i \rangle$ refers to the honest instances. Again,

$$\sigma = \{\Omega, \mathcal{I}_1, \ldots, \mathcal{I}_k\}$$

denotes the global state before the transition and

$$\sigma' = \{\Omega', \mathcal{I}'_1, \ldots, \mathcal{I}'_k\}$$

denotes the global state after the transition. In addition, $\overline{B}$ refers to the obvious extension of a set of bindings B from the domain of variables to the domain of message templates. In other words, $\overline{B}(m)$ is the result of substituting each variable v appearing in m, with its corresponding value $B(v)$. More formally:

Definition 3.1.2 *Let B be a set of bindings, let k be a key, and let m, m_1, and m_2 be message templates. Then we extend B to the set of message templates with variables in the domain of B as follows:*

- $\overline{B}(m) = m$ *if* $m \in \mathcal{D} \cup \mathcal{P}_\Omega \cup \mathcal{N}$.
- $\overline{B}(pubkey(X)) = pubkey(\overline{B}(X))$
- $\overline{B}(privkey(X)) = privkey(\overline{B}(X))$
- $\overline{B}(symkey(X)) = symkey(\overline{B}(X))$
- $\overline{B}(v) = B(v)$ *for* $v \in \mathcal{V}$.
- $\overline{B}(m_1 \cdot m_2) = \overline{B}(m_1) \cdot \overline{B}(m_2)$
- $\overline{B}(\{m\}_k) = \{\overline{B}(m)\}_{\overline{B}(k)}$

We can now concentrate on the definition of the transition relation. The transition relation definition has three parts. The transition relation must be defined for **send** actions, for **receive** actions, and for **internal** actions.

- $\sigma \overset{H_i \cdot \textbf{send} \cdot m}{\longrightarrow} \sigma'$

  An instance H_i can send message m in global state σ and the new global state is σ' if and only if the following hold:

  1. $I'_\Omega = I_\Omega \cup m$. (The adversary adds m to the set of messages it knows.)
  2. If in σ, $\mathcal{I}_i = \langle H_i, B_i, I_i, P_i \rangle$ and in σ', $\mathcal{I}'_i = \langle H'_i, B'_i, I'_i, P'_i \rangle$ then
     - $H'_i = H_i$ (The instance ID remains unchanged.)
     - $B'_i = B_i$ (The bindings remain unchanged.)
     - $I'_i = I_i$ (The set of information remains unchanged.)
     - $P_i =$ **send** $(s\text{-}msg) \cdot P'_i$ (The instance is ready to send a message, and that send action is removed from the process description in the new state.)
     - $m = \overline{B_i}(s\text{-}msg)$ (The message that the instance is ready to send and the actual message sent are the same.)
  3. $\mathcal{I}_j = \mathcal{I}'_j$ for all $j \neq i$. (All other instances remain unchanged.)

  Notice that the adversary intercepts all messages. In order for the intended recipient to receive the message, the adversary must forward it.

- $\sigma \overset{H_i \cdot \textbf{receive} \cdot m}{\longrightarrow} \sigma'$

  An instance H_i can receive message m in global state σ and the new global state is σ' if and only if the following hold:

  1. $m \in \overline{I_\Omega}$. (The adversary can generate the message m.)
  2. If in σ, $\mathcal{I}_i = \langle H_i, B_i, I_i, P_i \rangle$ and in σ', $\mathcal{I}'_i = \langle H'_i, B'_i, I'_i, P'_i \rangle$ then
     - $H_i = H'_i$ (The instance ID remains unchanged.)

- B'_i is the smallest extension of B_i such that $\overline{B'_i}(r\text{-}msg) = m$ (The bindings of the instance are updated correctly, and the message received matches the message template in the receive action.)
- $I'_i = I_i \cup m$ (The information of the instance is updated correctly.)
- $P_i =$ **receive** $(r\text{-}msg) \cdot P'_i$ (The instance is ready to receive a message, and that action is removed from the process description in the next state.)

3. $\mathcal{I}_j = \mathcal{I}'_j$ for all $j \neq i$. (All other instances remain unchanged.)

Notice that all messages come from the adversary, although as stated above they may simply have been forwarded unchanged. This allows us to also model an adversary than can modify, misdirect, and suppress messages.

- $\sigma \xrightarrow{H_i \cdot A \cdot m} \sigma'$

An instance H_i can perform some user defined **internal** action A with argument m in global state σ and the new global state is σ' if and only if the following hold:

1. If in σ, $\mathcal{I}_i = \langle H_i, B_i, I_i, P_i \rangle$, and in σ', $\mathcal{I}'_i = \langle H'_i, B'_i, I'_i, P'_i \rangle$ then
   - $H'_i = H_i$ (The instance ID remains unchanged.)
   - $B'_i = B_i$ (The bindings remain unchanged.)
   - $I'_i = I_i$ (The set of information remains unchanged.)
   - $P_i = A(msg) \cdot P'_i$ (The instance is ready to perform action A, and that action is removed from the process description in the new state.)
   - $m = \overline{B_i}(msg)$. (The message argument in the process description and the actual message occurring in the action are the same.)
2. $\mathcal{I}_j = \mathcal{I}'_j$ for all $j \neq i$. (All other instances remain unchanged.)

All that remains is to briefly formalize what is meant by a trace of the model:

Definition 3.1.3 *A* trace $\pi = \sigma_0\alpha_1\sigma_1\alpha_1 \ldots \alpha_n\sigma_n$ *is a finite alternating sequence of states* (σ_i) *and actions* (α_i) *such that* $\sigma_{i-1} \stackrel{\alpha_i}{\to} \sigma_i$ *for all* $0 < i \leq n$. *The number* n *is referred to as the* length *of* π.

3.2 Logic

In Brutus, a temporal logic is used to specify the requirements or the desired properties of the protocol. The quantifiers in this logic range over the finite set of instances in a model. In addition, the logic includes the past-time modal operator so that one can refer to things that happened in the history of a particular protocol trace. The atomic propositions of the logic can refer to the bindings of variables in the model, to actions that occur during execution of the protocol, and to the knowledge of the different agents participating in the protocol. This discussion begins with the syntax of the logic, followed by the formal semantics of the logic.

3.2.1 Syntax

Brutus uses a first order logic with quantifiers ranging over the finite set of instances. The atomic propositions are used to characterize states, actions, and knowledge in the model. The arguments to the atomic propositions are terms expressing instances or messages. Below is a formal description of the terms in the logic.

- If $H_i \in ID$ is an instance ID, then H_i is an instance term. Often, an instance ID like A_2 is used to refer to the second instance of principal A.

- If $m \in \mathcal{A}$ is an atomic message, m is a message term.

- If $H_i \in H$ is an honest instance ID and $v \in \mathcal{V}$ is a message variable then $H_i.v$ is a message term representing the binding of v in the instance H_i.

- If m_1 and m_2 are message terms, then $m_1 \cdot m_2$ is a message term.

- If m_1 and m_2 are message terms, then $\{m_1\}_{m_2}$ is a message term.

As in standard first order logic, atomic propositions are constructed from terms using relation symbols. The predefined relation symbols are "=" and "**Knows**". The user can also define other relation symbols which would correspond to user defined actions in the model. The relation symbols are used infix to construct atomic propositions as follows:

- If m_1 and m_2 are message terms, then $m_1 = m_2$ is an atomic proposition. Examples of this atomic proposition would include checking if a customer and merchant agree on the price of a purchase ($C_0.price = M_0.price$), or checking if a particular instance of A believes it is authenticating with B ($A_0.partner = B$).

- If H_i is any instance ID (including the adversary) and m is a message term, then H_i **Knows** m is an atomic proposition which intuitively means that instance H_i knows the message m. This proposition can be used to verify that the adversary has not compromised the session key as follows: $\neg(H_\Omega\ \textbf{Knows}\ K)$

- If H_i is an honest instance ID, m is a message term, and **Act** is a user defined action, then H_i **Act** m is an atomic proposition which intuitively means that instance H_i performed action **Act** with message m as an argument. For example, this could be used to check if a customer has committed to a transaction with identifier TID (C_0 **commit** TID).

Well-formed formulas (wffs) are built up from atomic propositions with the usual connectives from first-order logic and modal logic. These connectives are described below with an intuitive description of the intended meaning of the connective. The formal meaning will be given in the formal semantics.

- If ϕ is an atomic proposition, then ϕ is a wff.

- If ϕ is a wff, then $\neg\phi$ is a wff.

- If ϕ_1 and ϕ_2 are wffs, then $\phi_1 \wedge \phi_2$ is a wff.

- if ϕ is a wff, then $\Diamond_P \phi$ is a wff.

The following common shorthands are also used:

- $\phi_1 \vee \phi_2 \equiv \neg(\neg\phi_1 \wedge \neg\phi_2)$
- $\phi_1 \rightarrow \phi_2 \equiv \neg\phi_1 \vee \phi_2$
- $\phi_1 \leftrightarrow \phi_2 \equiv \phi_1 \rightarrow \phi_2 \wedge \phi_2 \rightarrow \phi_1$
- $\Box_P\phi \equiv \neg\Diamond_P\neg\phi$

Quantifiers range over instance IDs. Because all the models are finite, quantification is equivalent to a finite number of disjunctions or conjunctions. Therefore, quantification is also treated as a shorthand.

- $\exists x.\phi \equiv \bigvee_{H_i \in ID}[\phi/(x \mapsto H_i)]$.
  This wff is true if the formula $[\phi/(x \mapsto H_i)]$ is true for some instance H_i (i.e., if ϕ is true when some instance H_i is substituted for the variable x). Note that, strictly speaking, ϕ is not a formula allowed by the syntax because somewhere inside it contains the formal variable x instead of an instance ID. However, this notation is simply a shorthand for explicitly writing out all the disjuncts. The result of expanding this shorthand *is* a wff.
- $\forall x.\phi \equiv \neg\exists x.\neg\phi$.

3.2.2 Semantics

The formal meaning of these logic formulas is given by the semantics below. These semantics are given in terms of the formal model presented in Section 3.1. In other words, the semantics describe how to evaluate a formula in a given state of a given trace. As in the description of the syntax, the semantics begin with the terms of the logic.

Definition 3.2.1 *The notation $\sigma(m)$ is used to denote the interpretation of the message term m in the state σ. This is defined inductively on the structure of m.*

- *$\sigma(a) = a$ for any atomic message $a \in \mathcal{A}$.*
- *$\sigma(H_i.v) = B_i(v)$ where $\mathcal{I}_i = \langle H_i, B_i, I_i, P_i\rangle$ is an instance in the global state σ.*
- *$\sigma(m_1 \cdot m_2) = \sigma(m_1) \cdot \sigma(m_2)$ for message terms m_1 and m_2.*

- $\sigma(\{m\}_k) = \{\sigma(m)\}_{\sigma(k)}$ *for message terms* m *and* k.

The wffs of the logic are interpreted over the traces of a particular model. Recall that a trace π consists of a finite, alternating sequence of states and actions $\sigma_0\alpha_1\sigma_1\ldots\sigma_n$. The length of a trace π is denoted by length(π). The semantics of wffs in the model are given via a recursive definition of the satisfaction relation $\models$. The notation $\langle\pi, i\rangle \models \phi$ means that the i^{th} state in the trace π, satisfies the formula ϕ. The satisfaction relation is defined for atomic propositions as follows:

- $\langle\pi, i\rangle \models m_1 = m_2$ iff $\sigma_i(m_1) = \sigma_i(m_2)$. Thus the formula $m_1 = m_2$ is true in a state iff the interpretations of m_1 and m_2 are equal. In other words, two message terms are equal in a state if after applying the appropriate substitutions to the variables appearing in the message terms, the resulting messages are equal.

- $\langle\pi, i\rangle \models H_j$ **Knows** m iff $\sigma_i(m) \in \overline{I_j}$ for the instance $\mathcal{I}_j = \langle H_j, B_j, I_j, P_j\rangle$ in σ_i. In other words, the formula H_j **Knows** m is true in a state if the instance $\mathcal{I}_j$ can derive message m from its set of known messages in that state. The instance need not be an honest instance. For example, $\langle\pi, i\rangle \models H_\Omega$ **Knows** m iff $\sigma_i(m) \in \overline{I_\Omega}$.

- $\langle\pi, i\rangle \models H_j\ A\ m$ iff $\alpha_k = H_j \cdot A \cdot \sigma_i(m)$ for some $1 \leq k \leq i$. In other words, $\langle\pi, i\rangle \models H_j\ A\ m$ for some user defined action A iff the honest instance H_j has performed action A with argument m.

The extension of the satisfaction relation to the logical connectives is straightforward.

- $\langle\pi, i\rangle \models \neg\phi$ iff $\langle\pi, i\rangle \not\models \phi$.

- $\langle\pi, i\rangle \models \phi_1 \wedge \phi_2$ iff $\langle\pi, i\rangle \models \phi_1$ and $\langle\pi, i\rangle \models \phi_2$

- $\langle\pi, i\rangle \models \Diamond_P\phi$ iff there exists a $0 \leq j \leq i$ such that $\langle\pi, j\rangle \models \phi$ In other words, the formula $\Diamond_P\phi$ is true in a state of a trace π if the formula ϕ is true in any state of the trace up to and including the current state.

A formula ϕ is said to be true in a trace π (denoted as $\pi \models \phi$) iff ϕ is true in *every state* of the trace π.

3.2.3 Specification Examples

Examples of how interesting security properties can be expressed in the logic are given below:

- *Payment Authorization.* For the 1KP protocol, one wishes to show that whenever the customer's account is debited, the customer must have authorized that debit. This is expressed with the formula

$$\forall a\ .\ (a.pr = A) \wedge (a \text{ debit } (a.\text{CC}, a.\text{price})) \rightarrow$$
$$\exists c\ .\ (c.pr = C) \wedge (a.\text{CC} = c.\text{CC}) \wedge \Diamond_P (c \text{ auth } a.\text{price})$$

  This formula states that for all instances a, if a is an instance of the authority A, and a debits the credit card account CC by the amount $PRICE$, then there exists an instance c of the customer C with that same credit card number that authorized a debit of that amount.

  Note that this formula would be true of a trace where there were multiple debits of a credit card by the same amount, but where there was only one authorization. However, every electronic commerce model we tried with more than one instance of the credit card authority was too large to be searched successfully. Therefore only one debit action could possibly occur and this formula sufficed. The formula can be corrected by adding information identifying the transaction (like a transaction ID or nonce) to the argument lists for the auth and debit actions.

- *Privacy.* The 1KP electronic commerce protocol should not reveal information about the transaction. In other words, only the appropriate principals should know the order information. This can be expressed with the formula

$$\forall s\ .\ \forall c\ .\ (c.pr = C) \wedge\ (s\ \ \textbf{Knows}\ \ c.DESC) \rightarrow$$
$$(s.pr = C) \vee (s.pr = M)$$

  This formula states that for all instances s, if s knows the customer's description of the transaction, then s is an instance of either the customer or the merchant.

- *Non-repudiation.* One may want to check that a principal cannot deny knowledge of a particular value (a key or nonce). For instance, in the Needham-Schroeder protocol, one can check that whenever principal A ends a session with principal B, B must know the nonce created by A. This is a somewhat weak notion of non-repudiation. A may not be able to prove B's knowledge of the nonce. Indeed, A may not even be able to convince itself that B knows the nonce. The formula below simply states that there is no trace in which B does not know the nonce.

  $$\forall s \,.\, \forall t \,.\, s \text{ end } t.pr \rightarrow \exists r \; [(t.pr = r.pr) \wedge (r \;\; \textbf{Knows} \;\; s.Nonce)]$$

  This formula states that for all pairs of instances s and t, if s ends a protocol session with t's principal, then there is an instance r of that principal (possibly $r = t$) that knows the nonce generated by s.

- *Anonymity.* Certain protocols claim to maintain the anonymity of one or more of the parties involved. The requirement that A's anonymity is maintained can be expressed with the following formula:

  $$\forall s \,.\, s \;\; \textbf{Knows} \;\; A \rightarrow \Box_P(s \;\; \textbf{Knows} \;\; A)$$

  This formula states that for all instances s, if s knows the name of principal A, then it always knew the name of A. In other words, it did not learn the name during the execution of this protocol.

  Again, this is weaker than what is usually meant by anonymity. Anonymity would require that a particular message or transaction not be associated with a principal. While learning a principal's name during the execution of a protocol would suggest a failure to guarantee anonymity, the formula does not guarantee anonymity. In particular, if an adversary already knew the name of the principal in question, this formula would be satisfied regardless of what the protocol does.

Chapter 4

Algorithms

This chapter focuses on the algorithms used in BRUTUS. First, the depth-first search algorithm that is used to perform a state space exploration of the model is discussed. The second algorithm considered is the algorithm used to keep track of the messages known by the various agents, including the adversary. Finally, the use of partial order reductions and of symmetry reductions to reduce the size of the state space is presented. Only the actual algorithm is described (i.e., how and when BRUTUS decides to prune the computation tree). The proofs of correctness are presented in chapters 5 and 6.

4.1 Search Algorithm

Recall that a trace is an alternating sequence of global states and actions, and that we are interested in checking all possible traces. Clearly, everything in the model is finite except for the set of messages that the adversary can generate. If the adversary never generates an infinite set of messages, then the entire model is finite. This would mean that BRUTUS can perform a depth-first search to check that all traces of the model satisfy the given security properties. For the moment, we will assume that this is the case. The question of ensuring that the search is finite will be considered later.

A straightforward depth-first search algorithm is presented in Figure 4.1. This algorithm will form the basis of the explicit model checking algorithm in BRUTUS. Note that the specification formula ϕ is included

```
proc DFS(s, φ)
  push(S, s)
  while (not empty(S)) do
        s_cur = pop(S)
        if sat(s_cur, φ)
          then L = expand(s_cur)
                 foreach s_next ∈ L do
                   push(S, s_next)
                 od
           else counter-example(s_cur, φ)
        fi
  od
```

Figure 4.1: Search algorithm

as a parameter to *DFS*. This is because BRUTUS must check every state to ensure that it satisfies the specification ϕ. This is done by calling the function *sat*. This function is defined recursively and is a straightforward implementation of the $\models$ relation defined in Section 3.2. If the state does not satisfy ϕ, then BRUTUS prints out the partial trace that has led to the current state by calling the procedure *counter-example*. Note that both *sat* and *counter-example* need access to the history or path to the current state. This path is updated in the function *expand* by keeping a pointer in every reached state that points back to the previous state. It is important to note that, unlike the usual depth-first search algorithm, *DFS* does *not* check to see if an expanded state is one that has been reached already. In other words, BRUTUS does not keep track of the set of states already visited. This is because the specification language allows us to refer to past events. In some sense, then, the current state must include history information. Since each possible sequence of actions is tried at most once, one can never reach the same state via the same sequence of actions. This means that no two states (even otherwise identical states) ever have the same history; therefore, no two states are ever the same in our model.

The function *expand* generates the set of all possible next-states from a specific current state. Again, this is a straightforward implementation of the transition relation defined in section 3.1. BRUTUS simply checks all instances in the current state and determines what actions they can take. Instances can always take internal actions. Send actions are also always enabled, and when they are taken, the adversary's knowledge is updated appropriately. An instance can take a receive action only when the adversary can generate a message that matches the template of the message that the instance is waiting to receive. Often, the adversary can generate more than one message that matches this template. If this is the case, each message leads to a different receive action and a different next state. As mentioned in the description of *DFS*, when *DFS* constructs the next state a back pointer to the current state is created so that BRUTUS can keep track of the trace up to the current state. In the symmetry and the partial order discussion (Sections 4.4 and 4.5) I will describe how the *expand* function is modified to restrict the set of next states to be considered.

4.2 Ensuring Finite Branching

Before discussing how to maintain the set of messages known by the adversary, let us return to the question of ensuring that our algorithm terminates. We need to ensure that no part of the search becomes infinite. The only part of the model that is infinite is the set of known messages. The set of messages known by any instance is usually infinite. For example, an agent can repeatedly encrypt a message with a key. Therefore, simply checking to see if an agent knows a message could involve checking an infinite set unless some care is taken. This will be the topic of section 4.3. For now we concentrate on the other way that the infinite set of known messages could affect our search. Whenever an instance is waiting to receive a message, the adversary must be able to generate that message. But receivers specify message templates that they are willing to receive. This template could match an infinite number of messages. Since the adversary can also generate an infinite number of messages, this could lead to an infinite number of **receive** actions that an instance could take. This is a much more fundamental problem since now the number of states in the model becomes infinite. In particular, the branching factor at a particular state becomes infinite. The number of messages considered must somehow be restricted to a finite set. This is not a "safe" thing to do. By not considering certain messages, we might end up not considering a behavior that violates the specification. This, in turn, would lead us to incorrectly conclude that the model satisfies the specification. Although there has been work on finding a finite subset of messages that would still preserve all errors in the model [74], this subset is still too large to be practical for model checking. In what follows two "natural" (but not necessarily "safe") ways of limiting the number of messages are considered.

4.2.1 Bound on message complexity

The easiest and probably most straightforward way of limiting the number of messages tried by the adversary is to limit the number of message construction rules that it uses when constructing a message. In other words, one can bound the "size" or more accurately the complexity of the messages the adversary can generate. While this seems ad hoc, it also seems intuitive that some such bound should exist and that repeated concatenation and encryption could only help up to a certain point. Stoller has computed a bound on the number of nested encryptions that need to be allowed in order to prove the protocol correct [74]. Roughly, this bound is the product of the number of encryptions appearing in all messages sent and received in the protocol and the total number of instances in the protocol model. Since the number of messages the adversary can generate of a certain complexity (with a certain number of nested encryptions) is roughly exponential in the number of nested encryptions, this bound is too high to use in practice. Typically, one must severely restrict the power of the adversary by allowing it to encrypt only once or possibly at most twice when generating a message. Even if the number of encryptions allowed is equal to the number of nested encryptions appearing in the most complex message in the protocol, one can easily come up with a protocol for which the restriction prevents the model checker from finding an error. For example, consider the following pair of protocols:

1. $A \rightarrow B : \{\{\{x\}_{K_b}\}_{K_b}\}_{K_b}$
2. $B \rightarrow A : \{\{\{x\}_{K_b^{-1}}\}_{K_b^{-1}}\}_{K_b^{-1}}$

and

1. $A \rightarrow B : \{N_a\}_{K_b}$
2. $B \rightarrow A : \{N_a\}_{K_a}$

The first protocol is a somewhat contrived signature protocol where B signs messages on behalf of A. The second is a simple nonce challenge where A requires that B decrypt the nonce N_a using its private key. Neither of these protocols is intended to be general purpose. The idea here is that A and B are the only honest agents in the protocols and they always play the same roles. In other words, B is the only principal performing signatures and A is the only principal issuing nonce challenges. The adversary is still allowed to modify messages, but it is not allowed to participate in either protocol. The security requirement is that whenever A receives the correct response $\{N_a\}_{K_a}$ to its nonce challenge, B must have participated in the nonce challenge protocol. The most complex message has three nested encryptions. However, if we restrict the adversary to only four nested encryptions we will miss the following violating trace:

1. $A \rightarrow \Omega(B) : \{N_a\}_{K_b}$
2. $\Omega(A) \rightarrow B : \{\{\{\{\{\{N_a\}_{K_b}\}_{K_b}\}_{K_b}\}_{K_b}\}_{K_b}\}_{K_b}$
3. $B \rightarrow \Omega(A) : N_a$
4. $\Omega(B) \rightarrow A : \{N_a\}_{K_a}$

In this trace, A successfully completes the nonce challenge protocol, but B did not participate in the nonce challenge protocol. Principal B was participating in the signature protocol. This attack required the adversary to perform five nested encryptions in step 2 on the message $\{N_a\}_{K_b}$ it intercepted from A in step 1. Clearly, it is not safe to restrict arbitrarily the amount of encryption the adversary is allowed to perform when modifying messages. However, this kind of restriction may sometimes be necessary in order to be able to perform *some* analysis, even if not exhaustive.

4.2.2 Typed messages

A second way to restrict the set of messages generated by the adversary is to enforce a type on all messages received by honest agents. These types include the atomic types (nonces, keys, principal names, data) and compound types built up using encryption and concatenation (for example, $\{nonce, key\}_{key}$). Since the model contains only a finite number of atomic messages, there will only be a finite number of messages of any given type. When the adversary is trying to generate messages that an honest agent is willing to receive, the set of messages it can attempt will be restricted by the type associated with the message being received, thus making the set of messages finite.

While adding a type system may seem to be a much more reasonable restriction than the arbitrary bound on message complexity discussed earlier, it is not any safer. To see this, we now consider a protocol proposed by Woo and Lam [85]. The protocol is given below:

1. $A \to B : A$
2. $B \to A : N_b$
3. $A \to B : \{N_b\}_{K_{as}}$
4. $B \to S : \{A \cdot \{N_b\}_{K_{as}}\}_{K_{bs}}$
5. $S \to B : \{A \cdot N_b\}_{K_{bs}}$

This protocol can be attacked as follows:

1. $\Omega(A) \to B : A$
2. $B \to \Omega(A) : N_b$
3. $\Omega(A) \to B : N_b$
4. $B \to \Omega(S) : \{A \cdot N_b\}_{K_{bs}}$
5. $\Omega(S) \to B : \{A \cdot N_b\}_{K_{bs}}$

Note that attack requires that in step 3, B accept a nonce when it is expecting an encrypted nonce. If only messages that match the receiving type are allowed, then the adversary would not be allowed to send this message and this attack would not be found. This solution seems more satisfactory because one might be willing to believe that B can tell the difference between a nonce and an encrypted nonce although both would technically be random numbers from B's point of view. However, in general, this restriction would mean that receivers of encrypted messages could tell the types of components hidden "underneath" the encryption, even if the receiver had no way of checking because it does not possess the necessary key. For example, consider the following protocol:

1. $A \to S : \{A, B, \{A, B, K\}_{K_B}\}_{K_{AS}}$
2. $S \to B : \{A, B, K\}_{K_B}$

When the server S receives the first message, it cannot decrypt the component $\{A, B, K\}_{K_B}$ since it does not possess B's private key K_B^{-1}. Without knowledge of the new secret key K, it also cannot construct $\{A, B, K\}_{K_B}$ itself to compare against the received message. However, if we place the type restriction on received messages, we would in essence allow the server to reject a message of the form $\{A, B, X\}_{K_B}$ even though it could not really "look inside" to check that the message had components of the correct type. In other words, there is no way for the S to know that the X in $\{A, B, X\}_{K_B}$ is not a key, yet we are giving S this capability.

Although enforcing message types may seem reasonable, it is not safe. As was the case with the previous restriction on adversary capability, certain attacks may be lost when using this kind of restriction. However, some kind of restriction is often necessary. Without any restrictions, the model can be infinite and hence could not be analyzed using standard model checking.

4.3 Maintaining Information

While the solution to infinite branching is not completely satisfactory, the solution to the problem of maintaining an infinite set of messages known by the adversary is. Section 3.1 discussed how new messages can be generated from known messages using a set of derivation rules. In general the set of all known messages is infinite; however, the discussion that follows demonstrates that this set can be represented implicitly by a finite set of generators. The most obvious set of generators for the adversary is simply the set of all initially known messages together with all messages overheard during the execution of the protocol. However, there is still the question of how to decide whether a particular message m can be generated. In general, a derivation of m could be arbitrarily long. After finite computation one can only search a finite number of derivations, each with bounded length. At any moment in time how can we be sure that there is no derivation of m with a length longer than the ones tried so far? The solution to this problem is to restrict ourselves to *normalized derivations.*

4.3.1 Normalized derivations

A message that can be derived from some set of messages typically has an infinite number of derivations. We will prove that any message that can be derived from a set of messages also has a *normalized* derivation. This normalized derivation has a very specific structure for which Brutus can search. The existence of normalized derivations guarantees that if Brutus cannot build a derivation with this specific structure, then *no* derivation of that message exists and the message cannot be generated.

Before turning to the technical details and proofs, let us take an intuitive look at what is meant by a normalized derivation. Recall that the message derivation rules came in pairs. Encryption and decryption are inverses, as are pairing and projection. Borrowing terminology from [35], each pair has a shrinking rule that makes the messages smaller. The shrinking rules are decryption and projection. The corresponding expanding rules (encryption and pairing) yield more complex messages. A normalized derivation will be one in which all shrinking rules occur before all expanding rules. Intuitively this would mean that if one can derive a message from a set of known messages I, then there must be a derivation in which one first closes I under the shrinking rules to get a set of "generators". From this set of generators one then applies only expanding rules to generate the message in question. Note that this makes the question of whether or not the adversary can generate a message decidable. One first closes under shrinking rules. This is guaranteed to terminate because each simplifying rule yields a smaller message and this process must stop when atomic messages are reached. One then tries all combinations of expanding rules that yield a message of the desired size. Since each application of an expanding rule increases the length of the message, there is a maximum number M of expanding rules that can be applied without exceeding the size of the message in question. If, after applying all possible combinations of M expanding rules, the message has not been generated, then the message does not have a normalized derivation. If every derivable message has a normalized derivation, then any message that does not have a normalized derivation cannot be generated at all.

In practice one does not generate all messages of a specified size. The actual algorithm used by Brutus tries to generate the message by recursively trying to generate components until all the messages that need to be generated are in the set of generators or until an atomic message that is not in the set of generators is reached. In the later case, the message cannot be generated. The details can be found in Section 4.3.2 and in [12].

Most other model checkers used to verify security protocols use a similar idea and this result applies pretty directly to their work as well. The following proof also demonstrates why the simplifying assumption that all keys are atomic is so useful, and why this assumption is so common in the literature. Also, there is much similarity between this method and the use of shrinking rules and expanding rules by Kindred and Wing [34, 35], despite the fact that they are using theorem proving and theory generation instead of model checking. The high level idea is the same; however, they apply it to a more complicated set of proof rules. Because I use a more restricted set of rules, I am able to use results from natural deduction.

$$\frac{m_1 \qquad\qquad m_2}{m_1 \cdot m_2}\cdot\mathcal{I} \qquad\qquad \frac{m_1 \cdot m_2}{m_1}\cdot\mathcal{E}_l \qquad \frac{m_1 \cdot m_2}{m_2}\cdot\mathcal{E}_r$$

$$\frac{m \qquad\qquad k}{\{m\}_k}\{\}_k\mathcal{I} \qquad\qquad \frac{\{m\}_k \qquad\qquad k^{-1}}{m}\{\}_k\mathcal{E}$$

Figure 4.2: Derivation rules for messages

I will assume that the reader is familiar with derivation trees and so I will not formalize them here. We will say that a particular message m is derivable from a set of information I, if there exists a valid derivation tree using the inference rules in Figure 4.2, such that m appears at the bottom of the tree and all messages appearing at the top of the tree are contained in I. An example of such a tree can be found in Figure 4.3.

Note that in general there is more than one derivation tree for some message m. While all derivations trees are finite, their sizes are unbounded. In order to prove the decidability of checking $I \vdash m$ we would like to show that there is always a *normalized derivation* of bounded size for which one can search. Unlike normalized derivations in natural deduction, this normalized derivation will have the additional property that

$$\dfrac{\dfrac{\dfrac{\{a\}_k \cdot b}{\{a\}_k}\cdot\mathcal{E}_l \qquad k^{-1}}{a}\{\}_k\mathcal{E} \qquad \dfrac{\{a\}_k \cdot b}{b}\cdot\mathcal{E}_r}{a \cdot b}\cdot\mathcal{I}$$

Figure 4.3: A derivation tree for $\{\{a\}_k \cdot b, k^{-1}\} \vdash a \cdot b$

all elimination rules $(\cdot\mathcal{E}_l, \cdot\mathcal{E}_r, \{\}_k\mathcal{E})$ appear above all introduction rules $(\cdot\mathcal{I}, \{\}_k\mathcal{I})$. A similar idea for placing bounds on the lengths of derivations can be found in [5, 35, 40, 62]. However, this framework allows a straightforward translation into a proof search algorithm and explains the reasons behind certain assumptions made about the message space, in particular, the perfect encryption assumption and the atomic key assumption.

Each message construction operation (pairing and encryption) is characterized by a pair of inference rules. One is an introduction rule that creates a new message whose principal connective is that operation. For example the $\{\}_k\mathcal{I}$ rule creates a new encrypted message $\{m\}_k$ from the message m and the key k. The second is an elimination rule that removes that particular operation from the compound message assuming it is the principal (outermost) connective. For example the $\cdot\mathcal{E}_l$ rule takes a message $m_1 \cdot m_2$ and returns its left component m_1. The intuition behind normalized derivations is that an instance of an elimination rule appearing immediately below an instance of the corresponding introduction rule gains no new information. Therefore, such a derivation is transformed into a smaller derivation in which this redundant step is eliminated.

Using terminology similar to that found in [68], we will call the key k in an instance of the inference rule $\{\}_k\mathcal{E}$ a *minor premise*. Any other premise is a *major premise*. A message that appears in a derivation tree T as the conclusion of an introduction rule and as a major premise of an elimination rule is a *maximum message*. A derivation tree is a *normalized derivation* if it contains no maximum message. We now show that any derivation tree T can be transformed into a normalized derivation tree T' by eliminating maximum messages one at a time.

Let T be a derivation tree that is not atomic, and let M be a maximum message in T. Then T', the reduction of T at M, is constructed from T

·-reduction $(i = 1, 2)$

$$\begin{array}{c} \dfrac{\dfrac{\dfrac{\Sigma_1}{m_1} \qquad \dfrac{\Sigma_2}{m_2}}{m_1 \cdot m_2}}{m_i} \\ \Pi \end{array} \qquad \Rightarrow \qquad \begin{array}{c} \dfrac{\Sigma_i}{m_i} \\ \Pi \end{array}$$

$\{\}_k$-reduction

$$\begin{array}{c} \dfrac{\dfrac{\dfrac{\Sigma_1}{m} \qquad \dfrac{\Sigma_2}{k}}{\{m\}_k} \qquad \dfrac{\Sigma_3}{k^{-1}}}{m} \\ \Pi \end{array} \qquad \Rightarrow \qquad \begin{array}{c} \dfrac{\Sigma_1}{m} \\ \Pi \end{array}$$

Figure 4.4: Reduction rules for derivation trees

using one of the rules in Figure 4.4, depending on the form of M. In the diagrams, Π is what would remain of T after removing M and everything above it, while Σ_1, Σ_2, and Σ_3 represent sequences of derivation trees.

Theorem 4.3.1 *Any derivation tree T for m depending on assumptions in A can be transformed into a normalized derivation tree T' for m depending on the same assumptions in A.*

Proof: The proof is by induction on the number of maximum messages. If T has no maximum messages then it is already normalized and there is nothing to do. Otherwise, take any maximum message M. Because of the perfect encryption assumption, M cannot be the conclusion of an introduction rule for one operator and a major premise in the elimination rule for the other operator. Therefore, one of the reduction rules applies to M. After applying the appropriate reduction rule, one maximum message is removed and no new maximum messages are introduced. The result is a derivation tree for m that depends on the same assumptions and which

has one less maximum message. By the induction hypothesis this new derivation tree can be properly transformed.
□

In fact, the structure of these derivations trees is even more restricted as the following theorem demonstrates.

Theorem 4.3.2 *No introduction rule appears above an elimination rule in a normalized derivation tree.*

Proof: A derivation tree is normalized if no message appears as the conclusion of an introduction rule and a major premise of an elimination rule. Therefore, only minor premises need to be considered. The only minor premises are keys. Recall that keys are restricted to be atomic; therefore, no key can appear as the conclusion of an introduction rule. Hence no message can appear as the conclusion of an introduction rule and a premise of an elimination rule. It follows that no introduction rule appears above an elimination rule.
□

4.3.2 Information algorithms

Theorem 4.3.2 suggests an efficient algorithm for determining if $I \vdash m$. Since all elimination rules appear above all introduction rules in a normalized derivation, one can first construct I^*, the closure of the initial set of assumptions I under all elimination rules. A backwards search for a derivation of m from I^* using only introduction rules is then performed. (The notation $I^* \vdash_{\mathcal{I}} m$ is used to denote that such a normalized derivation tree exists.) We will now prove termination and correctness of this algorithm.

Theorem 4.3.3 $I \vdash m$ *iff* $I^* \vdash_{\mathcal{I}} m$ *(Correctness)*

Proof: ($\Rightarrow$) Consider the case when $I \vdash m$. Let T be a normalized derivation tree for m from I. By removing all elimination rules, we get a new tree T' for m from $I \cup \Delta$, where Δ is the set of all the messages appearing at the top of T' that are not in I. So T' is a derivation tree for $I \cup \Delta \vdash_{\mathcal{I}} m$. From the original tree T, each $\delta_i \in \Delta$ can be derived from I using only elimination rules, so $I \cup \Delta \subseteq I^*$. Therefore T' is also a derivation tree for $I^* \vdash_{\mathcal{I}} m$.

($\Leftarrow$) Consider the case when $I^* \vdash_{\mathcal{I}} m$. By definition, $I \vdash i$ for each $i \in I^*$. Therefore, we can transform the tree T' for $I^* \vdash_{\mathcal{I}} m$ into a tree T for $I \vdash m$ by placing a derivation tree T_i for $I \vdash i$ above each message $i \notin I$ at the top of T'.
$\square$

Theorem 4.3.3 proves the correctness of the algorithm. To prove termination, consider all the messages that are generated during the derivation. The elimination rules start from the assumptions and generate only submessages. Since the original messages are finite length, and since there are only a finite number of them, the process of closing under elimination rules terminates. Now consider the backwards search using introduction rules. Since these rules are applied backwards, the new major premise messages that become subgoals for the search must be proper submessages of the message being searched for. In other words, each subgoal is smaller then the goal that generates it. Therefore, this part of the search terminates when either a message in I^* is reached or an atomic message that is not in I^* is reached. The minor premises are all atomic keys. Therefore, any search for minor premises also involves a simple scan of I^* as well. Therefore, the entire algorithm terminates.

The implementation of this algorithm is given in the following two figures. Figure 4.5 shows how to update the adversary's set of information when it learns a new message. Any time the adversary gains a new message, it is added to the set of messages the adversary currently has and the new set is closed under elimination rules and redundant messages are removed. (A message in a set I is redundant if it can be generated from the other messages in I using only introduction rules.) Figure 4.6 describes how to search for a derivation of m from I^* using only introduction rules. This search involves first checking if $m \in I^*$. If this fails, then it recursively searches for the components of m.

4.4 Partial Order

As is the case with model checking in general, the use of model checking to analyze and verify security protocols is severely limited by the *state ex-*

```
funct add(I, m)
   foreach i ∈ I do
      if i = {x}_y ∧ m = y^{-1}
         then I := add(I, x)
                 if y ∈ I then I := I − i fi
      fi
   od
   if m ∈ 𝒜
      then return I ∪ {m}
   elsif m = x · y
         then return add(add(I, x), y)
   elsif m = {x}_y ∧ y^{-1} ∈ I
         then if y ∈ I
                    then return add(I, x)
                     else return add(I ∪ {m}, x)
                 fi
   else
         return I ∪ {m}
   fi
```

Figure 4.5: Augmenting the adversary's knowledge

```
funct in(I, m)
  if m ∈ I
    then return true
  elsif m = x · y
      then return in(I, x) ∧ in(I, y)
  elsif m = {x}_y
      then return in(I, x) ∧ in(I, y)
  else
      return false
  fi
```

Figure 4.6: Searching the adversary's knowledge

plosion problem. Intuitively, the problem is that small increases in the size of the model (for example adding another instance or another principal) often result in very large increases in the size of the state space. Typically, the state space grows exponentially with respect to the number of components in the model. Since Brutus essentially performs an exhaustive search, this can greatly restrict the models that can be analyzed.

One source of the state space explosion is the *interleaving semantics* given to the composition of two or more processes. Interleaving semantics models the behavior of a system by considering all possible execution orderings of the enabled actions. This is often pictured in the literature using a diamond as in Figure 4.7. The state at the top of the figure is one where two actions, a and b, are enabled. From that state one can perform the action sequence ab or the action sequence ba and end up in the state at the bottom of the figure.

While all interleavings must be considered when analyzing the behavior of a system, it may not be necessary to *explore* all interleavings when performing the model checking. This has been the idea behind a great body of research in what is called *partial order theory* [22, 66, 80]. The idea is to reduce the number of interleavings that are explored. One way to do this is to expand the states by taking actions from some *ample* set of actions. Since the ample set of actions is a subset of the set of all enabled actions only a subset of all the possible traces is generated or explored. The partial order theory helps us to choose an ample set in such a way that for every trace that is discarded, there is a trace that is considered

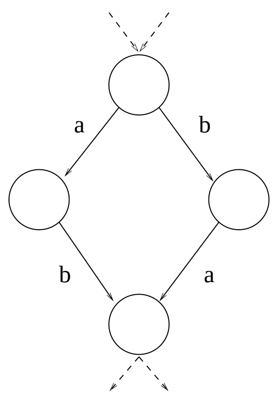

Figure 4.7: A state with two enabled actions

which agrees with the discarded trace on the specification.

This theory is discussed in more detail in Chapter 6. Specifically, I discuss how the general theory is simplified in the case of my models, and also how I have generalized the theory to handle models that deal with a notion of knowledge. For now, the discussion concentrates on how this partial order reduction is implemented in Brutus and the proof of correctness is postponed until Chapter 6.

Recall the definition of the depth first search performed by Brutus. The algorithm is duplicated in Figure 4.8. In this straightforward implementation, the function *expand* generates all next states that result from taking one of the actions enabled in the current state. I implemented the partial order reduction by modifying the definition of the *expand* function so that it does not necessarily expand all the enabled actions in a state. The new *expand* function is given in Figure 4.9. In the algorithm, if α is an action enabled in state s, then $\alpha(s)$ is the state resulting from taking action α in state s. Also, the function *enabled* returns the set of actions that are enabled in state s. Intuitively, the algorithm works as follows:

1. If there is an internal action enabled that does not appear in the specification, then expand using only "the first" such action.

2. Otherwise, if there is a send action enabled then expand using only "the first" send action.

3. Otherwise, expand all enabled actions to generate all possible next states.

```
proc DFS(s, φ)
  push(S, s)
  while (not empty(S)) do
        s_cur = pop(S)
        if sat(s_cur, φ)
          then L = expand(s_cur)
                foreach s_next ∈ L do
                  push(S, s_next)
                od
           else counter-example(s_cur, φ)
        fi
  od
```

Figure 4.8: Search algorithm

```
funct expand(s, φ)
  foreach α ∈ enabled(s) do
    if (α = INTERNAL) ∧ (¬occurs(α, φ))
      then return {α(s)}
    fi
  od
  foreach α ∈ enabled(s) do
    if α = SEND
      then return {α(s)}
    fi
  od
  ample = {}
  foreach α ∈ enabled(s) do
    ample = ample ∪ {α(s)}
  od
  return ample
```

Figure 4.9: Partial Order Expand Function

The words "the first" appear in quotes because the algorithm, breaks out of the foreach loop after encountering the first such action. However, there is no ordering on the actions. The order in which the actions are considered is arbitrary.

4.5 Symmetry

There is another source of state space explosion which has generated quite a bit of research. The source of the explosion is the replicated components that often appear in models. In the realm of hardware verification, one of the most obvious replicated components is memory. In security protocols verification, the replicated components appear for two reasons. Multiple principals can participate in the protocol; however they can play the **same** role and execute the **same** actions. Secondly, one principal may participate multiple times in a protocol. Since Brutus models each session or execution of the protocol with a different *instance*, the model will have replicated instances.

If a model contains replicated components, then the model possesses symmetries. If the components are indeed identical, one should be able to swap the roles of the components without changing the behaviors of the system. In the case of a hardware example, one should be able to swap the roles of two lines of memory or the roles of two general purpose registers without changing the possible behaviors of the system. In the case of security protocols, one should be able to swap principals that take on the same role in the protocol as well as the different attempts of the same principal to participate in the protocol.

A number of researchers have studied the possibility of exploiting symmetry to reduce the size of the state space that must be explored during model checking [9, 21, 28, 29, 30]. The reductions employed by Brutus follow from these general theories; however, Brutus does not require the heavy machinery used by more general model checkers that exploit symmetry. The reductions are proven correct in Chapter 5. The present discussion is restricted to a description of the implementation of the algorithm.

4.5.1 Principal symmetry

Currently, principal symmetry (the symmetry that arises because *different* principals play the same role) is not exploited by BRUTUS because the number of instances of any principal are fixed for any model. It is up to the user to realize that a model in which customer A can participate once and customer B can participate twice should be equivalent to a similar model where customer A can participate twice and customer B can only participate once. For this reason, the user only models one of these two possibilities. However, it may be useful to have a more general kind of model, one in which we allow a total of three customer instances which can be divided among A and B in any of four possible ways. One can see that two of these ways are redundant. For example, the case where A participates three times and B does not participate is equivalent to the case where B participates three times and A does not participate. One need only swap the names of A and B. The same holds for the two cases where one principal participates twice while the other participates once.

One can avoid these redundant cases by implementing a model generator for the current model checker. This model generator takes the names of the principals, what roles they can play in the protocol, and the total number of instances allowed for each role in the model. The model generator then generates a set of *configurations*. Each configuration is a set of functions, one for each role. Each such function maps a principal name to the number of times that principal can participate in the protocol in the given role. In our previous example then, there would be a customer function that maps A to the number of times A can participate as a customer and B to the number of times B can participate as a customer. The sum over all principal names should be the total number of instances allowed for that role. This then specifies a model which can currently be checked by BRUTUS. The symmetry reduction in this case involves making sure that symmetric configurations are avoided. If $\mathcal{C}$ is a particular configuration and π is a permutation on the principal names, then $\pi(\mathcal{C}) = \{\, f \circ \pi \mid f \in \mathcal{C} \,\}$. Using this definition then, the configuration where customer A can participate twice and customer B participates once is symmetric to the configuration where B participates twice and A once via the permutation that swaps A and B.

Because this feature has not been implemented, I do not go into more detail here, but I do prove the correctness of this reduction in Section 5.2.

4.5.2 Instance symmetry

The second source of symmetry arises because a principal is allowed to participate multiple times in a protocol. For example, imagine any protocol that has two parties, an initiator who sends the first message (A) and a responder who receives the first message (B). In the model of the protocol, one would like to allow the possibility of multiple executions of the protocol by the same parties. Let us assume then that the model has three instances for each participant ($A1, A2, A3, B1, B2,$ and $B3$) so that each A and B can attempt to execute the protocol three times. In the start state, the Ai are symmetric and the Bi are also symmetric. They are identical up to their names (instance IDs). At this point in time, anything $A1$ can do, $A2$ or $A3$ can do. The same holds for the Bi. Intuitively, if we assign IDs to the different instances in the model, then the initial state is symmetric with respect to these instance IDs. The instances are equivalent because they are instances of the same principal playing the same role. Two instances that either belong to different principals or are playing different roles in the protocol are not equivalent. Permutations that only swap equivalent start state instances are called *safe permutations*.

These safe permutations in turn allow us to permute traces. Safe permutations only permute instances that are equivalent. In others words, instances that can do the same things. This means that while the instances may only be equivalent in the start state, one can still permute an entire trace because whatever one instance can do throughout a trace, an equivalent instance could do in a permuted trace. For example, consider Figure 4.10. The figure on the left represents a trace in which $B1$ successfully responded to a protocol session with $A1$ and similarly for the pairs $(A2, B2)$ and $(A3, B3)$. The figure on the right represents a trace in which $B2$ successfully responded to $A1$ and similarly for the pairs $(A2, B3)$ and $(A3, B1)$. The traces are symmetric as are the graphs. This is demonstrated in the figure on the right. By permuting the labels on the vertices (the instance IDs in the model) as suggested by the cycling arrows, we arrive at the original graph (trace).

This symmetry induces an equivalence relation on the set of all traces of a model. Two traces are equivalent if there is a safe permutation that maps one to the other. Since the requirements for the models shouldn't depend on the instance IDs, the requirements should be insensitive to these permutations. When this is the case, one can restrict the exhaustive

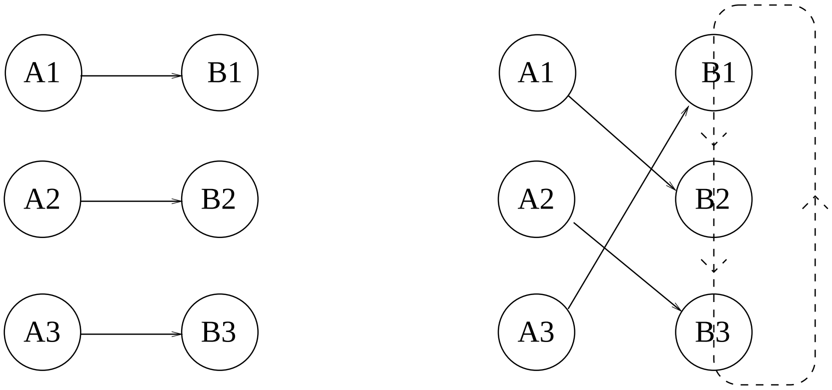

Figure 4.10: Exploiting Symmetry

search to include only a single representative trace from each equivalence class. The technical details as well as the proof of correctness is the topic of Section 5.3.

A difficulty that arises in general symmetry reduction is how to detect symmetries. Computing symmetries can be very difficult and costly. My solution is to demonstrate and exploit a few symmetries that are guaranteed to exist in any model in BRUTUS. In particular, there is a symmetry that always exists at the point in the protocol when the initiator instances have all sent their first message and the responder instances are ready to receive the first message. Because of the partial order reduction, all initiator instances send their first message before any responder instances receive any messages. Therefore, this state exists in *any* trace considered by BRUTUS. For the sake of illustration, we will consider a concrete example.

Assume that the model has one initiator A, that can participate twice and one responder B that can participate twice. The initiator is simply the principal that sends the first message and the responder receives the first message. Consider the point when B is ready to receive its first message. There is nothing in the execution so far that distinguishes between the two instances of B. Any trace that results from $B1$ receiving that first message has a symmetric trace in the model where $B2$ behaves as $B1$ and receives that first message. So at this point BRUTUS arbitrarily chooses $B1$ to receive the first message and ignores the case where $B2$ receives the first message. Because of the partial order reduction, both $A1$ and $A2$ have sent their first messages, so $B1$ could receive either one. So this state is also symmetric with respect to $A1$ and $A2$ (since no one but the adversary has received these messages and it has received both). Any trace that results from $B1$ receiving $A1$'s message has a symmetric trace where $A2$ behaves as $A1$ and $B1$ receives $A2$'s message instead. So at this point, BRUTUS arbitrarily chooses to have $B1$ receive $A1$'s message and ignores the case where $B1$ receives $A2$'s message. Note, however, that now the symmetry is broken. $A1$ can be distinguished from $A2$ because someone has received $A1$'s message while no one has received $A2$'s message. Similarly, $B1$ can be distinguished from $B2$ because $B1$ has received its first message while $B2$ has not. However, if there were a third instance of each principal ($A3$ and $B3$), then $A2$ and $A3$ would still be equivalent and $B2$ and $B3$ would also be equivalent. There would still be symmetric traces from this point. In general, any instances that are equivalent in the start state remain equivalent until they receive the first message (if they are a responder) or until some honest instance receives their first message (if they are an initiator).

The algorithm for performing the symmetry reduction can be found in Figure 4.11. The partial order reduction search algorithm is modified by filtering out any symmetric next states. However, to determine if two next states are symmetric, BRUTUS needs to maintain information about equivalence classes of instances. Therefore, the global state is augmented to include this information. BRUTUS also needs to keep track of the origin of any messages sent during the execution of the protocol.

BRUTUS keeps track of who sent what messages by maintaining a *get_origin* function. One can think of this function as an associative array that is used to augment the global state. Whenever an honest instance H_i performs a **send** action, BRUTUS records this with *set_origin*(s, m, H_i). In state s as well as in all successor states to s, the message m is tagged with its origin H_i. Because this information is maintained by augmenting the information in the next global state, this association is "forgotten" when BRUTUS backtracks to the previous state.

BRUTUS also needs to keep track of the set of instances that are symmetric in the current state. Recall that in the discussion above, two instances playing the same role on behalf of the same principal (for example they are both instances of principal A trying to initiate the protocol) are considered symmetric in the initial state. As the protocol executes, the symmetry between responders is broken when one of them receives its

```
funct expand(s, φ)
   foreach α ∈ enabled(s) do
      if (α = INTERNAL) ∧ (¬occurs(α, φ))
         then return {α(s)}
      fi
   od
   foreach α ∈ enabled(s) do
      if (α = i send m)
         then s' := α(s)
                 set_origin(s', m, i)
                 return {s'}
      fi
   od
   ample := {}
   foreach α ∈ enabled(s) do
      if (α = i receive m) ∧
         (i = rep(i)) ∧
         (get_origin(s, m) = rep(get_origin(s, m)))
         then
                 s' := α(s)
                 remove(s', i)
                 remove(s', get_origin(s, m))
                 ample = ample ∪ {s'}
      fi
   od
   return ample
```

Figure 4.11: Symmetry Expand Function

first message while the symmetry between initiators is broken when anyone receives one of their messages. This observation is used to maintain the set of equivalence classes. In the initial state, all instances executing the same role for the same principal are placed in the same equivalence class. As the protocol executes, an initiator instance is removed from its equivalence class when any honest agent receives that initiator's first message. Similarly, a responder instance is removed from its equivalence class when it performs its first **receive** action. The equivalence classes are implemented in BRUTUS via two functions. The $remove(s, H_i)$ function removes an instance H_i from its current equivalence class and places it in its own equivalence class in the augmented global state s. The function $rep(s, H_i)$ returns the representative of the equivalence class to which H_i belongs in the state s.

All this machinery is present in the symmetry reduction algorithm in Figure 4.11. The algorithm uses the same *expand* function that was used for the partial-order reduction. If there are any invisible internal actions one of those actions is expanded and the resulting singleton set is returned. Otherwise, if there are any **send** actions, one of those **send** actions is expanded (making sure to keep track of the origin of the sent message) and the resulting singleton set is returned. Otherwise a set of states that result from multiple **receive** actions is returned. In general, this set may lead to symmetric traces. Therefore, the set of next states is reduced by filtering out symmetric actions. Two actions are symmetric if all of the following hold:

1. They are both **receive** actions.
2. The instances performing the actions are symmetric. (They belong to the same equivalence class.)
3. The messages being received originated from symmetric instances. (The sending instances belong to the same equivalence class.)

This check is straightforward using the machinery for maintaining equivalence classes and message origins discussed above.

The algorithm never actually compares two different actions to see if they are symmetric. This is because there is a specific action that is constructed from each set of equivalent actions. This representative action is the one where the representative from the equivalence class of receiver instances receives a message that originated from the representative of the equivalence class of sender instances. Any **receive** action that involves instances that are symmetric to these representatives will be symmetric to this particular **receive** action. Since this representative action is equivalent to the other actions, BRUTUS can safely ignore the equivalent actions that would lead to symmetric traces.

Chapter 5

Symmetry

Researchers have successfully exploited the symmetry inherent in hardware containing replicated components to reduce the sizes of the models analyzed when performing verification [9, 21, 28, 29]. By arguing that some configuration is identical to a second configuration up to a change in the names of the components, one can safely analyze only one of the configurations, because the second will have an error only if the first does as well. In the analysis of security protocols using Brutus, the models have a specific number of participants. These participants, in turn, attempt some fixed number of sessions. Like the hardware examples, these models consist of a number of replicated components which generate a large number of symmetries that can be exploited.

In what follows, a permutation p will be defined on principal names or on instance IDs. In either case, the permutation p will be extended to a permutation on messages, $p_{\mathcal{M}}$. From $p_{\mathcal{M}}$, I will then define permutations on actions, states, traces, and formulas by simply applying $p_{\mathcal{M}}$ to any occurrence of atomic messages in these structures and leaving the rest unchanged. For example, let A and B be principal names and $p = (AB)$ be a permutation on these names. Then $p(A) = B$ and $p(A\ \underline{\textbf{send}}\ \{A \cdot B\}_{K_B}) = B\ \underline{\textbf{send}}\ \{B \cdot A\}_{K_A}$.

Typically, protocol models exhibit two kinds of symmetries that can be exploited. The first is a kind of top level symmetry that results from a renaming of the honest agents. For example, instead of having A initiate a session with B, we might have B initiate a session with A. The second kind of symmetry results from the fact that each agent may participate in multiple sessions. This is modeled with multiple instances of the same principal. These instances are equivalent in the sense that any sequence of actions performed by one instance could have been performed by another instance instead. Since the top level symmetry is easier to understand, it is discussed first. The proofs for the second kind of symmetry can then borrow some of the structure and ideas from the first kind. I begin by reviewing some of the properties of groups, permutations, and symmetries.

5.1 Groups and Permutations

I begin with a brief review of some basic terminology and properties of groups and permutations. Much of this information comes from Hungerford's text on modern algebra [26].

Theorem 5.1.1 *Let H be a nonempty finite subset of a group G. If H is closed under the operation in G, then H is a subgroup of G.*

Proof: Since H inherits the associative operator from G and since H is assumed to be closed, all that remains is to show that H has inverses. Once we show that $a \in H$ has an inverse $a^{-1} \in H$, then we know $aa^{-1} = 1_G = 1_H \in H$.

To show that every $a \in H$ has an inverse $a^{-1} \in H$ consider a^k for all positive integers k. Since H is finite and each $a^k \in H$, it must be the case that $a^i = a^j$ for some $0 < i < j$. Now choose n so that $j = i + n$. Then $a^i = a^i a^n$. Since $a^i \in G$, it has an inverse x. Multiplying on the left by x we get $xa^i = xa^i a^n$ or $1_H = a^n$. Which means that $1_H = aa^{n-1}$ and we have that $a^{-1} = a^{n-1}$.
□

Definition 5.1.1 *Let G be a group and S be a set. We say that G* **acts** *on S if there is a map $(g, s) \mapsto g(s)$ such that*

- $g(s) \in S$;
- $1_G(s) = s$; *and*
- $(g_1 g_2)(s) = g_1(g_2(s))$.

Example 5.1.1 *Let $\Sigma = \{a, b, c\}$ and consider Σ^*, the set of strings over Σ. Let G be the group of permutations on $\{a, b, c\}$. Then G acts on Σ^* as follows. Let $\sigma_1\sigma_2\cdots\sigma_n \in \Sigma^*$ and $g \in G$. Then $g(\sigma_1\sigma_2\cdots\sigma_n) = g(\sigma_1)g(\sigma_2)\cdots g(\sigma_n)$. For example,*

$$\begin{pmatrix} a & b & c \\ b & a & c \end{pmatrix}(acbac) = bcabc.$$

Note that the properties required in definition 5.1.1 are satisfied.

- $$g(\sigma_1\sigma_2\cdots\sigma_n) = g(\sigma_1)g(\sigma_2)\cdots g(\sigma_n) \in \Sigma^*$$

- $$\begin{aligned} 1_G(\sigma_1\sigma_2\cdots\sigma_n) &= 1_G(\sigma_1)1_G(\sigma_2)\cdots 1_G(\sigma_n) \\ &= \sigma_1\sigma_2\cdots\sigma_n \end{aligned}$$

- $$\begin{aligned} (g_1g_2)(\sigma_1\sigma_2\cdots,\sigma_n) &= (g_1g_2)(\sigma_1)(g_1g_2)(\sigma_2)\cdots,(g_1g_2)(\sigma_n) \\ &= g_1(g_2(\sigma_1))g_1(g_2(\sigma_2))\cdots,g_1(g_2(\sigma_n)) \\ &= g_1(g_2(\sigma_1)g_2(\sigma_2)\cdots,g_2(\sigma_n) \\ &= g_1(g_2(\sigma_1\sigma_2\cdots,\sigma_n)) \end{aligned}$$

Definition 5.1.2 *A group G acting on a set S induces a relation $\equiv_G$ on the elements of the set S. This relation is defined as follows:*

$$a \equiv_G b \text{ iff } b = g(a) \text{ for some } g \in G.$$

Theorem 5.1.2 *Let G be a group acting on a set S and let $\equiv_G$ be the relation induced by G on S. Then $\equiv_G$ is an equivalence relation and partitions S into equivalence classes.*

Proof:

- Reflexivity: G contains the identity 1_G and $a = 1_G(a)$; therefore, $a \equiv_G a$.

- Symmetry: Let $a \equiv_G b$. Then $b = g(a)$ for some $g \in G$. Since G is a group, g has an inverse $g^{-1} \in G$, and
$$g^{-1}(b) = g^{-1}(g(a)) = (g^{-1}g)a = 1_G(a) = a.$$
Therefore, $b \equiv_G a$.
- Transitivity: Let $a \equiv_G b$ and $b \equiv_G c$. Then there exist $g_1, g_2 \in G$ such that $b = g_1(a)$ and $c = g_2(b)$. Therefore,
$$c = g_2(g_1(a)) = (g_2 g_1)(a).$$
Since G is closed, $g_2 g_1 \in G$ and $a \equiv_G c$.

□

Since permutations are groups that can act on sets, they can be used to partition sets into equivalence classes. These equivalence classes are called *orbits*. The orbit of an element a is denoted $G(a)$.

Definition 5.1.3 *The term* **symmetry** *is used to denote a permutation on the components of some structure that results in an isomorphic structure.*

Example 5.1.2 *Let $G = (V, E)$ be a graph with n vertices labeled 1 to n. Let p be a permutation on the set $\{1, 2, \cdots n\}$. Then p is a symmetry of G if the graph G' derived from G by applying the relabeling p to the vertices is isomorphic to G. More formally, the permutation p on $\{1, 2, \cdots n\}$ induces a permutation, p^*, on graphs. So p is a symmetry on G when $p^*(G) \cong G$.*

Example 5.1.3 *Let $S \subseteq \Sigma^*$ be some set of strings over Σ. Let p be a permutation over Σ. We extend p to strings as in Example 5.1.1. Let $p(S) = \{\, p(s) \mid s \in S \,\}$. Then p is a symmetry of S if $p(S) = S$. For example, consider*
$$\Sigma = \{a, b, c\}, \quad S = \{abbca, cca, baacb, ccb\}, \text{ and}$$
$$G = \left\{ \iota = \begin{pmatrix} a & b & c \\ a & b & c \end{pmatrix}, p = \begin{pmatrix} a & b & c \\ b & a & c \end{pmatrix} \right\}.$$
Then $p(S) = S$. Note that G also partitions S into two equivalence classes:
$$G(abbca) = G(baacb) = \{abbca, baacb\}, \quad \text{and}$$
$$G(cca) = G(ccb) = \{cca, ccb\}.$$

This general theory can be applied to the kinds of models analyzed with Brutus. Permutations on the names of the honest agents, as well as permutations on the different instance IDs of different sessions belonging to the same principal, will induce permutations on messages, states, and traces in the model. In particular the symmetries exhibited on the sets of traces of the model partition the set of traces into equivalence classes. Traces in the same equivalence class will agree on all symmetric specifications.

5.2 Symmetry on Principals

First, let us consider the symmetries present in a model when two different principals play the same role in a protocol. Let us assume we are analyzing a protocol that has two roles, initiator and responder. Let us further assume that we are interested in bounding the problem by looking at models with at most three initiator instances and three responder instances. We will also limit ourselves to three different honest principals, call them A, B, and C. There are many different "configurations" that we could use and still stay within these parameters. For example, we could allow each principal to have one initiator instance and one responder instance. We could instead allow A to have 3 initiator instances, B to have two responder instances and C to have one responder instance. I will call each of these possibilities, a "configuration." The set of all possible configurations will usually be symmetric in the names of the honest agents. If the specification is also symmetric in the names of the honest agents, then it may only be necessary to analyze some subset of all the possible configurations. I formalize this intuition below.

5.2.1 Configuration Symmetries

Let $\mathcal{P}$ be the set of names of the honest agents. Each principal can play the role of either the initiator or the responder in a protocol. We will limit ourselves to these two roles in the discussion that follows to make it simpler; in general, there may be many roles and multiple protocols. Recall that the initiator is the sender of the first message and the responder is the receiver of the first message. Each initiator can try to initiate a protocol run with any other principal, including the adversary. Each responder can respond to any other principal, including the adversary. Each such assignment of instances to roles played by principals is called a *configuration*

Definition 5.2.1 *A configuration is a set of functions $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots\}$, one for each role in the protocol. Each function $\mathcal{C}_k : \mathcal{P} \rightarrow \mathbb{N}$ is map from the name of an honest principal to the number of times that principal plays that particular role in the model.*

For the sake of simplicity, I will consider protocols with only the roles of initiator and responder (and sometimes server); I will denote the corresponding functions $\mathcal{C}_i, \mathcal{C}_r$, and $\mathcal{C}_s$.

Theorem 5.2.1 *Assuming there are m honest agents, i initiator instances and r responder instances, there are $\binom{m+i-1}{i}\binom{m+r-1}{r}$ different configurations.*

Proof: We must choose i initiator instances from the m honest agents (with repetition). According to Lemma 5.2.2 proven below, there are $\binom{m+i-1}{i}$ ways to do this. Similarly, there are $\binom{m+r-1}{r}$ ways to choose r responder instances from the m honest agents. Since these choices are independent, the total number of ways of choosing both is the product $\binom{m+i-1}{i}\binom{m+r-1}{r}$.
□

Lemma 5.2.2 *The number of selections of n items (with repetition) from r different types of items is $\binom{n+r-1}{n}$.*

Proof: Consider an arbitrary sequence of n "x" marks and $r-1$ "/" marks. The "/" marks divide the "x"s into r groups. The "x"s before the first "/" signify the number of items of the first type. The "x"s between the first "/" and the second "/" signify the number of items of the second type. The "x"s after the last "/" signify the number of items of the r^{th} type. Notice that for each such sequence there is exactly one selection of n items from r different types and for each selection there is exactly one sequence of n "x"s and $r-1$ "/"s. So there is a one-to-one correspondence between the sequences and selections. So now we need to count the number of sequences. Notice that each sequence has $n+r-1$ elements and once the n "x"s (or the $r-1$ "/"s) are placed, the rest of

the sequence is determined. The number of such sequences is the number of ways of choosing the n "slots" in which to place the "x"s. The number of ways of choosing n slots from $n+r-1$ slots is $\binom{n+r-1}{n}$.
$\square$

A great number of these configurations are symmetric with respect to permuting the names of the honest agents. For example, the configuration

$$\{\mathcal{C}_i = \{(A,3),(B,0),(C,0)\}, \mathcal{C}_r = \{(A,0),(B,2),(C,1)\}\}$$

is symmetric to the configuration

$$\{\mathcal{C}_i = \{(A,0),(B,3),(C,0)\}, \mathcal{C}_r = \{(A,1),(B,0),(C,2)\}\}.$$

In general a permutation p on principal names will induce a permutation on the set of configurations by composing p with each of the configuration functions as follows:

$$p_{\mathcal{C}}(\{\mathcal{C}_i, \mathcal{C}_r\}) = \{\mathcal{C}_i \circ p, \mathcal{C}_r \circ p\}.$$

These permutations then correspond to the symmetries in the set of all configurations. In other words, as in Theorem 5.1.2, $\mathcal{S}_{\mathcal{P}}$, the group of permutations on the set of honest agents $\mathcal{P}$ will partition the set of configurations into equivalence classes. Continuing the example,

$$\begin{gathered}\{\mathcal{C}_i = \{(A,3),(B,0),(C,0)\}, \mathcal{C}_r = \{(A,0),(B,2),(C,1)\}\} \\ \equiv_{\mathcal{S}_{\mathcal{P}}} \\ \{\mathcal{C}_i = \{(A,0),(B,3),(C,0)\}, \mathcal{C}_r = \{(A,1),(B,0),(C,2)\}\}\end{gathered}$$

because

$$\begin{gathered}\{\mathcal{C}_i = \{(A,0),(B,3),(C,0)\}, \mathcal{C}_r = \{(A,1),(B,0),(C,2)\}\} \\ = \\ \begin{pmatrix} A & B & C \\ B & C & A \end{pmatrix} \{\mathcal{C}_i = \{(A,3),(B,0),(C,0)\}, \mathcal{C}_r = \{(A,0),(B,2),(C,1)\}\}.\end{gathered}$$

Theorem 5.2.3 *Let $\Gamma(\mathcal{P}, i, r)$ be the set of all possible configurations, where $\mathcal{P}$ is the set of honest agents, i is the number of initiator instances allowed, and r is the number of responder instances allowed. Let $\mathcal{S}_{\mathcal{P}}$ be the set of all permutations on the honest agents $\mathcal{P}$. For any $\mathcal{C} \in \Gamma(\mathcal{P}, i, r)$ and for any $p \in \mathcal{S}_{\mathcal{P}}$, $p_{\mathcal{C}}(\mathcal{C}) \in \Gamma(\mathcal{P}, i, r)$.*

Proof: To show that $p(\mathcal{C}) \in \Gamma(\mathcal{P}, i, r)$ we must show that $p_{\mathcal{C}}(\mathcal{C})$ is a valid configuration. In other words, it is sufficient to show that every function $\mathcal{C}_k \in \mathcal{C}$ maps principals in $\mathcal{P}$ to integers and that the number of initiator instances i and responder instances r, do not change. But every function $\mathcal{C}_k \circ p$ in $p_{\mathcal{C}}(\mathcal{C})$ has $\mathcal{P}$ as its domain because p is a permutation on $\mathcal{P}$. Since $p_{\mathcal{C}}$ does not affect the range of the functions $\mathcal{C}_k$, it does not affect the total number of initiators nor the total number of responders.
□

Corollary 5.2.4 *Let $\Gamma(\mathcal{P}, i, r)$ be the set of all possible configurations, where $\mathcal{P}$ is the set of honest agents, i is the number of initiator instances allowed, and r is the number of responder instances allowed. Let $\mathcal{S}_{\mathcal{P}}$ be the set of all permutations on the honest agents $\mathcal{P}$. Then $\Gamma(\mathcal{P}, i, r)$ is symmetric with respect to the set of honest agents. In other words, for any $p \in \mathcal{S}_{\mathcal{P}}$, $p_{\Gamma}(\Gamma(\mathcal{P}, i, r)) = \{ p_{\mathcal{C}}(\mathcal{C}) \mid \mathcal{C} \in \Gamma(\mathcal{P}, i, r) \}$ and $\Gamma(\mathcal{P}, i, r)$ are the same set.*

Proof:

$p_{\Gamma}(\Gamma(\mathcal{P}, i, r)) \subseteq \Gamma(\mathcal{P}, i, r)$

Let $\mathcal{C}' \in p_{\Gamma}(\Gamma(\mathcal{P}, i, r))$. Then $\mathcal{C}' = p_{\mathcal{C}}(\mathcal{C})$ for some $\mathcal{C} \in \Gamma(\mathcal{P}, i, r)$. By Theorem 5.2.3 $p_{\mathcal{C}}(\mathcal{C}) \in \Gamma(\mathcal{P}, i, r)$ and so $\mathcal{C}' \in \Gamma(\mathcal{P}, i, r)$.

$\Gamma(\mathcal{P}, i, r) \subseteq p_{\Gamma}(\Gamma(\mathcal{P}, i, r))$

Let $\mathcal{C} \in \Gamma(\mathcal{P}, i, r)$. Since $p \in \mathcal{S}_{\mathcal{P}}$ and $\mathcal{S}_{\mathcal{P}}$ is a group, $p^{-1} \in \mathcal{S}_{\mathcal{P}}$. Let $\mathcal{C}' = p_{\mathcal{C}}^{-1}(\mathcal{C})$. By Theorem 5.2.3, $\mathcal{C}' \in \Gamma(\mathcal{P}, i, r)$ since $p^{-1} \in \mathcal{S}_{\mathcal{P}}$. Therefore, $\mathcal{C} = p_{\mathcal{C}}(p_{\mathcal{C}}^{-1}(\mathcal{C})) = p_{\mathcal{C}}(\mathcal{C}') \in p_{\Gamma}(\Gamma(\mathcal{P}, i, r))$.

□

Again, by Theorem 5.1.2, the group $\mathcal{S}_{\mathcal{P}}$ partitions the set of configurations $\Gamma(\mathcal{P}, i, r)$, into equivalence classes. The fact that two configurations are equivalent means that they are identical up to a renaming of the principals. If the specification does not distinguish between agent names then one should only have to analyze a single representative from each equivalence class of configurations. Since each configuration $\mathcal{C}$ is symmetric to each configuration $p(\mathcal{C})$ for each $p \in \mathcal{S}_{\mathcal{P}}$, one need only analyze one representative from each orbit which has size roughly $|\mathcal{S}_{\mathcal{P}}| = |\mathcal{P}|!$.

When analyzing the reduction in the size of the problems, one should note that there are some permutations other than the identity, which still map a particular configuration to itself. For example, the configuration

$$\{\mathcal{C}_i = \{(A,1),(B,1),(C,2)\}, \mathcal{C}_r = \{(A,2),(B,2),(C,0)\}\}$$

is mapped to itself by the permutation

$$\begin{pmatrix} A & B & C \\ B & A & C \end{pmatrix}.$$

It is for this reason that the number of configurations that need to be analyzed is reduced by *roughly* a factor of $|\mathcal{P}|!$.

5.2.2 Trace Symmetries

While the the idea of configuration symmetries is fairly intuitive, it is at too high a level to prove the correctness of the symmetry reduction. In order to show that these symmetries respect a specification, one must look at the traces of the system and verify that all the traces belonging to the same orbit agree on the specification. In other words, we need to show that the symmetry is not only an equivalence on traces, but that it is a congruence with respect to the satisfaction relation $\models$.

First we must formally extend p from the domain of principal names to other domains with greater structure.

Definition 5.2.2 *Let $p \in \mathcal{S}_{\mathcal{P}}$ be a permutation on the set of principal names $\mathcal{P}$. Then $p : \mathcal{P} \to \mathcal{P}$ induces a permutation $p_{\mathcal{M}} : \mathcal{M} \to \mathcal{M}$ on the set of messages templates as follows:*

- $p_{\mathcal{M}}(d) = d$ *for all data* $d \in \mathcal{D}$.
- $p_{\mathcal{M}}(h) = p(h)$ *for all principal names* $h \in \mathcal{P}$. *In other words* $p_{\mathcal{M}}$ *maps principal names the same way* p *does.*
- $p_{\mathcal{M}}(n) = n$ *for all nonces* $n \in \mathcal{N}$.
- $p_{\mathcal{M}}(k_x) = k_{p(x)}$ *for all keys* $k_x \in \mathrm{keys} \subseteq \mathcal{A}$. *In other words,* $p_{\mathcal{M}}$ *permutes keys appropriately. If* p *switches principals* A *and* B, *then*

$p_{\mathcal{M}}$ maps the keys of A to the keys of B. Formally, in terms of the key functions, we have:

$$\begin{array}{rcl} p_{\mathcal{M}}(\mathit{pubkey}(x)) & = & \mathit{pubkey}(p(x)), \\ p_{\mathcal{M}}(\mathit{privkey}(x)) & = & \mathit{privkey}(p(x)), \text{ and} \\ p_{\mathcal{M}}(\mathit{symkey}(x)) & = & \mathit{symkey}(p(x)). \end{array}$$

- $p_{\mathcal{M}}(m_1 \cdot m_2) = p_{\mathcal{M}}(m_1) \cdot p_{\mathcal{M}}(m_2)$. *$p_{\mathcal{M}}$ works on a concatenated message by working on the components.*
- $p_{\mathcal{M}}(\{m\}_k) = \{p_{\mathcal{M}}(m)\}_{p_{\mathcal{M}}(k)}$. *$p_{\mathcal{M}}$ works on an encrypted message by working on the plaintext and on the key.*
- $p_{\mathcal{M}}(v) = v$ *for all variables* $v \in \mathcal{V}$.

It is important to note that $p_{\mathcal{M}}$ is also a permutation on the set of atomic messages $\mathcal{A}$. In other words, $p_{\mathcal{M}}(a) \in \mathcal{A}$ for all $a \in \mathcal{A}$. In addition, it is clear from the definition of the extension that the inverse of an extension of a permutation p is equal to the extension of the inverse permutation. In our notation this means $(p_{\mathcal{M}})^{-1} = (p^{-1})_{\mathcal{M}}$.

Definition 5.2.3 *Let $p \in \mathcal{S}_{\mathcal{P}}$ be a permutation on the set of honest agents $\mathcal{P}$ and let $p_{\mathcal{M}}$ be the permutation induced by p on the set of messages $\mathcal{M}$. Then p can be extended to actions, processes, instances, states, and traces as follows.*

- *p_a acts on actions by applying $p_{\mathcal{M}}$ to the message argument of the action. More formally,*

  $$p_a(H_i, A, M) = (H_i, A, p_{\mathcal{M}}(M)).$$

- *p_p acts on a process by applying $p_{\mathcal{M}}$ to every message appearing in the process. More formally,*
  - $p_p(\mathbf{nil}) = \mathbf{nil}$, *and*
  - $p_p(a(m) \cdot P') = a(p_{\mathcal{M}}(m)) \cdot p_p(P')$.

- p_i *acts on instances, again by applying* $p_{\mathcal{M}}$ *to all messages appearing in the instances. More formally, let* $\mathcal{I}_i = \langle H_i, B_i, I_i, P_i \rangle$. *Then*

$$p_i(\mathcal{I}_i) = \langle H_i, p_{\mathcal{M}} \circ B_i, p_{\mathcal{M}}(I_i), p_p(P_i) \rangle.$$

  Again $p_{\mathcal{M}}(I_i)$ *is an abuse of notation for* $\{ \, p_{\mathcal{M}}(m) \mid m \in I_i \, \}$.

- p_s *acts on states by applying* p_i *to all the instances. More formally,*

$$p_s(\{\Omega, \mathcal{I}_1, \ldots, \mathcal{I}_n\}) = \{p_i(\Omega), p_i(\mathcal{I}_1), \ldots, p_i(\mathcal{I}_n)\}.$$

- p_t *acts on a trace by applying* p_s *to all the states in the trace and* p_a *to all the actions appearing in the trace. More formally,*

$$p_t(\sigma_0 \alpha_1 \sigma_1 \alpha_1 \cdots \alpha_n \sigma_n) = p_s(\sigma_0) p_a(\alpha_1) p_s(\sigma_1) p_a(\alpha_1) \cdots p_a(\alpha_n) p_s(\sigma_n).$$

Again, note that the inverse of the extension of p is the extension of the inverse of p. Namely,

$$\begin{array}{rcl}
(p_a)^{-1} & = & (p^{-1})_a, \\
(p_p)^{-1} & = & (p^{-1})_p, \\
(p_i)^{-1} & = & (p^{-1})_i, \\
(p_s)^{-1} & = & (p^{-1})_s, \text{ and} \\
(p_t)^{-1} & = & (p^{-1})_t.
\end{array}$$

With this notation, we can now show that the set of all traces over all configurations $\Gamma(\mathcal{P}, i, r)$ is symmetric with respect to the names of the honest agents $\mathcal{P}$. We begin with a couple of lemmas.

Lemma 5.2.5 *Let* $p \in \mathcal{S}_{\mathcal{P}}$ *be a permutation on the names of honest agents and let* $p_{\mathcal{M}}$ *be the extension of* p *to messages. Let* $B : \mathcal{V} \to \mathcal{M}$ *be a set of variable bindings and* $\overline{B}$ *be the extension of* B *to message templates. Then* $p_{\mathcal{M}}(\overline{B}(m)) = \overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(m))$ *for all message templates* m *whose variables are in the domain of* B.

The intuition here is straightforward. If we take a message template m, apply the bindings B to the variables and then permute the message with $p_{\mathcal{M}}$ [i.e., $p_{\mathcal{M}}(\overline{B}(m))$] the result should be the same as if we first permute the message template with $p_{\mathcal{M}}$ (leaving the variables alone), then we apply the permuted bindings to the variables [i.e., $\overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(m))$]. The formal proof is below.

Proof: (by induction on the structure of m)

- $m = a \in \mathcal{A}$

$$\begin{array}{rcll} p_{\mathcal{M}}(\overline{B}(m)) & = & p_{\mathcal{M}}(\overline{B}(a)) & \\ & = & p_{\mathcal{M}}(a) & \text{Definition 3.1.2} \\ & = & \overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(a)) & \text{Definition 3.1.2 and } p_{\mathcal{M}}(a) \in \mathcal{A} \\ & = & \overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(m)) & \end{array}$$

- $m = v \in \mathcal{V}$

$$\begin{array}{rcll} p_{\mathcal{M}}(\overline{B}(m)) & = & p_{\mathcal{M}}(\overline{B}(v)) & \\ & = & p_{\mathcal{M}}(B(v)) & \text{Definition 3.1.2} \\ & = & (p_{\mathcal{M}} \circ B)(v) & \\ & = & \overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(v)) & \text{Definition 3.1.2} \\ & = & \overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(m)) & \end{array}$$

- $m = m_1 \cdot m_2$

$$\begin{array}{rcll} p_{\mathcal{M}}(\overline{B}(m)) & = & p_{\mathcal{M}}(\overline{B}(m_1 \cdot m_2)) & \\ & = & p_{\mathcal{M}}(\overline{B}(m_1) \cdot \overline{B}(m_2)) & \text{Definition 3.1.2} \\ & = & p_{\mathcal{M}}(\overline{B}(m_1)) \cdot p_{\mathcal{M}}(\overline{B}(m_2)) & \text{Definition 5.2.2} \\ & = & \overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(m_1)) \cdot \overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(m_2)) & \text{ind. hyp.} \\ & = & \overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(m_1) \cdot p_{\mathcal{M}}(m_2)) & \text{Definition 3.1.2} \\ & = & \overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(m_1 \cdot m_2)) & \text{Definition 5.2.2} \\ & = & \overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(m)) & \end{array}$$

- $m = \{m'\}_k$

$$\begin{array}{rcll} p_{\mathcal{M}}(\overline{B}(m)) & = & p_{\mathcal{M}}(\overline{B}(\{m'\}_k)) & \\ & = & p_{\mathcal{M}}(\{\overline{B}(m')\}_{\overline{B}(k)}) & \text{Definition 3.1.2} \\ & = & \{p_{\mathcal{M}}(\overline{B}(m'))\}_{p_{\mathcal{M}}(\overline{B}(k))} & \text{Definition 5.2.2} \\ & = & \{\overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(m'))\}_{\overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(k))} & \text{ind. hyp.} \\ & = & \overline{p_{\mathcal{M}} \circ B}(\{p_{\mathcal{M}}(m')\}_{p_{\mathcal{M}}(k)}) & \text{Definition 3.1.2} \\ & = & \overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(\{m'\}_k)) & \text{Definition 5.2.2} \\ & = & \overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(m)) & \end{array}$$

□

Lemma 5.2.6 *Let $k \in \mathcal{K}$ be a key. Let $p \in \mathcal{S}_{\mathcal{P}}$ be a permutation on honest agent names and let $p_{\mathcal{M}}$ be the extension of p to the set of message templates. Then the inverse of a permuted key is the same as the permuted inverse key. More formally, $p_{\mathcal{M}}(k^{-1}) = (p_{\mathcal{M}}(k))^{-1}$.*

Proof: Consider the three possibilities for k. Using Definition 5.2.2 and the definition of inverse keys:

- $k = pubkey(X)$

$$
\begin{aligned}
p_{\mathcal{M}}((pubkey(X))^{-1}) &= p_{\mathcal{M}}(privkey(X)) \\
&= privkey(p_{\mathcal{M}}(X)) \\
&= (pubkey(p_{\mathcal{M}}(X)))^{-1}
\end{aligned}
$$

- $k = privkey(X)$

$$
\begin{aligned}
p_{\mathcal{M}}((privkey(X))^{-1}) &= p_{\mathcal{M}}(pubkey(X)) \\
&= pubkey(p_{\mathcal{M}}(X)) \\
&= (privkey(p_{\mathcal{M}}(X)))^{-1}
\end{aligned}
$$

- $k = symkey(X)$

$$
\begin{aligned}
p_{\mathcal{M}}((symkey(X))^{-1}) &= p_{\mathcal{M}}(symkey(X)) \\
&= symkey(p_{\mathcal{M}}(X)) \\
&= (symkey(p_{\mathcal{M}}(X)))^{-1}
\end{aligned}
$$

□

Lemma 5.2.7 *Let T be a derivation tree for m with assumptions in A. Let $p \in \mathcal{S}_{\mathcal{P}}$ be a permutation on honest agent names $\mathcal{P}$ and let $p_{\mathcal{M}}$ be the permutation induced by p over the set of messages $\mathcal{M}$. When $p_{\mathcal{M}}$ is applied to every message appearing in T, the result is a new derivation tree $p_{\mathcal{M}}(T)$ for $p_{\mathcal{M}}(m)$ with assumptions in $p_{\mathcal{M}}(A)$.*

Proof: (by induction on the height of the derivation tree T)

- Assume T has height 0 and hence is atomic.
  Then T consists of the single message m and $m \in A$. This means that $p_{\mathcal{M}}(T)$ consists of the single message $p_{\mathcal{M}}(m)$, and that $p_{\mathcal{M}}(m) \in p_{\mathcal{M}}(A)$.

- Assume T has height n and T has the form

$$\frac{\dfrac{\Sigma}{m_1 \cdot m_2}}{m_i}$$

  and $m_i = m$. The subtree above m_i has height $n - 1$. By the inductive hypothesis,

$$\frac{p_{\mathcal{M}}(\Sigma)}{p_{\mathcal{M}}(m_1 \cdot m_2)}$$

  is a derivation tree for $p_{\mathcal{M}}(m_1 \cdot m_2)$ with assumptions in $p_{\mathcal{M}}(A)$. By Definition 5.2.2, $p_{\mathcal{M}}(m_1 \cdot m_2) = p_{\mathcal{M}}(m_1) \cdot p_{\mathcal{M}}(m_2)$. Clearly,

$$\frac{p_{\mathcal{M}}(m_1) \cdot p_{\mathcal{M}}(m_2)}{p_{\mathcal{M}}(m_i)}$$

  is a valid inference. Putting these together results in a derivation tree for $p_{\mathcal{M}}(m_i)$ with assumptions in $p_{\mathcal{M}}(A)$.

- Assume T has height n and T has the form

$$\frac{\dfrac{\Sigma_1}{m_1} \qquad \dfrac{\Sigma_2}{m_2}}{m_1 \cdot m_2}$$

  Then the subtrees above $m_1 \cdot m_2$ have height at most $n - 1$. By the inductive hypothesis,

$$\frac{p_{\mathcal{M}}(\Sigma_i)}{p_{\mathcal{M}}(m_i)}$$

  is a derivation tree for $p_{\mathcal{M}}(m_i)$ with assumptions in $p_{\mathcal{M}}(A)$. Clearly,

$$\frac{p_{\mathcal{M}}(m_1) \qquad p_{\mathcal{M}}(m_2)}{p_{\mathcal{M}}(m_1) \cdot p_{\mathcal{M}}(m_2)}$$

is a valid inference. Putting these together results in a new derivation tree $p_{\mathcal{M}}(T)$ for $p_{\mathcal{M}}(m_1) \cdot p_{\mathcal{M}}(m_2) = p_{\mathcal{M}}(m_1 \cdot m_2)$ with assumptions in $p_{\mathcal{M}}(A)$.

- Assume T has height n and T has the form

$$\frac{\dfrac{\Sigma_1}{\{m\}_k} \qquad \dfrac{\Sigma_2}{k^{-1}}}{m}$$

Then the subtrees above m have height at most $n-1$. By the inductive hypothesis,

$$\frac{p_{\mathcal{M}}(\Sigma_1)}{p_{\mathcal{M}}(\{m\}_k)} \qquad \text{and} \qquad \frac{p_{\mathcal{M}}(\Sigma_2)}{p_{\mathcal{M}}(k^{-1})}$$

are derivation trees for $p_{\mathcal{M}}(\{m\}_k)$ and $p_{\mathcal{M}}(k^{-1})$ with assumptions in $p_{\mathcal{M}}(A)$. By Definition 5.2.2, $p_{\mathcal{M}}(\{m\}_k) = \{p_{\mathcal{M}}(m)\}_{p_{\mathcal{M}}(k)}$, and by Lemma 5.2.6, $p_{\mathcal{M}}(k^{-1}) = (p_{\mathcal{M}}(k))^{-1}$. Clearly,

$$\frac{\{p_{\mathcal{M}}(m)\}_{p_{\mathcal{M}}(k)} \qquad (p_{\mathcal{M}}(k))^{-1}}{p_{\mathcal{M}}(m)}$$

is a valid inference. Putting these together results in a derivation tree for $p_{\mathcal{M}}(m)$ with assumptions in $p_{\mathcal{M}}(A)$.

- Assume T has height n and T has the form

$$\frac{\dfrac{\Sigma_1}{m} \qquad \dfrac{\Sigma_2}{k}}{\{m\}_k}$$

Then the subtrees above $\{m\}_k$ have height at most $n-1$. By the inductive hypothesis,

$$\frac{p_{\mathcal{M}}(\Sigma_1)}{p_{\mathcal{M}}(m)} \qquad \text{and} \qquad \frac{p_{\mathcal{M}}(\Sigma_2)}{p_{\mathcal{M}}(k)}$$

are derivation trees for $p_{\mathcal{M}}(m)$ and $p_{\mathcal{M}}(k)$ with assumptions in $p_{\mathcal{M}}(A)$. Clearly,

$$\frac{p_{\mathcal{M}}(m) \qquad p_{\mathcal{M}}(k)}{\{p_{\mathcal{M}}(m)\}_{p_{\mathcal{M}}(k)}}$$

is a valid inference. Putting these together results in a new derivation tree $p_{\mathcal{M}}(T)$ for $\{p_{\mathcal{M}}(m)\}_{p_{\mathcal{M}}(k)} = p_{\mathcal{M}}(\{m\}_k)$ with assumptions in $p_{\mathcal{M}}(A)$.

□

Lemma 5.2.8 *Let $p \in \mathcal{S}_{\mathcal{P}}$ be a permutation on the names of the honest agents. Let $p_{\mathcal{M}}$, p_a, and p_s be the extensions of p to messages, actions, and states respectively. If $\sigma \overset{H_i \cdot A \cdot m}{\longrightarrow} \sigma'$ is a valid transition then so is $p_s(\sigma) \overset{p_a(H_i \cdot A \cdot m)}{\longrightarrow} p_s(\sigma')$.*

Proof: It is clear from the definition of p_s that if σ and σ' are states, then so are $p_s(\sigma)$ and $p_s(\sigma')$. From the definition of p_a, if $H_i \cdot A \cdot m$ is an action, then so is $p_a(H_i \cdot A \cdot m) = H_i \cdot A \cdot p_{\mathcal{M}}(m)$. It remains to show that each of the three different kinds of actions results in a valid transition.

- $H_i \cdot A \cdot m$ is a **send** action
  1. $I'_\Omega = I_\Omega \cup m$. (The adversary adds m to the set of messages it knows.)
  2. If in σ, $\mathcal{I}_i = \langle H_i, B_i, I_i, P_i \rangle$, and in σ', $\mathcal{I}'_i = \langle H'_i, B'_i, I'_i, P'_i \rangle$, then
     - $H'_i = H_i$ (The instance ID remains unchanged.)
     - $B'_i = B_i$ (The bindings remain unchanged.)
     - $I'_i = I_i$ (The set of information remains unchanged.)
     - $P_i =$ **send** $(s\text{-}msg) \cdot P'_i$ (The instance is ready to send a message, and that send action is removed from the process description in the new state.)
     - $m = \overline{B_i}(s\text{-}msg)$ (The message that the instance is ready to send and the actual sent message are the same.)
  3. $\mathcal{I}_j = \mathcal{I}'_j$ for all $j \neq i$. (All other instances remain unchanged.)

  We now need to show the corresponding properties for the permuted transition $p_s(\sigma) \overset{H_i \cdot A \cdot p_{\mathcal{M}}(m)}{\longrightarrow} p_s(\sigma')$.

1. In $p_s(\sigma')$,

$$p_{\mathcal{M}}(I'_\Omega) = p_{\mathcal{M}}(I_\Omega \cup m) = p_{\mathcal{M}}(I_\Omega) \cup p_{\mathcal{M}}(m).$$

   (The adversary adds $p_{\mathcal{M}}(m)$ to the set of messages it knows.)

2. In $p_s(\sigma)$,

$$p_i(\mathcal{I}_i) = \langle H_i, p_{\mathcal{M}} \circ B_i, p_{\mathcal{M}}(I_i), p_p(P_i)\rangle,$$

   and in $p_s(\sigma')$,

$$p_i(\mathcal{I}'_i) = \langle H'_i, p_{\mathcal{M}} \circ B'_i, p_{\mathcal{M}}(I'_i), p_p(P'_i)\rangle,$$

   and

   – $H'_i = H_i$ (The instance ID remains unchanged.)

   – $B'_i = B_i$ so
$$p_{\mathcal{M}} \circ B'_i = p_{\mathcal{M}} \circ B_i$$
   (The bindings remain unchanged.)

   – $I'_i = I_i$ so
$$p_{\mathcal{M}}(I'_i) = p_{\mathcal{M}}(I_i)$$
   (The set of information remains unchanged.)

   – $P_i =$ **<u>send</u>** $(s\text{-}msg) \cdot P'_i$ so

$$p_p(P_i) = \underline{\textbf{send}}\ (p_{\mathcal{M}}(s\text{-}msg)) \cdot p_p(P'_i)$$

   (The instance is ready to send a message, and that send action is removed from the process description in the new state.)

   – $m = \overline{B_i}(s\text{-}msg)$ so

$$p_{\mathcal{M}}(m) = p_{\mathcal{M}}(\overline{B_i}(s\text{-}msg)) = \overline{p_{\mathcal{M}} \circ B_i}(p_{\mathcal{M}}(s\text{-}msg))$$

   by Lemma 5.2.5. (The message that the instance is ready to send and the actual sent message are the same.)

3. $\mathcal{I}_j = \mathcal{I}'_j$ for all $j \neq i$ so

$$p_i(\mathcal{I}_j) = p_i(\mathcal{I}'_j) \text{ for all } j \neq i.$$

   (All other instances remain unchanged.)

- $H_i \cdot A \cdot m$ is a **receive** action

  1. $m \in \overline{I_\Omega}$. (The adversary can generate the message m.)
  2. If in σ, $\mathcal{I}_i = \langle H_i, B_i, I_i, P_i \rangle$, and in σ', $\mathcal{I}'_i = \langle H'_i, B'_i, I'_i, P'_i \rangle$, then
     - $H_i = H'_i$ (The instance ID remains unchanged.)
     - B'_i is the smallest extension of B_i such that $\overline{B'_i}(r\text{-}msg) = m$. (The bindings of the instance are updated correctly, and the message received matches the message template in the receive action.)
     - $I'_i = I_i \cup m$ (The information of the instance is updated correctly.)
     - $P_i =$ **receive** $(r\text{-}msg) \cdot P'_i$ (The instance is ready to receive a message, and that action is removed from the process description in the next state.)
  3. $\mathcal{I}_j = \mathcal{I}'_j$ for all $j \neq i$. (All other instances remain unchanged.)

Again, we must show the corresponding properties for the permuted transition $p_s(\sigma) \stackrel{H_i \cdot A \cdot p_{\mathcal{M}}(m)}{\longrightarrow} p_s(\sigma')$.

1. By Lemma 5.2.7, we can apply $p_{\mathcal{M}}$ to the derivation tree for $m \in \overline{I_\Omega}$ to obtain a derivation tree for $p_{\mathcal{M}}(m) \in \overline{p_{\mathcal{M}}(I_\Omega)}$. (The adversary can generate the message $p_{\mathcal{M}}(m)$.)
2. In $p_s(\sigma)$,

$$p_i(\mathcal{I}_i) = \langle H_i, p_{\mathcal{M}} \circ B_i, p_{\mathcal{M}}(I_i), p_p(P_i) \rangle,$$

   and in $p_s(\sigma')$,

$$p_i(\mathcal{I}'_i) = \langle H'_i, p_{\mathcal{M}} \circ B'_i, p_{\mathcal{M}}(I'_i), p_p(P'_i) \rangle,$$

   and

   - $H_i = H'_i$ (The instance ID remains unchanged.)
   - B'_i is the smallest extension of B_i such that $\overline{B'_i}(r\text{-}msg) = m$ so using Lemma 5.2.5, $p_{\mathcal{M}} \circ B'_i$ is the smallest extension of $p_{\mathcal{M}} \circ B_i$ such that $\overline{p_{\mathcal{M}} \circ B'_i}(p_{\mathcal{M}}(r\text{-}msg)) = p_{\mathcal{M}}(m)$. (The bindings of the instance are updated correctly, and the message received matches the message template in the receive action.)

– $I'_i = I_i \cup m$ so

$$p_{\mathcal{M}}(I'_i) = p_{\mathcal{M}}(I_i \cup m) = p_{\mathcal{M}}(I_i) \cup p_{\mathcal{M}}(m)$$

(The information of the instance is updated correctly.)

– $P_i =$ **receive** $(r\text{-}msg) \cdot P'_i$ so

$$p_p(P_i) = \underline{\textbf{receive}}\ (p_{\mathcal{M}}(r\text{-}msg)) \cdot p_p(P'_i)$$

(The instance is ready to receive a message, and that action is removed from the process description in the next state.)

3. $\mathcal{I}_j = \mathcal{I}'_j$ for all $j \neq i$ so

$$p_i(\mathcal{I}_j) = p_i(\mathcal{I}'_j) \text{ for all } j \neq i.$$

(All other instances remain unchanged.)

- $H_i \cdot A \cdot m$ is an internal action,

1. If in σ, $\mathcal{I}_i = \langle H_i, B_i, I_i, P_i \rangle$, and in σ', $\mathcal{I}'_i = \langle H'_i, B'_i, I'_i, P'_i \rangle$, then
   – $H'_i = H_i$ (The instance ID remains unchanged.)
   – $B'_i = B_i$ (The bindings remain unchanged.)
   – $I'_i = I_i$ (The set of information remains unchanged.)
   – $P_i = A(msg) \cdot P'_i$ (The instance is ready to perform action A, and that action is removed from the process description in the new state.)
   – $m = \overline{B_i}(msg)$. (The message argument in the process description and the actual message occurring in the taken action are the same.)
2. $\mathcal{I}_j = \mathcal{I}'_j$ for all $j \neq i$. (All other instances remain unchanged.)

We must show the corresponding properties for the permuted transition $p_s(\sigma) \stackrel{H_i \cdot A \cdot p_{\mathcal{M}}(m)}{\longrightarrow} p_s(\sigma')$.

1. In $p_s(\sigma)$,

$$p_i(\mathcal{I}_i) = \langle H_i, p_{\mathcal{M}} \circ B_i, p_{\mathcal{M}}(I_i), p_p(P_i) \rangle,$$

and in $p_s(\sigma')$,

$$p_i(\mathcal{I}'_i) = \langle H'_i, p_{\mathcal{M}} \circ B'_i, p_{\mathcal{M}}(I'_i), p_p(P'_i) \rangle,$$

and

- $H_i' = H_i$ (The instance ID remains unchanged.)
- $B_i' = B_i$ so
$$p_{\mathcal{M}} \circ B_i' = p_{\mathcal{M}} \circ B_i$$
(The bindings remain unchanged.)
- $I_i' = I_i$ so
$$p_{\mathcal{M}}(I_i') = p_{\mathcal{M}}(I_i)$$
(The set of information remains unchanged.)
- $P_i = A(msg) \cdot P_i'$ so
$$p_p(P_i) = A(p_{\mathcal{M}}(s\text{-}msg)) \cdot p_p(P_i')$$
(The instance is ready to perform action A, and that action is removed from the process description in the new state.)
- $m = \overline{B_i}(msg)$ so
$$p_{\mathcal{M}}(m) = p_{\mathcal{M}}(\overline{B_i}(msg)) = \overline{p_{\mathcal{M}} \circ B_i}(p_{\mathcal{M}}(msg))$$
by Lemma 5.2.5. (The message argument in the process description and the actual message occurring in the taken action are the same.)

2. $\mathcal{I}_j = \mathcal{I}_j'$ for all $j \neq i$ so
$$p_i(\mathcal{I}_j) = p_i(\mathcal{I}_j') \text{ for all } j \neq i.$$
(All other instances remain unchanged.)

$\square$

Theorem 5.2.9 *Let $Tr(\mathcal{P}, i, r)$ be the set of all traces with honest agents in $\mathcal{P}$, with i initiators, and with r responders, and let $\pi \in Tr(\mathcal{P}, i, r)$. If $p \in \mathcal{S}_{\mathcal{P}}$ and p_t is the extension of p to traces, then $p_t(\pi) \in Tr(\mathcal{P}, i, r)$.*

Proof: First we show that for any valid trace π, $p_t(\pi)$ is also a trace. We prove this inductively on the length of a trace.

- Let $\pi = \sigma_0$ be a trace of length 0. Then $p_t(\pi) = p_s(\sigma_0)$. Clearly, $p_s(\sigma_0)$ is a state as well, and so it is a trace of length 0.

- Let $\pi = \sigma_0\alpha_1\sigma_1\alpha_2\cdots\alpha_{n+1}\sigma_{n+1}$ be a trace of length $n+1$. Then $\pi_n = \sigma_0\alpha_1\sigma_1\alpha_2\cdots\alpha_n\sigma_n$ is a trace of length n and by the inductive hypothesis so is

$$p_t(\pi_n) = p_s(\sigma_0)p_a(\alpha_1)p_s(\sigma_1)p_a(\alpha_2)\cdots p_a(\alpha_n)p_s(\sigma_n).$$

From the trace π we see that $\sigma_n \xrightarrow{\alpha_n} \sigma_{n+1}$ is a valid transition. By Lemma 5.2.8, $p_s(\sigma_n) \xrightarrow{p_a(\alpha_{n+1})} p_s(\sigma_{n+1})$ is also a valid transition. Appending this to the end of $p_t(\pi_n)$ results in the valid trace $p_t(\pi)$.

We still need to show that the trace $p_t(\pi)$ is a trace in $Tr(\mathcal{P}, i, r)$. Since $\pi \in Tr(\mathcal{P}, i, r)$, π has i initiators and r responders, and all the honest agents are in $\mathcal{P}$. But p_t does not affect the number of initiators or responders; it only changes the names of the honest agents. Therefore, $p_t(\pi)$ has i initiators and r responders. Furthermore, p_t is an extension of $p \in \mathcal{S}_\mathcal{P}$ which is a permutation on the names of the honest agents, so the names of the honest agents in p_t are also all members of $\mathcal{P}$. Therefore, $p_t(\pi) \in Tr(\mathcal{P}, i, r)$.
□

One can relate this to the intuitive description of symmetries on configurations by noting that if π is a trace of some configuration $\mathcal{C} \in \Gamma(\mathcal{P}, i, r)$, then $p_t(\pi)$ is a trace of the configuration $p_\mathcal{C}(\mathcal{C})$.[1]

Corollary 5.2.10 *Let $Tr(\mathcal{P}, i, r)$ be the set of all traces with honest agents in $\mathcal{P}$ and with i initiators and r responders. Let $\mathcal{S}_\mathcal{P}$ be the set of all permutations on the honest agents $\mathcal{P}$. Then $Tr(\mathcal{P}, i, r)$ is symmetric with respect to the set of honest agents. In other words, for any $p \in \mathcal{S}_\mathcal{P}$, $p_t(Tr(\mathcal{P}, i, r)) = \{\, p(\pi) \mid \pi \in Tr(\mathcal{P}, i, r) \,\}$ is equal to $Tr(\mathcal{P}, i, r)$.*

Proof:

$p_t(Tr(\mathcal{P}, i, r)) \subseteq Tr(\mathcal{P}, n)$

Let $\pi' \in p_t(Tr(\mathcal{P}, i, r))$. Then $\pi' = p_t(\pi)$ for some $\pi \in Tr(\mathcal{P}, i, r)$. By Theorem 5.2.9, $\pi' = p_t(\pi) \in Tr(\mathcal{P}, i, r)$.

[1]A proof of this would require a formalization of what it means for a trace to belong to a configuration. Since the purpose of the configuration construction is to provide intuition and not technical details, this would run counter to our purposes.

$Tr(\mathcal{P}, i, r) \subseteq p_t(Tr(\mathcal{P}, i, r))$

Let $\pi \in Tr(\mathcal{P}, i, r)$. Since $p \in \mathcal{S}_\mathcal{P}$ and $\mathcal{S}_\mathcal{P}$ is a group, $p^{-1} \in \mathcal{S}_\mathcal{P}$. Let $\pi' = p_t^{-1}(\pi)$. By Theorem 5.2.9, $\pi' \in Tr(\mathcal{P}, i, r)$ because $p^{-1} \in \mathcal{S}_\mathcal{P}$. Therefore, $\pi = p_t(p_t^{-1}(\pi)) = p_t(\pi') \in p_t(Tr(\mathcal{P}, i, r))$.

□

As is the case with configurations, Theorem 5.1.2 tells us that the group of permutations induced by $\mathcal{S}_\mathcal{P}$ partitions the set of traces $Tr(\mathcal{P}, i, r)$, into equivalence classes. One trace π is equivalent to another trace π' if and only if there exists some $p \in \mathcal{S}_\mathcal{P}$ such that $\pi' = p_t(\pi)$. This symmetry partitions the traces into sets with equivalent structure, in the sense that the same actions and relationships are present, however the names of the honest agents have been permuted. It remains to prove that these equivalence classes of traces respect the truth values of formulas that display the same symmetry. First we must define the extension of $p \in \mathcal{S}_\mathcal{P}$ to the set of formulas.

Definition 5.2.4 *Let $\mathcal{S}_\mathcal{P}$ be the set of permutations on the names of the honest agents. Let ϕ be any formula in our logic. Then we extend $p \in \mathcal{S}_\mathcal{P}$ to formulas as follows:*

- $p_f(H_i.v) = H_i.v$ *for any instance ID i and any message variable v.*
- $p_f(a) = p_\mathcal{M}(a)$ *for any atomic message $a \in \mathcal{A}$.*
- $p_f(m_1 \cdot m_2) = p_f(m_1) \cdot p_f(m_2)$ *for all message terms m_1 and m_2.*
- $p_f(\{m\}_k) = \{p_f(m)\}_{p_f(k)}$ *for all message terms m and k.*
- $p_f(m_1 = m_2) = [p_f(m_1) = p_f(m_2)]$ *for any atomic proposition of the form $m_1 = m_2$.*
- $p_f(H_i$ **Knows** $m) = H_i$ **Knows** $p_f(m)$ *for any atomic proposition of the form H_i* **Knows** *m.*
- $p_f(H_i$ **Act** $m) = H_i$ **Act** $p_f(m)$ *for any atomic proposition of the form H_i* **Act** *m.*
- $p_f(\neg\phi) = \neg(p_f(\phi))$.
- $p_f(\phi_1 \wedge \phi_2) = p_f(\phi_1) \wedge p_f(\phi_2)$.

- $p_f(\Diamond_P \phi) = \Diamond_P p_f(\phi)$.

We now need to show that a trace satisfies a formula if and only if the permuted trace satisfies the permuted formula. First we need a lemma analogous to Lemma 5.2.5, but for message terms instead of message templates.

Lemma 5.2.11 *Let $p \in \mathcal{S_P}$ be a permutation on the names of honest agents. Let $p_\mathcal{M}$ be the extension of p to messages, p_s be the extension of p to states, and p_f be the extension of p to formulas. Let m be a message term and let σ be a state in which all variables appearing in m are bound. Then $p_\mathcal{M}(\sigma(m)) = [p_s(\sigma)](p_f(m))$.*

Proof: (by induction on the structure of the message term m)

- $m = a \in \mathcal{A}$

$$\begin{array}{rcll} p_\mathcal{M}(\sigma(m)) & = & p_\mathcal{M}(\sigma(a)) & \\ & = & p_\mathcal{M}(a) & \text{Definition 3.2.1} \\ & = & [p_s(\sigma)](p_\mathcal{M}(a)) & \text{Definition 3.2.1 and } p_\mathcal{M}(a) \in \mathcal{A} \\ & = & [p_s(\sigma)](p_f(a)) & \text{Definition 5.2.4} \\ & = & [p_s(\sigma)](p_f(m)) & \end{array}$$

- $m = H_i.v$

$$\begin{array}{rcll} p_\mathcal{M}(\sigma(m)) & = & p_\mathcal{M}(\sigma(H_i.v)) & \\ & = & p_\mathcal{M}(B_i(v)) & \text{Definition 3.2.1} \\ & = & (p_\mathcal{M} \circ B_i)(v) & \\ & = & [p_s(\sigma)](H_i.v) & \text{Definition 5.2.3 and Definition 3.2.1} \\ & = & [p_s(\sigma)](p_f(H_i.v)) & \text{Definition 5.2.4} \\ & = & [p_s(\sigma)](p_f(m)) & \end{array}$$

- $m = m_1 \cdot m_2$

$$\begin{array}{rcll} p_\mathcal{M}(\sigma(m)) & = & p_\mathcal{M}(\sigma(m_1 \cdot m_2)) & \\ & = & p_\mathcal{M}(\sigma(m_1) \cdot \sigma(m_2)) & \text{Definition 3.2.1} \\ & = & p_\mathcal{M}(\sigma(m_1)) \cdot p_\mathcal{M}(\sigma(m_2)) & \text{Definition 5.2.2} \\ & = & [p_s(\sigma)](p_f(m_1)) \cdot [p_s(\sigma)](p_f(m_2)) & \text{ind. hyp.} \\ & = & [p_s(\sigma)](p_f(m_1) \cdot p_f(m_2)) & \text{Definition 3.2.1} \\ & = & [p_s(\sigma)](p_f(m_1 \cdot m_2)) & \text{Definition 5.2.4} \\ & = & [p_s(\sigma)](p_f(m)) & \end{array}$$

- $m = \{m'\}_k$

$$\begin{array}{rcll} p_{\mathcal{M}}(\sigma(m)) & = & p_{\mathcal{M}}(\sigma(\{m'\}_k)) & \\ & = & p_{\mathcal{M}}(\{\sigma(m')\}_{\sigma(k)}) & \text{Definition 3.2.1} \\ & = & \{p_{\mathcal{M}}(\sigma(m'))\}_{p_{\mathcal{M}}(\sigma(k))} & \text{Definition 5.2.2} \\ & = & \{[p_s(\sigma)](p_f(m'))\}_{[p_s(\sigma)](p_f(k))} & \text{ind. hyp.} \\ & = & [p_s(\sigma)](\{(p_f(m')\}_{p_f(k)}) & \text{Definition 3.2.1} \\ & = & [p_s(\sigma)](p_f(\{m'\}_k)) & \text{Definition 5.2.4} \\ & = & [p_s(\sigma)](p_f(m)) & \end{array}$$

□

Theorem 5.2.12 *Let π be a trace, ϕ be a formula, and $\mathcal{S}_{\mathcal{P}}$ be the permutations on the names of the honest agents. Then for any $p \in \mathcal{S}_{\mathcal{P}}$, $\langle \pi, i \rangle \models \phi$ if and only if $\langle p_t(\pi), i \rangle \models p_f(\phi)$.*

Proof: (by induction on the structure of the formula ϕ)

- $\phi = [m_1 = m_2]$
  Using the definition of $\models$, Lemma 5.2.11, and Definition 5.2.4 (the definition of p_f) we have the following:

$$\begin{array}{rcl} \langle \pi, i \rangle \models [m_1 = m_2] & \Leftrightarrow & \sigma_i(m_1) = \sigma_i(m_2) \\ & \Leftrightarrow & p_{\mathcal{M}}(\sigma_i(m_1)) = p_{\mathcal{M}}(\sigma_i(m_2)) \\ & \Leftrightarrow & [p_s(\sigma_i)](p_f(m_1)) = [p_s(\sigma_i)](p_f(m_2)) \\ & \Leftrightarrow & \langle p_t(\pi), i \rangle \models [p_f(m_1) = p_f(m_2)] \\ & \Leftrightarrow & \langle p_t(\pi), i \rangle \models p_f([m_1 = m_2]) \end{array}$$

- $\phi = H_j$ **Knows** m
  Let $\mathcal{I}_j = \langle H_j, B_j, I_j, P_j \rangle$ in σ_i. Then in $p_s(\sigma_i)$,

$$p_i(\mathcal{I}_j) = \langle H_j, p_{\mathcal{M}} \circ B_j, p_{\mathcal{M}}(I_j), p_p(P_j) \rangle$$

  Using the definition of $\models$, Lemma 5.2.7 about applying $p_{\mathcal{M}}$ to derivation trees, Lemma 5.2.11, and Definition 5.2.4 (the definition of p_f) we have the following:

$$\begin{array}{rcl} \langle \pi, i \rangle \models H_j \ \ \textbf{Knows} \ \ m & \Leftrightarrow & \sigma_i(m) \in \overline{I_j} \\ & \Leftrightarrow & p_{\mathcal{M}}(\sigma_i(m)) \in \overline{p_{\mathcal{M}}(I_j)} \\ & \Leftrightarrow & [p_s(\sigma_i)](p_f(m)) \in \overline{p_{\mathcal{M}}(I_j)} \\ & \Leftrightarrow & \langle p_t(\pi), i \rangle \models H_j \ \ \textbf{Knows} \ \ p_f(m) \\ & \Leftrightarrow & \langle p_t(\pi), i \rangle \models p_f(H_j \ \ \textbf{Knows} \ \ m) \end{array}$$

- $\phi = H_j\ A\ m$
  Using the definition of $\models$, Definition 5.2.3 (definition of p_a), Lemma 5.2.11, and Definition 5.2.4 (the definition of p_f) we have the following:

$$\begin{aligned}
\langle \pi, i\rangle \models H_j\ A\ m &\Leftrightarrow \alpha_k = H_j \cdot A \cdot \sigma_{k-1}(m) \text{ for some } 1 \leq k \leq i \\
&\Leftrightarrow p_a(\alpha_k) = p_a(H_j \cdot A \cdot \sigma_{k-1}(m)) \\
&\Leftrightarrow p_a(\alpha_k) = H_j \cdot A \cdot p_{\mathcal{M}}(\sigma_{k-1}(m)) \\
&\Leftrightarrow p_a(\alpha_k) = H_j \cdot A \cdot [p_s(\sigma_{k-1})](p_f(m)) \\
&\Leftrightarrow \langle p_t(\pi), i\rangle \models H_j\ A\ p_f(m) \\
&\Leftrightarrow \langle p_t(\pi), i\rangle \models p_f(H_j\ A\ m)
\end{aligned}$$

- $\phi = \neg\phi'$
  Using the definition of $\models$, the inductive hypothesis, and Definition 5.2.4 (the definition of p_f) we have the following:

$$\begin{aligned}
\langle \pi, i\rangle \models \neg\phi' &\Leftrightarrow \langle \pi, i\rangle \not\models \phi' \\
&\Leftrightarrow \langle p_t(\pi), i\rangle \not\models p_f(\phi') \\
&\Leftrightarrow \langle p_t(\pi), i\rangle \models \neg p_f(\phi') \\
&\Leftrightarrow \langle p_t(\pi), i\rangle \models p_f(\neg\phi')
\end{aligned}$$

- $\phi = \phi_1 \wedge \phi_2$
  Using the definition of $\models$, the inductive hypothesis, and Definition 5.2.4 (the definition of p_f) we have the following:

$$\begin{aligned}
\langle \pi, i\rangle \models \phi_1 \wedge \phi_2 &\Leftrightarrow \langle \pi, i\rangle \models \phi_1 \text{ and } \langle \pi, i\rangle \models \phi_2 \\
&\Leftrightarrow \langle p_t(\pi), i\rangle \models p_f(\phi_1) \text{ and } \langle p_t(\pi), i\rangle \models p_f(\phi_2) \\
&\Leftrightarrow \langle p_t(\pi), i\rangle \models p_f(\phi_1) \wedge p_f(\phi_2) \\
&\Leftrightarrow \langle p_t(\pi), i\rangle \models p_f(\phi_1 \wedge \phi_2)
\end{aligned}$$

- $\phi = \Diamond_P\phi'$
  Using the definition of $\models$, the inductive hypothesis, and Definition 5.2.4 (the definition of p_f) we have the following:

$$\langle \pi, i\rangle \models \Diamond_P\phi' \Leftrightarrow \langle \pi, j\rangle \models \phi' \text{ for some } 0 \leq j \leq i$$

$$\begin{aligned}
&\Leftrightarrow \langle p_t(\pi), j\rangle \models p_f(\phi') \text{ for the same } 0 \leq j \leq i \\
&\Leftrightarrow \langle p_t(\pi), i\rangle \models \Diamond_P p_f(\phi') \\
&\Leftrightarrow \langle p_t(\pi), i\rangle \models p_f(\Diamond_P \phi')
\end{aligned}$$

□

Note that often the principal names are arbitrary and the correctness of the protocol should not depend on them. In other words, the specification is usually insensitive to permuting the principal names. In this case the following corollary gives the correctness of the symmetry reduction.

Corollary 5.2.13 *Given the set of traces $Tr(\mathcal{P}, i, r)$, a formula ϕ, and a permutation on the names of the honest agents $p \in \mathcal{S}_\mathcal{P}$ such that $p_f(\phi) = \phi$ (the formula is symmetric in p), then for any trace $\pi \in Tr(\mathcal{P}, i, r)$, $\pi \models \phi$ if and only if $p_t(\pi) \models \phi$.*

Proof: Let $l = \mathit{length}(\pi)$. Using the definition of $\models$, Theorem 5.2.12, and the fact that $p_f(\phi) = \phi$ we have the following:

$$\begin{aligned}
\pi \models \phi &\Leftrightarrow \langle \pi, i\rangle \models \phi \text{ for all } 0 \leq i \leq l \\
&\Leftrightarrow \langle p_t(\pi), i\rangle \models p_f(\phi) \text{ for all } 0 \leq i \leq l \\
&\Leftrightarrow p_t(\pi) \models p_f(\phi) \\
&\Leftrightarrow p_t(\pi) \models \phi
\end{aligned}$$

□

In practice this allows one to perform the following reduction. Assume a model has i initiators and r responders, and $\mathcal{P}$ is the set of honest agent names. Then the model has $Tr(\mathcal{P}, i, r)$ as its set of traces. Assume the formula to be verified is ϕ. Let $G \subseteq \mathcal{S}_\mathcal{P}$ be the subset of permutations on the honest agents that preserve the formula ϕ. In other words, let $G = \{\, p \in \mathcal{S}_\mathcal{P} \mid p_f(\phi) = \phi \,\}$. If G is a group (proven below in Theorem 5.2.15) then by Theorem 5.1.2, G partitions $Tr(\mathcal{P}, i, r)$ into equivalence classes. Furthermore, these equivalence classes are a congruence with respect to satisfying the formula ϕ. In other words, for any $\pi \in Tr(\mathcal{P}, i, r)$, $\pi \models \phi$ if and only if $\pi' \models \phi$ for every $\pi' \in G(\pi)$. This means one need only check one representative from each equivalence class. Each equivalence class has size $|G|$. Thus, the amount of model checking performed is reduced by a factor of $|G|$. In the case when the formula is completely symmetric with respect to the names of the honest agents, then $G = \mathcal{S}_\mathcal{P}$ and the number of traces checked is reduced by a factor of $|\mathcal{P}|!$.

Lemma 5.2.14 *Let $p, q \in \mathcal{S}_{\mathcal{P}}$. Then $(pq)_f = p_f \circ q_f$.*

Proof: The equality is proven pointwise. In other words, we show

$$(pq)_f(\phi) = (p_f \circ q_f)(\phi)$$

for all formulas ϕ by induction on the structure of ϕ.

- $\phi = P$, where $P \in \mathcal{P}$
  In this case, $p_f(P) = p_{\mathcal{M}}(P) = p(P)$ for all $p \in \mathcal{S}_{\mathcal{P}}$, so

$$\begin{aligned}(pq)_f(P) &= (pq)(P)\\ &= p(q(P))\\ &= p(q_f(P))\\ &= p_f(q_f(P))\\ &= (p_f \circ q_f)(P).\end{aligned}$$

- $\phi = a$, where $a \in \mathcal{A} - \mathcal{P}$
  In this case, $p_f(a) = p_{\mathcal{M}}(a) = a$ for all $p \in \mathcal{S}_{\mathcal{P}}$, so

$$\begin{aligned}(pq)_f(a) &= a\\ &= q_f(a)\\ &= p_f(q_f(a))\\ &= (p_f \circ q_f)(a).\end{aligned}$$

- $\phi = H_i.v$
  In this case, $p_f(H_i.v) = H_i.v$ for all $p \in \mathcal{S}_{\mathcal{P}}$, so

$$\begin{aligned}(pq)_f(H_i.v) &= H_i.v\\ &= p_f(H_i.v)\\ &= p_f(q_f(H_i.v)\\ &= (p_f \circ q_f)(H_i.v).\end{aligned}$$

- $\phi = m_1 \cdot m_2$
  In this case, we can use the inductive hypothesis, so

$$\begin{aligned}(pq)_f(m_1 \cdot m_2) &= (pq)_f(m_1) \cdot (pq)_f(m_2)\\ &= (p_f \circ q_f)(m_1) \cdot (p_f \circ q_f)(m_2)\\ &= (p_f \circ q_f)(m_1 \cdot m_2).\end{aligned}$$

- All other cases consist of a straightforward use of the induction hypothesis like the $\phi = m_1 \cdot m_2$ case above.

Theorem 5.2.15 *Let ϕ be a formula and let G be the subset of permutations in $\mathcal{S}_\mathcal{P}$ that preserve the formula ϕ. In other words*

$$G = \{\, p \in \mathcal{S}_\mathcal{P} \mid p_f(\phi) = \phi \,\}.$$

Then G itself is a group.

Proof: Since $\mathcal{S}_\mathcal{P}$ is finite, by Theorem 5.1.1, it is sufficient to show that G is closed under the group operation of composition. Let $p, q \in G$. Therefore, $p_f(\phi) = \phi$ and $q_f(\phi) = \phi$. Using the lemma just proved,

$$(pq)_f(\phi) = (p_f \circ q_f)(\phi) = p_f(q_f(\phi)) = p_f(\phi) = \phi.$$

Thus $pq \in G$ and G is closed. $\Box$

5.3 Symmetry on Instances

The second kind of symmetry inherent in my model arises because a model may contain multiple instances of the same principal. For example, principal A in an authentication protocol may try to initiate the protocol twice. The model will then have two distinct initiator instances for A, call them A_1 and A_2. The model could also have two distinct responder instances for B, call them B_1 and B_2. In one trace of this model, it might be the case that A_1 successfully authenticates with B_1, and A_2 successfully authenticates with B_2. Now each of A's instances are identical up to the name given to them (the instance ID). The same is true for B's instances. It should not matter which instance of A authenticates with which instance of B. In other words, there should be an "equivalent" trace in which A_1 and A_2 switch roles. That is, A_1 authenticates with B_2, and A_2 authenticates with B_1. They are "equivalent" in the sense that everything that A_1 did in the first trace, A_2 does in the second trace, and vice versa. Intuitively they should be equivalent because the protocol should not depend on which particular instance does what. The name or ID given an instance should have no affect on the behavior or the requirements of that instance. Clearly, in the start state, before any instance has performed any actions, all instances of the same principal performing the same role are identical. In what follows, we will formalize this notion by examining a permutation on the instances that will again partition the traces of a model into equivalence classes. First, we take a high level look at this symmetry by describing something similar to the configurations of Section 5.2.

5.3.1 Instance Configurations

Consider some initial state σ_0. It has a particular number of initiator instances for each honest agent and a particular number of responder instances for each honest agent. These numbers remain constant along any given trace. What is not determined is which initiator instances of a particular principal A will communicate with which responder instances of a particular instance B. We will try to visualize this using a graph.

Definition 5.3.1 *An* **instance configuration graph** *for a particular trace π, is a graph G_π where the set of vertices is the set of honest instance IDs in that trace. In other words, the nodes are labeled $\{H_1, \ldots, H_k\}$, where k is the number of honest instances in each state of the trace π. There is a directed edge from H_i to H_j iff $\mathcal{I}_j$ is a responder instance and it has received the first message sent by initiator instance $\mathcal{I}_i$.*

Note that each instance of an honest agent can be either an initiator instance or a responder instance, but not both. Therefore, no node corresponding to an honest agent can have both incoming edges and outgoing edges. Secondly, each honest responder instance can receive only one initial message so each honest node will have at most one directed edge into it. Thirdly, the adversary may duplicate a particular initial message and send it to multiple responders; therefore, initiators in the graph may have multiple edges coming out of them.

Example 5.3.1 *Figure 5.1 displays two different instance configuration graphs. Both graphs have seven vertices. As in previous sections, the instance with ID X_i refers to the i^{th} instance of principal X. There are three instances of principal A trying to initiate the protocol. They are labeled A_1, A_2, and A_3. There are three instances of principal B trying to respond to the protocol. They are labeled B_1, B_2, and B_3. Also present,*

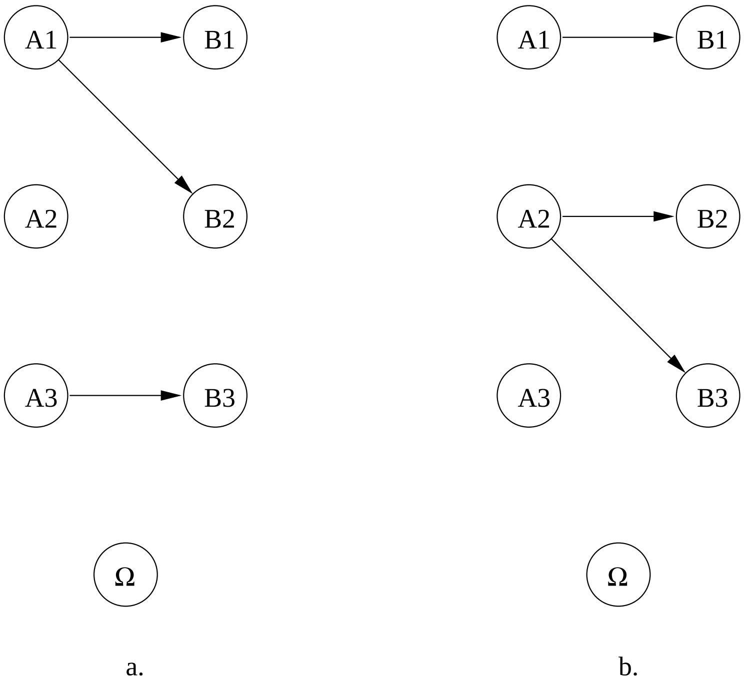

Figure 5.1: A pair of instance configuration graphs

is the instance of the adversary, Ω. In Figure 5.1.a, instance A_1's initial message has been duplicated by the adversary and received by both B_1 and B_2. Intuitively, B_1 and B_2 both believe they are responding to A_1. In addition, instance B_3 has received A_3's initial message. Note that there is no edge from A_2 since the adversary has prevented its initial message from being received by any of B's instances. In Figure 5.1.b, B_2 and B_3 receive duplicate initial messages, this time from A_2 while B_1 receives A_1's initial message.

I now define *safe permutations* on the vertices of an instance configuration graph (i.e., on instance IDs). Intuitively, a permutation is safe if it only permutes the instances of the same principal that have the same role. Two instances ($\mathcal{I}_i$ and $\mathcal{I}_j$) are instances of the same principal if the variable pr is bound to the same principal name in both instances (i.e., $B_i(pr) = B_j(pr)$). Two instances ($\mathcal{I}_i$ and $\mathcal{I}_j$) play the same roll in the protocol if, in the initial state, σ_0, the sequence of actions they take are equal (i.e., $P_i = P_j$). In addition, two instances of the same principal playing the same role must have the same set of known messages except for "freshly generated" messages (i.e., $I_i^\flat = I_j^\flat$).

Before continuing, I should point out a slight abuse of notation in the discussion that follows. I will be using p to denote a permutation on the set of instance IDs $\{H_1, \ldots, H_k\}$. However, it is often convenient to consider it as a permutation on the indices $\{1, \ldots k\}$ in the obvious way. In other words, for $i, j \in \{1, \ldots, k\}, p(i) = j$ whenever $p(H_i) = H_j$.

Definition 5.3.2 *Let $\mathcal{S}_H$ be the set of all permutations on the set of honest instance IDs, $H = \{H_1, H_2, \ldots, H_k\}$. We say that $p \in \mathcal{S}_H$ is a* safe permutation *with respect to a trace π if in the initial state of π*

- *$B_i(pr) = B_{p(i)}(pr)$, and*
- *$P_i = P_{p(i)}$.*

A permutation $p \in \mathcal{S}_H$ results in a relabeling of the configuration graph. This relabeling respects the names of the principals. Instance $\mathcal{I}_i$ and instance $\mathcal{I}_{p(i)}$ are both instances of the same principal. And of course, the new graph is isomorphic to the original graph.

Example 5.3.2 *Again, Figure 5.1 serves as an example. In this case, the graph in part a becomes the graph in part b via the following permutation on the labels.*

$$p = \begin{pmatrix} A_1 & A_2 & A_3 & B_1 & B_2 & B_3 \\ A_2 & A_3 & A_1 & B_2 & B_3 & B_1 \end{pmatrix}$$

Theorem 5.3.1 *Let $\mathcal{S}_{\overline{H}}$ be the set of safe permutations with respect to trace π. Then $\mathcal{S}_{\overline{H}}$ is a group.*

Proof:

- Closure: Let $p, q \in \mathcal{S}_{\overline{H}}$. Clearly $p \circ q$ is a permutation. To see that $p \circ q$ is safe, notice p and q are safe, therefore:
  - $B_i(\mathrm{pr}) = B_{q(i)}(\mathrm{pr}) = B_{p(q(i))}(\mathrm{pr}) = B_{(p\circ q)(i)}(\mathrm{pr})$, and
  - $P_i = P_{q(i)} = P_{p(q(i))} = P_{(p\circ q)(i)}$.
- Associativity: This follows because function composition is associative.
- Identity: Clearly the identity permutation is safe, therefore $\iota \in \mathcal{S}_{\overline{H}}$.
- Inverse: Since $p \in \mathcal{S}_{\overline{H}}$ is permutation, so is it's inverse p^{-1}. To see that p^{-1} is safe, let $p^{-1}(i) = j$. But this implies $p(j) = i$. Since p is safe we have
  - $B_i(\mathrm{pr}) = B_{p(j)}(\mathrm{pr}) = B_j(\mathrm{pr}) = B_{p^{-1}(i)}(\mathrm{pr})$, and
  - $P_i = P_{p(j)} = P_j = P_{p^{-1}(i)}$.

  Therefore, p^{-1} is safe.

□

Note that the requirements for a permutation to be safe with respect to a trace refer only to the initial state. Therefore, if a permutation p is safe with respect to a trace with initial state σ_0, then p is safe with respect to *any* trace with initial state σ_0.

Theorem 5.3.2 *Let $\Delta(\sigma_0)$ be the set of all possible instance configuration graphs for all traces starting in initial state σ_0. Let $\mathcal{S}_{\overline{H}}$ be the set of all safe permutations on the instances of σ_0 (vertices on the graph). For any graph $G_\pi \in \Delta(\sigma_0)$ and any permutation $p \in \mathcal{S}_{\overline{H}}$, $p(G_\pi) \in \Delta(\sigma_0)$.*

Proof: A formal proof of this theorem would require us to prove that there exists another trace π' starting from state σ_0 for which $p(G_\pi) = G_{\pi'}$. Since this is precisely what we want to avoid with this high level discussion, I do not give a formal proof here. Instead here is an intuitive proof.

The permutation does not change the number of edges or the number of vertices in the graph. The relabeling maps initiator instances of principal X to other initiator instances of principal X and responder instances of X to other responder instances of X. So the principals are still doing the same thing; however, which specific instance does what may have changed. Intuitively, such a symmetric trace must exist (i.e., one in which each instance $\mathcal{I}_{p(i)}$ in the trace π' behaves identically to the instance $\mathcal{I}_i$ in the trace π). Therefore, $p(G_\pi)$ is a valid instance configuration graph for some trace starting from state σ_0.

Corollary 5.3.3 *Let $\Delta(\sigma_0)$ be the set of all possible instance configuration graphs of traces starting from the initial state σ_0. Let $\mathcal{S}_{\overline{H}}$ be the set of safe permutations on the instances of σ_0 (vertices on the graph). Then for any $p \in \mathcal{S}_{\overline{H}}$, $\Delta(\sigma_0) = p(\Delta(\sigma_0)) = \{\, p(G) \mid G \in \Delta(\sigma_0) \,\}$.*

Proof:

$p(\Delta(\sigma_0) \subseteq \Delta(\sigma_0)$

> Let $G' \in p(\Delta(\sigma_0))$. Then $G' = p(G)$ for some $G \in \Delta(\sigma_0)$. By Theorem 5.3.2, $p(G) \in \Delta(\sigma_0)$. Therefore $G' \in \Delta(\sigma_0)$.

$\Delta(\sigma_0) \subseteq p(\Delta(\sigma_0))$

> Let $G \in \Delta(\sigma_0)$. Since $p \in \mathcal{S}_{\overline{H}}$ and $\mathcal{S}_{\overline{H}}$ is a group (Theorem 5.3.1), $p^{-1} \in \mathcal{S}_{\overline{H}}$. Let $G' = p^{-1}(G)$. By Theorem 5.3.2, $G' \in \Delta(\sigma_0)$. Therefore $G = p(p^{-1}(G)) = p(G') \in p(\Delta(\sigma_0))$.

□

Again, by Theorem 5.1.2, the group $\mathcal{S}_{\overline{H}}$ partitions the set of instance configuration graphs, $\Delta(\sigma_0)$, into equivalence classes. Two graphs are equivalent when there is a particular safe permutation $p \in \mathcal{S}_{\overline{H}}$ that is the isomorphism for the two graphs. When two instance configuration graphs are isomorphic, the same relationships exist between instances; only the labels or names of the instances have changed. If the specification does not distinguish between the different instances of the same principal (which it definitely should not) then BRUTUS need analyze only one representative from each equivalence class of instance configuration graphs. Each orbit has size roughly

$$|\mathcal{S}_{\overline{H}}| = \prod_{X \in \mathcal{P}} [\mathcal{C}_k(X)!]$$

where the $\mathcal{C}_k$'s are the configuration functions for each role in the protocol.

5.3.2 Instance Traces

The instance configuration graphs capture the relationships between the instances and help to illustrate the inherent symmetry in my models. To prove formally that it is safe to exploit this symmetry, one must analyze the traces of the system and verify that symmetric traces satisfy the same set of formulas. Again, I begin by extending $p \in \mathcal{S}_{\overline{H}}$ to the set of messages templates. As was the case for principal symmetry, permuting a compound message is done by permuting its components. The only interesting part is how to permute atomic messages. We have to be careful because when instance IDs are permuted all private information such as freshly generated nonces and session keys must also be permuted.

Definition 5.3.3 *Let $p \in \mathcal{S}_{\overline{H}}$ be a safe permutation on the instances of some trace π. Let $\sigma_0 = \{\Omega, \mathcal{I}_1, \ldots \mathcal{I}_k\}$ be the initial state of π. Let B_i^0 be the value of B_i in the state σ_0 (the initial bindings for instance $\mathcal{I}_i$). Then p induces a permutation $p_{\mathcal{M}}$ on the set of messages and message templates as follows:*

- $p_{\mathcal{M}}(a) = \begin{cases} B^0_{p(i)}(v) & \text{if } a = B^0_i(v) \text{ for some variable } v \\ & \text{and instance } \mathcal{I}_i \\ a & \text{otherwise} \end{cases}$
  for all atomic messages a.
- $p_{\mathcal{M}}(v) = v$ *for all variables v.*
- $p_{\mathcal{M}}(m_1 \cdot m_2) = p_{\mathcal{M}}(m_1) \cdot p_{\mathcal{M}}(m_2)$. *In other words, $p_{\mathcal{M}}$ works on a concatenated message by working on the components.*
- $p_{\mathcal{M}}(\{m\}_k) = \{p_{\mathcal{M}}(m)\}_{p_{\mathcal{M}}(k)}$. *In other words, $p_{\mathcal{M}}$ works on an encrypted message by working on the plaintext and on the key.*

This definition is a straightforward inductive definition on compound messages, but the definition on atomic messages requires some explanation. Instances may begin with some "private" information such as fresh nonces or session keys. When swapping instance $\mathcal{I}_i$ with instance $\mathcal{I}_j$ one must also swap occurrences of this kind of private information. For example, assume $\mathcal{I}_i$ has a fresh nonce n_i. If $\mathcal{I}_i$ is swapped with instance $\mathcal{I}_j$, then $\mathcal{I}_j$ is an instance of the same principal playing the same role and therefore it has a corresponding fresh nonce n_j. Since the intention is to swap the behaviors of $\mathcal{I}_i$ and $\mathcal{I}_j$ in the trace, these nonces must also be swapped in any messages appearing in a trace. In other words, if the original trace has an action consisting of instance $\mathcal{I}_i$ sending nonce n_i, then in the permuted trace, the corresponding action would be instance $\mathcal{I}_j$ sending nonce n_j.

It is important to note that $p_{\mathcal{M}}(m)$ is well defined since this is not clear from the definition. In particular, it is not clear that $p_{\mathcal{M}}(a)$ is well defined for atomic messages a. We need to consider the case when $a = B_i^0(x)$ and $a = B_j^0(y)$ for $i \neq j$ or $x \neq y$ because then there may be more than one value for $p_{\mathcal{M}}(a)$ (namely $B_{p(i)}^0(x)$ and $B_{p(j)}^0(y)$). Recall, however, that the initial bindings B^0 contain freshly generated messages (see Definition 3.1.1). Freshly generated messages cannot be equal to one another. This is the assumption that nonces and session keys are "fresh". The symmetry reduction is restricted to models that have this property. If this is the case, the only initial bindings that are not newly generated messages are the bindings to pr. In other words, $B_i^0(x) = B_j^0(y)$ implies $(x = y \wedge i = j)$ or $(x = y = pr)$. If $x = y$ and $i = j$ then $B_{p(i)}^0(x) = B_{p(j)}^0(y)$ and $p_{\mathcal{M}}(a)$ is well defined. If instead $x = y = pr$, then $B_i(pr) = B_{p(i)}(pr)$ and $B_j(pr) = B_{p(j)}(pr)$ because p is safe. Therefore, $B_i(pr) = B_j(pr) = a$ implies $B_{p(i)}(pr) = B_{p(j)}(pr) = a$, and $p_{\mathcal{M}}(m)$ is also well defined in this case.

There is an additional restriction. If $pubkey(X) = B_i^0(u)$ for some variable u then there exists a variable v such that $privkey(X) = B_i^0(v)$ and vice versa. This is a reasonable restriction because $pubkey(X) = B_i^0(u)$ or $privkey(X) = B_i^0(v)$ for some instance $\mathcal{I}_i$ implies that this public key pair is freshly generated. If a public key pair is freshly generated, then there should be only one instance that initially knows either the public key or the private key, and that instance should know both since it generated the public key pair. We shall see that this restriction is necessary to ensure that $p_{\mathcal{M}}(k^{-1}) = [p_{\mathcal{M}}(k)]^{-1}$ for public-private key pairs (Lemma 5.3.7).

As was the case with principal symmetry, $p_{\mathcal{M}}$ is also a permutation on the set of atomic messages $\mathcal{A}$. In other words, $p_{\mathcal{M}}(a) \in \mathcal{A}$ for all $a \in \mathcal{A}$. In addition, it is clear from the definition of $p_{\mathcal{M}}$, that the inverse of an extension of a permutation p is equal to the extension of the inverse permutation. In other words $(p_{\mathcal{M}})^{-1} = (p^{-1})_{\mathcal{M}}$.

Definition 5.3.4 *Let $p \in \mathcal{S}_{\overline{H}}$ be a permutation on the instances of some trace π and let $p_{\mathcal{M}}$ be the extension of p to the set of message templates. Then p is extended to actions, instances, states, and traces as follows.*

- *p_a acts on actions by applying p to the instance label and $p_{\mathcal{M}}$ to the message argument of the action. More formally,*

$$p_a(H_i \cdot A \cdot M) = p(H_i) \cdot A \cdot p_{\mathcal{M}}(M).$$

- *p_i acts on instances, again by applying $p_{\mathcal{M}}$ to all messages appearing in the instances and by applying p to the instance ID. More formally, let $\mathcal{I}_i = \langle H_i, B_i, I_i, P_i \rangle$. Then*

$$p_i(\mathcal{I}_i) = \langle p(H_i), p_{\mathcal{M}} \circ B_i, p_{\mathcal{M}}(I_i), P_i \rangle.$$

  Again $p_{\mathcal{M}}(I_i)$ is an abuse of notation for $\{\, p_{\mathcal{M}}(m) \mid m \in I_i \,\}$. Also, it is not necessary to permute the process description P_i because p is safe, which ensures that $P_i = P_{p(i)}$.

- *p_s acts on states by applying p_i to all the instances. More formally,*

$$p_s(\{\Omega, \mathcal{I}_1, \ldots, \mathcal{I}_k\}) = \{p_i(\Omega), p_i(\mathcal{I}_1), \ldots, p_i(\mathcal{I}_n)\}.$$

- *p_t acts on a trace by applying p_s to all the states in the trace and p_a to all the actions appearing in the trace. More formally,*

$$p_t(\sigma_0\alpha_1\sigma_1\alpha_1 \cdots \alpha_n\sigma_n) = p_s(\sigma_0)p_a(\alpha_1)p_s(\sigma_1)p_a(\alpha_1)\cdots p_a(\alpha_n)p_s(\sigma_n).$$

As in the case of $p_{\mathcal{M}}$, the inverse of the extension of p is the extension of the inverse of p. Namely,

$$\begin{aligned} (p_a)^{-1} &= (p^{-1})_a, \\ (p_i)^{-1} &= (p^{-1})_i, \\ (p_s)^{-1} &= (p^{-1})_s, \text{ and} \\ (p_t)^{-1} &= (p^{-1})_t. \end{aligned}$$

Safe permutations have an interesting and important property. They map the initial state to itself. This is important because our models have a unique initial state and all traces must start from that state. Therefore, a permuted trace would also have to start from that same initial state. This follows from the following theorem.

Theorem 5.3.4 *Let π be a trace with initial state σ_0 and p be a safe permutation with respect to π. Then $p_i(\mathcal{I}_j) = \mathcal{I}_{p(j)}$ for all instances $\mathcal{I}_j \in \sigma_0$.*

Proof: In order to show that

$$p_i(\mathcal{I}_j) = p_i(\langle H_j, B_j, I_j, P_j\rangle) = \langle p(H_j), p_{\mathcal{M}} \circ B_j, p_{\mathcal{M}}(I_j), P_j\rangle$$

is equal to $\mathcal{I}_{p(j)} = \langle H_{p(j)}, B_{p(j)}, I_{p(j)}, P_{p(j)}\rangle$ we will show that the components are equal.

$p(H_j) = H_{p(j)}$

We know this from the definition of p.

$p_{\mathcal{M}} \circ B_j = B_{p(j)}$

Recall that we are dealing with the initial state σ_0 and so $B_j = B_j^0$ and $B_{p(j)} = B_{p(j)}^0$. Since these initial bindings are all atomic (nonces, keys, etc.), it is clear from Definition 5.3.3 that $p_{\mathcal{M}} \circ B_j^0 = B_{p(j)}^0$.

$p_{\mathcal{M}}(I_j) = I_{p(j)}$

Recall that in the initial state we have $I_j^0 = I_j^\sharp \cup I_j^\flat$.

- Let us first consider $I_j^\flat$. Since p is safe, $\mathcal{I}_j$ and $\mathcal{I}_{p(j)}$ are instances of the same principal and so, except for freshly generated messages, their initial knowledge is the same (i.e., $I_j^\flat = I_{p(j)}^\flat$). Since the messages in $I_j^\flat$ are not freshly generated, Definition 5.3.3 gives us $p_{\mathcal{M}}(I_j^\flat) = I_j^\flat$. Therefore $p_{\mathcal{M}}(I_j^\flat) = I_j^\flat = I_{p(j)}^\flat$.
- Now we consider $I_j^\sharp$. Recall that

$$I_j^\sharp = B_j^0(\mathcal{V} - \{pr\})$$

Since p is safe, $\mathcal{I}_j$ and $\mathcal{I}_{p(j)}$ are playing the same role and they have the same initial set of variables and so there is a corresponding set of bindings $B^0_{p(j)}$ for instance $\mathcal{I}_{p(j)}$ such that

$$I^\sharp_{p(j)} = B^0_{p_j}(\mathcal{V} - \{pr\})$$

Therefore, by Definition 5.3.3 $p_\mathcal{M}(I^\sharp_j) = I^\sharp_{p(j)}$.

Combining these two results, we get

$$\begin{aligned} p_\mathcal{M}(I^0_j) &= p_\mathcal{M}(I^\sharp_j \cup I^\flat_j) \\ &= p_\mathcal{M}(I^\sharp_j) \cup p_\mathcal{M}(I^\flat_j) \\ &= I^\sharp_{p(j)} \cup I^\flat_{p(j)} \\ &= I^0_{p(j)} \end{aligned}$$

$P_j = P_{p(j)}$

This is trivial because p is safe and the process descriptions P_j and $P_{p(j)}$ are identical in the initial state σ_0.

□

Corollary 5.3.5 *Let π be a trace with initial state σ_0 and let p be a safe permutation with respect to π. Then $p_s(\sigma_0) = \sigma_0$.*

Proof: Theorem 5.3.4 gives us that p_i is a permutation on the instances of the initial state σ_0. Therefore,

$$\begin{aligned} p_s(\sigma_0) &= p_i(\{\, \mathcal{I}_j \mid \mathcal{I}_j \in \sigma_0 \,\}) \\ &= \{\, p_i(\mathcal{I}_j) \mid \mathcal{I}_j \in \sigma_0 \,\} \\ &= \{\, \mathcal{I}_j \mid \mathcal{I}_j \in \sigma_0 \,\} \\ &= \sigma_0 \end{aligned}$$

□

I now show that the set of all traces starting from some initial state σ_0 is symmetric with respect to the safe permutations on the instances of σ_0. I begin with a couple of lemmas.

Lemma 5.3.6 *Let $p \in \mathcal{S}_{\overline{H}}$ be a safe permutation on the instances of some trace and let $p_{\mathcal{M}}$ be the extension of p to message templates. Let B be a set of variable bindings. Let $\overline{B}$ be the extension of B to message templates. Then $p_{\mathcal{M}}(\overline{B}(m)) = \overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(m))$ for all message templates m whose variables are in the domain of B.*

The intuition is straightforward. If we take a message template m, apply the bindings B to the variables and then permute the message with $p_{\mathcal{M}}$ [i.e., $p_{\mathcal{M}}(\overline{B}(m))$] the result should be the same as if we first permute the message template with $p_{\mathcal{M}}$ (leaving the variables alone), then we apply the permuted bindings to the variables [i.e., $\overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(m))$]. The formal proof is below.

Proof: (by induction on the structure of m)

- $m = a \in \mathcal{A}$

$$\begin{array}{rcll} p_{\mathcal{M}}(\overline{B}(m)) & = & p_{\mathcal{M}}(\overline{B}(a)) & \\ & = & p_{\mathcal{M}}(a) & \text{Definition 3.1.2} \\ & = & \overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(a)) & \text{Definition 3.1.2 and } p_{\mathcal{M}}(a) \in \mathcal{A} \\ & = & \overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(m)) & \end{array}$$

- $m = v \in \mathcal{V}$

$$\begin{array}{rcll} p_{\mathcal{M}}(\overline{B}(m)) & = & p_{\mathcal{M}}(\overline{B}(v)) & \\ & = & p_{\mathcal{M}}(B(v)) & \text{Definition 3.1.2} \\ & = & (p_{\mathcal{M}} \circ B)(v) & \\ & = & \overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(v)) & \text{Definition 3.1.2} \\ & = & \overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(m)) & \end{array}$$

- $m = m_1 \cdot m_2$

$$\begin{array}{rcll} p_{\mathcal{M}}(\overline{B}(m)) & = & p_{\mathcal{M}}(\overline{B}(m_1 \cdot m_2)) & \\ & = & p_{\mathcal{M}}(\overline{B}(m_1) \cdot \overline{B}(m_2)) & \text{Definition 3.1.2} \\ & = & p_{\mathcal{M}}(\overline{B}(m_1)) \cdot p_{\mathcal{M}}(\overline{B}(m_2)) & \text{Definition 5.3.3} \\ & = & \overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(m_1)) \cdot \overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(m_2)) & \text{ind. hyp.} \\ & = & \overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(m_1) \cdot p_{\mathcal{M}}(m_2)) & \text{Definition 3.1.2} \\ & = & \overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(m_1 \cdot m_2)) & \text{Definition 5.3.3} \\ & = & \overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(m)) & \end{array}$$

- $m = \{m'\}_k$

$$\begin{array}{rcll} p_{\mathcal{M}}(\overline{B}(m)) & = & p_{\mathcal{M}}(\overline{B}(\{m'\}_k)) & \\ & = & p_{\mathcal{M}}(\{\overline{B}(m')\}_{\overline{B}(k)}) & \text{Definition 3.1.2} \\ & = & \{p_{\mathcal{M}}(\overline{B}(m'))\}_{p_{\mathcal{M}}(\overline{B}(k))} & \text{Definition 5.3.3} \\ & = & \{\overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(m'))\}_{\overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(k))} & \text{ind. hyp.} \\ & = & \overline{p_{\mathcal{M}} \circ B}(\{p_{\mathcal{M}}(m')\}_{p_{\mathcal{M}}(k)}) & \text{Definition 3.1.2} \\ & = & \overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(\{m'\}_k)) & \text{Definition 5.3.3} \\ & = & \overline{p_{\mathcal{M}} \circ B}(p_{\mathcal{M}}(m)) & \end{array}$$

□

Lemma 5.3.7 *Let $k \in \mathcal{K}$ be a key. Let $p \in \mathcal{S}_{\overline{H}}$ be a safe permutation on the instances of some trace and let $p_{\mathcal{M}}$ be the extension of p to message templates. Then the inverse of a permuted key is the same as the permuted inverse key. More formally, $p_{\mathcal{M}}(k^{-1}) = (p_{\mathcal{M}}(k))^{-1}$.*

Proof: If k is not freshly generated then $k \neq B_i^0(x)$ for any instance $\mathcal{I}_i$ or any variable x. Since k is atomic, then by Definition 5.3.3 $p_{\mathcal{M}}(k) = k$ and $p_{\mathcal{M}}(k^{-1}) = k^{-1}$. Therefore,

$$p_{\mathcal{M}}(k^{-1}) = k^{-1} = (p_{\mathcal{M}}(k))^{-1}.$$

If k is freshly generated, then there is exactly one instance $\mathcal{I}_i$ such that $B_i^0(x) = k$. That same instance must know the inverse key, so $B_i^0(y) = k^{-1}$ for some variable y. Note that if k is a symmetric key, then $x = y$. Now $p_{\mathcal{M}}(k) = B_{p(i)}^0(x)$ and $p_{\mathcal{M}}(k^{-1}) = B_{p(i)}^0(y)$. Since $\mathcal{I}_{p(i)}$ is an instance of the same role, $B_{p(i)}^0(x)$ and $B_{p(i)}^0(y)$ must also be inverses. Therefore,

$$\begin{array}{rcll} p_{\mathcal{M}}(k^{-1}) & = & p_{\mathcal{M}}(B_i^0(y)) & \text{for some variable } y \text{ and instance } \mathcal{I}_i \\ & = & B_{p(i)}^0(y) & \text{for the same variable } y \\ & = & (B_{p(i)}^0(x))^{-1} & \text{for some variable } x \\ & = & (p_{\mathcal{M}}(B_i^0(x)))^{-1} & \text{for the same variable } x \\ & = & (p_{\mathcal{M}}(k))^{-1}. & \end{array}$$

□

Lemma 5.3.8 *Let T be a derivation tree for m with assumptions in A. Let $p \in \mathcal{S}_{\overline{H}}$ be a safe permutation on the instances of some trace and let $p_{\mathcal{M}}$ be the extension of p to the set of message templates. Applying $p_{\mathcal{M}}$ to every message appearing in T, results in a new derivation tree $p_{\mathcal{M}}(T)$ for $p_{\mathcal{M}}(m)$ with assumptions in $p_{\mathcal{M}}(A)$.*

Proof: (by induction on the height of the derivation tree T)

- Assume T has height 0 and hence is atomic.
  Then T consists of the single message m and $m \in A$. This means that $p_{\mathcal{M}}(T)$ consists of the single message $p_{\mathcal{M}}(m)$, and that $p_{\mathcal{M}}(m) \in p_{\mathcal{M}}(A)$.

- Assume T is a derivation tree of height n with assumptions in A, and that it has the following form:

  $$\cfrac{\cfrac{\Sigma}{m_1 \cdot m_2}}{m_i}$$

  Then the subtree above m_i has height $n - 1$. By the inductive hypothesis,

  $$\frac{p_{\mathcal{M}}(\Sigma)}{p_{\mathcal{M}}(m_1 \cdot m_2)}$$

  is a derivation tree for $p_{\mathcal{M}}(m_1 \cdot m_2)$ with assumptions in $p_{\mathcal{M}}(A)$. Now by Definition 5.3.3, $p_{\mathcal{M}}(m_1 \cdot m_2) = p_{\mathcal{M}}(m_1) \cdot p_{\mathcal{M}}(m_2)$. Clearly,

  $$\frac{p_{\mathcal{M}}(m_1) \cdot p_{\mathcal{M}}(m_2)}{p_{\mathcal{M}}(m_i)}$$

  is a valid inference. Putting these together results in a derivation tree for $p_{\mathcal{M}}(m_i)$ with assumptions in $p_{\mathcal{M}}(A)$.

- Assume T is a derivation tree of height n with assumptions in A, and that it has the following form:

  $$\cfrac{\cfrac{\Sigma_1}{m_1} \qquad \cfrac{\Sigma_2}{m_2}}{m_1 \cdot m_2}$$

  Then the subtrees above $m_1 \cdot m_2$ have height at most $n - 1$. By the inductive hypothesis,

  $$\frac{p_{\mathcal{M}}(\Sigma_i)}{p_{\mathcal{M}}(m_i)}$$

is a derivation tree for $p_{\mathcal{M}}(m_i)$ with assumptions in $p_{\mathcal{M}}(A)$. Clearly,

$$\frac{p_{\mathcal{M}}(m_1) \qquad p_{\mathcal{M}}(m_2)}{p_{\mathcal{M}}(m_1) \cdot p_{\mathcal{M}}(m_2)}$$

is a valid inference. Putting these together results in a derivation tree for $p_{\mathcal{M}}(m_1) \cdot p_{\mathcal{M}}(m_2) = p_{\mathcal{M}}(m_1 \cdot m_2)$ with assumptions in $p_{\mathcal{M}}(A)$.

- Assume T is a derivation tree of height n with assumptions in A, and that it has the following form:

$$\frac{\dfrac{\Sigma_1}{\{m\}_k} \qquad \dfrac{\Sigma_2}{k^{-1}}}{m}$$

Then the subtrees above m have height at most $n-1$. By the inductive hypothesis,

$$\frac{p_{\mathcal{M}}(\Sigma_1)}{p_{\mathcal{M}}(\{m\}_k)} \qquad \text{and} \qquad \frac{p_{\mathcal{M}}(\Sigma_2)}{p_{\mathcal{M}}(k^{-1})}$$

are derivation trees for $p_{\mathcal{M}}(\{m\}_k)$ and $p_{\mathcal{M}}(k^{-1})$ with assumptions in $p_{\mathcal{M}}(A)$. Now by Definition 5.3.3, $p_{\mathcal{M}}(\{m\}_k) = \{p_{\mathcal{M}}(m)\}_{p_{\mathcal{M}}(k)}$ and by Lemma 5.3.7, $p_{\mathcal{M}}(k^{-1}) = (p_{\mathcal{M}}(k))^{-1}$. Clearly,

$$\frac{\{p_{\mathcal{M}}(m)\}_{p_{\mathcal{M}}(k)} \qquad (p_{\mathcal{M}}(k))^{-1}}{p_{\mathcal{M}}(m)}$$

is a valid inference. Putting these together results in a derivation tree for $p_{\mathcal{M}}(m)$ with assumptions in $p_{\mathcal{M}}(A)$.

- Assume T is a derivation tree of height n with assumptions in A, and that it has the following form:

$$\frac{\dfrac{\Sigma_1}{m} \qquad \dfrac{\Sigma_2}{k}}{\{m\}_k}$$

Then the subtrees above $\{m\}_k$ have height at most $n-1$. By the inductive hypothesis,

$$\frac{p_{\mathcal{M}}(\Sigma_1)}{p_{\mathcal{M}}(m)} \qquad \text{and} \qquad \frac{p_{\mathcal{M}}(\Sigma_2)}{p_{\mathcal{M}}(k)}$$

are derivation trees for $p_{\mathcal{M}}(m)$ and $p_{\mathcal{M}}(k)$ with assumptions in $p_{\mathcal{M}}(A)$. Clearly,

$$\frac{p_{\mathcal{M}}(m) \qquad p_{\mathcal{M}}(k)}{\{p_{\mathcal{M}}(m)\}_{p_{\mathcal{M}}(k)}}$$

is a valid inference. Putting these together results in a derivation tree for $\{p_{\mathcal{M}}(m)\}_{p_{\mathcal{M}}(k)} = p_{\mathcal{M}}(\{m\}_k)$ with assumptions in $p_{\mathcal{M}}(A)$.

□

Lemma 5.3.9 *Let $p \in \mathcal{S}_{\overline{H}}$ be a safe permutation on the instances. Let $p_{\mathcal{M}}$, p_a, and p_s be the extensions of p to message templates, actions, and states respectively. If $\sigma \xrightarrow{H_i \cdot A \cdot m} \sigma'$ is a valid transition then so is $p_s(\sigma) \xrightarrow{p_a(H_i \cdot A \cdot m)} p_s(\sigma')$.*

Proof: It is clear from the definition of p_s that if σ and σ' are states, then so are $p_s(\sigma)$ and $p_s(\sigma')$. From the definition of p_a, if $H_i \cdot A \cdot m$ is an action, then so is $p_a(H_i \cdot A \cdot m) = p(H_i) \cdot A \cdot p_{\mathcal{M}}(m)$. It remains to show that each of the three different kinds of actions results in a valid transition.

- $H_i \cdot A \cdot m$ is a **send** action
  1. $I'_\Omega = I_\Omega \cup m$. (The adversary adds m to the set of messages it knows.)
  2. If in σ, $\mathcal{I}_i = \langle H_i, B_i, I_i, P_i \rangle$, and in σ', $\mathcal{I}'_i = \langle H'_i, B'_i, I'_i, P'_i \rangle$, then
     - $H'_i = H_i$ (The instance ID remains unchanged.)
     - $B'_i = B_i$ (The bindings remain unchanged.)
     - $I'_i = I_i$ (The set of information remains unchanged.)
     - $P_i =$ **send** $(s\text{-}msg) \cdot P'_i$ (The instance is ready to send a message, and that send action is removed from the process description in the new state.)

  - $m = \overline{B_i}(s\text{-}msg)$ (The message that the instance is ready to send and the actual message sent are the same.)

3. $\mathcal{I}_j = \mathcal{I}'_j$ for all $j \neq i$. (All other instances remain unchanged.)

We now need to show the corresponding properties for the permuted transition $p_s(\sigma) \stackrel{p(H_i)\cdot A\cdot p_{\mathcal{M}}(m)}{\longrightarrow} p_s(\sigma')$.

1. In $p_s(\sigma')$,

$$p_{\mathcal{M}}(I'_\Omega) = p_{\mathcal{M}}(I_\Omega \cup m) = p_{\mathcal{M}}(I_\Omega) \cup p_{\mathcal{M}}(m).$$

   (The adversary adds $p_{\mathcal{M}}(m)$ to the set of messages it knows.)

2. In $p_s(\sigma)$,

$$p_i(\mathcal{I}_i) = \langle p(H_i), p_{\mathcal{M}} \circ B_i, p_{\mathcal{M}}(I_i), p_p(P_i)\rangle,$$

   and in $p_s(\sigma')$,

$$p_i(\mathcal{I}'_i) = \langle p(H'_i), p_{\mathcal{M}} \circ B'_i, p_{\mathcal{M}}(I'_i), p_p(P'_i)\rangle,$$

   and

  - $H'_i = H_i$ so
$$p(H'_i) = p(H_i)$$
    (The instance ID remains unchanged.)
  - $B'_i = B_i$ so
$$p_{\mathcal{M}} \circ B'_i = p_{\mathcal{M}} \circ B_i$$
    (The bindings remain unchanged.)
  - $I'_i = I_i$ so
$$p_{\mathcal{M}}(I'_i) = p_{\mathcal{M}}(I_i)$$
    (The set of information remains unchanged.)
  - $P_i =$ **send** $(s\text{-}msg) \cdot P'_i$ so
$$p_p(P_i) = \underline{\textbf{send}}\ (p_{\mathcal{M}}(s\text{-}msg)) \cdot p_p(P'_i)$$
    (The instance is ready to send a message, and that send action is removed from the process description in the new state.)

- $m = \overline{B_i}(s\text{-}msg)$ so

$$p_{\mathcal{M}}(m) = p_{\mathcal{M}}(\overline{B_i}(s\text{-}msg)) = \overline{p_{\mathcal{M}} \circ B_i}(p_{\mathcal{M}}(s\text{-}msg))$$

by Lemma 5.3.6. (The message that the instance is ready to send and the actual sent message are the same.)

3. $\mathcal{I}_j = \mathcal{I}'_j$ for all $j \neq i$ so

$$p_i(\mathcal{I}_j) = p_i(\mathcal{I}'_j) \text{ for all } j \neq i.$$

(All other instances remain unchanged.)

- $H_i \cdot A \cdot m$ is a **receive** action

1. $m \in \overline{I_\Omega}$. (The adversary can generate the message m.)
2. If in σ, $\mathcal{I}_i = \langle H_i, B_i, I_i, P_i \rangle$, and in σ', $\mathcal{I}'_i = \langle H'_i, B'_i, I'_i, P'_i \rangle$, then
   - $H_i = H'_i$ (The instance ID remains unchanged.)
   - B'_i is the smallest extension of B_i such that $\overline{B'_i}(r\text{-}msg) = m$ (The bindings of the instance are updated correctly, and the message received matches the message template in the receive action.)
   - $I'_i = I_i \cup m$ (The information of the instance is updated correctly.)
   - $P_i =$ **receive** $(r\text{-}msg) \cdot P'_i$ (The instance is ready to receive a message, and that action is removed from the process description in the next state.)
3. $\mathcal{I}_j = \mathcal{I}'_j$ for all $j \neq i$. (All other instances remain unchanged.)

Again, we must show the corresponding properties for the permuted transition $p_s(\sigma) \overset{p(H_i) \cdot A \cdot p_{\mathcal{M}}(m)}{\longrightarrow} p_s(\sigma')$.

1. By Lemma 5.3.8, we can apply $p_{\mathcal{M}}$ to the derivation tree for $m \in \overline{I_\Omega}$ to obtain a derivation tree for $p_{\mathcal{M}}(m) \in \overline{p_{\mathcal{M}}(I_\Omega)}$. (The adversary can generate the message $p_{\mathcal{M}}(m)$.)
2. In $p_s(\sigma)$,

$$p_i(\mathcal{I}_i) = \langle p(H_i), p_{\mathcal{M}} \circ B_i, p_{\mathcal{M}}(I_i), p_p(P_i) \rangle,$$

and in $p_s(\sigma')$,

$$p_i(\mathcal{I}'_i) = \langle p(H'_i), p_{\mathcal{M}} \circ B'_i, p_{\mathcal{M}}(I'_i), p_p(P'_i)\rangle,$$

and

- $H_i = H'_i$ so
$$p(H_i) = p(H'_i).$$
(The instance ID remains unchanged.)
- B'_i is the smallest extension of B_i such that $\overline{B'_i}(r\text{-}msg) = m$ so using Lemma 5.3.6, $p_{\mathcal{M}} \circ B'_i$ is the smallest extension of $p_{\mathcal{M}} \circ B_i$ such that $\overline{p_{\mathcal{M}} \circ B'_i}(p_{\mathcal{M}}(r\text{-}msg)) = p_{\mathcal{M}}(m)$ (The bindings of the instance are updated correctly, and the message received matches the message template in the receive action.)
- $I'_i = I_i \cup m$ so
$$p_{\mathcal{M}}(I'_i) = p_{\mathcal{M}}(I_i \cup m) = p_{\mathcal{M}}(I_i) \cup p_{\mathcal{M}}(m)$$
(The information of the instance is updated correctly.)
- $P_i =$ **receive** $(r\text{-}msg) \cdot P'_i$ so
$$p_p(P_i) = \textbf{receive } (p_{\mathcal{M}}(r\text{-}msg)) \cdot p_p(P'_i)$$
(The instance is ready to receive a message, and that action is removed from the process description in the next state.)

3. $\mathcal{I}_j = \mathcal{I}'_j$ for all $j \neq i$ so
$$p_i(\mathcal{I}_j) = p_i(\mathcal{I}'_j) \text{ for all } j \neq i.$$
(All other instances remain unchanged.)

- $H_i \cdot A \cdot m$ is an internal action

 1. If in σ, $\mathcal{I}_i = \langle H_i, B_i, I_i, P_i\rangle$, and in σ', $\mathcal{I}'_i = \langle H'_i, B'_i, I'_i, P'_i\rangle$, then
   - $H'_i = H_i$ (The instance ID remains unchanged.)
   - $B'_i = B_i$ (The bindings remain unchanged.)
   - $I'_i = I_i$ (The set of information remains unchanged.)

  - $P_i = A(msg) \cdot P'_i$ (The instance is ready to perform action A, and that action is removed from the process description in the new state.)
  - $m = \overline{B_i}(msg)$. (The message argument in the process description and the actual message occurring in the taken action are the same.)

2. $\mathcal{I}_j = \mathcal{I}'_j$ for all $j \neq i$. (All other instances remain unchanged.)

We must show the corresponding properties for the permuted transition $p_s(\sigma) \stackrel{p(H_i) \cdot A \cdot p_{\mathcal{M}}(m)}{\longrightarrow} p_s(\sigma')$.

1. In $p_s(\sigma)$,

$$p_i(\mathcal{I}_i) = \langle p(H_i), p_{\mathcal{M}} \circ B_i, p_{\mathcal{M}}(I_i), p_p(P_i) \rangle,$$

and in $p_s(\sigma')$,

$$p_i(\mathcal{I}'_i) = \langle p(H'_i), p_{\mathcal{M}} \circ B'_i, p_{\mathcal{M}}(I'_i), p_p(P'_i) \rangle,$$

and

  - $H'_i = H_i$ so
$$p(H'_i) = p(H_i)$$
(The instance ID remains unchanged.)
  - $B'_i = B_i$ so
$$p_{\mathcal{M}} \circ B'_i = p_{\mathcal{M}} \circ B_i$$
(The bindings remain unchanged.)
  - $I'_i = I_i$ so
$$p_{\mathcal{M}}(I'_i) = p_{\mathcal{M}}(I_i)$$
(The set of information remains unchanged.)
  - $P_i = A(msg) \cdot P'_i$ so
$$p_p(P_i) = A(p_{\mathcal{M}}(s\text{-}msg)) \cdot p_p(P'_i)$$
(The instance is ready to perform action A, and that action is removed from the process description in the new state.)

– $m = \overline{B_i}(msg)$ so

$$p_{\mathcal{M}}(m) = p_{\mathcal{M}}(\overline{B_i}(msg)) = \overline{p_{\mathcal{M}} \circ B_i}(p_{\mathcal{M}}(msg))$$

by Lemma 5.3.6. (The message argument in the process description and the actual message occurring in the taken action are the same.)

2. $\mathcal{I}_j = \mathcal{I}'_j$ for all $j \neq i$ so

$$p_i(\mathcal{I}_j) = p_i(\mathcal{I}'_j) \text{ for all } j \neq i.$$

(All other instances remain unchanged.)

□

Theorem 5.3.10 *Let $Tr(\sigma_0)$ be the set of all possible traces with initial state σ_0 and let $\pi \in Tr(\sigma_0)$ be a trace. Let p be a safe permutation on the instances of σ_0, let p_s be the extension of p to states, and let p_t be the extension of p to traces. Then $p_t(\pi) \in Tr(\sigma_0)$.*

Proof: (by induction on the length of a trace)

- Let $\pi = \sigma_0$ be a trace of length 0. Then $p_t(\pi) = p_s(\sigma_0)$. Clearly, $p_s(\sigma_0)$ is a state as well, and so it is a trace of length 0. By Corollary 5.3.5, $p_s(\sigma_0) = \sigma_0$, and so $p_t(\pi)$ is a trace with initial state σ_0. Hence $p_t(\pi) \in Tr(\sigma_0)$.

- Let $\pi = \sigma_0\alpha_1\sigma_1\alpha_2 \cdots \alpha_{n+1}\sigma_{n+1}$ be a trace of length $n+1$ in $Tr(\sigma)$. Then $\pi_n = \sigma_0\alpha_1\sigma_1\alpha_2 \cdots \alpha_n\sigma_n$ is a trace of length n in $Tr(\sigma_0)$ and by the inductive hypothesis so is

$$p_t(\pi_n) = p_s(\sigma_0)p_a(\alpha_1)p_s(\sigma_1)p_a(\alpha_2) \cdots p_a(\alpha_n)p_s(\sigma_n).$$

From the trace π we see that $\sigma_n \xrightarrow{\alpha_{n+1}} \sigma_{n+1}$ is a valid transition. By Lemma 5.3.9, $p_s(\sigma_n) \xrightarrow{p_a(\alpha_{n+1})} p_s(\sigma_{n+1})$ is also a valid transition. Appending this to the end of $p_t(\pi_n)$ we get that $p_t(\pi)$ is a trace of length $n+1$ in $Tr(\sigma_0)$.

□

There is an intuitive relationship between instance configuration graphs and instance symmetries on traces. If p is a safe permutation with respect to a trace π, and G_π is an instance configuration graph for the trace π, then $p(G_\pi) = G_{p_t(\pi)}$. In other words, the permuted graph $p(G_\pi)$ is the configuration graph for the permuted trace $p_t(\pi)$. Since the purpose of the graphs is to provide an intuitive picture of the symmetries, no proof is provided.

Corollary 5.3.11 *Let $Tr(\sigma_0)$ be the set of all traces starting from the initial state σ_0. Let $\mathcal{S}_{\overline{H}}$ be the set of safe permutations on the instance IDs in σ_0. Then $Tr(\sigma_0)$ is symmetric with respect to the set of safe permutations $\mathcal{S}_{\overline{H}}$. In other words, for any $p \in \mathcal{S}_{\overline{H}}$, $p_t(Tr(\sigma_0)) = \{\, p(\pi) \mid \pi \in Tr(\sigma_0) \,\}$ is equal to $Tr(\sigma_0)$.*

Proof:

$p_t(Tr(\sigma_0)) \subseteq Tr(\sigma_0)$

Let $\pi' \in p_t(Tr(\sigma_0))$. Then $\pi' = p_t(\pi)$ for some $\pi \in Tr(\sigma_0)$. By Theorem 5.3.10, $\pi' = p_t(\pi) \in Tr(\sigma_0)$.

$Tr(\sigma_0) \subseteq p_t(Tr(\sigma_0)$

Let $\pi \in Tr(\sigma_0)$. Since $p \in \mathcal{S}_{\overline{H}}$ and $\mathcal{S}_{\overline{H}}$ is a group (Theorem 5.3.1), $p^{-1} \in \mathcal{S}_{\overline{H}}$. Let $\pi' = p_t^{-1}(\pi)$. By Theorem 5.3.10, $\pi' \in Tr(\sigma_0)$ because $p^{-1} \in \mathcal{S}_{\overline{H}}$. Therefore, $\pi = p_t(p_t^{-1}(\pi)) = p_t(\pi') \in p_t(Tr(\sigma_0))$.

□

Theorem 5.1.2 states that $\{\, p_t \mid p \in \mathcal{S}_{\overline{H}} \,\}$, the group of safe permutations on traces, partitions the set of traces $Tr(\sigma)$ into equivalence classes. One trace π is equivalent to another trace π' if and only if there exists some $p \in \mathcal{S}_{\overline{H}}$ such that $\pi' = p_t(\pi)$. This symmetry partitions the traces into sets with equivalent structure, in the sense that the same actions and relationships are present; however, the instances performing the actions have been permuted. I would now like to show that these equivalence classes of traces also respect the truth values of formulas if the formulas also display the same symmetry. First, I must define the extension of $p \in \mathcal{S}_{\overline{H}}$ to the set of formulas.

Definition 5.3.5 *Let $\mathcal{S}_{\overline{H}}$ be the set of safe permutations on the instances of some trace. Let ϕ be any formula in our logic. Then $p \in \mathcal{S}_{\overline{H}}$ is extended to p_f, a permutation on formulas, as follows:*

- $p_f(H_i.v) = p(H_i).v$ *for any instance ID* H_i *and any message variable* v.
- $p_f(a) = p_{\mathcal{M}}(a)$ *for any atomic message* $a \in \mathcal{A}$.
- $p_f(m_1 \cdot m_2) = p_f(m_1) \cdot p_f(m_2)$ *for all message terms* m_1 *and* m_2.
- $p_f(\{m\}_k) = \{p_f(m)\}_{p_f(k)}$ *for all message terms* m *and* k.
- $p_f(m_1 = m_2) = [p_f(m_1) = p_f(m_2)]$ *for any atomic proposition of the form* $m_1 = m_2$.
- $p_f(H_i \textbf{ Knows } m) = p(H_i) \textbf{ Knows } p_f(m)$ *for any atomic proposition of the form* $H_i \textbf{ Knows } m$.
- $p_f(H_i \textbf{ Act } m) = p(H_i) \textbf{ Act } p_f(m)$ *for any atomic proposition of the form* $H_i \textbf{ Act } m$.
- $p_f(\neg\phi) = \neg(p_f(\phi))$.
- $p_f(\phi_1 \wedge \phi_2) = p_f(\phi_1) \wedge p_f(\phi_2)$.
- $p_f(\Diamond_P \phi) = \Diamond_P p_f(\phi)$.

We now need to show that a trace satisfies a formula if and only if the permuted trace satisfies the permuted formula. First we need a lemma analogous to Lemma 5.3.6, but for message terms instead of message templates.

Lemma 5.3.12 *Let* $p \in \mathcal{S}_{\overline{H}}$ *be a safe permutation on the instances of some trace* π. *Let* $p_{\mathcal{M}}$, p_s, *and* p_f *be the extensions of* p *to message templates, states, and formulas, respectively. Let* m *be a message term and let* σ *be a state in* π *in which all variables appearing in* m *are bound. Then* $p_{\mathcal{M}}(\sigma(m)) = [p_s(\sigma)](p_f(m))$.

Proof: (by induction on the structure of the message term m)

- $m = a \in \mathcal{A}$

$$
\begin{array}{rcll}
p_{\mathcal{M}}(\sigma(m)) & = & p_{\mathcal{M}}(\sigma(a)) & \\
 & = & p_{\mathcal{M}}(a) & \text{Definition 3.2.1} \\
 & = & [p_s(\sigma)](p_{\mathcal{M}}(a)) & \text{Definition 3.2.1 and } p_{\mathcal{M}}(a) \in \mathcal{A} \\
 & = & [p_s(\sigma)](p_f(a)) & \text{Definition 5.3.5} \\
 & = & [p_s(\sigma)](p_f(m)) &
\end{array}
$$

- $m = H_i.v$

$$\begin{array}{rcll} p_{\mathcal{M}}(\sigma(m)) & = & p_{\mathcal{M}}(\sigma(H_i.v)) & \\ & = & p_{\mathcal{M}}(B_i(v)) & \text{Definition 3.2.1} \\ & = & (p_{\mathcal{M}} \circ B_i)(v) & \\ & = & [p_s(\sigma)](H_i.v) & \text{Definition 5.3.4 and Definition 3.2.1} \\ & = & [p_s(\sigma)](p_f(H_i.v)) & \text{Definition 5.3.5} \\ & = & [p_s(\sigma)](p_f(m)) & \end{array}$$

- $m = m_1 \cdot m_2$

$$\begin{array}{rcll} p_{\mathcal{M}}(\sigma(m)) & = & p_{\mathcal{M}}(\sigma(m_1 \cdot m_2)) & \\ & = & p_{\mathcal{M}}(\sigma(m_1) \cdot \sigma(m_2)) & \text{Definition 3.2.1} \\ & = & p_{\mathcal{M}}(\sigma(m_1)) \cdot p_{\mathcal{M}}(\sigma(m_2)) & \text{Definition 5.3.3} \\ & = & [p_s(\sigma)](p_f(m_1)) \cdot [p_s(\sigma)](p_f(m_2)) & \text{ind. hyp.} \\ & = & [p_s(\sigma)](p_f(m_1) \cdot p_f(m_2)) & \text{Definition 3.2.1} \\ & = & [p_s(\sigma)](p_f(m_1 \cdot m_2)) & \text{Definition 5.3.5} \\ & = & [p_s(\sigma)](p_f(m)) & \end{array}$$

- $m = \{m'\}_k$

$$\begin{array}{rcll} p_{\mathcal{M}}(\sigma(m)) & = & p_{\mathcal{M}}(\sigma(\{m'\}_k)) & \\ & = & p_{\mathcal{M}}(\{\sigma(m')\}_{\sigma(k)}) & \text{Definition 3.2.1} \\ & = & \{p_{\mathcal{M}}(\sigma(m'))\}_{p_{\mathcal{M}}(\sigma(k))} & \text{Definition 5.3.3} \\ & = & \{[p_s(\sigma)](p_f(m'))\}_{[p_s(\sigma)](p_f(k))} & \text{ind. hyp.} \\ & = & [p_s(\sigma)](\{(p_f(m')\}_{p_f(k)}) & \text{Definition 3.2.1} \\ & = & [p_s(\sigma)](p_f(\{m'\}_k)) & \text{Definition 5.3.5} \\ & = & [p_s(\sigma)](p_f(m)) & \end{array}$$

□

Theorem 5.3.13 *Let π be a trace, ϕ be a formula, and $\mathcal{S}_{\overline{H}}$ be the safe permutations on the instances in π. Then for any $p \in \mathcal{S}_{\overline{H}}, \langle \pi, i \rangle \models \phi$ if and only if $\langle p_t(\pi), i \rangle \models p_f(\phi)$.*

Proof: (by induction on the structure of the formula ϕ)

- $\phi = [m_1 = m_2]$
  Using the definition of $\models$, Lemma 5.3.12, and Definition 5.3.5 (the

definition of p_f) we have the following:

$$\begin{aligned}
\langle \pi, i\rangle \models [m_1 = m_2] &\Leftrightarrow \sigma_i(m_1) = \sigma_i(m_2) \\
&\Leftrightarrow p_{\mathcal{M}}(\sigma_i(m_1)) = p_{\mathcal{M}}(\sigma_i(m_2)) \\
&\Leftrightarrow [p_s(\sigma_i)](p_f(m_1)) = [p_s(\sigma_i)](p_f(m_2)) \\
&\Leftrightarrow \langle p_t(\pi), i\rangle \models [p_f(m_1) = p_f(m_2)] \\
&\Leftrightarrow \langle p_t(\pi), i\rangle \models p_f([m_1 = m_2])
\end{aligned}$$

- $\phi = H_j$ **Knows** m
  Let $\mathcal{I}_j = \langle H_j, B_j, I_j, P_j\rangle$ in σ_i. Then in $p_s(\sigma_i)$,

  $$p_i(\mathcal{I}_j) = \langle p(H_j), p_{\mathcal{M}} \circ B_j, p_{\mathcal{M}}(I_j), P_j\rangle.$$

  Using the definition of $\models$, Lemma 5.3.8 about applying $p_{\mathcal{M}}$ to derivation trees, Lemma 5.3.12, and Definition 5.3.5 (the definition of p_f) we have the following:

  $$\begin{aligned}
  \langle \pi, i\rangle \models H_j \;\; \textbf{Knows} \;\; m &\Leftrightarrow \sigma_i(m) \in \overline{I_j} \\
  &\Leftrightarrow p_{\mathcal{M}}(\sigma_i(m)) \in \overline{p_{\mathcal{M}}(I_j)} \\
  &\Leftrightarrow [p_s(\sigma_i)](p_f(m)) \in \overline{p_{\mathcal{M}}(I_j)} \\
  &\Leftrightarrow \langle p_t(\pi), i\rangle \models p(H_j) \;\; \textbf{Knows} \;\; p_f(m) \\
  &\Leftrightarrow \langle p_t(\pi), i\rangle \models p_f(H_j \;\; \textbf{Knows} \;\; m)
  \end{aligned}$$

- $\phi = H_j \ A \ m$
  Using the definition of $\models$, Definition 5.3.4 (definition of p_a), Lemma 5.3.12, and Definition 5.3.5 (the definition of p_f) we have the following:

  $$\begin{aligned}
  \langle \pi, i\rangle \models H_j \ A \ m &\Leftrightarrow \alpha_k = H_j \cdot A \cdot \sigma_{k-1}(m) \text{ for some } 1 \le k \le i \\
  &\Leftrightarrow p_a(\alpha_k) = p_a(H_j \cdot A \cdot \sigma_{k-1}(m)) \\
  &\Leftrightarrow p_a(\alpha_k) = p(H_j) \cdot A \cdot p_{\mathcal{M}}(\sigma_{k-1}(m)) \\
  &\Leftrightarrow p_a(\alpha_k) = p(H_j) \cdot A \cdot [p_s(\sigma_{k-1})](p_f(m)) \\
  &\Leftrightarrow \langle p_t(\pi), i\rangle \models p(H_j) \ A \ p_f(m) \\
  &\Leftrightarrow \langle p_t(\pi), i\rangle \models p_f(H_j \ A \ m)
  \end{aligned}$$

- $\phi = \neg\phi'$
  Using the definition of $\models$, the inductive hypothesis, and Definition 5.3.5 (the definition of p_f) we have the following:

$$\begin{aligned}\langle \pi, i\rangle \models \neg\phi' &\Leftrightarrow \langle \pi, i\rangle \not\models \phi' \\ &\Leftrightarrow \langle p_t(\pi), i\rangle \not\models p_f(\phi') \\ &\Leftrightarrow \langle p_t(\pi), i\rangle \models \neg p_f(\phi') \\ &\Leftrightarrow \langle p_t(\pi), i\rangle \models p_f(\neg\phi')\end{aligned}$$

- $\phi = \phi_1 \wedge \phi_2$
  Using the definition of $\models$, the inductive hypothesis, and Definition 5.3.5 (the definition of p_f) we have the following:

$$\begin{aligned}\langle \pi, i\rangle \models \phi_1 \wedge \phi_2 &\Leftrightarrow \langle \pi, i\rangle \models \phi_1 \text{ and } \langle \pi, i\rangle \models \phi_2 \\ &\Leftrightarrow \langle p_t(\pi), i\rangle \models p_f(\phi_1) \text{ and } \langle p_t(\pi), i\rangle \models p_f(\phi_2) \\ &\Leftrightarrow \langle p_t(\pi), i\rangle \models p_f(\phi_1) \wedge p_f(\phi_2) \\ &\Leftrightarrow \langle p_t(\pi), i\rangle \models p_f(\phi_1 \wedge \phi_2)\end{aligned}$$

- $\phi = \Diamond_P\phi'$
  Using the definition of $\models$, the inductive hypothesis, and Definition 5.3.5 (the definition of p_f) we have the following:

$$\begin{aligned}\langle \pi, i\rangle \models \Diamond_P\phi' &\Leftrightarrow \langle \pi, j\rangle \models \phi' \text{ for some } 0 \leq j \leq i \\ &\Leftrightarrow \langle p_t(\pi), j\rangle \models p_f(\phi') \text{ for the same } 0 \leq j \leq i \\ &\Leftrightarrow \langle p_t(\pi), i\rangle \models \Diamond_P p_f(\phi') \\ &\Leftrightarrow \langle p_t(\pi), i\rangle \models p_f(\Diamond_P\phi')\end{aligned}$$

□

Instance IDs are arbitrary. The correctness of the protocol should not depend on them. The specification should be insensitive to permutations on the IDs of instances performing the same role on behalf of the same principal. In other words, the specification should be insensitive to safe permutations. If this is the case, one need only check one trace from the equivalence class of traces that are related via a safe permutation of instance IDs. This is formalized in the corollary below, but first a minor technicality should be clarified.

Sometimes, when p_f is applied to a formula, the result is an equivalent formula although it may not be syntactically equal. This situation arises because I have defined quantification as a shorthand for finite conjunction/disjunction, and both operations are commutative. For example, assuming the message term m contains only variables and messages that are **not** freshly generated, then

$$\begin{aligned}
p_f(\forall H_i \,.\, H_i \;\; \mathbf{Knows}\; m) &\equiv p_f\left(\bigwedge_{i=1}^{k} H_i \;\; \mathbf{Knows}\; m\right) \\
&\equiv \bigwedge_{i=1}^{k} p(H_i) \;\; \mathbf{Knows}\; p_f(m) \\
&\equiv \bigwedge_{i=1}^{k} p(H_i) \;\; \mathbf{Knows}\; m \text{ , assuming } p_f(m) = m \\
&\approx \bigwedge_{i=1}^{k} H_i \;\; \mathbf{Knows} \;\; m \\
&\equiv \forall H_i \,.\, H_i \;\; \mathbf{Knows}\; m
\end{aligned}$$

The second to the last line is equivalent to the previous line because p is a permutation. All the disjuncts are present, although not necessarily in the same order. However, since disjunction (and conjunction) are commutative, they are semantically equivalent ($\approx$). In other words, the two formulas agree on all traces. This semantic equivalence can be defined syntactically as follows:

Definition 5.3.6 *Given two formulas ϕ and ϕ', $\phi \approx \phi'$ if and only if ϕ and ϕ' are syntactically equal up to the ordering of conjuncts and disjuncts. Note that "$\approx$" is clearly an equivalence.*

So, in the previous example, if $p_f(m) = m$, then

$$p_f(\forall H_i \,.\, H_i \;\; \mathbf{Knows}\; m) \approx \forall H_i \,.\, H_i \;\; \mathbf{Knows}\; m.$$

The question then is how to ensure that $p_f(m) = m$. A sufficient condition for this is to restrict m to contain only "public knowledge" and variables. In other words, if m does not contain any freshly generated messages, then $p_f(m) = m$. It is clear from the definition of p_f (Definition 5.3.5), that this condition is sufficient. Also, note that this is a reasonable restriction, because freshly generated messages must be bound to a variable in the initial state. If necessary, the specification can refer to freshly generated messages through these variables.

The relation "$\approx$" is now used to characterize the formulas for which a symmetry reduction is possible:

Corollary 5.3.14 *Given an initial state σ, the set of traces $Tr(\sigma)$, a formula ϕ, and $p \in \mathcal{S}_{\overline{H}}$, a safe permutation on the instances of σ such that $p_f(\phi) \approx \phi$ (the formula is symmetric in p), then for any trace $\pi \in Tr(\sigma)$, $\pi \models \phi$ if and only if $p_t(\pi) \models \phi$.*

Proof: Let $l = \mathit{length}(\pi)$. Using the definition of $\models$, Theorem 5.3.13, and the fact that $p_f(\phi) \approx \phi$ implies $\pi_0 \models \phi \Leftrightarrow \pi_0 \models p_f(\pi)$ for any trace π_0, we have the following:

$$\begin{aligned}
\pi \models \phi \quad & \Leftrightarrow \quad \langle \pi, i\rangle \models \phi \text{ for } 0 \le i \le l \\
& \Leftrightarrow \quad \langle p_t(\pi), i\rangle \models p_f(\phi) \text{ for } 0 \le i \le l \\
& \Leftrightarrow \quad p_t(\pi) \models p_f(\phi) \\
& \Leftrightarrow \quad p_t(\pi) \models \phi
\end{aligned}$$

□

In practice this allows the following reduction. Assume σ is the initial state of the model to be analyzed. Then the model has $Tr(\sigma)$ as its set of traces. Assume ϕ is the formula to be verified. Let $\mathcal{S}_{\overline{H}}$ be the subset of safe permutations on the instances in σ. Then $\mathcal{S}_{\overline{H}}$ partitions $Tr(\sigma)$ into equivalence classes. Furthermore, these equivalence classes are a congruence with respect to satisfying the formula ϕ. In other words, for any $\pi \in Tr(\sigma)$, $\pi \models \phi$ if and only if $\pi' \models \phi$ for every $\pi' \in \mathcal{S}_{\overline{H}}(\pi)$. This means one need only check a single representative from each equivalence class. Each equivalence class has size $|\mathcal{S}_{\overline{H}}|$. This means the amount search required is reduced by a factor of $|\mathcal{S}_{\overline{H}}|$.

How large is $\mathcal{S}_{\overline{H}}$, the set of safe permutations? Any safe permutation can be constructed by independently permuting the instances playing the same role on behalf of the same principal. Each of these subgroups has size $\mathcal{C}_k(X)!$. Recall that $\mathcal{C}_k(X)$ is a function in the configuration $\mathcal{C}$ that gives the number of instances of principal X playing role k. The total number of safe permutations is the product of these subgroups, namely $\Pi[\mathcal{C}_k(X)!]$. Figure 5.1, is an instance configuration graph for a model with three initiator instances for principal A and with three responder instances for principal B. The size of $\mathcal{S}_{\overline{H}}$, and therefore, the amount of reduction in this example is

$$\begin{aligned}
\prod[\mathcal{C}_k(X)!] &= \mathcal{C}_i(A)! \cdot \mathcal{C}_i(B)! \cdot \mathcal{C}_r(A)! \cdot \mathcal{C}_r(B)! \\
&= 3! \cdot 0! \cdot 0! \cdot 3! \\
&= 36
\end{aligned}$$

Chapter 6

Partial Order

Another important technique used to combat the state explosion problem in model checking is the partial order reduction. Partial order reduction prunes the set of traces of a system by reducing the number of interleavings that need to be considered. For example, if the system is insensitive to permuting two actions α and β, then one can consider a single interleaving (say $\alpha\beta$) and ignore the other interleaving ($\beta\alpha$) while exploring the system. Consider a simple system with two processes, one performing the actions $\alpha_1, \ldots \alpha_n$ in order and the other performing $\beta_1, \ldots \beta_n$ in order. All possible traces are depicted in Figure 6.1. The full product model contains n^2 states. Each path in the figure corresponds to a different trace. As noted in [31], the number of different paths in an $n \times n$ grid is $\binom{2n}{n}$. (There are $2n$ total actions and one must choose where to place the n α-actions). If the specification is insensitive to the ordering of these actions, then the entire graph can be collapsed into one representative trace that has n states. The state space for three processes has n^3 states and

$$\binom{3n}{2n} \cdot \binom{2n}{n} = \frac{3n!}{n!n!n!}$$

different possible interleavings. It is easy to see that the complexity (both time and space) grows quite quickly in the number of processes. However, the set of traces that need to be checked could theoretically contain a single trace containing $3n$ states. The trick lies in being able to perform the reduction without having to construct the entire model first. This is done by pruning the state space as the search proceeds. Typically, this involves expanding a subset of the enabled actions in a state.

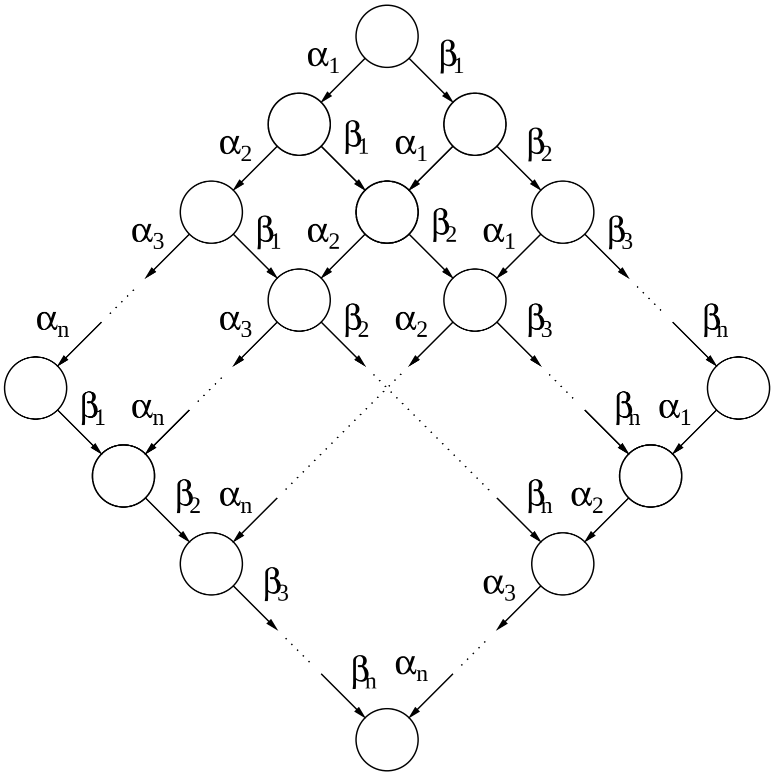

Figure 6.1: Two processes with n actions each

While the set of executions generally cannot be reduced to a single trace as in the example above, the use of partial order reduction has proved valuable in verifying reactive systems [22, 66, 80]. I now show how to apply the partial order reduction technique to the verification of security protocols. I also present a proof of correctness for this reduction. While my implementation is specific to the BRUTUS model checker, the proof of correctness is fairly general and other researchers working in this area should be able to adapt it to their tools.

While this reduction in BRUTUS is influenced heavily by the traditional partial order reduction theory, it is not solely a special case of this theory. In particular, traditional partial order techniques do not directly apply to BRUTUS models because these models explicitly keep track of the knowledge of various agents, and the specification logic can refer to this knowledge in a meaningful way. In addition, the enabledness of **receive** actions depends heavily on the adversary's knowledge. These differences combine to give us a fundamental difference in the proof of correctness. Traditional partial order techniques rely on the equivalence of traces. Any trace that is not explored is equivalent to another trace that is explored. Equivalent traces agree on the specification in the sense that they either both satisfy the specification, or neither satisfies the specification. I will argue that for any trace that is not considered there is a trace that I do consider such that if the ignored trace violates the requirement, then the trace I do consider also violates the requirement. However, I do not insist that related traces agree on the specification.

6.1 Preliminaries

In order for this partial order reduction to be sound, I must restrict the set of formulas that are allowed as specifications. Consider $AP(\phi)$, the set of atomic propositions appearing in the formula ϕ. This set can be partitioned into the following two sets.

- $AP_\Omega(\phi)$: the set of atomic propositions referring to the adversary.
- $AP_H(\phi)$: the set of atomic propositions referring to the honest agents.

Note that because the adversary performs no internal actions, and because the adversary has no variables, the atomic propositions in $AP_\Omega(\phi)$, must be of the form H_Ω **Knows** m for some message term m. All other atomic propositions appearing in ϕ are in $AP_H(\phi)$.

In order for this partial order reduction to be sound, any atomic propositions in $AP_\Omega(\phi)$ must occur *negatively* in ϕ. To simplify this discussion as well as the proof, I will assume that ϕ is in negation normal form. This means that all negations are "pushed in" as far as possible so that the only negated subformulas in ϕ are atomic propositions. If ϕ is in negation normal form, the restriction that atomic propositions in $AP_\Omega(\phi)$ occur negatively is equivalent to the restriction that all propositions in $AP_\Omega(\phi)$ appear as negated literals. Such a specification will be called *admissible*.

Intuitively, the restriction to admissible formulas means that one can only write specifications that require messages to remain unknown to the adversary. One cannot write specifications that require the adversary to know something. Should such a specification be necessary, the partial order reduction can be disabled to ensure that the verification is still correct.

Before continuing with the formal description of the partial order reduction, let us consider the intuition behind the reduction. The truth value of a specification ϕ on a trace π is completely determined by the values of the atomic propositions in the set AP_H and by the adversary's knowledge which determines the values of the atomic propositions in the set AP_Ω. I will refer to the set of atomic propositions in AP_H that are true in a state $\langle \pi, i \rangle$ as $L(\langle \pi, i \rangle)$, the labeling of $\langle \pi, i \rangle$. I will refer to the set of messages known to the adversary (equivalently, the "adversary's knowledge") in a state $\langle \pi, i \rangle$, as $I_\Omega(\langle \pi, i \rangle)$.

As in the case of the usual partial order reduction, I look for *invisible actions*. Invisible actions do not affect the truth value of the formula in question. Any actions that do not affect the truth value of the atomic propositions will be invisible. It is precisely these invisible actions that can be permuted in a trace without affecting the truth value of the specification formula. This means we must identify actions α such that

$$L(\langle \pi, i \rangle) = L(\langle \pi, i+1 \rangle) \text{ and } I_\Omega(\langle \pi, i \rangle) = I_\Omega(\langle \pi, i+1 \rangle)$$

for all pairs of states $\langle \pi, i \rangle$ and $\langle \pi, i+1 \rangle$ where $\langle \pi, i \rangle \xrightarrow{\alpha} \langle \pi, i+1 \rangle$.

This notion of invisible actions can be extended to the idea of *semi-invisible actions*. Semi-invisible actions can only make the specification false. Because our specifications can only require that the adversary **not** know things, any action that increases the adversary's knowledge but is otherwise "invisible" is semi-invisible (can only make the specification false). In other words, we also look for actions β such that

$$L(\langle \pi, i \rangle) = L(\langle \pi, i+1 \rangle) \text{ and } I_\Omega(\langle \pi, i \rangle) \subsetneq I_\Omega(\langle \pi, i+1 \rangle)$$

for any two states $\langle \pi, i \rangle$ and $\langle \pi, i+1 \rangle$ such that $\langle \pi, i \rangle \xrightarrow{\beta} \langle \pi, i+1 \rangle$. I show that these actions are precisely the **send** actions. Since **send** actions only increase the adversary's knowledge, these actions can be moved forward (sooner) in the trace, and the resulting trace will still be a counterexample if the original trace was a counterexample.

It is not sufficient to ensure that the permutations on the actions preserve the truth value of traces; the permutations must also be allowed. In other words, the traces that result from the permutations must be valid traces (possible traces) of the system or model. In traditional partial order techniques this is accomplished by analyzing the enabledness of different actions. A check has to be made that two actions which are permuted cannot disable each other. This task is simplified in my setting because the actions I move to the front (invisible actions and semi-invisible actions) never disable any other actions. With this overview in mind, we now turn to the formal proof of this result.

6.2 A Relation on Traces

I now define a relation "$\mathcal{K}$" on traces that respects satisfaction of ϕ. In other words, $\pi_1 \models \phi$ and $\pi_1 \mathcal{K} \pi_2$ should imply that $\pi_2 \models \phi$. If this is the case, then whenever $\pi_1 \mathcal{K} \pi_2$ for any pair of traces π_1 and π_2, one need only check the trace π_1. Because this relation can be detected at individual states (for traces with a common prefix), entire subtrees of the computation tree can be collapsed into a single trace or branch. Informally, $\pi_1 \mathcal{K} \pi_2$ can be thought to hold whenever the adversary might learn messages sooner in π_1 than in π_2, but the traces are otherwise "almost identical." This is defined formally below.

Definition 6.2.1 *For any two traces π_1 and π_2, $\pi_1 \mathcal{K} \pi_2$ if and only if there exist a partition $A = \{A_0, A_1, A_2, \ldots, A_l\}$ of the trace π_1 and a partition $B = \{B_0, B_1, B_2, \ldots B_l\}$ of the trace π_2 such that the following conditions hold:*

1. There exist indices

$$0 = a_0 < a_1 < a_2 < \cdots < a_l < a_{l+1} = \mathit{length}(\pi_1) + 1$$

   such that $A_i = \{\langle \pi_1, a_i \rangle, \ldots, \langle \pi_1, a_{i+1} - 1 \rangle\}$. In other words, A_i is the subtrace of π_1 starting at index a_i and ending at index $a_{i+1} - 1$.

2. There exist indices

$$0 = b_0 < b_1 < b_2 < \cdots < b_{l+1} = \mathit{length}(\pi_2) + 1$$

   such that $B_i = \{\langle \pi_2, b_i \rangle, \ldots, \langle \pi_2, b_{i+1} - 1 \rangle\}$. In other words, B_i is the subtrace of π_2 starting at index b_i and ending at index $b_{i+1} - 1$.

3. For any two corresponding partitions A_k and B_k ($0 \leq k \leq l$), the following holds:

$$\forall \langle \pi_1, i \rangle \in A_k \;.\; \forall \langle \pi_2, j \rangle \in B_k \;.\; L(\langle \pi_1, i \rangle) = L(\langle \pi_2, j \rangle).$$

   In other words, all states in any two corresponding partitions agree on all the atomic propositions referring to the honest agents.

4. For any partition index $0 \leq k \leq l$, the following holds:

$$I_\Omega(\langle \pi_2, b_{k+1} - 1 \rangle) \subseteq I_\Omega(\langle \pi_1, a_k \rangle).$$

   In other words, the adversary knows at least as much in the first state of A_k as it does in the last state of B_k. Since the adversary's knowledge is monotonically increasing, this also means that the adversary knows at least as much in every state of A_k as it does in any state of B_k. In other words,

$$I_\Omega(\langle \pi_2, j \rangle) \subseteq I_\Omega(\langle \pi_1, i \rangle)$$

   for any indices i and j within the partition boundaries of A_k and B_k (i.e., for $a_k \leq i < a_{k+1}$ and $b_k \leq j < b_{k+1}$).

Recall that the original idea was to define a relation that respects the satisfaction relation. The proof follows.

Theorem 6.2.1 *Let ϕ be an admissible specification. Let π_1 and π_2 be two traces such that $\pi_1 \mathcal{K} \pi_2$. Let $A = \{A_1, \cdots A_l\}$ and $B = \{B_1, \cdots B_l\}$ be the partitions of π_1 and π_2 that satisfy the definition of "$\mathcal{K}$". If $\langle \pi_1, i \rangle \models \phi$ for some state in partition A_k (i.e., for some $a_k \leq i < a_{k+1}$), then $\langle \pi_2, j \rangle \models \phi$ for all states in partition B_k (i.e., for all $b_k \leq j < b_{k+1}$).*

Proof: (by induction on the formula ϕ)

I will assume that the formula is in negation normal form, and so I will ignore general negation and treat negated literals as a base case for the induction.

- $\phi = p$ or $\phi = \neg p$ where $p \in AP_H$

  This case is trivial, because A_k and B_k agree on the atomic propositions in AP_H. (For all states $\langle \pi_1, i \rangle \in A_k$ and for all states $\langle \pi_2, j \rangle \in B_k$, $L(\langle \pi_1, i \rangle) = L(\langle \pi_2, j \rangle)$.) Therefore, if any state $\langle \pi_1, i \rangle \in A_k$ satisfies p then every state $\langle \pi_2, j \rangle \in B_k$ satisfies p. If any state $\langle \pi_1, i \rangle \in A_k$ does not satisfy p then no state $\langle \pi_2, j \rangle \in B_k$ satisfies p.

- $\phi = \neg p$ where $p \in AP_\Omega$

  Again, this case is straightforward because p must have the form Ω **Knows** m and the adversary knows more in every state of A_k than in any state of B_k. (For all states $\langle \pi_1, i \rangle \in A_k$ and for all states $\langle \pi_2, j \rangle \in B_k$ $I_\Omega(\langle \pi_1, i \rangle) \supseteq I_\Omega(\langle \pi_2, j \rangle)$.) If $\langle \pi_1, i \rangle \models \neg\ H_\Omega$ **Knows** m then m is not derivable from $I_\Omega(\langle \pi_1, i \rangle)$. Since $I_\Omega(\langle \pi_2, j \rangle) \subseteq I_\Omega(\langle \pi_1, i \rangle)$ for all states $\langle \pi_2, j \rangle \in B_k$, m is also not derivable from $I_\Omega(\langle \pi_2, j \rangle)$ which means that $\langle \pi_2, j \rangle \models \neg\ H_\Omega$ **Knows** m for all $b_k \leq j < b_{k+1}$.

- $\phi = \phi_1 \wedge \phi_2$

  By hypothesis, $\langle \pi_1, i \rangle \models \phi_1 \wedge \phi_2$ for some state in partition A_k (i.e., for some $a_k \leq i < a_{k+1}$). Therefore $\langle \pi_1, i \rangle \models \phi_1$ and $\langle \pi_1, i \rangle \models \phi_2$ for some $a_k \leq i < a_{k+1}$. By the induction hypothesis $\langle \pi_2, j \rangle \models \phi_1$ and $\langle \pi_2, j \rangle \models \phi_2$ for all states in partition B_k (i.e., for all $b_k \leq j < b_{k+1}$). Therefore, $\langle \pi_2, j \rangle \models \phi_1 \wedge \phi_2$ for all $b_k \leq j < b_{k+1}$.

- $\phi = \phi_1 \vee \phi_2$

  By hypothesis, $\langle \pi_1, i \rangle \models \phi_1 \vee \phi_2$ for some state in partition A_k (i.e., for some $a_k \leq i < a_{k+1}$). Therefore, $\langle \pi_1, i \rangle \models \phi_1$ or $\langle \pi_1, i \rangle \models \phi_2$ for some $a_k \leq i < a_{k+1}$. Without loss of generality, assume $\langle \pi_1, i \rangle \models \phi_1$. By the induction hypothesis $\langle \pi_2, j \rangle \models \phi_1$ for all states in partition B_k (i.e., for all $b_k \leq j < b_{k+1}$). Therefore, $\langle \pi_2, j \rangle \models \phi_1 \vee \phi_2$ for all $b_k \leq j < b_{k+1}$.

- $\phi = \Diamond_P \phi_1$

  By hypothesis, $\langle \pi_1, i \rangle \models \Diamond_P \phi_1$ for some state in partition A_k (i.e., for some $a_k \leq i < a_{k+1}$). Therefore $\langle \pi_1, i' \rangle \models \phi_1$ for some $i' \leq i$. There are two cases depending on whether $\langle \pi_1, i' \rangle$ falls in the same partition A_k or some earlier partition.

  - **case:** $\langle \pi_1, i' \rangle \in A_k$
    In this case $\langle \pi_1, i' \rangle$, the state satisfying ϕ_1, is also in partition A_k (i.e., $a_k \leq i' \leq i < a_{k+1}$). By the inductive hypothesis, $\langle \pi_2, j \rangle \models \phi_1$ for all states in partition B_k (i.e., for all $b_k \leq j < b_{k+1}$). Therefore, $\langle \pi_2, j \rangle \models \Diamond_P \phi_1$ for all states in partition B_k (i.e., for all $b_k \leq j < b_{k+1}$).
  - **case:** $\langle \pi_1, i' \rangle \notin A_k$
    In this case $\langle \pi_1, i' \rangle$, the state satisfying ϕ_1, is in some partition $A_{k'}$ appearing before partition A_k. In other words,

    $$a_{k'} \leq i' < a_{k'+1} \leq a_k \leq i < a_{k+1}$$

    for some partition index $k' < k$. By the inductive hypothesis, $\langle \pi_2, j' \rangle \models \phi_1$ for all states in partition $B_{k'}$ (i.e., for all $b_{k'} \leq j' < b_{k'+1}$). Since $b_{k'+1} \leq b_k$, $\langle \pi_2, j \rangle \models \Diamond_P \phi_1$ for all $b_k \leq j < b_{k+1}$. In other words, $\langle \pi_2, j \rangle \models \Diamond_P \phi_1$ for all states in partition B_k.

- $\phi = \Box_P \phi_1$

  By hypothesis, $\langle \pi_1, i \rangle \models \Box_P \phi_1$ for some state in partition A_k (i.e., for some $a_k \leq i < a_{k+1}$). Therefore $\langle \pi_1, i' \rangle \models \phi_1$ for every $i' \leq i$. Since the smallest possible value for i is a_k, in particular, $\langle \pi_1, i' \rangle \models \phi_1$ for every $i' \leq a_k$. Since this is true for every $i' \leq a_k$ it is also true for every partition boundary $a_{k'} \leq a_k$ (i.e., for every k' where $0 \leq k' \leq k$). But this means that there is at least one state (namely the state $\langle \pi_1, a_{k'} \rangle$) in every partition $A_{k'}$ up to and including the partition A_k (for $0 \leq k' \leq k$) that satisfies ϕ_1. By the induction hypothesis, every state in every partition $B_{k'}$ up to and including B_k (for $0 \leq k' \leq k$) also satisfies ϕ_1. In other words $\langle \pi_2, j' \rangle \models \phi_1$ for all $j' < b_{k+1}$. Again, using the definition of $\Box_P$, $\langle \pi_2, j' \rangle \models \Box_P \phi_1$ for all $j' < b_{k+1}$. In particular then, $\langle \pi_2, j \rangle \models \Box_P \phi_1$ for all states in partition B_k, (i.e., for all $b_k \leq j < b_{k+1}$).

□

So, some state σ_i in partition A_k of trace π_1 satisfies ϕ only if every state σ'_j in the corresponding partition B_k of trace π_2 also satisfies ϕ. Since I show this true of every partition A_k, this implies that if $\langle \pi_1, i \rangle \models \phi$ for all i then $\langle \pi_2, j \rangle \models \phi$ for all j. This corollary is restated below.

Corollary 6.2.2 *Let π_1 and π_2 be two traces such that $\pi_1 \mathcal{K} \pi_2$ and let ϕ be an admissible specification. Then $\pi_1 \models \phi$ implies $\pi_2 \models \phi$.*

6.3 Transformations on Traces

I now define a transformation on traces that respects the relation defined in the previous section. The idea is to define a transformation that permutes only invisible and semi-invisible actions in a trace π so as to result in a new trace π' such that $\pi' \mathcal{K} \pi$. This resulting trace should also be a valid trace of the system being modeled. By Corollary 6.2.2, I can ignore the original trace (π in this case, but π_2 in Corollary 6.2.2) if the transformed trace satisfies the specification. This transformation is defined below.

Definition 6.3.1 *An action is* invisible *if it is an internal action and it does not affect the value of any of the atomic propositions in the specification. In other words, the action must be an internal action to which no proposition in AP_H refers.*

Definition 6.3.2 *Two traces π and π' are related by a transformation step ($\pi \Rightarrow \pi'$), if the trace π' results from the trace π after the application of one of the following operations:*

1. Move an invisible actions forward.

   This operation allows an invisible action to be swapped with the action immediately preceding it, as long as both actions are not being performed by the same instance (otherwise it would not be a valid trace). Consider a trace $\pi = \sigma_0\alpha_1\sigma_1 \dots \sigma_n$. If there exists a sequence of transitions

   $$\sigma_i\alpha_{i+1}\sigma_{i+1}\alpha_{i+2}\sigma_{i+2}$$

   such that α_{i+2} is an invisible action, and actions α_{i+1} and α_{i+2} do not belong to the same instance, then the actions can be swapped to get the new trace given below:

   $$\sigma_0\alpha_1\sigma_1 \dots \sigma_i\alpha_{i+2}{\sigma'}_{i+1}\alpha_{i+1}\sigma_{i+2} \dots \sigma_n$$

2. Move a **send** actions forward.

   This operation allows a **send** action to be swapped with the action immediately preceding it, as long as both actions are not being performed by the same instance. Consider a trace $\pi = \sigma_0\alpha_1\sigma_1 \dots \sigma_n$. If there exists a sequence of transitions

   $$\sigma_i\alpha_{i+1}\sigma_{i+1}\alpha_{i+2}\sigma_{i+2}$$

   such that α_{i+2} is a **send** action, and actions α_{i+1} and α_{i+2} do not belong to the same instance, then the actions can be swapped to get the new trace given below:

   $$\sigma_0\alpha_1\sigma_1 \dots \sigma_i\alpha_{i+2}{\sigma'}_{i+1}\alpha_{i+1}\sigma_{i+2} \dots \sigma_n$$

The two transformations just described are called the *allowable operations* on a trace. I use $\pi \Rightarrow \pi'$ to denote that trace π' is obtained by applying one of the *allowable operations* to the trace π. The reflexive transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^\star$. The following lemma regarding $\Rightarrow$ is crucial in proving the correctness of the partial order reduction.

Lemma 6.3.1 *Consider two traces π and π' such that $\pi \Rightarrow \pi'$. In this case $\pi'\mathcal{K}\pi$.*

Proof: Let π and π' be the following traces:

$$\pi = \sigma_0\alpha_1\sigma_1 \ldots \sigma_i\alpha_{i+1}\sigma_{i+1}\alpha_{i+2}\sigma_{i+2} \ldots \sigma_n$$

$$\pi' = \sigma_0\alpha_1\sigma_1 \ldots \sigma_i\alpha_{i+2}{\sigma'}_{i+1}\alpha_{i+1}\sigma_{i+2} \ldots \sigma_n$$

Consider two different cases, one for each of the allowable operations.

- α_{i+2} is an invisible action

  Since α_{i+2} is invisible, it does not change the values of any of the atomic propositions either in AP_H or in AP_Ω. Therefore this action does not change the labeling nor the adversary knowledge of any state in either π or π', regardless of when it is executed. This means that both the state before the invisible action and the state after the invisible action can be included in the same partition that is used in the definition of "$\mathcal{K}$". Therefore the traces can be partitioned as follows:

  $$\overbrace{\sigma_0}^{A_0}\alpha_1\overbrace{\sigma_1}^{A_1}\ldots\overbrace{\sigma_i}^{A_i}\alpha_{i+1}\overbrace{\sigma_{i+1}\alpha_{i+2}\sigma_{i+2}}^{A_{i+1}}\alpha_{i+3}\overbrace{\sigma_{i+3}}^{A_{i+2}}\ldots\overbrace{\sigma_{n-1}}^{A_{n-2}}\alpha_n\overbrace{\sigma_n}^{A_{n-1}}$$

  $$\underbrace{\sigma_0}_{B_0}\alpha_1\underbrace{\sigma_1}_{B_1}\ldots\underbrace{\sigma_i\alpha_{i+2}{\sigma'}_{i+1}}_{B_i}\alpha_{i+1}\underbrace{\sigma_{i+2}}_{B_{i+1}}\alpha_{i+3}\underbrace{\sigma_{i+3}}_{B_{i+2}}\ldots\underbrace{\sigma_{n-1}}_{B_{n-2}}\alpha_n\underbrace{\sigma_n}_{B_{n-1}}$$

  Since $A_j = B_j = \{\sigma_j\}$ for $0 \leq j < i$, these partitions have matching labels and adversary knowledge. The same holds for the partitions $A_j = B_j = \{\sigma_{j+1}\}$ for $i + 1 < j < n$. In partitions A_i and B_i, all states agree on both labels and adversary knowledge because $A_i \cup B_i = \{\sigma_i, {\sigma'}_{i+1}\}$ and $\sigma_i \stackrel{\alpha_{i+2}}{\to} {\sigma'}_{i+1}$ with α_{i+2} being invisible, so σ_i and ${\sigma'}_{i+1}$ must have matching labels and adversary knowledge. Similarly, in partitions A_{i+1} and B_{i+1}, all states agree on both labels and adversary knowledge because $A_{i+1} \cup B_{i+1} = \{\sigma_{i+1}, \sigma_{i+2}\}$ and $\sigma_{i+1} \stackrel{\alpha_{i+2}}{\to} \sigma_{i+2}$ with α_{i+2} being invisible, so σ_{i+1} and σ_{i+2} must have matching labels and adversary knowledge.

- α_{i+2} is a **send** action

  Since α_{i+2} is a **send** action, it is semi-invisible. As in the previous case, α_{i+2} does not affect the labeling of any states; however it can increase the adversary's knowledge. The restrictions on the partitioning still allow the state preceding and the state following the action α_{i+2} to be placed in the same partition. The two traces are partitioned as before:

$$\overbrace{\sigma_0}^{A_0}\ \alpha_1\ \overbrace{\sigma_1}^{A_1} \ldots \overbrace{\sigma_i}^{A_i}\ \alpha_{i+1}\ \overbrace{\sigma_{i+1}\alpha_{i+2}\sigma_{i+2}}^{A_{i+1}}\ \alpha_{i+3}\ \overbrace{\sigma_{i+3}}^{A_{i+2}} \ldots \overbrace{\sigma_{n-1}}^{A_{n-2}}\ \alpha_n\ \overbrace{\sigma_n}^{A_{n-1}}$$

$$\underbrace{\sigma_0}_{B_0}\ \alpha_1\ \underbrace{\sigma_1}_{B_1} \ldots \underbrace{\sigma_i\alpha_{i+2}\sigma'_{i+1}}_{B_i}\ \alpha_{i+1}\ \underbrace{\sigma_{i+2}}_{B_{i+1}}\ \alpha_{i+3}\ \underbrace{\sigma_{i+3}}_{B_{i+2}} \ldots \underbrace{\sigma_{n-1}}_{B_{n-2}}\ \alpha_n\ \underbrace{\sigma_n}_{B_{n-1}}$$

As in the previous case, $A_j = B_j = \{\sigma_j\}$ for $0 \leq j < i$ and $A_j = B_j = \{\sigma_{j+1}\}$ for $i+1 < j < n$ so each of those pairs of partitions agree on both state labeling and adversary knowledge. Also as in the previous case, all states in partitions A_i and B_i agree on the state labeling because $A_i \cup B_i = \{\sigma_i, \sigma'_{i+1}\}$ and $\sigma_i \overset{\alpha_{i+2}}{\rightarrow} \sigma'_{i+1}$ with α_{i+2} being semi-invisible. Since α_{i+2} is semi-invisible, $I_\Omega(\sigma_i) \subseteq I_\Omega(\sigma'_{i+1})$ which is enough to satisfy condition 4 of the definition of $\mathcal{K}$. Similarly A_{i+1} and B_{i+1} agree on the state labeling because $A_{i+1} \cup B_{i+1} = \{\sigma_{i+1}, \sigma_{i+2}\}$ and $\sigma_{i+1} \overset{\alpha_{i+2}}{\rightarrow} \sigma_{i+2}$ with α_{i+2} being semi-invisible. Since α_{i+2} is semi-invisible, $I_\Omega(\sigma_{i+1}) \subseteq I_\Omega(\sigma_{i+2})$ which is enough to satisfy condition 4 of the definition of $\mathcal{K}$.

We have considered all of the *allowable operations*. For each operation, we constructed partitions for π and π' that satisfy Definition 6.2.1. Therefore, we conclude that $\pi'\mathcal{K}\pi$.

□

Using Corollary 6.2.2 and Lemma 6.3.1, the proof of the following theorem is transparent.

Theorem 6.3.2 *Assume that ϕ is an admissible specification and that π and π' are two traces such that $\pi \Rightarrow^\star \pi'$. Then $\pi' \models \phi$ implies that $\pi \models \phi$, or equivalently $\pi \not\models \phi$ implies that $\pi' \not\models \phi$.*

Proof If $\pi \Rightarrow^\star \pi'$, then there is some sequence of traces π_i such that

$$\pi = \pi_0 \Rightarrow \pi_1 \Rightarrow \pi_2 \Rightarrow \cdots \Rightarrow \pi_n = \pi'.$$

It follows from Lemma 6.3.1, that

$$\begin{aligned} \pi' = \pi_n \quad &\mathcal{K} \quad \pi_{n-1}; \\ \pi_{n-1} \quad &\mathcal{K} \quad \pi_{n-2}; \\ &\vdots \\ \pi_1 \quad &\mathcal{K} \quad \pi_0 = \pi. \end{aligned}$$

Through repeated use of Corollary 6.2.2, we then get

$$\begin{array}{rcr} \pi_n \models \phi & \text{implies} & \pi_{n-1} \models \phi; \\ \pi_{n-1} \models \phi & \text{implies} & \pi_{n-2} \models \phi; \\ & \vdots & \\ \pi_1 \models \phi & \text{implies} & \pi_0 \models \phi. \end{array}$$

We conclude $\pi' \models \phi$ implies $\pi \models \phi$.
□

6.4 The Algorithm

We have seen that if $\pi \Rightarrow^* \pi'$, then the correctness of the trace π' implies the correctness of trace π. In other words, a model checker would only need to check the trace π', and it could disregard the trace π. What remains is to see how BRUTUS computes what traces to consider and what traces to ignore. It should be the case that for any trace π **not** considered by BRUTUS, there is another trace π' that **is** considered by BRUTUS, such that, $\pi \Rightarrow^* \pi'$.

In the discussion that follows, I borrow heavily from the established terminology of partial order reduction in the literature. I have already introduced the notion of invisible actions. I now introduce the notion of *independent actions*. Intuitively, the order in which independent actions are executed is really immaterial to the behavior of the system. The partial order reduction relies on exploiting these independent actions so that one can avoid enumerating all possible interleavings. Since the order of the actions does not matter, an arbitrary order can be chosen. However, it is not enough to construct the entire state space and then consider only some of the traces; one must avoid constructing the entire state space to begin with.

In order to avoid constructing the entire state space, typically one does not fully expand all successor states to a particular state. Computing an optimal set of states to expand is NP-hard [65], so different heuristics are used when computing the set of actions to expand. These heuristics are given different names in the literature. Peled uses the notion of *ample sets* [65, 67], while Wolper and Godefroid introduce *persistent sets* [82], and Valmari uses *stubborn sets* [79]. While the details of how these sets are computed differ, the underlying idea is the same: from any state, expand only a subset of the possible actions.

I also follow this approach. Although I borrow the ample set terminology used by Peled, I do not compute ample sets the same way. I do use the same dependency criteria used by all three to determine when actions can be permuted. However, I have different criteria for when such permutations are allowed with respect to a specification. Recall that I am **not** interested in finding equivalence classes of traces that agree on the specification. In the previous section, I demonstrated that this was not necessary for typical security properties. However, I still need to make sure that BRUTUS does not throw away too many traces, and that the transformations performed do result in valid traces of the system (traces that **are** checked). First, some notation and definitions are needed.

The set of actions enabled in a state σ is $en(\sigma)$. The set of states in which the action α is enabled will be denoted en_α. The state resulting from executing action α in state σ is $\alpha(\sigma)$. Independent actions are commonly defined in the literature as follows:

Definition 6.4.1 *A pair of actions α and β are called* independent *if the following two conditions hold:*

1. *For any state $\sigma \in en_\alpha$, $\beta \in en(\sigma)$ implies $\beta \in en(\alpha(s))$.*

2. *For any state $\sigma \in en_\alpha \cap en_\beta$, $\alpha(\beta(\sigma)) = \beta(\alpha(\sigma))$.*

Intuitively, the first condition ensures that independent actions cannot disable each other, while the second condition ensures that the actions can be permuted safely (the resulting state is the same in either order).

Originally, partial order theory required that independent actions not disable *nor enable* each other. This is equivalent to changing the first condition above to:

1. For any state $\sigma \in en_\alpha,\ \beta \in en(\sigma)$ **iff** $\beta \in en(\alpha(s))$.

As noted in [67], this original condition would ensure that whenever independent actions in a valid trace π are swapped to obtain a trace π', π' is also a valid trace of the model. If actions that are independent according to Definition 6.4.1 are arbitrarily swapped, the resulting trace may not be a valid trace of the model. This is because an action may be moved before the action that enables it. However, the original requirement is stronger than is needed; the ample set methods (as well as the persistent set methods and the stubborn set methods) do not arbitrarily permute independent actions. Instead, they expand only subsets of all the enabled actions. Therefore we need not concern ourselves with the possibility of expanding actions that are disabled.

Using this definition of independence, we now consider which actions in our model are dependent and which are independent.

- Any action is **dependent** with itself. In our model any action is disabled once it is taken.
- Two instantiated receive actions that correspond to the same read template in an instance are **dependent**. This is clear because an instance only accepts one message per receive action in its process description. In effect, the receive action as given in the process description is more of an action *template*, and once one matching receive is performed, all other matching receives are disabled.
- All other actions are **independent**. There is no other way for an action to become disabled than the two ways mentioned above, and the order in which the actions are taken does not affect the final resulting state.

Now that we know what the independent actions in the model are, we must define the ample sets for the states in the model. I require that $ample(\sigma)$, the ample set for a state σ, satisfy the following:

Definition 6.4.2 *A set A of actions enabled at a state σ (i.e., $A \subseteq en(\sigma)$) is called an* ample set *if it satisfies the the following 4 conditions.*

C0 *If* ample$(\sigma) = \{\}$ *then* en$(\sigma) = \{\}$.

C1 *For any trace*

$$\sigma = \sigma_0 \stackrel{\alpha_1}{\to} \sigma_1 \stackrel{\alpha_2}{\to} \sigma_2 \stackrel{\alpha_3}{\to} \cdots \stackrel{\alpha_n}{\to} \sigma_n$$

starting from σ and consisting of actions $\alpha_i \notin$ ample(σ), all α_i are independent with respect to all actions in ample(σ).

C2 *If* en(σ) *contains any invisible or semi-invisible actions, then* ample(σ) *is a set containing a single invisible or semi-invisible action.*

C3 *If* en(σ) *contains no invisible actions and no semi-invisible actions, then* ample$(\sigma) =$ en(σ).

I use *ample*(σ) to denote the ample for σ that is computed by Brutus.

The first two conditions (**C0** and **C1**) are the same first two conditions that occur in all three partial order reduction techniques mentioned before (ample sets, persistent sets, and stubborn sets). Condition **C0** guarantees that artificial deadlocks are not introduced into the model. We shall see that condition **C1** guarantees that any trace not considered can be transformed into a trace that is considered. Conditions **C2** and **C3** satisfy the restrictions on the allowed transformations that appear in the definition of "$\Rightarrow$" in Definition 6.3.2. As discussed in Section 6.3, these guarantee that the trace that is considered violates the specification whenever the trace that is ignored violates the specification. For readers familiar with partial order reduction, the proofs that follow will be similar to what is found in the literature.

Theorem 6.4.1 *A depth-first search algorithm that only expands actions in an ample set satisfying conditions* **C0**-**C3** *is "safe" in the sense that for any trace π that it does not consider, there is a trace π' that it does consider such that $\pi \Rightarrow^* \pi'$.*

Proof: Let $\mathcal{A}_{\mathcal{PO}}$ be the depth-first search algorithm that uses conditions **C0**-**C3** above to determine the set of actions it will expand from a particular state. Let

$$\pi = \sigma_0\alpha_1\sigma_1\alpha_2 \cdots \alpha_n\sigma_n$$

be a trace of the system that would be considered by a normal exhaustive depth-first search. A simple re-arrangement of the actions appearing in π will lead to a trace π' that is considered by $\mathcal{A}_{\mathcal{PO}}$.

To construct the new trace π', take the longest prefix of π that is considered by $\mathcal{A}_{\mathcal{PO}}$. Let this prefix be

$$\pi_k = \sigma_0\alpha_1\sigma_1\alpha_2\cdots\alpha_k\sigma_k.$$

If $k = n$ then $\pi = \pi'$. Otherwise, for $0 < i \leq k$, $\alpha_i \in \mathit{ample}(\sigma_{i-1})$ and $\alpha_{k+1} \notin \mathit{ample}(\sigma_k)$. By condition **C0**, $\mathit{ample}(\sigma_k) \neq \{\}$ since $\mathit{en}(\sigma_k)$ contains at least α_{k+1}. In addition, some action $\beta \in \mathit{ample}(\sigma_k)$ must appear in

$$\pi^k = \sigma_k\alpha_{k+1}\sigma_{k+1}\alpha_{k+2}\cdots\alpha_n\sigma_n,$$

the suffix of π starting at σ_k. This is because condition **C1** guarantees that until an action from $\mathit{ample}(\sigma_k)$ is executed, all actions occurring after the state σ_k are independent with respect to $\mathit{ample}(\sigma_k)$. Therefore, all the actions in $\mathit{ample}(\sigma_k)$ would continue to be enabled until one of them is executed. (This is why Wolper and Godefroid call these *persistent sets* [82].) Since all the traces must be finite (our models contain no loops), some action in $\mathit{ample}(\sigma_k)$ must eventually be executed in π^k. Let α_l be the first action in π^k that appears in $\mathit{ample}(\sigma_k)$. Because condition **C1** guarantees independence with respect to all actions α_i where $k \leq i < l$, the action α_l can be permuted with the preceding action, and the action before that, and so on until it is permuted with action α_{k+1}.

The result is a trace π'' such that $\pi \Rightarrow^* \pi''$. This is because the action that was moved earlier in the trace (α_l) must be invisible or semi-invisible. Recall that α_l was chosen because it is the first action in $\mathit{ample}(\sigma_k)$ that appears after σ_k. Since α_l is not taken from state σ_k, it must be the case that $\mathit{ample}(\sigma_k) \neq \mathit{en}(\sigma_k)$. By condition **C2**, $\mathit{ample}(\sigma_k)$ must be a set containing a single invisible or semi-invisible action, and so $\alpha_l \in \mathit{ample}(\sigma_k)$ must be an invisible or semi-invisible action.

While $\pi \Rightarrow^* \pi''$, it is not necessarily the case that π'' is the π' we are looking for. (Recall that π' is the trace we are interested in because it is considered by $\mathcal{A}_{\mathcal{PO}}$.) However, π'' is "closer" to the trace π' in the sense that the largest prefix of π considered by $\mathcal{A}_{\mathcal{PO}}$ has length k while the largest prefix of π'' considered by $\mathcal{A}_{\mathcal{PO}}$ has length at least $k+1$. Since the traces are finite, this step can be repeated a finite number of times before the trace π' is reached.

Formally, the theorem is proved via induction on $D = n - k$, the difference in length between trace π and π_k, the largest prefix of π considered by $\mathcal{A}_{\mathcal{PO}}$.

Base case: If $D = 0$, then $n = k$ and the largest prefix of π considered by $\mathcal{A}_{\mathcal{PO}}$ is π itself. In this case, let $\pi' = \pi$ and clearly $\pi \Rightarrow^* \pi'$.

Inductive case: If $D > 0$ then the largest prefix of π considered by $\mathcal{A}_{\mathcal{PO}}$ has length $k < n$. In this case, the discussion above shows that there is some action $\alpha_l \in \mathit{ample}(\sigma_k)$ for $l > k+1$ that can be moved earlier in the trace π so that it is executed from state σ_k. This results in a new trace π'' whose largest prefix considered by $\mathcal{A}_{\mathcal{PO}}$ has length at least $k+1$ and which is related to the original trace π via $\pi \Rightarrow^* \pi''$. For this trace then, the new difference D' is no more than $n - k - 1$ which is smaller than $D = n - k$, so by the inductive hypothesis, there is a trace π' related to π'' via $\pi'' \Rightarrow^* \pi'$ that is considered by $\mathcal{A}_{\mathcal{PO}}$. Since $\pi \Rightarrow^* \pi''$ and $\pi'' \Rightarrow^* \pi'$, it follows that $\pi \Rightarrow^* \pi'$.

□

All that remains is to show that the algorithm described in Section 4.4 and given in Figure 4.9 guarantees conditions **C0**-**C3**. Fortunately, this is fairly straightforward.

C0 This condition is trivially met because either $\mathit{ample}(\sigma)$ contains one action or it is equal to $\mathit{en}(\sigma)$. Therefore, if $\mathit{ample}(\sigma) = \{\}$, then $\mathit{en}(\sigma) = \{\}$.

C1 Recall that this condition requires that for any trace

$$\sigma = \sigma_0 \stackrel{\alpha_1}{\rightarrow} \sigma_1 \stackrel{\alpha_2}{\rightarrow} \sigma_2 \stackrel{\alpha_3}{\rightarrow} \cdots \stackrel{\alpha_n}{\rightarrow} \sigma_n$$

starting from σ and consisting of actions $\alpha_i \notin \mathit{ample}(\sigma)$, all α_i are independent with respect to all transitions in $\mathit{ample}(\sigma)$. There are two cases depending on whether $\mathit{en}(\sigma)$ contains any invisible or semi-invisible actions.

Case 1: $\mathit{en}(\sigma)$ contains an invisible or a semi-invisible action. In this case $\mathit{ample}(\sigma) = \{\beta\}$ for some invisible or semi-invisible action β. Invisible actions are internal actions and semi-invisible actions are **<u>send</u>** actions. As discussed earlier, an internal action is always independent with any action other than itself. Recall that an internal action cannot disable any action other than itself. The same is true if β is a **<u>send</u>** action. A **<u>send</u>** action does not disable any action other than itself. Therefore, β is independent of *all* other actions, and in particular, $ample(\sigma) = \{\beta\}$ is independent with respect to all the α_i's.

Case 2: $en(\sigma)$ contains no invisible or semi-invisible action. In this case, the algorithm sets $ample(\sigma)$ to be all of $en(\sigma)$. This set trivially satisfies condition **C1** since there can then be no trace starting from σ that *does not* use an action from $ample(\sigma)$. In other words, there is no action α_1 such that $\alpha_1 \in en(\sigma)$ and $\alpha_1 \notin ample(\sigma)$.

C2 This condition is trivial because the first thing the algorithm does is look for a single invisible or semi-invisible action that is enabled and expands it.

C3 Again, this condition is trivial because it is exactly what the algorithm does. If there is no invisible or semi-invisible action enabled in the state, then all actions are expanded from that state.

I conclude with a short comparison of my partial order reduction with other reductions described in the literature. As mentioned earlier, I borrow heavily from previous work in partial order reduction. My conditions **C0** and **C1** appear in the other well known reduction techniques. Condition **C1** is basically the condition required of *persistent sets* [81, 82]. When the subset of actions to be expanded is computed, there is a further restriction that the persistent set be non-empty (condition **C0**). These conditions are sufficient to preserve deadlocks or terminal states for systems with finite traces.

If the system has infinite traces (the model contains cycles), then an additional condition is required. This condition basically requires that at least one state in a cycle is fully expanded [67, 79, 81]. Since my models contain no cycles and all traces are finite, the theory is simplified in this case. I am not investigating infinite behavior nor fairness issues and so I do not need to enforce this extra condition.

Finally, other restrictions are placed on ample sets to ensure that the partial order reduction preserves specifications (instead of just deadlocks).

The key restriction here is that actions be *invisible* [66, 78]. In other words, the actions cannot affect the atomic propositions appearing in the specification. This is necessary to preserve formulas in LTL-X (LTL without the next-time operator). This is also the restriction I place on atomic propositions that refer to the honest agents. Because my models include a notion of knowledge that is monotonically increasing, I am able to generalize the notion of invisibility to semi-invisibility. Here, I exploit the fact that atomic propositions referring to the adversary require that the adversary **not** know something, and that **send** actions can only increase the adversary's knowledge. Moving a **send** action forward (sooner) in a trace, can only make a specification "more false". While this generalization does not preserve the truth value of formulas in the logic (two "equivalent" traces may not agree on the specification), I am able to guarantee that for every faulty trace that is not considered, there is a faulty trace that is considered (errors are preserved). For the sake of completeness, I should mention that there is yet another restriction often placed on the ample sets. This restriction is that for any state σ, either $ample(\sigma)$ contains exactly one action or $ample(\sigma) = en(\sigma)$ [67]. This restriction is required to preserve branching properties. While my ample sets also satisfy this property, this is a result of the kinds of models being investigated. I am not currently checking for branching properties.

Chapter 7

Experiments

I now describe some of the experiments I conducted using Brutus. For each experiment, I introduce and describe the protocol that is being analyzed. I also describe how it is modeled in Brutus. I then indicate the specifications that were checked and the results of the verification. In the last section, I discuss how effective the symmetry and partial order reductions are in reducing the time required to perform the verification.

7.1 Needham-Schroeder Public Key

The Needham-Schroeder public key authentication protocol has received much attention since a new attack was found by Gavin Lowe in 1996 [37]. What follows is a presentation of my analysis of this protocol using Brutus. The structure of this protocol is given below. I will assume that the initiator is agent A and that it wishes to authenticate with agent B.

1. First, the initiator, A, generates a nonce, N_a, (which we can assume is a random number), and then encrypts the pair N_a, A with B's public key. A then constructs the message $A, B, \{N_a, A\}_{K_B}$ which it sends to B.

$$A \rightarrow B : A, B, \{N_a, A\}_{K_B}$$

2. Upon receiving message number 1, B uses its private key to decrypt $\{N_a, A\}_{K_B}$ and recover the identity of the initiator, A, and its nonce,

N_a. It then generates its own nonce, N_b, encrypts the pair N_a, N_b with A's public key, and constructs the message $A, B, \{N_a, N_b\}_{K_A}$ which it sends to A.

$$B \to A : A, B, \{N_a, N_b\}_{K_A}$$

3. Upon receiving message number 2, A uses its private key to decrypt $\{N_a, N_b\}_{K_A}$. It is now convinced of B's identity and that B possesses N_a, a shared secret that A can include in new messages for identification. It now replies to B by encrypting N_b with B's public key and sending the message $A, B, \{N_b\}_{K_B}$ to B.

$$A \to B : A, B, \{N_b\}_{K_B}$$

4. Upon receiving message number 3, B can once again use its private key to decrypt $\{N_b\}_{K_B}$. Now B is convinced of A's identity and that A possesses N_b, a shared secret that B can include in new messages for identification.

Let us now look at how this protocol is modeled in Brutus. Each role in the protocol will be modeled as a process (a sequence of actions). Every instance in the model represents a principal participating once in the protocol. The role of the initiator is modeled by the sequence of actions in Figure 7.1. This role is parameterized in pr, the name of the initiator, and n_a, the value of its random number or nonce. The internal action "begin-initiate" is used to mark the point in the protocol when the initiator has begun execution. The internal action "end-initiate" is used to mark the point in the protocol when the initiator has finished executing the protocol. Each of these internal actions has a variable containing the name of the principal with whom the initiator is trying to authenticate. These actions are included in the model so that the requirements can refer to when an instance begins or ends execution of the protocol.

There is an analogous role for the responder. The definition for this process is given by the sequence of actions in Figure 7.2. It is parameterized in pr, the name of the responder, and n_b, the value of its random number. Again, the internal actions "begin-respond" and "end-respond"

$INITIATOR =$
choose (b)
internal (“`begin-initiate`”, b)
send $\langle pr, b, \{n_a, pr\}_{pubkey(b)} \rangle$
receive $\langle pr, b, \{n_a, n_b\}_{pubkey(pr)} \rangle$
send $\langle pr, b, \{n_b\}_{pubkey(b)} \rangle$
internal (“`end-initiate`”, b)

Figure 7.1: Needham-Schroeder Initiator

$RESPONDER =$
receive $\langle a, pr, \{n_a, a\}_{pubkey(pr)} \rangle$
internal (“`begin-respond`”, a)
send $\langle a, pr, \{n_a, n_b\}_{pubkey(a)} \rangle$
receive $\langle a, pr, \{n_b\}_{pubkey(pr)} \rangle$
internal (“`end-respond`”, a)

Figure 7.2: Needham-Schroeder Responder

mark the points in the protocol where the responder has begun execution and where the responder has finished execution. The argument, a, indicates the principal with whom the responder is authenticating.

One of the models I verified contained 4 instances. This model contained one initiator instance and one responder instance for principal "A" and the same for principal "B". The model is the composition of the four instances in Figure 7.3 with an instance of the adversary.

To perform the analysis, one must also specify the requirements for the protocol. Three different properties are checked. The first is that if principal A has finished executing a protocol session with B then B must have participated in a protocol session with A. Similarly, if principal B has finished executing a protocol session with A then A must have participated in a protocol session with B. The second property checks that the nonces which are intended to be shared secrets are kept secret from the adversary. The final property is a non-repudiation property and states that when A finishes a protocol session with B, B knows A's nonce and vice-versa. Each of these is presented in more detail below.

- *Authentication.* This property, which is also called correspondence by Woo and Lam [83], can be used as a generic requirement for authentication protocols. Intuitively, the requirement is that if some principal A has finished executing an authentication protocol with B, then B must have participated in the protocol. This is formalized in our logic with two formulas, one for the initiator and one for the responder. The first formula is:

$$\forall A_0 \;.\; A_0 \;\; \underline{\textbf{internal}} \;\; (\text{“}\texttt{end-initiate}\text{”}, A_0.b) \rightarrow$$

$$\exists B_0 \left[\begin{array}{c} (B_0.pr = A_0.b) \;\wedge \\ \Diamond_P (B_0 \;\; \underline{\textbf{internal}} \;\; (\text{“}\texttt{begin-respond}\text{”}, A_0.pr)) \end{array} \right]$$

This formula states that for all instances A_0, if A_0 has performed an "end-initiate" internal action with a principal that it believes is its partner in the authentication protocol, then there exists an instance B_0 of that partner such that at some point in the past B_0 performed a "begin-respond" internal action with the principal of instance A_0. In other words, for all initiator instances I, if I has finished executing with some principal, then some instance R of that other principal must have at least started executing the protocol with the principal

$$\mathcal{I}_1 : \begin{cases} H_1 = \text{“}\texttt{A1}\text{”} \\ B_1 = \{(pr, \text{“}\texttt{A}\text{”}), (n_a, \text{“}\texttt{Na1}\text{”})\} \\ I_1 = \{A, B, \Omega, K_A, K_B, K_\Omega, K_A^{-1}, Na1\} \\ P_1 = \mathit{INITIATOR} \end{cases}$$

$$\mathcal{I}_2 : \begin{cases} H_2 = \text{“}\texttt{A2}\text{”} \\ B_2 = \{(pr, \text{“}\texttt{A}\text{”}), (n_b, \text{“}\texttt{Na2}\text{”})\} \\ I_2 = \{A, B, \Omega, K_A, K_B, K_\Omega, K_A^{-1}, Na2\} \\ P_2 = \mathit{RESPONDER} \end{cases}$$

$$\mathcal{I}_3 : \begin{cases} H_3 = \text{“}\texttt{B1}\text{”} \\ B_3 = \{(pr, \text{“}\texttt{B}\text{”}), (n_b, \text{“}\texttt{Nb1}\text{”})\} \\ I_3 = \{A, B, \Omega, K_B, K_B, K_\Omega, K_B^{-1}, Nb1\} \\ P_3 = \mathit{INITIATOR} \end{cases}$$

$$\mathcal{I}_4 : \begin{cases} H_4 = \text{“}\texttt{B2}\text{”} \\ B_4 = \{(pr, \text{“}\texttt{B}\text{”}), (n_b, \text{“}\texttt{Nb2}\text{”})\} \\ I_4 = \{A, B, \Omega, K_B, K_B, K_\Omega, K_B^{-1}, Nb2\} \\ P_4 = \mathit{RESPONDER} \end{cases}$$

Figure 7.3: Needham-Schroeder Model

executing I. This requirement guarantees the participation of the responder in an authentication protocol. The model satisfies this property.

There is an analogous property requiring the participation of the initiator. The formula for this property is:

$$\forall B_0 \,.\, B_0 \;\underline{\textbf{internal}}\; (\text{“}\texttt{end-respond}\text{”}, B_0.a) \rightarrow$$

$$\exists A_0 \left[\begin{array}{c} (A_0.pr = B_0.a) \wedge \\ \Diamond_P (A_0 \;\underline{\textbf{internal}}\; (\text{“}\texttt{begin-initiate}\text{”}, B_0.pr)) \end{array} \right]$$

This formula specifies that, if a responder instance B_0 has finished executing the protocol with some principal $B_0.a$, then there must be some initiator instance A_0, executing on behalf of the the principal $B_0.a$, that has participated in the protocol. This property is violated by the protocol. Figure 7.4 contains the counterexample trace provided by BRUTUS. Note that at the end of the trace, $A2$ has finished responding with B, but there is no instance of B, ($B1$ or $B2$) that has initiated with A. This attack actually occurs in a model with a single initiator and a single responder instance. Figure 7.5 illustrates this attack with a single initiator A and a single responder B.

- *Secrecy.* The nonces exchanged in the Needham-Schroeder protocol are intended to be shared secrets. As such, the adversary should have no knowledge of them unless an honest agent is trying to authenticate with the adversary. The formula specifying secrecy is:

$$\forall X \,.\, (H_\Omega \;\textbf{Knows}\; X.n_a \vee H_\Omega \;\textbf{Knows}\; X.n_b) \rightarrow$$

$$\Diamond_P \left[\begin{array}{c} X \;\underline{\textbf{internal}}\; (\text{“}\texttt{begin-initiate}\text{”}, \Omega) \vee \\ X \;\underline{\textbf{internal}}\; (\text{“}\texttt{begin-respond}\text{”}, \Omega) \end{array} \right]$$

In other words, if the adversary knows the value of someone else's nonce, then that instance must be executing the protocol with the adversary. This property is also violated by the same trace as before. On line 21 of the counterexample trace (Figure 7.4), $B2$ sends $A2$'s nonce encrypted with the key of the adversary (Ω). At that point the adversary knows $A2$'s nonce but $A2$ is not trying to authenticate with the adversary.

```
A1 choose Principal B
A1 internal ("begin-initiate", Principal B)
A1 send Principal A, Principal B,
        {Nonce Na1, Principal A}_Pubkey(Principal B)
B2 choose Principal Ω
B2 internal ("begin-initiate", Principal Ω)
B2 send Principal B, Principal Ω,
        {Nonce Nb2, Principal B}_Pubkey(Principal Ω)
B1 receive Principal A, Principal B,
           {Nonce Nb2, Principal A}_Pubkey(Principal B)
B1 internal ("begin-respond", Principal A)
B1 send Principal B, Principal A,
        {Nonce Nb2, Nonce Nb1}_Pubkey(Principal A)
A2 receive Principal B, Principal A,
           {Nonce Nb2, Principal B}_Pubkey(Principal A)
A2 internal ("begin-respond", Principal B)
A2 send Principal A, Principal B,
        {Nonce Nb2, Nonce Na2}_Pubkey(Principal B)
B2 receive Principal Ω, Principal B,
           {Nonce Nb2, Nonce Na2}_Pubkey(Principal B)
B2 send Principal B, Principal Ω,
        {Nonce Na2}_Pubkey(Principal Ω)
B2 internal ("end-initiate", Principal Ω)
A2 receive Principal B, Principal A,
           {Nonce Na2}_Pubkey(Principal A)
A2 internal ("end-respond", Principal B)
```

Figure 7.4: Needham-Schroeder counterexample

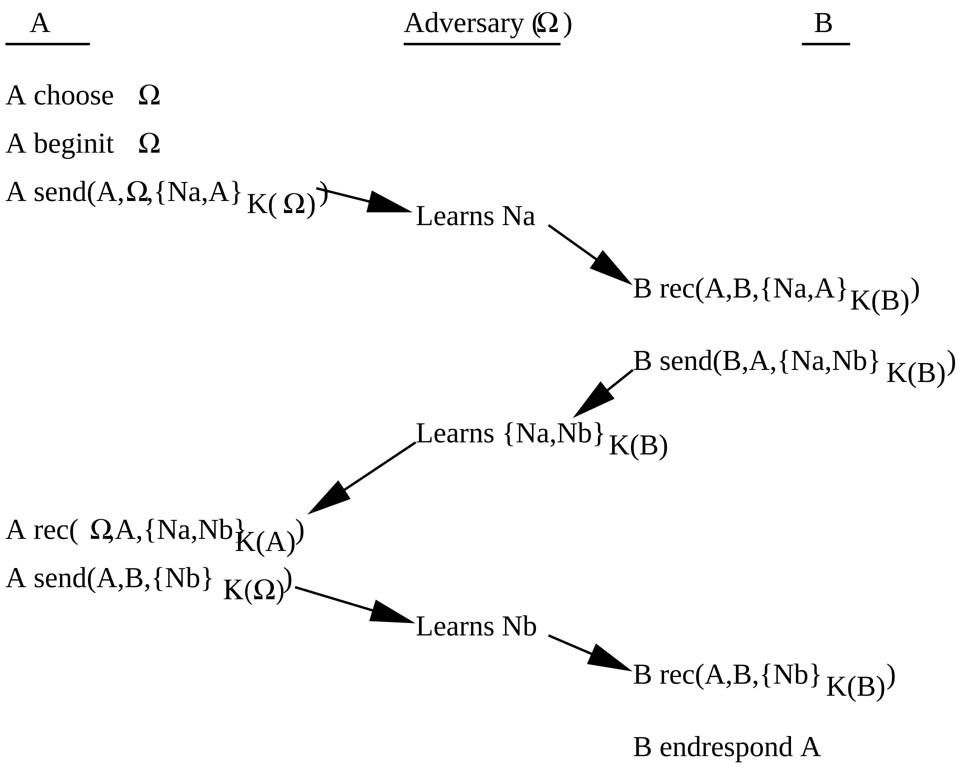

Figure 7.5: Needham-Schroeder attack

- *Non-repudiation.* When the protocol finishes executing, both parties should possess the correct shared secrets (the nonces). This is a weaker property than what is typically meant by non-repudiation. This requirement does not guarantee that one principal can prove that the other possesses the secret. This specification simply states that there is no execution in which some principal A finishes authenticating with B but B does not know A's nonce.

$$\forall A_0 \;.\; A_0 \;\; \underline{\textbf{internal}} \;\; (\text{“}\texttt{end-initiate}\text{”}, A_0.b) \rightarrow$$

$$\exists B_0 \; [(B_0.pr = A_0.b) \;\wedge\; (B_0 \;\; \textbf{Knows} \;\; A_0.n_a)]$$

  Like the authentication requirement, this requirement has a second formula with the roles of the initiator and responder reversed.

$$\forall B_0 \;.\; B_0 \;\; \underline{\textbf{internal}} \;\; (\text{“}\texttt{end-respond}\text{”}, B_0.a) \rightarrow$$

$$\exists A_0 \; [(A_0.pr = B_0.a) \;\wedge\; (A_0 \;\; \textbf{Knows} \;\; B_0.n_b)]$$

  Unlike the authentication requirement, however, the protocol satisfies this property. It is the case that if an agent A believes it has finished authenticating with agent B, it has sent out its nonce encrypted with B's key and has received the nonce back again. The only way this is possible is if B decrypted the nonce. Therefore, B must have possession of it, even in the traces where the other two requirements are violated.

Gavin Lowe has suggested a very simple fix for this protocol [37]. In the second message, the responder's name B is added to the encrypted portion so that the message now looks like $A, B, \{N_a, N_b, B\}_{K_A}$. This slightly changes the protocol so that now at step 2, B generates this new message and at step 3, A decrypts the message and checks not only for its own nonce N_a, but also for the identity of B, the principal with whom A is trying to authenticate. When this change is made to the model of the Needham-Schroeder protocol, Brutus does not find any counterexample to the properties described above.

7.2 1KP

The next case study is the 1KP protocol. This protocol is a member of the iKP family of protocols for secure electronic payments over the

$\underline{SALT_C}$: random number generated by C used to salt *DESC*

$\underline{PRICE}$: amount and currency

$\underline{DATE}$: merchant's date/time stamp

$\underline{NONCE_M}$: merchant's nonce (random number)

$\underline{ID_M}$: merchant's ID

$\underline{TID_M}$: transaction ID (unique)

$\underline{DESC}$: description of the goods and delivery information

$\underline{CAN}$: customer's account number

$\underline{R_C}$: random number chosen by C to form *CID*

$\underline{Y/N}$: yes or no response from credit card authority.

Figure 7.6: 1KP atomic messages

Internet [4]. The protocol has three participants, a customer, a merchant, and a credit card authority which I refer to as C, M, and A respectively. This protocol is quite a bit more complicated than the Needham-Schroeder protocol. The atomic messages for the protocol are given in Figure 7.6 and some composite fields are defined in Figure 7.7. Also, I use $\mathcal{H}(\cdot)$ to denote a one-way hash function. This hash function is modeled in Brutus by having a special private key called "HASH" that has no inverse. This way, the result of encrypting with "HASH" (applying the hash) can be checked, but there is no way to decrypt the hash (invert the hash function).

The definition of the protocol is given below. It should be noted that there is an assumption that the customer and merchant somehow arrive at the description of the transaction outside of the 1KP protocol. In other words, at the time the protocol is executed, the customer and merchant should already know the values of *DESC* and *PRICE* due to some previous negotiation step.

1. **Initiate:** The customer generates two random numbers, $SALT_C$ and R_C. It also computes $CID = \mathcal{H}(R_C, CAN)$. It then sends a

<u>*CID*</u>**:** a customer pseudo-ID formed by $\mathcal{H}(R_C, CAN)$

<u>*Common*</u>**:** $\mathit{PRICE}, \mathit{ID}_M, \mathit{TID}_M, \mathit{DATE}, \mathit{NONCE}_M, \mathit{CID}, \mathcal{H}(\mathit{DESC}, \mathit{SALT}_C)$

<u>*Clear*</u>**:** $\mathit{ID}_M, \mathit{TID}_M, \mathit{DATE}, \mathit{NONCE}_M, \mathcal{H}(\mathit{Common})$

<u>*SLIP*</u>**:** $\mathit{PRICE}, \mathcal{H}(\mathit{Common}), \mathit{CAN}, R_C$

Figure 7.7: 1KP composite messages

message to the merchant consisting of the random number SALT_C and the customer pseudo-ID *CID*.

$$C \to M : \mathit{SALT}_C, \mathit{CID}$$

2. **Invoice:** The merchant recovers the values of SALT_C and *CID*. It also generates the values NONCE_M and TID_M. The merchant already knows *PRICE*, *DATE*, and its own identity ID_M so it can create the composite message *Common*. It uses all these components, along with the hash function to construct the compound message *Clear* as defined in Figure 7.7 and sends it to the customer.

$$M \to C : \mathit{Clear}$$

3. **Payment:** The customer receives *Clear* and retrieves the values ID_M, *DATE*, TID_M, and NONCE_M. Since the customer already has *PRICE* and *CID*, it can form *Common*. It computes $\mathcal{H}(\mathit{Common})$ and checks that this matches what was received in *Clear*. It already has the information necessary to form *SLIP* (Figure 7.7) and then encrypts this and sends it to the merchant. At this point the customer commits to the transaction.

$$C \to M : \{\mathit{SLIP}\}_{K_A}$$

4. **Auth-Request:** The merchant receives the encrypted payment slip and now needs to get authorization from the credit card authority. It forwards the encrypted slip to the authority, along with *Clear* and

$\mathcal{H}(DESC, SALT_C)$ so that the authority can check the validity of the *SLIP*. At this point, the merchant is committing to the transaction.

$$M \rightarrow A : Clear, \mathcal{H}(DESC, SALT_C), \{SLIP\}_{K_A}$$

5. **Auth-Response:** The authority receives the authorization request and performs the following actions.

   - The authority extracts the values ID_M, TID_M, *DATE*, and $NONCE_M$ and checks that there is no previous request with these same values. It also extracts the value h_1 which is supposed to be $\mathcal{H}(Common)$.
   - The authority decrypts the encrypted *SLIP*. If the decryption is successful, it now has *SLIP* and can extract *PRICE*, *CAN*, R_C, and the value h_2 which is supposed to again be $\mathcal{H}(Common)$.
   - The authority verifies that $h_1 = h_2$ which ensures that the customer and merchant agree on the transaction.
   - It now also has all the components to construct *Common* and does so. It computes $\mathcal{H}(Common)$ and compares this to the value $h_1 = h_2$.
   - Assuming everything is in order, it can authorize the payment by signing the pair $Y, \mathcal{H}(Common)$ and returning this to the merchant.

$$A \rightarrow M : Y, \{Y, \mathcal{H}(Common)\}_{K_A^{-1}}$$

6. **Confirm:** The merchant receives the authorization response. Assuming the response is Y, it then verifies that it has received a valid signature of $Y, \mathcal{H}(Common)$ from the authority. If so, it forwards this on to the customer who can also verify the signature.

$$M \rightarrow C : Y, \{Y, \mathcal{H}(Common)\}_{K_A^{-1}}$$

Like the Needham-Schroeder protocol, the BRUTUS definition of the 1KP protocol consists of a set of instances with at least one instance for each role in the protocol. These roles also consist of mostly send and receive actions with a few internal actions to mark commit points in the protocol as well as when the authority debits and credits accounts. The major difference between the two models is the much larger size of the 1KP protocol. Not only are there three roles instead of two and six messages instead of three, but the size and complexity of the messages is greatly increased. The messages appearing in the Needham-Schroeder protocol were quite simple. In 1KP, the messages contain many more fields (so many, in fact, that composite fields like *Common* and *SLIP* had to be defined in order to simplify the presentation). The messages also have multiple levels of nested encryption, signatures, and hashes. Because of this, the number of messages that the adversary can generate in an attempt to subvert the protocol is much greater as is the number of reachable states. This is because the number of messages that match a template grows exponentially in the length of the message. This is analogous to how the number of strings of a given length grows exponentially with respect to the length of the string. While using typed messages did reduce the number of messages considered by the adversary, this was not enough. Using type information for a particular variable or "slot" in a message, reduces the number of messages that can fit in that "slot"; however, the total number of composite messages is still exponential (with a smaller base). This situation is similar to the difference between the number of octal strings of length n and the number of binary strings of length n. Therefore, to make the verification tractable, I had to make the model smaller. In particular, the BRUTUS model for the protocol has only a single instance of a customer and merchant and two instances of the authority. Still, I was able to perform the analysis, and verify the following properties proposed by Bellare et al. in [4].

- *Proof of transaction by customer.* When the authority debits a certain credit card account by a certain amount, it must have unforgeable proof that the credit card owner has authorized the payment. Bellare et al. argue that *SLIP* provides this unforgeable proof. While verifying that *SLIP* does provide a *proof* is outside the capabilities of BRUTUS, one can check that the authority only debits credit cards when it possesses *SLIP*.

$$\forall A_0 \;.\; (A_0.pr = A) \wedge (A_0 \;\; \underline{\textbf{internal}} \;\; (\text{“}\texttt{debit}\text{”}, \langle A_0.c, A_0.price\rangle)) \rightarrow$$

$$A_0 \;\; \textbf{Knows} \;\; \mathit{SLIP'}$$

Here *SLIP'* is identical to *SLIP* in Figure 7.7 except that $A_0.c$ is used in place of the constant customer account number *CAN*, and $A_0.price$ is used instead of the constant *PRICE*. In other words, one checks that the authority does indeed have a proof with values that are consistent with this particular debit.

- *Unauthorized payment is impossible.* This property requires that the authority debit a customer's account only if the customer has authorized the debit. One is no longer asking if the authority has some kind of proof, but whether or not the customer did actually authorize the debit.

$$\forall A_0 \;.\; (A_0.pr = A) \wedge (A_0 \;\; \underline{\textbf{internal}} \;\; (\text{“}\texttt{debit}\text{”}, \langle A_0.c, A_0.price\rangle)) \rightarrow$$

$$\exists C_0 \;.\; (C_0.pr = A_0.c) \wedge \Diamond_P (C_0 \; \underline{\textbf{internal}} \;\; (\text{“}\texttt{authorize}\text{”}, A_0.price))$$

In other words, if some authority instance A_0 debits the account belonging to principal $A_0.c$ (presumably the customer) by the amount $A_0.price$, then there must be some instance executing on behalf of the principal $A_0.c$, who has authorized a debit of the same amount ($A_0.price$). However, this does not guarantee the absence of a replay attack, where the adversary simply replays previous messages to cause a second transaction to take place. In fact, recall that the first thing the credit card authority does when it receives an authorization request is to make sure that there is no previous request with the same transaction ID, date, and nonce. Since BRUTUS does not allow for this kind of check, one would expect there to be a replay attack in the model. So I checked the following formula.

$$\neg[A1 \;\; \underline{\textbf{internal}} \;\; (\text{“}\texttt{debit}\text{”}, \langle C, A1.price\rangle) \wedge$$

$$A2 \;\; \underline{\textbf{internal}} \;\; (\text{“}\texttt{debit}\text{”}, \langle C, A2.price)]$$

If there is no replay attack, this formula should hold. Recall that the model has a single customer instance and a single merchant instance; therefore, there is at most one debit authorization. This should mean that there is at most one debit. However, there is a counterexample to this formula in which the adversary simply replays the message it sent to one authority instance to the other instance and both instances end up performing a debit.

- *Privacy.* Customers want to ensure that the merchant is the only other party that knows the details of the transaction. Also, the customer's credit card number should be kept secret as well. The following two formulas specify these requirements. Note that $C1$ is the customer instance in the model.

$$\forall S_0 \,.[\ S_0 \ \ \mathbf{Knows} \ \ C1.\mathit{DESC} \rightarrow (S_0.\mathit{pr} = C \vee S_0.\mathit{pr} = M)]$$

$$\forall S_0 \,.[\ S_0 \ \ \mathbf{Knows} \ \ C1.\mathit{CAN} \rightarrow (S_0.\mathit{pr} = C \vee S_0.\mathit{pr} = A)]$$

In other words, if some instance S_0 knows the customer's description of the transaction, then S_0 must be the customer or the merchant. If some instance S_0 knows the customer's account number, then S_0 must be either the customer or the authority.

- *Proof of transaction authorization by merchant.* This particular property is not claimed to hold of 1KP. It is meant to hold for 2KP and 3KP, but I tried to verify a variant of it in the model for 1KP. The requirement is that all transactions allowed by the authority must be authorized by the *merchant*.

$$\forall A_0 \,.\, (\mathit{pr}(A_0) = A) \wedge (A_0 \ \underline{\mathbf{internal}}\ (\text{“}\mathtt{credit}\text{”}, \langle A_0.m, A_0.\mathit{price}\rangle)) \rightarrow$$

$$\exists M_0 \,.\, (\mathit{pr}(M_0) = A_0.m) \wedge \Diamond_P (M_0 \ \underline{\mathbf{internal}}\ (\text{“}\mathtt{Mauthorize}\text{”}, A_0.\mathit{price}))$$

In other words, if the authority credits principal $A_0.m$ (presumably the merchant) by the amount $A_0.\mathit{price}$, then it must be the case that $A_0.m$ authorized a payment of that amount. This particular analysis did provide some insight. Although the authors do not claim that

1KP guarantees this property, there was nothing about the protocol that suggested to me that this property could be violated. BRUTUS does find a counterexample. All the adversary needs is the merchant's ID (ID_M), and some account number to debit (possibly the adversary's own account). With this information, no one else other than the authority need participate in order to have the authority make payments.

7.3 Wide Mouthed Frog

The Wide Mouthed Frog protocol is intended to distribute a freshly generated session key to another party. In the description that follows, A is the initiator wishing to communicate with the responder, B. S is a trusted third party with whom both A and B share a different symmetric key. The protocol proceeds as follows:

1. The initiator A, generates a new session key K_{ab} that it intends to distribute to B. It pairs the key with B's name and a timestamp T_a and encrypts the pair with K_{as}, the key A shares with the third party S. It sends this along with its own name to S.

$$A \rightarrow S : A, \{T_a, B, K_{ab}\}_{K_{as}}$$

2. The trusted third party S now decrypts $\{T_a, B, K_{ab}\}_{K_{as}}$ and checks the timestamp T_a. If it is fresh (i.e., if the value T_a falls within some window of the actual current time), then S takes the key K_{ab} and pairs it with A's name and it's own timestamp and encrypts this with K_{bs}, the key it shares with B. It then sends this encrypted message to B.

$$S \rightarrow B : \{T_s, A, K_{ab}\}_{K_{bs}}$$

3. The responder receives this message and decrypts with the key K_{bs}. If the decryption succeeds, it checks the timestamp. If the timestamp is fresh, then it accepts K_{ab} as a session key for communication with A.

$INITIATOR =$
$\underline{\textbf{choose}}\ (b)$
$\underline{\textbf{internal}}\ (\text{“}\texttt{begin-initiate}\text{”}, b)$
$\underline{\textbf{send}}\ \langle pr, \{b, k'\}_{symkey(pr,s)}\rangle$

$RESPONDER =$
$\underline{\textbf{receive}}\ \{a, k'\}_{symkey(pr,s)}$
$\underline{\textbf{internal}}\ (\text{“}\texttt{end-respond}\text{”}, a)$

$SERVER =$
$\underline{\textbf{receive}}\ \langle a, \{b, k'\}_{symkey(a,pr)}\rangle$
$\underline{\textbf{send}}\ \{a, k'\}_{symkey(b,pr)}$

Figure 7.8: Roles in the Wide Mouthed Frog protocol

BRUTUS does not have a notion of time, so I cannot include timestamps in the BRUTUS model of the Wide Mouthed Frog protocol. Because of this, the model will not be secure against replay. However, the main purpose in analyzing this protocol was to explore how exploiting symmetry reduces the size of the state space and increases the size of the model that can be analyzed. The Wide Mouthed Frog protocol is ideal for this because of its small size. The sequence of actions defining the roles of the initiator, responder, and server (trusted third party) are given in Figure 7.8.

I constructed and checked a model of the Wide Mouthed Frog protocol which contained five initiator instances, five responder instances, and five server instances. Figure 7.9 contains two example instances for each of the three roles in the protocol.

The same authentication property that was checked for the Needham-Schroeder protocol can be checked for this protocol. However, only half of the property is satisfied. The formula

$$\forall B_0 \,.\, B_0\ \underline{\textbf{internal}}\ (\text{“}\texttt{end-respond}\text{”}, B_0.a) \rightarrow$$

$$\exists A_0 \,.\, (A_0.pr = B_0.a) \wedge \Diamond_P (A_0\ \underline{\textbf{internal}}\ (\text{“}\texttt{begin-initiate}\text{”}, B_0.pr))$$

$$\mathcal{I}1 : \begin{cases} H_1 = \text{``}\texttt{A1}\text{''} \\ B_1 = \{(pr, \text{``}\texttt{A}\text{''}), (k_{as}, K_{as}), (k', K_1)\} \\ I_1 = \{A, B, \Omega, K_{as}, K_1\} \\ P_1 = \mathit{INITIATOR} \end{cases}$$

$$\mathcal{I}_2 : \begin{cases} H_2 = \text{``}\texttt{B1}\text{''} \\ B_2 = \{(pr, \text{``}\texttt{B}\text{''}), (k_{bs}, K_{bs})\} \\ I_2 = \{A, B, \Omega, K_{bs}\} \\ P_2 = \mathit{RESPONDER} \end{cases}$$

$$\mathcal{I}_3 : \begin{cases} H_3 = \text{``}\texttt{S1}\text{''} \\ B_3 = \{(pr, \text{``}\texttt{S}\text{''}), (k_{as}, K_{as}), (k_{bs}, K_{bs})\} \\ I_3 = \{A, B, \Omega, K_{as}, K_{bs}\} \\ P_3 = \mathit{SERVER} \end{cases}$$

$$\mathcal{I}_4 : \begin{cases} H_4 = \text{``}\texttt{A2}\text{''} \\ B_4 = \{(pr, \text{``}\texttt{A}\text{''}), (k_{as}, K_{as}), (k', K_4)\} \\ I_4 = \{A, B, \Omega, K_{as}, K_4\} \\ P_4 = \mathit{INITIATOR} \end{cases}$$

$$\mathcal{I}_5 : \begin{cases} H_5 = \text{``}\texttt{B2}\text{''} \\ B_5 = \{(pr, \text{``}\texttt{B}\text{''}), (k_{bs}, K_{bs})\} \\ I_5 = \{A, B, \Omega, K_{bs}\} \\ P_5 = \mathit{RESPONDER} \end{cases}$$

$$\mathcal{I}_6 : \begin{cases} H_6 = \text{``}\texttt{S2}\text{''} \\ B_6 = \{(pr, S), (k_{as}, K_{as}), (k_{bs}, K_{bs})\} \\ I_6 = \{A, B, \Omega, K_{as}, K_{bs}\} \\ P_6 = \mathit{SERVER} \end{cases}$$

Figure 7.9: Wide Mouthed Frog model

does hold for this model. The other half of the property (if the initiator finishes then the responder must have participated) does not hold in this model. This is obvious because the only thing the initiator does in the protocol is send the first message. Hence it could finish executing before/without a corresponding responder executing if the adversary simply prevents the message from reaching the responder.

One can also check a secrecy requirement on the session key K_{ab}. One would like to verify that the adversary does not gain knowledge of the session key. Again, this requirement need only hold when the initiator is trying to execute the protocol with someone other than the adversary. In other words, if the adversary knows the session key, then the initiator must be trying to execute the protocol with the adversary. The formula is:

$$\forall X \; . \; H_\Omega \;\; \mathbf{Knows} \;\; X.k' \rightarrow \Diamond_P[X \;\; \underline{\mathbf{internal}} \;\; (\text{“}\texttt{begin-initiate}\text{”}, \Omega)]$$

Somewhat surprisingly, this property does hold of the model, despite the fact that it does not use any nonces or any timestamps.

7.4 A Composition Example

One of the advantages of Brutus is the fact that it has a built in model of the adversary. This is particularly useful when one wants to analyze possible interactions between different protocols. Because the adversary is built in, one does not need to anticipate what messages or submessages might interact. Brutus will catch those possible interactions automatically.

The protocol that follows is a simple toy example that illustrates how two protocols could each satisfy a requirement in isolation, but when composed in the same model, the adversary could use one to attack the other. This example borrows from an unpublished protocol suggested by Dawn Song for this very reason. The original pair of protocols satisfied different requirements. I had to change the example slightly so that both protocols would satisfy the *same* requirement in isolation.

The first protocol is the Needham-Schroeder public key authentication protocol as modified by Lowe in [37]. It is identical to the original protocol, except that when sending the second message, the responder includes her name in the encrypted portion of the message. This small modification is enough to prevent the attack described in Section 7.1. In fact all of the requirements discussed in Section 7.1 are satisfied by all models of this new protocol with up to two initiators and three responders. The messages for this corrected protocol appear in Figure 7.10. The BRUTUS roles for the Needham-Schroeder-Lowe protocol are given in Figure 7.11. Note that there is an additional argument to the internal actions marking the start and end of execution, namely *Data "N_S_L"*. This is so that when the model composed of two different protocols is checked, BRUTUS can determine not only when a principal has started or ended executing a protocol, but also *which* protocol has started or ended. This would force a corresponding change in the specification being checked. The new authentication formula is:

$$\forall A_0 \,.\, A_0 \ \underline{\textbf{internal}} \ (\text{``}\texttt{end-initiate}\text{''}, \langle A_0.b, Data\text{``}\texttt{N\_S\_L}\text{''}\rangle) \rightarrow$$

$$\exists B_0 \left[\begin{array}{c} (B_0.pr = A_0.b) \wedge \\ \Diamond_P(B_0 \ \underline{\textbf{internal}} \ (\text{``}\texttt{begin-respond}\text{''}, \langle A_0.pr, Data\text{``}\texttt{N\_S\_L}\text{''}\rangle)) \end{array} \right]$$

The instances in the BRUTUS model are otherwise unchanged from what was used in the original Needham-Schroeder protocol in Figure 7.3.

The second protocol is meant to be a one-way authentication protocol. In this protocol, the responder B is authenticated to A, but there is no guarantee made to B about A. This protocol contains the following messages.

1. First, the initiator, A, generates a nonce, N_a, (which we can assume is a random number), and then encrypts the pair $\langle N_a, A\rangle$ with B's public key. It then sends this message to B.

$$A \rightarrow B : \{N_a, A\}_{K_B}$$

2. Upon receiving message number 1, B uses its private key to decrypt $\{N_a, A\}_{K_B}$ and recover the identity of the initiator, A, and the initiator's nonce, N_a. It then encrypts the pair $\langle N_a, B\rangle$ with its own private key, and sends this to A.

$$B \rightarrow A : \{N_a, B\}_{K_B^{-1}}$$

1. $A \rightarrow B : A, B, \{N_a, A\}_{K_B}$
2. $B \rightarrow A : B, A, \{N_a, N_b, B\}_{K_A}$
3. $A \rightarrow B : A, B, \{N_b\}_{K_B}$

Figure 7.10: Needham-Schroeder-Lowe protocol

$INITIATOR =$
choose (b)
internal $(\text{“}\texttt{begin-initiate}\text{”}, \langle b, Data\text{“}\texttt{N\_S\_L}\text{”}\rangle)$
send $\langle pr, b, \{n_a, pr\}_{pubkey(b)} \rangle$
receive $\langle pr, b, \{n_a, n_b, b\}_{pubkey(pr)} \rangle$
send $\langle pr, b, \{n_b\}_{pubkey(b)} \rangle$
internal $(\text{“}\texttt{end-initiate}\text{”}, \langle b, Data\text{“}\texttt{N\_S\_L}\text{”}\rangle)$

$RESPONDER =$
receive $\langle a, pr, \{n_a, a\}_{pubkey(pr)} \rangle$
internal $(\text{“}\texttt{begin-respond}\text{”}, \langle a, Data\text{“}\texttt{N\_S\_L}\text{”}\rangle)$
send $\langle a, pr, \{n_a, n_b, pr\}_{pubkey(a)} \rangle$
receive $\langle a, pr, \{n_b\}_{pubkey(pr)} \rangle$
internal $(\text{“}\texttt{end-respond}\text{”}, \langle a, Data\text{“}\texttt{N\_S\_L}\text{”}\rangle)$

Figure 7.11: Needham-Schroeder-Lowe Roles

$INITIATOR\_1WAY =$
choose (b)
internal $(\text{“}\texttt{begin-initiate}\text{”}, \langle b, Data\text{“}\texttt{1way}\text{”}\rangle)$
send $\langle \{n_a, pr\}_{pubkey(b)} \rangle$
receive $\langle \{n_a, b\}_{privkey(b)} \rangle$
internal $(\text{“}\texttt{end-initiate}\text{”}, \langle b, Data\text{“}\texttt{1way}\text{”}\rangle)$

$RESPONDER\_1WAY =$
receive $\langle \{n_a, a\}_{pubkey(pr)} \rangle$
internal $(\text{“}\texttt{begin-respond}\text{”}, \langle a, Data\text{“}\texttt{1way}\text{”}\rangle)$
send $\langle \{n_a, pr\}_{privkey(pr)} \rangle$
internal $(\text{“}\texttt{end-respond}\text{”}, \langle a, Data\text{“}\texttt{1way}\text{”}\rangle)$

Figure 7.12: One-way Protocol Roles

3. Upon receiving message number 2, A uses B's public key to decrypt $\{N_a, B\}_{K_B^{-1}}$ and verify its own nonce and B's name. Assuming a match, A should now be guaranteed of B's participation in the protocol.

The Brutus roles for this new one-way authentication protocol appear in Figure 7.12. These roles were then used in a model of the protocol containing two initiator instances and two responder instances given in Figure 7.13. This model was analyzed (independently of the Needham-Schroeder-Lowe protocol) and was found to satisfy the authentication property that whenever the initiator, A, finishes the protocol with a responder, B, that responder must have participated in the protocol with A. The formula expressing this property is identical to the one for the Needham-Schroeder-Lowe protocol, except that Data “N_S_L” is replaced with Data “1way”.

$$\forall A_0 \;.\; A_0 \;\; \underline{\textbf{internal}} \;\; (\text{“}\texttt{end-initiate}\text{”}, \langle A_0.b, Data\text{“}\texttt{1way}\text{”}\rangle) \rightarrow$$

$$\exists B_0 \left[\begin{array}{c} (B_0.pr = A_0.b) \wedge \\ \Diamond_P(B_0 \;\; \underline{\textbf{internal}} \;\; (\text{“}\texttt{begin-respond}\text{”}, \langle A_0.pr, Data\text{“}\texttt{1way}\text{”}\rangle)) \end{array} \right]$$

The next model was a composed model with one initiator and one responder from each of the two protocols (four instances total). It simply

$$\mathcal{I}_5 : \begin{cases} H_5 = \text{“}\texttt{A3}\text{”} \\ B_5 = \{(pr, \text{“}\texttt{A}\text{”}), (n_a, \text{“}\texttt{Na3}\text{”})\} \\ I_5 = \{A, B, \Omega, K_A, K_B, K_\Omega, K_A^{-1}, Na3\} \\ P_5 = \mathit{INITIATOR\_1WAY} \end{cases}$$

$$\mathcal{I}_6 : \begin{cases} H_6 = \text{“}\texttt{A4}\text{”} \\ B_6 = \{(pr, \text{“}\texttt{A}\text{”}), (n_a, \text{“}\texttt{Na4}\text{”})\} \\ I_6 = \{A, B, \Omega, K_A, K_B, K_\Omega, K_A^{-1}, Na4\} \\ P_6 = \mathit{INITIATOR\_1WAY} \end{cases}$$

$$\mathcal{I}_7 : \begin{cases} H_7 = \text{“}\texttt{B3}\text{”} \\ B_7 = \{(pr, \text{“}\texttt{B}\text{”})\} \\ I_7 = \{A, B, \Omega, K_B, K_B, K_\Omega, K_B^{-1}\} \\ P_7 = \mathit{RESPONDER\_1WAY} \end{cases}$$

$$\mathcal{I}_8 : \begin{cases} H_8 = \text{“}\texttt{B4}\text{”} \\ B_8 = \{(pr, \text{“}\texttt{B}\text{”})\} \\ I_8 = \{A, B, \Omega, K_B, K_B, K_\Omega, K_B^{-1}\} \\ P_8 = \mathit{RESPONDER\_1WAY} \end{cases}$$

Figure 7.13: One-way Protocol Model

$A1$ **choose** *Principal B*
$A1$ **internal** ("`begin-initiate`", $\langle$*Principal B*, *Data N_S_L*$\rangle$)
$A1$ **send** *Principal A*, *Principal B*,
  $\{$*Nonce Na*1, *Principal A*$\}$_*Pubkey*(*Principal B*)
$A3$ **choose** *Principal B*
$A3$ **internal** ("`begin_initiate`", $\langle$*Principal B*, *Data* 1*way*$\rangle$)
$A3$ **send** $\{$*Nonce Na*3, *Principal A*$\}$_*Pubkey*(*Principal B*)
$B1$ **receive** *Principal* Ω, *Principal B*,
  $\{$*Nonce* N_Ω, *Principal* $\Omega\}$_*Pubkey*(*Principal B*)
$B1$ **internal** ("`begin-respond`", $\langle$*Principal* Ω, *Data N_S_L*$\rangle$)
$B1$ **send** *Principal B*, *Principal* Ω,
  $\{$*Nonce* N_Ω, *Nonce Nb*1$\}$_*Pubkey*(*Principal* Ω)
$B1$ **receive** *Principal* Ω, *Principal B*,
  $\{$*Nonce Nb*1$\}$_*Pubkey*(*Principal B*)
$B1$ **internal** ("`end-respond`", $\langle$*Principal* Ω, *Data N_S_L*$\rangle$)
$B3$ **receive** $\{$*Nonce Na*1, *Principal A*$\}$_*Pubkey*(*Pubkey B*)
$B3$ **internal** ("`beg-respond`", $\langle$*Principal A*, *Data* 1*way*$\rangle$)
$B3$ **send** $\{$*Nonce Na*1, *Principal B*$\}$_*Privkey*(*Principal B*)
$B3$ **internal** ("`end-respond`", $\langle$*Principal A*, *Data* 1*way*$\rangle$)
$A1$ **receive** *Principal B*, *Principal A*,
  $\{$*Nonce Na*1, *Nonce Na*1$\}$_*Pubkey*(*Principal A*)
$A1$ **send** *Principal A*, *Principal B*,
  $\{$*Nonce Na*1$\}$_*Pubkey*(*Principal B*)
$A1$ **internal** ("`end-initiate`", $\langle$*Principal B*, *Data N_S_L*$\rangle$)

Figure 7.14: Composition counterexample

consisted of instances $\mathcal{I}_1$ and $\mathcal{I}_3$ from the Needham-Schroeder-Lowe protocol (Figure 7.3) and instances $\mathcal{I}_5$ and $\mathcal{I}_7$ from the one-way authentication protocol (Figure 7.13). While each of these protocols satisfied its corresponding authentication formula in isolation (even in models containing multiple instances), the formula is not satisfied in this composed model. In other words, the adversary is able to use one protocol to attack the other. The attack appears in Figure 7.14.

The attack is possible in part because both protocols share the message $\{N_a, A\}_{K_B}$. The Needham-Schroeder-Lowe protocol requires that the nonce N_a remain secret while the one-way protocol reveals the nonce N_a.

protocol	init	resp	none	p.o.	symm	p.o+symm
1KP	1	1	17,905,267	906,307	17,905,267	906,307
N-S	1	1	1,208	146	1,208	146
N-S	1	2	1,227,415	6,503	613,713	3,257
N-S	2	2	X	372,977	X	186,340
N-S	2	3	X	78,917,569	X	12,148,470
WMF	1	1	18	18	18	18
WMF	2	2	665,827	1,285	157,275	223
WMF	3	3	X	1,286,074	X	7,004
WMF	4	4	X	X	X	455,209
WMF	5	5	X	X	X	47,651,710

Figure 7.15: Table of results

Since both protocols expect the same message format, the adversary can simply forward the message from the Needham-Schroeder-Lowe protocol to the one-way protocol for decryption. This is exactly what happens in lines 17 and 19 in Figure 7.14. Notice that while instance $A3$ has correctly sent a message containing its own nonce $Na3$ in line 8, this was replaced in line 17 by the adversary with the equivalent portion from the message in line 3. In line 19, the one-way protocol decrypts the message for the adversary giving the adversary access to the nonce $Na1$ which leads to the attack.

7.5 Results

The table in Figure 7.15, summarizes the results of applying the symmetry reduction and the partial order reduction to the models of most of the protocols discussed in this chapter. I examined the 1KP secure payment protocol [4], the Needham-Schroeder public key protocol [58], and the Wide Mouthed Frog protocol [7, 71]. In Figure 7.15 these protocols are labeled as **1KP**, **N-S**, and **WMF**. Columns two and three give the number of initiator and responder instances used in building the model. The other columns give the number of states encountered during the state space traversal of the model. The column marked with **none** refers to results when no reductions were applied. Results corresponding to the partial order and symmetry reductions are presented in columns marked with **p.o.** and **symm** respectively. The column labeled with **p.o.+symm** presents results when both reductions (partial order and symmetry) were applied simultaneously. The entries with an "X" represent computations that were aborted after 24 hours of computation (over 700,000,000 states).

The results in the table demonstrate that the reduction achieved due to the two techniques is significant. Notice that for the Needham-Schroeder protocol with one initiator and two responders (third row) the state space is reduced by a factor of around 400. The partial order techniques by themselves seem to yield a much larger reduction than the symmetry techniques, but this is somewhat misleading because the models contain very few instances. In order to appreciate the reduction possible via symmetry techniques I would need to have models with many instances of the same roles. The best example is the Wide Mouthed Frog protocol. This protocol is simple enough to allow a model that has fifteen concurrent instances. However, an exhaustive search of this model would not be possible without the symmetry reduction. In addition, although the model with twelve concurrent instances can be searched without the symmetry reduction (using only the partial order reduction), the reduction in the size of the state space when the symmetry reduction is applied is over 180 fold. Also note that there is no reduction in the size of the state space when applying symmetry reductions to models that do not have more than one initiator or responder since there are no replicated components in these models.

While these experiments (and the theoretical results in previous chapters) are specific to Brutus, it is worthwhile to speculate how these ideas can be generalized and applied elsewhere. First, there is nothing necessarily specific to the Brutus tool in the discussion. These ideas should be applicable to any tool using a similar system model for analyzing security protocols. Specifically, these ideas should be applicable to anyone model checking a Dolev-Yao type intruder. Second, while symmetry has already been used in many model checking and theorem proving domains, the generalized partial order reduction discussed here has really only been used for model checking security protocols. It may be applicable to other verification domains. The problem is finding other domains with the same monotonicity properties. In particular, in order for an action to be semi-invisible, the atomic propositions relating to that action must be monotonic (once the atomic proposition becomes true it remains true), and the proposition must appear negatively in the specification formula.

Chapter 8

Related Work

A number of researchers have seen the potential in trying to apply formal methods to the analysis of security protocols. The approaches being used include theorem proving, non-automated reasoning, model checking, and rule rewrite systems. Almost all of them use the same basic adversary model which was originally proposed by Dolev and Yao [19]. However, the approaches often differ in how one specifies the protocol, how one specifies the requirements, how one specifies the adversary, and how the tool goes about trying to perform the analysis.

8.1 Logic of Authentication

One of the earliest successful attempts at formally reasoning about security protocols involved the development of a new logic in which one could express and deduce security properties. The earliest such logic is the Logic of Authentication proposed by Burrows, Abadi, and Needham [7], and is commonly referred to as the BAN logic. Their syntax provided constructs for expressing intuitive properties like

- "A said X." ($A \mid\!\sim X$)
- "A sees X." ($A \triangleleft X$)
- "A believes X." ($A \mid\equiv X$)
- "K is a good key for communication between A and B." ($A \overset{K}{\leftrightarrow} B$)

- "X is a fresh message." ($\sharp(X)$)
- "S is an authority on X." ($S \mapsto X$).

The authors also provide a set of proof rules which can then be used to try to deduce security properties such as "A and B believe K is a good key" ($A \models A \overset{K}{\leftrightarrow} B$ and $B \models A \overset{K}{\leftrightarrow} B$) from a list of explicit assumptions made about the protocol. For example, their inference system provides rules for the following:

- If a message X encrypted with a key K is received by a principal P and K is believed to be a good key, then P believes that the other party possessing K said the message.

$$\frac{P \models Q \overset{K}{\leftrightarrow} P, \quad P \triangleleft \{X\}_K}{P \models Q \mid\sim X}$$

- A principal only says things that it believes. Worded differently, if a principal P receives a recent message X from Q, then P believes that Q believes X.

$$\frac{P \models \sharp(X), \quad P \models Q \mid\sim X}{P \models Q \models X}$$

- If a principal P believes some principal Q has jurisdiction over the statement X, then P trusts Q on the truth of X. If statement X is about a session key generated by a server, this rule would allow one to infer that participants in a protocol will believe that the session key is good if it came from a trusted server that has jurisdiction over good keys.

$$\frac{P \models Q \mapsto X, \quad P \models Q \models X}{P \models X}$$

- A principal can see the components of compound messages and a principal can decrypt messages with good keys.

$$\frac{P \triangleleft (XY)}{P \triangleleft X} \qquad \frac{P \models Q \overset{K}{\leftrightarrow} P, \quad P \triangleleft \{X\}_K}{P \triangleleft X}$$

This formalism was successful in uncovering implicit assumptions that had been made in a number of protocols. Its success has inspired other work related to the BAN logic. In fact, there have been a couple of attempts at automating the process of a BAN analysis [17, 35]. In addition, extensions have been made to the logic in an attempt to add a notion of accountability to the logic [33].

However, the logic has been criticized for the "protocol idealization" step required when using this formalism. Protocols in the literature are typically given as a sequence of messages. Use of the BAN logic requires that the user transform each message in the protocol into formulas *about* that message, so that the inferences can be made within the logic. For example, if the server sends a message containing the key K_{ab}, then that step might need to be converted into a step where the server sends a message containing $A \overset{K_{ab}}{\leftrightarrow} B$, meaning that the key K_{ab} is a good key for communication between A and B. This simple example is pretty straightforward. In general, however, the idealization step requires that one assign a *meaning* to the messages that appear in the protocol, thus introducing an informal step into the protocol analysis. Because of these criticisms, an attempt was made to give the logic a formal model [1]; however, this was not entirely successful. Probably the most successful attempt at formalizing the idealization gap is the work of Kindred and Wing [34, 36].

The second objection to this kind of analysis is that all principals are honest and so it does not allow for a malicious adversary. In other words, one tries to argue about how different participants might come to certain beliefs about keys and secrets, but one cannot investigate how a malicious adversary might try to subvert the protocol by possibly modifying and misdirecting messages. For this reason, a number of researchers have looked to analyzing security protocols in a framework that allows for a malicious adversary. In fact, one group has even investigated the benefits of combining REVERE (Kindred's theory generator for the BAN Logic) with my model checking tool, BRUTUS [25].

In the tools that follow, researchers have a concrete operational model for how the protocol executes. This operational model includes a description of how the honest participants in the protocol behave (i.e., what it means to execute the protocol) and a description of how an adversary can interfere with the execution of the protocol. The model of the adversary has evolved from one proposed by Dolev and Yao [19]. Typically, the adversary model allows for the maximum amount of interference while maintaining encryption as a black box. The model of the adversary usually allows it to overhear and to intercept all messages, to misdirect messages, and to send fake messages. The adversary can send any message it can generate from previously overheard messages by concatenating and projecting onto components as well as by encrypting and decrypting with known keys. The adversary is also allowed to participate in the protocol. In other words, it can try to initiate protocol sessions with honest agents, and honest agents are willing to try to initiate sessions with the adversary. While the details of *how* this behavior is modeled is different among the different tools, all the tools described below (including BRUTUS) do implement this high-level description of an adversary.

8.2 FDR

Gavin Lowe has investigated the use of FDR to analyze CSP [24] models of cryptographic protocols [37, 38]. CSP seems to be a natural language in which to model the communication of an inherently asynchronous composition of protocol sessions. Each instance of an agent trying to execute the protocol is modeled by a CSP process that alternates between waiting for a message and sending a message (replying). Channels are used for communication between processes (between participants in the protocol). Channels are also used to model adversary interference. For example, the channel *intercept* models the possibility of the adversary intercepting a message intended for an honest agent. The channel *fake* models the possibility of an honest agent receiving a message that it believes came from an honest agent, but was actually generated by the adversary. Channels are also used to keep track of important events in the protocol, such as commit points. For example, the event *I_commit.a.b* can be generated by initiator *a* on channel *I_commit* to represent the fact that it is committing to a session with responder *b*.

A description of the initiator in the Needham-Schroeder protocol is given in Figure 8.1. The definition is parameterized in *a*, the name of the initiator, and n_a, the nonce used by the initiator in the session. This definition follows the abstract description of the protocol fairly closely. The initiator waits for a request from the user then begins running the protocol and sends message 1. When it receives message 2, it checks if

$$
\begin{array}{ll}
\mathit{INITIATOR}(a, n_a) \equiv & \mathit{user.a?b} \rightarrow \mathit{I\_running.a.b} \rightarrow \\
 & \mathit{comm!Msg1.a.b.Encrypt.key(b)}.n_a.a \rightarrow \\
 & \mathit{comm.Msg2.a.b.Encrypt.key(a)?}n'_a.n_b \rightarrow \\
 & \underline{\mathbf{if}}\ n_a = n'_a \\
 & \quad \underline{\mathbf{then}}\ \mathit{comm!Msg3.a.b.Encrypt.key(b)}.n_b \rightarrow \\
 & \qquad\quad \mathit{I\_commit.a.b} \rightarrow \mathit{session.a.b} \rightarrow \mathit{Skip} \\
 & \quad\ \underline{\mathbf{else}}\ \mathit{Stop}
\end{array}
$$

Figure 8.1: FDR example

the nonce in message 2 matches the nonce sent in message 1. If so, it sends message 3, commits to the protocol session, and continues with the session execution (the actual work to be done), otherwise it halts. To model the interception of messages by the adversary and the introduction of fake messages by the adversary, a renaming is applied to this process so that actions that occur on the channel *comm* can occur on the channel *intercept* or *fake* instead.

Originally, the user also had to provide a description of the adversary. With the development of Casper, the construction of the adversary became automated [38]. The adversary can be thought of as the parallel composition of n processes, one for each of n facts or messages that the adversary may learn during the execution of the protocol. Each process basically has two states, one in which it knows the message and one in which it does not. Each of these processes then can generate a number of events.

- It will synchronize with an agent when overhearing a message sent by that agent that contains the fact.
- It will synchronize with an agent when it sends a message to that agent that contains the fact.
- It will synchronize with other processes representing the adversary when knowledge of those other facts can be used to derive this fact.
- It will synchronize with another process representing the adversary when this fact can contribute to the derivation of the fact represented by that other process.

- It can generate a *leak* event to signal that the adversary has acquired knowledge of the fact it represents.

Casper will also construct the specification process for the verification. FDR is then used to check that the protocol in parallel with the adversary is a refinement of the specification process. The specification process for secrecy is simply the CSP process $RUN(\Sigma - L)$ where Σ is the set of all possible events, L is the set of leak events that correspond to the adversary knowing a secret, and $RUN(S)$ is the process that can perform any sequence of events in S. So the specification is the process that can perform any sequence of events that do not include the *leak* events in question. The authentication specification process AS is defined below.

$$
\begin{aligned}
AS_0 &\equiv R\_running.A.B \rightarrow I\_commit.A.B \rightarrow AS_0 \\
A &\equiv \{R\_running.A.B, I\_commit.A.B\} \\
AS &\equiv AS_0 \parallel RUN(\Sigma - A)
\end{aligned}
$$

Intuitively, the specification is the process that can perform all events in any order, except that the subsequence consisting of all $R\_running.A.B$ events and all $I\_commit.A.B$ events must alternate and must begin with $R\_running.A.B$. This means that every time the initiator commits to a protocol session, there must be a separate responder instance that has started responding to the initiator. Woo and Lam have a similar correctness requirement which they call *correspondence* [83]. Their correspondence requirement is not as strict as Lowe's since it does not require strict alternations of the responder running and initiator committing events. Most investigators using operational models, myself included, use a specification closer to the one proposed by Woo and Lam.

In a BRUTUS model, the process description for each role in the protocol is very similar to the CSP model above. Each honest agent process consists of a sequence of events that define its role in the protocol. In addition, each agent (including the adversary), maintains a set of known facts or knowledge. Because this knowledge is represented as a set of "atomic facts" together with a set of re-write rules that can be applied to that set, it actually represents an infinite set of facts. This becomes especially important for the model of the adversary. Because the set of known facts is represented implicitly in BRUTUS, one is not forced to artificially limit the set of words that the adversary may learn in order to construct a finite state model for the adversary.

8.3 Murϕ

Mitchell and others have investigated using a general purpose state enumeration based model checking tool, Murϕ, for analyzing cryptographic protocols [56, 57]. In Murϕ, the state of the system is determined by the values of a set of global state variables, including the shared variables that are used to model communication. For example, each participant has a variable describing which state it is in and a different variable containing the name of its "partner" in the protocol. There is also a set of variables that contain the messages that are sent on the network with one variable describing the type of the message and other variables containing the fields of the message. Transition rules are used to describe how honest agents transition between states and how new messages are inserted into the network.

For example, the structure of the rule describing how the initiator in the Needham-Schroeder protocol responds to message number two with message number three can be found in Figure 8.2. The rule specifies that if there is some initiator i waiting for message number two and there is some message j on the network whose recipient is i then j is removed from the network and if j is a message two encrypted with i's key and containing i's nonce, then message three is constructed and added to the network, and i enters the COMMIT state. Similar rules would be written for each of the other messages used in the protocol. The authors mention that rules that capture the behavior of the adversary must also be constructed. These rules are not provided in the paper, and the authors concede that formulating the adversary is complicated [56]. Presumably, these rules would capture how an adversary can intercept messages and misdirect them and modify them. However, because the description must necessarily be finite state, it cannot capture the infinite behavior of the adversary. In particular, the description can only keep track of a finite number of words that the adversary may know or may learn. Also, this adversary model would be specific to the particular protocol being analyzed.

The specification for the protocol is given by providing an invariant on the reachable global states of the system. The "usual" correctness property is used. Namely, if an initiator i commits to a protocol run with responder r, then r must have at least started to respond to i. The actual Murϕ specification for this property is given in Figure 8.3. The verification would include an analogous invariant for all responders as well. However,

```
foreach i ∈ 1..num_initiators
  foreach j ∈ 1..network_size
    if (init[i].state = WAITING_FOR_MESSAGE_2 ∧
      net[j].destination = i)
      then
           remove j from the network
           if (net[j].key = i ∧
             net[j].type = MESSAGE_2 ∧
             net[j].nonce1 = i)
             then
                  set the fields of outgoing message out
                  add out to the network
                  init[i].state := COMMIT
           fi
    fi
```

Figure 8.2: Murϕ example

$$\forall i \in 1..\mathrm{num\_initiators}\ .$$

$$(\mathit{init[i].state} = \mathrm{COMMIT} \ \wedge\ \mathit{init[i].responder} \in \mathrm{Responders}) \rightarrow$$

$$(\mathit{resp[init[i].responder].initiator} = i \wedge \mathit{resp[init[i].responder].state} \neq \mathrm{INITIAL})$$

Figure 8.3: Murϕ specification

the authors do not provide a specification for secrecy. Since keeping track of the knowledge of each of the agents is somewhat cumbersome in this approach, this would probably involve a non-trivial extension to the model.

8.4 NRL Protocol Analyzer

Catherine Meadows developed the NRL Protocol Analyzer, a special-purpose verification tool for the analysis of cryptographic protocols [43, 46]. Like the model checkers, each participant has its own local state and the global state of the entire system is the composition of these local states with some state information for the environment or adversary. The state of each local participant is maintained by a store of learned facts or *lfacts*. This is represented by the following store with four indices (which can be thought of as a function of four arguments)

$$\mathit{lfact}(p, r, n, t) = v$$

where

- p is the participant that knows the fact.
- r identifies the run of the protocol (a run or session identifier).
- n describes the nature or name of the fact.
- t is the local time as kept by the participant's counter.
- v is a list of words or values that make up the content of the fact.

For example,

$$\mathit{lfact}(\mathit{user}(A), N, \mathit{init\_conv}, T) = [\mathit{user}(B)]$$

expresses the fact that A has initiated a conversation with B in run N at local time T. If A has not yet initiated a conversation with B, then the value of the fact would be empty and so the value of the function would be [].

New *lfact* values are computed using the transition rules that describe the behavior of the protocol. For example, consider a fired rule that causes A to perform some action C in run B. Also assume that

$lfact(A, B, C, X) = Y$. If the rule fires at local time X, it sets A's local counter to $s(X)$. If the rule changes the value of the *lfact* to Z, then $lfact(A, B, C, s(X)) = Z$, otherwise $lfact(A, B, C, s(X)) = Y$.

Each transition rule encodes some action taken by a principal participating in the protocol. The actual firing of a rule representing a particular action is recorded via a store called *event*. This store has the same four arguments as *lfact*. The first identifies the participant in the event, the second identifies the protocol round or session, the third identifies the event, and the fourth is the value of the principal's local clock after the rule fires. Like *lfact*, the value of *event* is a list of words relevant to the event.

For the sake of concreteness, an example that parallels the Murϕ example is given in Figure 8.4. In this example, the first rule checks to see if the initiator has sent message number one but still has not received message number two. If this is the case, it accepts any message Z that the intruder knows and records whether or not Z has the correct format for message number two in an *lfact* with the name *init_gotnonce*. It checks for the correct format with *id_check* which is simply the identity function. The event recorded when this rule fires is

$$event(user(A, honest), N, init\_decrypt, s(M)) = [user(B, W), X, Y, Z].$$

This *event* records the fact that the honest agent A acting as the initiator has decrypted message number two at time $s(M)$ of round N. The value of the event consists of the following four words:

1. $user(B, W)$, the name of the responder;
2. X, the initiator's nonce;
3. Y, the responder's nonce; and
4. Z, the ciphertext being decrypted.

The second rule checks for this *lfact* and if it exists and the value computed by *id_check* was *ok* (true), it produces message number three. This event is recorded as agent A, acting as the initiator in round N at time $s(M)$, replies with the responder's name B and the responder's nonce Y.

While this description looks somewhat similar to the Murϕ description (they both use transition rules), this similarity is mostly superficial. Murϕ

```
rule(1)
If:
count(user(A,honest)) = [M],
intruderknows(Z),
lfact(user(A,honest), N, init_nonce, M) = [user(B,W), X].
lfact(user(A,honest), N, init_gotnonce, M) = [],
then:
count(user(A,honest)) = [s(M)],
lfact(user(A,honest), N, init_gotnonce, s(M)) =
  [user(B, W), Y,
   id_check(pke(privkey(user(A,honest)), Z), (X, Y))],
EVENT:
event(user(A,honest), N, init_decrypt, s(M)) =
  [user(B, W), X, Y, Z].

rule(2)
If:
count(user(A, honest)) = [M],
lfact(user(A, honest), N, init_gotnonce, M) =
  [user(B, W), Y, ok],
lfact(user(A, honest), N, init_nonce, M) = [user(B, W), X],
lfact(user(A, honest), N, init_final, M) = [],
then:
count(user(A, honest)) = [s(M)],
intruderlearns(pke(pubkey(user(B, W)), Y)),
lfact(user(A, honest), N, init_final, s(M)) = [user(B, W), Y],
EVENT:
event(user(A, honest), N, init_reply, s(M)) =
[user(B, W), Y].
```

Figure 8.4: NRL example

performs a state space search on explicit state descriptions. The NRL Analyzer uses unification to work on a possibly incomplete state description that would represent a set of states. In addition, the search performed by the NRL Analyzer is goal driven. The Analyzer searches backwards from a goal to an initial state. Unlike other finite-state systems, there is no a priori bound placed on the number of instances of the protocol that can be executed; therefore, the number of states to be searched is infinite. The NRL Analyzer provides a way to prove certain sets of states (often these sets are infinite) unreachable in an attempt to prune the search; however, the procedure is still not guaranteed to terminate [45].

It is also interesting to compare how the requirements of a protocol are specified when using the NRL Analyzer. Syverson and Meadows have developed a logic in which to express the properties required of the protocol [75]. The atomic propositions are action symbols with four arguments that describe the actions taken by the principals at a high level. For example, if a particular message coming from a server at local time M conveys a key K for communication between A and B, this might be encoded as the following action:

$$\mathbf{send}(S, (\mathit{user}(A, X), \mathit{user}(B, Y)), K, M)$$

The required relationships between actions are specified using the usual logical connectives as well as a past-time operator. In this sense, the logic is very similar to the one used by BRUTUS. The logic has a learn predicate which is similar to the **Knows** predicate used in BRUTUS. Also, BRUTUS allows universal quantification while variables are implicitly universally quantified in the Analyzer because it uses unification. However, unlike BRUTUS, the Analyzer does not have a direct interpretation of the atomic propositions in terms of the model of computation. A requirement written in this logic must be transformed into a goal for the Analyzer. This is done by first negating the requirement and then translating all action predicates into event statements used by the model of computation. This must then be converted into a goal state that is made up of some combination of the following:

- a set of words known by the intruder,
- a set of values of local state variables,
- a sequence of events that occurred, and

- a sequence of events that must not have occurred.

The authors note that these constructs have been adequate for all the requirements they have attempted to specify; however, the authors do not discuss any non-repudiation type properties which require the honest agents to know a particular word. Because of the restrictions on the goal state, it would seem that there is no way to specify a requirement about what an honest agent knows or does not know, although it may be possible to encode such a requirement in terms of events.

The NRL Analyzer has been used to analyze a number of computer security protocols. The analysis of the Needham-Schroeder public key authentication protocol is of particular interest because Meadows compares her use of the NRL Analyzer with Lowe's use of FDR [44]. Meadows has also used her tool to analyze the Internet Key Exchange protocol (IKE) [47] and the SET electronic commerce protocol [49]. In addition, a group of researchers have developed a CAPSL Interface for the NRL Analyzer [6]. CAPSL stands for common authentication protocol specification language. It is meant to be a common specification language for security protocols which tool designers can use as a front end for their analysis tools [18, 52, 54].

8.5 Athena

Song and others have developed a promising new checker for security protocols called Athena [53, 72, 73]. Like the NRL Analyzer, this tool works backwards, starting from a faulty state and trying to discover what initial conditions are necessary to reach that faulty state. Also similar to the NRL Analyzer, the "states" are kept as general as possible. A "most general error" state is described. The tool then tries to unify this state with the right side of a rule. In the case of the NRL Analyzer, this would be a kind of rewrite rule. In the case of Athena, this is an inference rule. As the search continues, the "states" become more concrete in the sense that more variables become bound; therefore, the abstract states represent fewer and fewer concrete states.

Athena uses an extension of the strand space model proposed by Thayer, Herzog and Guttman as the underlying model of computation [76]. Because this model is intuitive and because it plays such an important role in the efficiency of Athena it is briefly described below.

$$
\begin{array}{ccc}
\underline{\mathit{Init}[A, B, N_a, N_b]} & & \underline{\mathit{Resp}[A, B, N_a, N_b]} \\
1 : \langle +\{N_a \cdot A\}_{K_b} \rangle & \longrightarrow & 1 : \langle -\{N_a \cdot A\}_{K_b} \rangle \\
\Downarrow & & \Downarrow \\
2 : \langle -\{N_a \cdot N_b \cdot\}_{K_a} \rangle & \longleftarrow & 2 : \langle +\{N_a \cdot N_b \cdot\}_{K_a} \rangle \\
\Downarrow & & \Downarrow \\
3 : \langle +\{N_b\}_{K_b} \rangle & \longrightarrow & 3 : \langle -\{N_b\}_{K_b} \rangle
\end{array}
$$

Figure 8.5: Parameterized strands for the Needham-Schroeder protocol

The actions taken during the execution of a protocol are modeled with *nodes*. To simplify the discussion, we consider only **send** and **receive** actions. A node is a pair $\langle \pm, t \rangle$ consisting of a sign (+ for send and - for receive) and a message term t. By convention, these nodes are aligned vertically in the order in which they occur in the protocol. A *strand* is the sequence of such nodes (sends and receives) performed by a particular instance. A *role* is a parameterized strand, where variables are allowed to appear in the message terms. To make this discussion more concrete, parameterized strands for the initiator and responder roles of the Needham-Schroeder public key authentication protocol are given in Figure 8.5. Recall that the message flows for this protocol are:

1. $A \rightarrow B : \{N_a, A\}_{K_B}$
2. $B \rightarrow A : \{N_a, N_b\}_{K_A}$
3. $A \rightarrow B : \{N_b\}_{K_B}$

The variables A, B, N_a, N_b can be bound to specific values (specific principal names for A and B and specific nonces for N_a and N_b) to instantiate the roles. Such an instantiation would correspond to an instance in BRUTUS.

Note that nodes belonging to the same role are connected sequentially via a double arrow ($\Rightarrow$). In general, for any two nodes n_1 and n_2, $n_1 \Rightarrow n_2$ means that both nodes occur in the same strand and that the node n_2 represents the action immediately following the action n_1. In other words, $\Rightarrow$ represents the sequencing/ordering of actions in individual strands.

$$\frac{\mathit{Intercept}[a]}{\langle -, a\rangle} \qquad\qquad \frac{\mathit{Duplicate}[a]}{\begin{array}{c}\langle -, a\rangle\\ \Downarrow \\ \langle +, a\rangle \\ \Downarrow \\ \langle +, a\rangle\end{array}} \qquad\qquad \frac{\mathit{Decrypt}[m, k, k^{-1}]}{\begin{array}{c}\langle -, \{m\}_k\rangle\\ \Downarrow \\ \langle -, k^{-1}\rangle \\ \Downarrow \\ \langle +, m\rangle\end{array}}$$

Figure 8.6: Parameterized strands for the adversary

This graph can be augmented with single arrows ($\rightarrow$) as is done Figure 8.5. This notation represents the send action and the receive action corresponding to a particular message. More formally, for two nodes n_1 and n_2, $n_1 \rightarrow n_2$ means that $n_1 = \langle +, a\rangle$ and $n_2 = \langle -, a\rangle$ for some message term a.

The behavior of the adversary is modeled with penetrator strands, one for each capability of the adversary. For instance, the ability to intercept messages, the ability to duplicate messages, and the ability to decrypt messages with known keys can be modeled with the strands in Figure 8.6. Note that in this case, the $\Rightarrow$ relation is too restrictive. For example, the strand modeling decryption requires that the adversary first learn the encrypted message and then the decryption key. One can work around this restriction by including a second decryption strand in which this order is reversed.

In this formalism, the two relations, $\rightarrow$ and $\Rightarrow$ together, define a causal precedence. In other words, one can define a new relation $\preceq$ that is defined to be $(\rightarrow \cup \Rightarrow)^*$, the reflexive transitive closure of the union of the two arrow relations. This relation has the property that $n_1 \preceq n_2$ if and only if n_1 *must occur before* n_2. It captures which events must precede which other events. In [76], Thayer, Herzog, and Guttman prove that this relation is a partial order, so one immediately gets something analogous to the trace semantics of Mazurkiewicz [41]. Because the only ordering imposed on nodes is this causal precedence relation, Athena avoids the full interleaving semantics and so eliminates the need to perform a partial order reduction in the first place. In a sense, the partial order reduction is already built into the model itself.

This causal precedence relation then defines how one does a backward search in Athena. The search starts from a collection of partial strands, perhaps even a single node. This collection of strands must then be closed (backwards) under the causal relation. (A collection of strands that is closed under the causality relation like this is called a *bundle*.) This means that for any node in the collection, all preceding nodes along the same strand are added to the collection. In addition, all received messages must originate from somewhere. This origin may be a node already in the collection, or a new strand containing the node may be added to the collection. There may be multiple ways of satisfying this origination requirement. Each way would lead to a different history. All of these histories may eventually be explored.

While using bundles as a model of computation seems promising, the language used to specify requirements in Athena is a little cumbersome. In this language, the atomic propositions have the form $s \in C$ where s is a strand constant and C is a bundle variable that is typically universally quantified. Specifications then have the form:

$$\forall C \;.\; \bigwedge \Phi \Rightarrow \bigvee \Psi$$

where Φ and Ψ are sets of atomic propositions. Intuitively, the specification states that for any bundle C (for any causally closed collection of strands), if certain strands are in the bundle C (as specified by Φ), then some other strand (as specified by Ψ) must also be in the bundle C. While this is sufficient to specify typical authentication/agreement properties, it is not clear how to extend this to a more general logic.

To conclude, let us see how a typical specification might be written in Athena. For example, the authentication property for the Needham-Schroeder public key authentication protocol is specified as follows:

$$\forall C.\ \mathit{Resp}[A, B, N_a, N_b] \in C \Rightarrow \mathit{Init}[A, B, N_a, N_b] \in C$$

This specification states that any bundle that contains a responder role instantiated with arguments A, B, N_a, and N_b, must also contain an initiator role instantiated with the same constants. The secrecy of the nonces can be specified as follows:

$$\forall C.\ \mathit{Learn}[N_a] \in C \Rightarrow \mathbf{false}$$

$$\forall C.\ \mathit{Learn}[N_b] \in C \Rightarrow \mathbf{false}$$

```
NS3 [| evs ∈ ns_public;
        Says A B (Crypt (pubK B) | Nonce NA, Agent A |)
              ∈ set_of_list evs;
        Says B' A (Crypt (pubK A) {| Nonce NA, Nonce NB |})
              ∈ set_of_list evs |]
      ⟹
        Says A B (Crypt (pubK B) (Nonce NB))
              # evs ∈ ns_public
```

Figure 8.7: Isabelle example

These specifications basically state that no bundle can contain the adversary intercept role instantiated with the nonce N_a or the nonce N_b. In other words, the adversary never learns these nonces.

8.6 Isabelle

Paulson has investigated the use of Isabelle to prove protocols correct [3, 62]. Like the models used in Murϕ and in the NRL Analyzer, the protocol is encoded with a set of rules that describe how the honest participants in the protocol behave. These rules describe under what circumstances an agent will generate and send a certain message. Murϕ and the NRL Analyzer use these rules to describe the state that results when a particular action is taken or a particular message is sent. In contrast, Paulson uses these rules to inductively define the set of possible traces. In other words, each of Paulson's rules has the form "**if** the trace *evs* contains certain events, **then** it can be augmented by concatenating the new event *ev* to the end of the trace." For example, in the Needham-Schroeder public key protocol, the message the initiator sends in step number three is modeled by the rule in Figure 8.7. If the trace *evs* contains the actions where some A sent message one containing nonce NA to B and A receives a message two containing NA in the first field and NB in the second field, then the trace is augmented with the action where A sends message three containing NB.

Since this is a theorem proving environment, the requirement is given in a syntax identical to that used to model the protocol. Figure 8.8 gives

```
[| Says A B (Crypt (pubK B) {|Nonce NA, Agent A|})
        ∈ set_of_list evs;
   Says B' A (Crypt (pubK A) {|Nonce NA, Nonce NB|})
        ∈ set_of_list evs;
   A ∉ lost; B ∉ lost; evs ∈ ns_public |]
⟹
   Says B A (Crypt (pubK A) {|Nonce NA, Nonce NB|})
        ∈ set_of_list evs
```

Figure 8.8: Isabelle requirement example

a possible requirement for the Needham-Schroeder public key protocol. The requirement states that if A sends the nonce NA to B in message one and receives a message two back that contains NA in the first slot, then B must have sent this message. Since this protocol is flawed and does not satisfy the property, no proof is found. Unfortunately, no counter example is found either. The complete report on the analysis of the Needham-Schroeder public key protocol using Isabelle can be found in [61].

Unlike Murϕ, the NRL Analyzer, and Brutus, Paulson's technique is based entirely on theorem proving. Because he gives an inductive definition for the set of traces in the protocol, there is no limit placed on the number of protocol sessions considered. In other words, Paulson's proof of correctness is for an arbitrary number of protocol sessions and not for a specific finite-state model. However, like the NRL Analyzer, there is no guarantee of termination. In addition, it is not clear how to get feedback about possible errors in the protocol from a failed proof. This suggests that while Paulson's verification technique may be able to prove stronger statements about a protocol, model-checking techniques would be more useful to a protocol designer for debugging purposes.

Paulson is not the only one to investigate the use of theorem proving to verify security protocols. Recently, Cohen has developed a theorem prover called TAPS for verifying security protocols. He claims TAPS is faster and easier to use than Isabelle [15, 16]. However, all such theorem proving approaches have the general criticism that a significant amount of user interaction and user insight is required to carry out the verification.

8.7 Revere

I conclude this chapter by briefly returning to Logic of Authentication. REVERE is the name of Kindred's tool for automating protocol analysis using the BAN logic. REVERE is based on theory generation. Since the full theory of a set of formulas or axioms is often infinite, the aim of theory generation becomes generating a finite representation (set of formulas) of the full theory that has some set of useful properties. According to Kindred [34], not only must this set be finite, but it should also be small enough to be manipulated and examined directly. In addition, the set should be generated efficiently, and there should be an efficient way of testing specific formulas for membership in the full theory. Finally, the set should be canonical so that direct comparison of sets is meaningful. If these goals can be attained, then theory generation can be used not only to check whether certain formulas can be derived from the axioms, but also to compare the "interesting" formulas in the theories of two sets of axioms.

Kindred is able to attain these goals by restricting himself to logics that are simpler than the logic used in Isabelle or in TAPS. In particular, the BAN logic is one such simple logic. However, Kindred has shown the feasibility of theory generation for a broad class of logics that also includes AUTLOG, Kailar's accountability logic, and his own logic, RV, which is used to formalize the idealization gap for the BAN logic. Because the approach is a kind of theorem proving approach, REVERE has the same basic capability of proving positive statements about protocols. And while it cannot provide counter-examples, the generated theory does provide other avenues for protocol analysis that are unavailable to standard theorem proving approaches. So by restricting himself to simple logics, Kindred has been able to maintain the flavor of theorem proving while providing the same level of automation that is enjoyed by model checkers.

Chapter 9

Conclusions

In this chapter, I conclude by summarizing my results as well as highlighting promising avenues for future work.

9.1 Summary of Results

Brutus is a special purpose model checker for analyzing security protocols. It was inspired by the model proposed by Woo and Lam [83, 84]. This model provided a way to *reason* about security protocols. I took this model and investigated the possibility of automating the analysis using model checking techniques. A description of this original prototype appears in [39].

During the early part of this research, other researchers turned to existing model checkers to try to analyze authentication protocols. One of the biggest drawbacks to these pre-existing tools was having to model how the adversary acquires new knowledge. In essence, the derivation rules of section 4.3 would have to be hand coded into every model that is verified. While this process could be automated, the result would be a finite model of what is really an infinite message space. In other words, the user would have to specify ahead of time what messages are *interesting* so that the model can keep track of when or if they are learned by the adversary. This means either keeping a separate variable or a separate process for each interesting message and its submessages.

In contrast, I do not need to restrict the set of messages before beginning the analysis, since Brutus maintains the list of all known messages implicitly as described in section 4.3. While proving the correctness of the algorithms that maintain the adversary's knowledge, I have clarified why most models are limited to *atomic* keys. Ultimately, this is not a necessary restriction; however, it does allow us to more efficiently test if a specific message can be derived. In the language of Section 4.3, this restriction allows us to construct a set of "generators" by closing under elimination rules. All messages can then be derived from this set of generators using only introduction rules. If keys were allowed to be non-atomic, then we might have to use introduction rules to construct a key that would allow us to use an elimination rule (decryption). This result appears in [12].

In chapter 3, I describe the formal model of computation for Brutus. More importantly, I describe a new logic of knowledge with which to specify properties about security protocols. Apart from the standard authentication and secrecy properties, one can also express properties for electronic commerce protocols, including non-repudiation and a limited form of anonymity. This logic was first introduced in [11]. Hopefully, this logic will prove general enough to express new security properties as they come up.

Chapter 5 described how Brutus exploits the symmetry inherent in security protocols. Because symmetry can be consistently exploited in the same place in all protocols, this reduction can be performed without having to search for or to compute the symmetry. The symmetry is proven to exist in all models. This result eliminates one of the large drawbacks of general symmetry reduction, namely, computing the symmetry orbit relation, which can be prohibitively expensive. Since the symmetry reduction is hard wired into the algorithm, Brutus may miss some other symmetries that are present in the model. These results appear in [14].

One of the biggest contributions of this research is the partial order reduction described in Chapter 6, which also appeared in [13]. Like traditional partial order reductions, this reduction is implemented by expanding a subset of the enabled transitions in a state. The novelty of this reduction does not lie solely in its application to security protocol verification. I have also generalized the traditional partial order reduction by lifting the restriction that two "equivalent traces" agree on the specification. I have accomplished this by introducing the notion of *semi-invisible* actions. These are actions which can only make the specification false. By moving these actions sooner in a trace π one may end up with a new trace π' that does not agree with π on the specification, but which is guaranteed to violate the specification if π does. This kind of partial order reduction need not be limited to security protocol verification, but could be applicable to other domains where semi-invisible actions can be identified. Recall that for an action to be semi-invisible, all atomic propositions associated with that action must be monotonic in the model (once it becomes true, it remains true), and the atomic proposition must appear negatively in the specification formula.

Finally, in addition to the four protocols discussed earlier, I have modeled and analyzed the following protocols:

- "fixed" Needham-Schroeder-Lowe public-key protocol
- Needham-Schroeder symmetric-key protocol
- enhanced Needham-Schroeder symmetric-key protocol
- Kerberos
- TMN protocol
- Otway-Rees
- Andrew Secure RPC
- Yahalom
- Neuman Stubblebine protocol
- two protocols by Woo and Lam
- SPLICE/AS
- A protocol discussed by Meadows in "The NRL Protocol Analyzer: An Overview"

9.2 Conclusions

While analyzing all these protocols, I found all previously known attacks and I confirmed one limitation on the iKP protocol mentioned in [4]. However, I did not find any new attacks. While certainly disheartening, the lack of new attacks does not undermine the usefulness of the tool.

Certainly, a lack of new attacks is due in large part to the fact that so much of the work in this area has been done using the same model of computation (Dolev-Yao model). I believe that the usefulness of all these methods has already been demonstrated by prior results.

I believe the automatic addition of a "most general intruder" to any model being verified is a significant addition to the use of model checkers for security protocol verification. Certainly, the process of manually creating such an intruder would be highly error prone. In fact, Lowe augmented his tool by providing a means of generating such an intruder. However, even with this addition, the set of words which the intruder can learn is still artificially fixed to a specific set. While BRUTUS must also insure that the set of words it considers is finite, there is no a priori restriction on the set of words considered. This approach allows one to increase the set of words considered by the adversary easily and incrementally.

The experiments I ran demonstrate the usefulness of the symmetry and partial order reductions when applied to security protocols. They certainly increased the size of the models that can be analyzed by BRUTUS. Perhaps the real question should be how much more they allow us to verify. In the case of the Needham-Schroeder protocol they allowed us to investigate the possible interaction between multiple runs of the protocol. This level of analysis was not possible without the reductions. While the increase in the number of concurrent runs of a protocol that can be modeled is significant, the total number is still quite small. Certainly, the number is not large enough to justify great confidence in a model that is successfully verified. However, in my opinion, this is the wrong way to look at model checking. If one views model checking not as a way of verifying a system correct, but instead, as a way of debugging a system, then certainly the gains are significant. If one really wants to try to prove a protocol correct, theorem proving techniques are more appropriate. Perhaps a combination of the two approaches is best. One can first use a model checker to debug the system. Once no bugs are found, a theorem prover can be used to gain even more confidence in the system.

I am disappointed with my results when verifying electronic commerce protocols. While the partial order reduction and the symmetry reduction significantly decrease the number of states and traces considered during the verification, the reductions are not enough to enable me to consider multiple runs of the protocols. The greatest limitation when analyzing these systems was the complexity of the messages. This complexity significantly increased the number of messages that an honest agent is willing to receive. I believe that this limitation must be addressed before larger electronic commerce models (or other models with complex messages) can be analyzed.

Finally, I want to address two general weakness of the Dolev-Yao intruder model. This model abstracts away arithmetic properties of cryptographic functions. Although some work has been done in terms of introducing arithmetic properties [56], modeling all of the properties would make the analysis infeasible. So, in one sense, the adversary model is not powerful enough. This by itself is not a fatal flaw. One just needs to be aware of the assumptions being made and of the kinds of flaws that the analysis will miss. Another simplification that is usually made is giving the intruder complete control of the network. This simplifies the model since it is then unnecessary to model when or how an intruder may intercept, suppress, and modify messages on a network. When checking safety properties (as was done in all of the examples) this simplification works well. However, it does give the adversary too much power, and it prohibits the modeling of denial of service attacks. Since BRUTUS uses this Dolev-Yao model, it also cannot find denial of service attacks. Researchers will have to break away from the traditional Dolev-Yao model in order to perform this kind of analysis. Indeed, Meadows has already started looking in this direction [48].

9.3 Future Work

There are several avenues of research that I hope are eventually explored. Many have to do with improving the BRUTUS model checker so that it can handle larger models. Others apply to security model checking in general. In addition, there are entire new families of security protocols with different security properties waiting to be analyzed.

9.3.1 Improving BRUTUS

While the symmetry and partial order reductions presented in this thesis significantly increase the efficiency of BRUTUS, more still has to be done. I see some promising directions for research in this area. The first is to implement a parallel model checking algorithm. This idea is extremely promising. Since the history at a particular state differentiates two states that would otherwise be identified, the underlying computations of a model form a tree. Therefore, performing the analysis in parallel should be as simple as assigning different branches of the tree to different processors. Another direction would be to restrict the specification logic so that it cannot distinguish states that differ only in their histories. I believe this can be done by simply disallowing the nesting of the past-time operators $\Diamond_P$ and $\Box_P$. In this case, states could be hashed in order to avoid expanding the same state more than once. However, it is not clear how many unique states there would be in this case. The size of the hash table may become prohibitively large.

I believe that BRUTUS could also be improved by adding semantics to internal actions. Recall that in the logic presented, an internal action (for example **debit**) does not have any semantics associated with it. Currently, the internal actions are used solely to mark when certain important events take place. In the future, I want to explore the possibility of having internal actions that can alter the state of the system. For example, it may be useful to have a **debit** action that actually debits a customer's account. However, it is not entirely clear what affect this will have on the symmetry and partial order reductions.

9.3.2 Improving Security Model Checking

Model checking has been around for quite some time, and there has been much research dedicated to trying to overcome or to circumvent the state explosion problem. It may be possible to adapt some of these techniques to the domain of security protocol verification.

The first such technique is data abstraction. This technique is particularly attractive because it is the enormous number of messages that seems to lead to such large state spaces when analyzing security protocols. When an honest agent is ready to receive a message, there are typically many messages the adversary can generate that the honest agent is willing to receive. If a number of those messages can be collapsed into a single abstract message, a significant reduction in the state space may be possible.

A second technique that may be applicable to security protocol verification is compositional model checking. At first glance, this technique also seems attractive. There are many complex security protocols that are made up of smaller protocols. For example, an electronic transaction might consist of an electronic commerce protocol that first requires authentication and session key exchange. The key exchange may require the use of public keys that must be retrieved using certificates. It is unlikely that the entire system can be analyzed in a single monolithic model. However, if the requirements of the individual components can be separated in such a way that the conjunction of their specifications implies the correctness of the entire system, then it would suffice to analyze the smaller components in isolation. Of course such a decomposition is not easy and is not always possible.

9.3.3 Other Families of Protocols

This discussion of future work would not be complete without at least mentioning the possibility of applying model checking to other families of protocols. In particular, schemes for electronic voting, electronic auctions, and digital cash, as well as new uses of smart cards have been proposed and implemented. The application of formal methods to these protocols is highly desirable. It is very likely that a more expressive modeling language and specification language will be required. What is unclear is how much must be added or changed in existing analysis techniques. While there is no direct evidence, the results so far are encouraging. For example, BRUTUS was originally developed to analyze authentication protocols. In particular, the only properties originally checked were secrecy and correspondence. However the logic and the model checker proved capable of specifying and verifying authorization and non-repudiation properties. It is my hope that with some work, BRUTUS can be applied to other security applications.

Bibliography

[1] M. Abadi and M. Tuttle. A semantics for a logic of authentication. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, pages 201–216, August 1991.

[2] R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In J. W. deBakker, C. Huizing, W.-P.deRoever, and G. Rozenberg, editors, *Real-Time: Theory in Practice. REX Workshop Proceedings*, volume 600 of *Lecture Notes in Computer Science*, pages 74–106. Springer-Verlag, 1991.

[3] G. Bella and L. C. Paulson. Using Isabelle to prove properties of the Kerberos authentication system. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997.

[4] M. Bellare, J. Garay, R. Hauser, A. Herberg, H. Krawczyk, M. Steiner, G. Tsudik, and M. Waidner. *i*KP - a family of secure electronic payment protocols. In *Proceedings of the 1st USENIX Workshop on Electronic Commerce*, July 1995.

[5] D. Bolignano. An approach to the formal verification of cryptographic protocols. In *Proceedings of the 3rd ACM Conference on Computer and Communication Security*, 1996.

[6] S. Brackin, C. Meadows, and J. Millen. CAPSL interface for the NRL Protocol Analyzer. In *Proceedings of ASSET 99*. IEEE Computer Society Press, 1999.

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

[8] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[9] E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1/2):77–104, 1996.

[10] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[11] E. M. Clarke, S. Jha, and W. Marrero. A machine checkable logic of knowledge for specifying security properties of electronic commerce protocols. In *Workshop on Formal Methods and Security Protocols*, 1998.

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

[13] E. M. Clarke, S. Jha, and W. Marrero. Partial order reductions for security protocol verification. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1785 of *Lecture Notes in Computer Science*, pages 503–518. Springer-Verlag, 2000.

[14] E. M. Clarke, S. Jha, and W. Marrero. Verifying security protocols with BRUTUS. *ACM Transactions on Software Engineering and Methodology*, 9(2), October 2000.

[15] E. Cohen. TAPS: A first-order verifier for cryptographic protocols. In *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 568–571. Spinger-Verlag, 2000.

[16] E. Cohen. TAPS: A first-order verifier for cryptographic protocols. In *Proceedings of the Computer Security Foundations Workshop*. IEEE Computer Society Press, 2000.

[17] D. Craigen and M. Saaltink. Using EVES to analyze authentication protocols. Technical Report TR-96-5508-05, ORA Canada, 1996.

[18] G. Denker and J. Millen. CAPSL integrated protocol environment. In *DARPA Information Survivability Conference (DISCEX 2000)*, pages 207–221. IEEE Computer Society Press, 2000.

[19] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, March 1989.

[20] E. A. Emerson. *Handbook of Theoretical Computer Science: Volume B, Formal Models and Semantics*, chapter Temporal and Modal Logic, pages pp. 995–1072. North-Holland Pub. Co./MIT Press, 1990.

[21] E.A. Emerson and A.P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1/2):105–130, 1996.

[22] P. Godefroid, D. Peled, and M. Staskauskas. Using partial order methods in the formal validation of industrial concurrent programs. In *ISSTA'96, International Symposium on Software Testing and Analysis*, pages 261–269, San Diego, California, USA, 1996. ACM Press.

[23] T. A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In W. Kuich, editor, *Proceedings of the 19th International Colloquium on Automata, Languages, and Programming*, volume 623 of *Lecture Notes in Computer Science*, pages 545–558. Springer-Verlag, 1992.

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

[25] N. Hopper, S. Seshia, and J. Wing. A comparison and combination of theory generation and model checking for security protocol analysis. In *Workshop on Formal Methods in Computer Security*, 2000.

[26] H. W. Hungerford. *Abstract Algebra: An Introduction.* Saunders College Publishing, 1990.

[27] SRI International Computer Science Laboratory. PVS web site. http://pvs.csl.sri.com/, 2000.

[28] C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–76, 1996.

[29] K. Jensen. Condensed state spaces for symmetrical coloured Petri nets. *Formal Methods in System Design*, 9(1/2):7–40, 1996.

[30] S. Jha. *Symmetry and Induction in Model Checking*. PhD thesis, Carnegie Mellon University, 1996.

[31] R. Johnsonbaugh. *Discrete Mathematics*. Prentice Hall, fourth edition, 1997.

[32] D. Kahn. *The Codebreakers: The Story of Secret Writing*. Scribner, revised edition, 1996.

[33] R. Kailar. Accountability in electronic commerce protocols. *IEEE Transactions on Software Engineering*, 22(5), May 1996.

[34] D. Kindred. *Theory Generation for Security Protocols*. PhD thesis, Carnegie Mellon University, 1999.

[35] D. Kindred and J. M. Wing. Fast, automatic checking of security protocols. In *USENIX 2nd Workshop on Electronic Commerce*, 1996.

[36] D. Kindred and J. M. Wing. Closing the idealization gap with theory generation. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997.

[37] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, 1996.

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

[39] W. Marrero, E. M. Clarke, and S. Jha. A model checker for authentication protocols. In *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997.

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

[41] A. Mazurkiewicz. Trace theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Advances in Petri Nets*, number 225 in LNCS, pages 279–324, Bad Honnef, Germany, 1986. Springer-Verlag.

[42] K. L. McMillan. *Symbolic model checking - an approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, 1992.

[43] C. Meadows. A model of computation for the NRL Protocol Analyzer. In *Proceedings of the 1994 Computer Security Foundations Workshop.* IEEE Computer Society Press, June 1994.

[44] C. Meadows. Analyzing the Needham-Schroeder public key protocol: A comparison of two approaches. In *Proceedings of ESORICS.* Springer-Verlag, 1996.

[45] C. Meadows. Language generation and verification in the NRL Protocol Analyzer. In *Proceedings of the 9th Computer Security Foundations Workshop.* IEEE Computer Society Press, 1996.

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

[47] C. Meadows. Analysis of the internet key exchange protocol using the NRL Protocol Analyzer. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, 1999.

[48] C. Meadows. A formal framework and evaluation method for network denial of service. In *Proceedings of the IEEE Computer Security Foundations Workshop.* IEEE Computer Society Press, 1999.

[49] C. Meadows and P. Syverson. A formal specification of requirements for payment transactions in the SET protocol. In *Preproceedings of Financial Cryptography*, 1998.

[50] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.

[51] J. Millen. The Interrogator model. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 251–260. IEEE Computer Society Press, 1995.

[52] J. Millen. CAPSL: Common authentication protocol specification language. Technical Report MP 97B48, The MITRE Corporation, 1997.

[53] J. Millen. A CAPSL connector to Athena. In *Workshop on Formal Methods in Computer Security*, 2000.

[54] J. Millen. CAPSL web site. http://www.csl.sri.com/~millen/capsl, 2000.

[55] M. Minea. *Partial Order Reduction for Verification of Timed Systems.* PhD thesis, Carnegie Mellon University, 1999.

[56] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murϕ. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, 1997.

[57] J. C. Mitchell, V. Shmatikov, and U. Stern. Finite-state analysis of SSL 3.0. In *The 7th USENIX Security Symposium*, pages 201–216, 1998.

[58] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[59] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Eleventh International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1993.

[60] F. Pagani. Partial orders and verification of real-time systems. In B. Jonsson and J. Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 327–346. Springer-Verlag, 1996.

[61] L. Paulson. Mechanized proofs of security protocols: Needham-schroeder with public keys. Technical Report 413, University of Cambridge Computer Laboratory, 1997.

[62] L. Paulson. Proving properties of security protocols by induction. In *Proceedings of the 1997 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 70–83, 1997.

[63] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[64] L. C. Paulson. Isabelle web site. http://www.cl.cam.ac.uk/Research/HVG/Isabelle/, 2000.

[65] D. Peled. All from one, one from all: On model checking using representatives. In *5th International Conference on Computer Aided Verification, Greece*, number 697 in LNCS, pages 409–423, Elounda Crete, Greece, 1993. Springer-Verlag.

[66] D. Peled. Combining partial order reductions with on-the-fly model-checking. *Journal of Formal Methods in Systems Design*, 8 (1):39–64, 1996. also appeared in 6th International Conference on Computer Aided Verification 1994, Stanford CA, USA, LNCS 818, Springer-Verlag, 377-390.

[67] D. Peled. Partial order reduction: Linear and branching temporal logics and process algebras. In *Partial Order Methods in Verification*, DIMACS, Princeton, NJ, USA, 1996. American Mathematical Society.

[68] D. Prawitz. *Natural Deduction: A Proof-Theoretical Study*. Almqvist & Wiksell, 1965.

[69] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[70] R. L. Rivest, A. Shamir, and L. M. Adleman. On digital signatures and public key cryptosystems. Technical Report MIT/LCS/TR-212, MIT Laboratory for Computer Science, 1979.

[71] B. Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., second edition, 1996.

[72] D. Song. Athena: An automatic checker for security protocol analysis. In *12th IEEE Computer Security Foundation Workshop*, 1999.

[73] D. Song, S. Berezin, and A. Perrig. Athena: A novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*. Submitted for publication.

[74] S. Stoller. Justifying finite resources for adversaries in automated analysis of authentication protocols. In *Workshop on Formal Methods and Security Protocols*, 1998.

[75] P. Syverson and C. Meadows. A formal language for cryptographic protocol requirements. *Designs, Codes, and Cryptography*, 7(1/2):27–59, 1996.

[76] F. J. Thayer, J. C. Herzog, and J. D. Guttman. Strand spaces: Why is a security protocol correct? In *Proceedings of the 1998 IEEE Sympoposium on Security and Privacy*, 1998.

[77] Ioannis Trithemii. *Polygraphiae libri sex.* 1518.

[78] A. Valmari. A stubborn attack on state explosion. In *CAV90*, volume 3 of *DIMACS*, pages 25–42, 1991.

[79] A. Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, number 483 in LNCS, pages 491–515. Springer-Verlag, 1991.

[80] A. Valmari. Stubborn sets of colored Petri nets. In *Proceedings of the 12th International Conference on Application and Theory of Petri Nets*, pages 102–121, Gjern, Denmark, 1991.

[81] B. Willems and P. Wolper. Partial-order methods for model checking: From linear time to branching time. In *Elevneth Annual IEEE Symposium on Logic in Computer Science*, pages 294–303, New Brunswick, 1996. IEEE Computer Society Press.

[82] P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In *CONCUR '93*, number 715 in LNCS, pages 223–246, Hildesheim, 1993. Springer-Verlag.

[83] T. Y. C. Woo and S. S. Lam. A semantic model for authentication protocols. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, 1993.

[84] T. Y. C. Woo and S. S. Lam. Verifying authentication protocols: Methodology and example. In *Proceedings of the International Conference on Network Protocols*, 1993.

[85] T. Y. C. Woo and S. S. Lam. A lesson on authentication protocol design. In *Operating Systems Review*, pages 24–37, 1994.