An Automatic Approach for Building Secure Systems

by

Xiaodong Dawn Song

B.S. (Tsinghua University) 1996
M.S. (Carnegie Mellon University) 1999

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Doug Tygar, Chair
Professor David Wagner
Professor John Chuang

Fall 2002

The dissertation of *Xiaodong Dawn Song* is approved:

Chair Date

Date

Date

University of California at Berkeley

Fall 2002

An Automatic Approach for Building Secure Systems

by

Xiaodong Dawn Song

Abstract

An Automatic Approach for Building Secure Systems

by

Xiaodong Dawn Song

Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Doug Tygar, Chair

Building a secure system is one of the most complex and error-prone processes in computing. Unfortunately, the current design and development process is primarily manual, ad hoc, and far from satisfactory. In this thesis, I propose a new automatic approach for building secure systems. With this approach, users only need to specify desired security and system requirements. Such an automatic system can then explore the possible design space, generate design candidates, evaluate those candidates to find the most efficient correct design, and finally generate source code implementing the optimal design. Compared to the manual design and development process, this automatic approach is faster, more economical, yields protocols and implementations of higher security and efficiency.

To demonstrate this automatic approach for building secure systems, we have

focused on one of the most important aspects, designing and developing security protocols. In particular, I have designed and developed an automatic toolbox, called *Athena*, for automatic protocol generation, verification and implementation. The system contains three components: the automatic protocol generator (APG), the automatic protocol verifier (APV), and the automatic code generator (ACG). APV proposes a new automatic approach for security protocol analysis. Given a security protocol and desired security requirements, APV analyzes the protocol automatically. When APV terminates, it can either generate a proof of correctness, if the protocol satisfies the given requirements, or generate a counterexample, if the protocol is flawed. APV is the first automatic approach that has both the capability of providing a proof of correctness when the protocol is correct and the capability of generating a counterexample when the protocol is flawed. Athena runs efficiently and this high efficiency enables the approach of automatic protocol generation. APG uses powerful pruning techniques to drastically reduce the protocol search space. It is the first approach for automatic generation of security protocols. In our experiments, APG has generated new protocols that are more efficient than previous protocols proposed in the literature for different system settings. The composition of APG, APV, and ACG forms an end-to-end system, Athena, which is the first system able to automatically generate, verify, and implement security

protocols.

Professor Doug Tygar
Dissertation Committee Chair

This dissertation is dedicated to my parents, sister Weiwei, and Adrian.

Contents

List of Figures

List of Tables

Acknowledgements

Foremost, I am greatly indebted to Doug Tygar. I would like to thank him for his invaluable help and guidance, continuous and unconditional support throughout my graduate study. I would also like to thank him for encouraging me to pursue an academic career.

I would also like to thank other members of my thesis committee, David Wagner and John Chuang. This dissertation greatly benefitted from their comments, feedback, and high standards. I have learned invaluable lessons from interacting with my committee, and I am grateful for their guidance. I would also like to thank John Millen and George Necula. Even though they are not on my thesis committee, but I have learned invaluable lessons from interacting with them as well.

I would like to thank my collaborators Sergey Berezin, Adrian Perrig, and Doantam Phan for their help in this work.

I would like to thank Ed Clarke and his group for their support especially during the early stage of this work. I would also like to thank many colleagues, researchers, and friends for their great feedback and wonderful exchanges of ideas, including Sergey Berezin, Nikita Borosov, Fay Chang, Monica Chew, Joshua Guttman, Jonathan Herzog, Yihchun Hu, Rob Johnson, Catherine Meadows, John Mitchell, Rob O'Callohan, Sylvan Pinsky, Andrey Scedrov, Vitaly Shmatikov Javier Thayer, Jeannette Wing.

In particular, I would like to thank Adrian Perrig for his continuous collabora-

tion and help.

I would like to thank my parents and my sister Weiwei for their unconditional love.

The National Science Foundation, the Defense Advanced Research Projects Agency, and the US Postal Service have provided grants and contracts that have helped support much of the development of this material. I gratefully acknowledge this support. In particular, this research was sponsored in part by the United States Postal Service (contract USPS 102592-01-Z-0236), by the United States Defense Advanced Research Projects Agency (contract N66001-99-2-8913), and by the United States National Science Foundation (grant FD99-79852). DARPA Contract N66001-99-2-8913 is under the supervision of the Space and Naval Warfare Systems Center, San Diego. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the United States government, of DARPA, NSF, USPS, any of its agencies.

Chapter 1

Introduction

1.1 Motivation

As the Internet continues to expand, more and more systems become interconnected through the Internet. On-line supply-chain management, stock trading over the Internet, rapid information dissemination, video conferencing, Internet telephony, and e-commerce are just a few examples. Even critical infrastructure, such as the control system of the power grid, is migrating onto the Internet.

Another trend is the proliferation of wireless services. We are witnessing an explosion of the number of small devices interconnected through wireless communication. Wireless handheld devices to read email, cell phones with cameras and digital assistants, and wireless devices to enable physical access are a few examples.

This rapid growth of interconnectivity enables new applications and higher efficiency than ever before, but also opens doors to cyber attacks. Hence security has become a central concern. Many systems were designed assuming trusted environments where parties cooperate and follow specified behaviors. However, these systems often break down under the hostile attacks in cyberspace. Even when security was a criterion during the design, many systems were designed with ad hoc security solutions and are not secure against malicious attacks. There is a crying need for making systems more secure.

Building a secure system is one of the most complex and error-prone processes in computing. Experience has shown that even systems designed by security experts are often plagued with vulnerabilities:

- Many security protocols are flawed even though security experts designed them. For example, Needham and Schroeder proposed a public-key authentication protocol in 1978 [70] and the protocol was used widely in authentication systems such as Kerboros [28]. However, 18 years later Gavin Lowe discovered a serious flaw in the protocol when he tried to formally verify the protocol [54]. Other examples include: early versions of Secure Sockets Layer (SSL), a widely used security protocol for establishing secure web connections; and CCITT X. 509, a protocol standard for secure communication recommended by CCITT [16].

- Security flaws not only occur in the design, but also in implementation. For

example, Pretty Good Privacy (PGP) is a widely distributed public-key encryption and signature software package [98]. To generate a secure public/private key pair used for encryption/decryption and signature verification/generation, the software needs to call a random number generation function to obtain random numbers. The random number generation function returns two values, a randomly generated number and an error code that indicates whether the function call has been successful. Because a careless programmer mistook the error code as the output random number in PGP, the PGP implementation will generate predictable keys in certain circumstances. This makes it possible for an attacker to guess a user's private key and to then be able to decrypt a user's encrypted documents and forge digital signatures [17].

One of the main difficulties in building a secure system is that the complexity, subtlety, and interactions among different components in a large system are often beyond the reach of even the most experienced security experts, not to mention average programmers who lack security expertise. In order to build secure systems, we need to build automatic tools to help system designers and developers to understand, analyze, optimize, and implement secure systems. In particular:

- Given a security protocol, we need a mechanism to guarantee the correctness of security protocols, especially since even security experts make mistakes when designing security protocols. However, manual proofs are often

tedious (so they may not get used) and are error-prone (still leaving the security protocol vulnerable). Therefore, we need automatic protocol verifiers that can check the correctness of security protocols automatically and efficiently.

- Given a set of desired security properties, we need a mechanism to design secure and efficient protocols that satisfy the given requirements. A manual design process may take a long time and a great deal of effort. Also, even though many different protocols may satisfy the given security requirements, they may differ greatly in performance. We would like to find the *optimal* protocol that satisfies the security properties and gives the lowest overhead according to the system metric. A manual design process often fails to find the optimal protocol simply because the designer cannot examine a large space of candidates. We need an protocol generation tool that can automatically generate security protocols that satisfy the given security requirements and are efficient with respect to the given system metric.

- Given a correctly designed security protocol, we need a mechanism to efficiently generate a correct implementation of the protocol. Manual implementation takes a long time and almost inevitably introduces programming errors that can leave the system vulnerable, even if the design is correct. We need an automatic implementation tool that can translate directly from

a high-level proven correct protocol description to a low-level implementation.

A toolbox containing an automatic protocol verifier, an automatic protocol generation tool, and an automatic protocol implementation tool can simplify and automate the design and implementation of security protocols. Each individual component is useful and when put together, they form an end-to-end system. Given a set of desired security properties, the protocol generation tool automatically generates candidate protocols. The automatic protocol verifier then scrutinizes the candidate protocols and outputs the correct protocols. The output correct protocols are then input into the automatic implementation tool that then translates the protocols into source code. With these tools, the system design and development process will become more productive and effective, and the systems will become more secure.

1.2 Challenges

Building a toolbox that can automatically generate, analyze, and implement security protocols is a challenge.

First, it is a difficult problem to design efficient algorithms to automatically verify the correctness of security protocols. We first consider a concrete example of how a security protocol may be broken. We describe the two-party public-

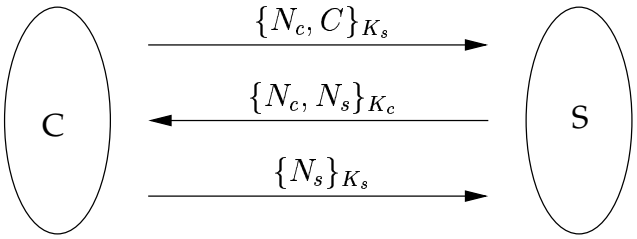


Figure 1.1: The Needham-Schroeder Protocol

key authentication protocol proposed by Needham and Schroeder [70]. We will use this protocol as a running example in the rest of the thesis and refer to it as the Needham-Schroeder protocol.

In the Needham-Schroeder protocol, a client C and a server S communicate with each other and try to reach mutual authentication and establish shared secrets at the end of the protocol execution. The protocol uses public-key encryption algorithms, such as RSA [83]. Each party has its own public/private key pair: K_s represents the public key of the server S, and K_c represents the public key of the client C. Throughout the thesis, we will use the following notation: $\{m\}_k$ represents the encryption of plaintext message m with encryption key k, m, m' represents the concatenation of messages m and m'. The the Needham-Schroeder protocol protocol is shown in Figure 1.1 and described below:

1. When the client C wants to start a protocol session with a server S, C generates a nonce N_c, encrypts the concatenation of N_c and C with S's public key K_s, and sends the ciphertext $\{N_c, C\}_{K_s}$ to S.

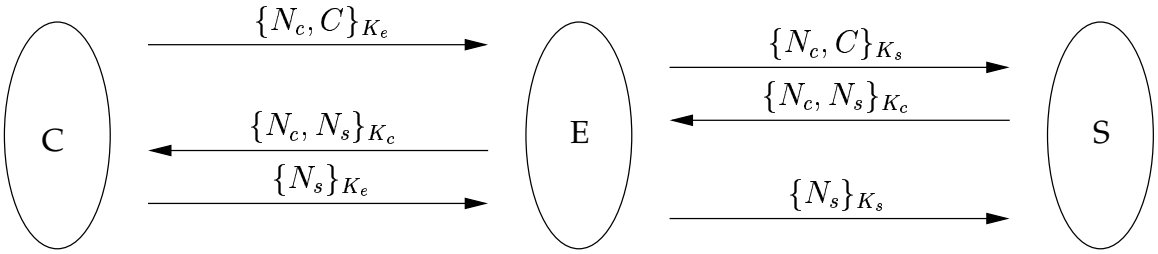


Figure 1.2: The Security Flaw in the Needham-Schroeder Protocol

2. Upon receiving the message, S decrypts the message and learns about N_c and C. S then generates its own nonce N_s, encrypts the concatenation of N_c and N_s with C's public key K_c, and sends the ciphertext $\{N_c, N_s\}_{K_c}$ to C.

3. Upon receiving the message, C decrypts the message and checks that the message is well formed, meaning the contained N_c is equal to the nonce N_c it generated in this session. C then learns about N_s, encrypts N_s with S's public key and sends the ciphertext $\{N_s\}_{K_s}$ to S.

This is a typical example of an authentication protocol. Supposedly, at the end of the protocol execution, C and S should reach mutual authentication and establish shared secrets N_c and N_s. Unfortunately, 18 years after the initial publication of the protocol, Lowe discovered the following serious flaw in the protocol when he tried to formally verify the protocol. Suppose a client C wants to start a protocol session with a server E, but server E is malicious and does not follow the protocol steps. Server E then tries to start a protocol session with another server S and tries to impersonate S as C. The protocol execution is shown in Figure 1.2.

At the end of the protocol execution, the authentication and secrecy properties of the protocol are violated—S thinks that it was talking to C (which is not the case), and N_c and N_s are supposed to be shared secrets between C and S, but E knows both of them.

Security protocols specify concurrent communication among different parties. Therefore, security protocol analysis has to consider all possible interactions among the different parties in protocol executions. Unfortunately the number of different configurations of protocol executions grows exponentially as the number of parties in the protocol execution grows, and the space of possible protocol executions may even be infinite because an attacker can perform arbitrary polynomial-time actions.

Many researchers have proposed different approaches for security protocol verification, ranging from a theorem-proving approach, model-checking approach, to logic-based approaches. Unfortunately, previous approaches have many drawbacks. Previous theorem-proving approaches are not fully automatic and require expert knowledge to use the tools. They also have the disadvantage that they cannot find counterexamples when a protocol is flawed. Previous model-checking approaches can automatically test whether a protocol is correct for some small test cases, but cannot provide a proof of correctness of the protocol for all cases. They also severely suffer from the state space explosion problem that prevents them from analyzing complicated protocols or bigger test cases.

For automatic protocol generation, the main challenge is that the protocol search space is enormous. To search for an efficient protocol satisfying the required security properties is like looking for a diamond in the desert. The search requires efficient search strategies and pruning algorithms to reduce the search space dramatically to make this approach viable. We were the first to explore the area of automatic generation of security protocols.

1.3 Thesis Contributions

This thesis proposes a new automatic approach for building secure systems. With this approach, users only need to specify desired security requirements and system requirements. An automatic system can then explore the possible design space, generate design candidates, evaluate them to find the most efficient correct design, and finally generate source code implementing the optimal design. Compared to the manual design and development process, this automatic approach is faster, more economical, yields protocols and implementations of higher security and efficiency.

To demonstrate this automatic approach for building secure systems, we have focused on one of the most important aspects, designing and developing security protocols. In particular, we have designed and developed an automatic toolbox, called *Athena*, for automatic protocol generation, verification and implementation.

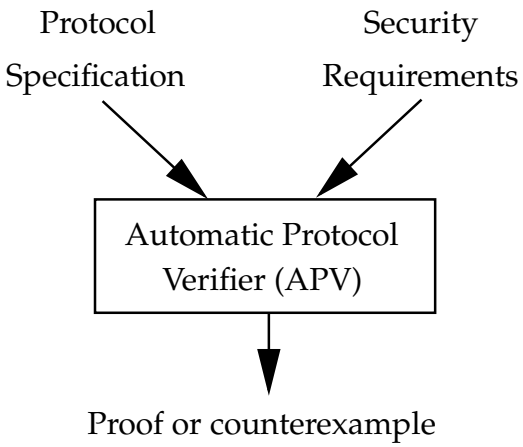


Figure 1.3: The Automatic Protocol Verifier (APV)

The system contains three components: the automatic protocol generator (APG), the automatic protocol verifier (APV), and the automatic code generator (ACG):

- APV (Figure 1.3): Given a security protocol specification and security requirements, APV automatically analyzes the security protocol. When APV terminates, it either generates a proof of correctness if the protocol satisfies the given requirements, or an example of a flaw (called a *counterexample*) if the protocol does not.

- APG (Figure 1.4): Given a system specification and desired security requirements, APG automatically generates candidate protocols that satisfy the given system requirements. The candidate protocols are passed to APV in increasing order of the overhead according to a specified metric function in the system requirements. APV then analyzes the candidate protocols. The first correct output security protocol will be the most efficient protocol with

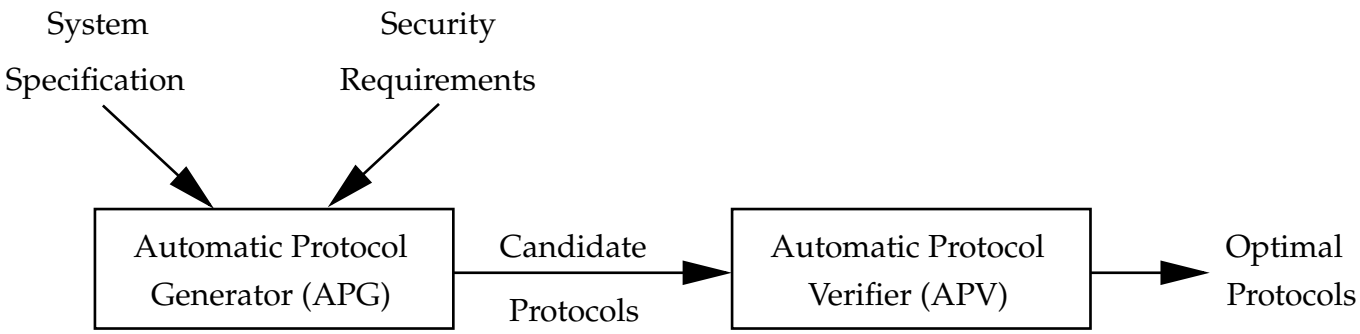


Figure 1.4: The Automatic Protocol Generator (APG)

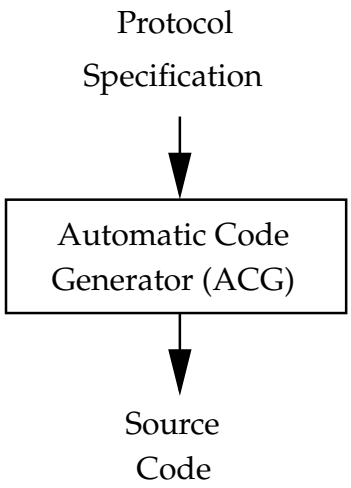


Figure 1.5: The Automatic Code Generator (ACG)

respect to the specified metric function.

- ACG (Figure 1.5): Given a security protocol specification, ACG automatically implements the protocol into Java source code.

Each individual component can be used separately. And the composition of the APG, APV, and ACG forms an end-to-end system, Athena. To automatically design and implement a security protocol, the user specifies desired security and

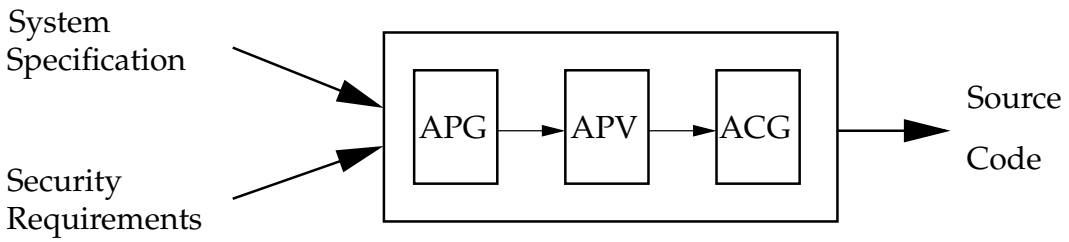


Figure 1.6: The Athena System

system requirements, and then Athena runs automatically. The APG and APV together find an efficient protocol that satisfies the desired security and system requirements and the ACG produces the Java source code for the generated protocol (Figure 1.6).

In order to design and develop Athena, we have to overcome many research challenges. This work has resulted in a number of technical contributions:

- APV proposes a new approach for security protocol analysis. Previous approaches for security protocol analysis fall mainly into two camps: theorem-proving and model-checking. Previous theorem-proving approaches prove properties of security protocols by induction. They have great expressive power, but suffer from the fact that they are not automatic and require a lot of human interaction [74, 75]. They also have the disadvantage that they do not have the capability to find counterexamples for flawed protocols because they prove properties by induction [74, 75].

  Previous model-checking approaches can automatically test whether a protocol is correct for small test cases, but cannot provide a proof of correctness

of the protocol for all cases. They start their analysis by selecting a small number of participants that form the test case. They then enumerate various possible interactions of the different participants and traces of possible protocol executions. For each protocol execution trace, they check whether the specified security property is satisfied or not. If the specified security property is not satisfied in a trace, then the trace gives the counterexample. Even if the specified security property is satisfied in all the traces enumerated by the model checker, there may still exist a trace not covered by the test case that does not satisfy the specified security property. Previous model-checking approaches also suffer severely from state space explosion, preventing them from analyzing complicated protocols or bigger test cases. These approaches are based on the traditional *trace-based model*, where each principal is modeled by a *process*. The global state space of the protocol is the Cartesian product of local state spaces of individual processes. The state transition of each local process is based on sending or receiving a message. The processes are composed *asynchronously*, and the global transition relation is an interleaving of local transitions. Due to such parallel composition and the interleaving semantics, the number of states and transitions to be explored grows exponentially with the number of participants involved in the protocol execution [76, 77]. Because of the state space explosion problem, they are often only able to verify protocol executions with small number of

participants, e.g. fewer than five.

APV is the first automatic approach that has both the capability of providing a proof of correctness when the protocol is correct and the capability of generating a counterexample when the protocol is flawed. APV achieves this by combining techniques from both theorem-proving and model-checking, while leveraging a new compact state representation and a search algorithm to reduce the search space. Unlike previous model checking approaches for security protocol analysis, APV starts the protocol analysis from a final state derived from the specified security property and performs a backward reachability analysis. In this way, APV does not need to predefine the set of principals in the protocol execution as the case in the previous model checking approaches. At each state, APV exhaustively enumerates what must have happened in order to lead to this state. In previous model checking approaches, each state is a trace of protocol execution. APV uses a new compact state representation. A trace representation of a protocol execution specifies the sequence of actions in the protocol execution. However, many of the actions may be irrelevant to the security property, and many traces may exhibit the same evaluation of the security property. The compact state representation in APV is based on this observation. Instead of keeping the information about every action in a trace, the compact state representation in APV omits actions that are proven to be irrelavant to checking the security property, and only

keep the necessary information, i.e., the exact causal relationships among actions. APV also uses symbolic representation – a state may contain free variables, and a state containing a free variable represents an infinite number of instatiations of the free variable. In this way, each state in APV natually represents an equivalence class of an infinite number of traces that all exhibit the same evaluation of the security property. A state transition in APV represents the transition of a set of infinite number of traces. Therefore, by checking the security property of a finite number of states, APV can check the security property of the inifinite number of all possible traces of protocol executions. When APV terminates, if the security property is satisfied in every state in the proof search, then APV proves that the security property holds for all possible protocol executions that satisfy the protocol specification; otherwise, if the security property is not satisfied in a state, APV can reconstruct a counterexample for this state. Because the state representation in APV is no longer an asynchronous composition of local processes as in the trace model used by previous model checking approaches, APV natually avoids the state space explosion problem caused by asynchronous composition. As a result, APV runs efficiently, often analyzing a protocol in a fraction of a second.

In addition, the development of APV also enables the approach of automatic protocol generation, which has not been achieved before partially because no such an efficient automatic protocol verifier existed.

- APG is the first approach for automatic generation of security protocols. APG uses powerful pruning techniques to reduce the protocol search space drastically and it runs efficiently. In our experiments, APG was able to generate desired security protocols within hours. APG has generated new protocols that are more efficient than previous protocols proposed in the literature for different system settings.

- Our Athena system proposes a new approach for security protocol design and development—the automatic approach for security protocol generation, verification and implementation. Our toolbox provides an end-to-end system for automatic protocol design and implementation. The user only needs to input the desired security properties and system requirements and the system will automatically find an efficient protocol satisfying the desired properties and output an implementation in Java source code. Our experiments demonstrate that this approach is efficient and viable, and yields protocols and implementations of higher security and efficiency.

In the rest of the thesis, we will first introduce background material in Chapter 2, and then describe the three components APV, APG, and ACG in Chapters 3, 4, and 5. We will then conclude and discuss future work in Chapter 6. Chapter 3 represents work done in part in collaboration with Sergey Berezin and Adrian Perrig [89, 87]. Chapter 4 represents work done in collaboration with Adrian Perrig [78, 80]. Chapter 5 represents work done in collaboration with Adrian Perrig

and Doantam Phan [88].

Chapter 2

Background

In this chapter we review some background material. We first review some material about relevant cyrptographic primitives in Section 2.1, and then review material about formal methods in Section 2.2.

2.1 Security Building Blocks

Security protocols are built on top of underlying cryptographic primitives. The properties of cryptographic primitives enable the properties of the security protocols. In this section, we review two types of cryptographic primitives that we use in later discussion: nonces and encryption/decryption schemes.

2.1.1 Random Numbers and Nonces

A nonce is a freshly generated random number. Because generating truly random numbers may be inefficient, in practice we often use pseudo-random generators to generate pseudo-random bit strings. A pseudo-random generator takes a small seed as input and generates a long sequence of bit string which is indistinguishable from a sequence of truly random bit string of equal length to a polynomial-time bounded attacker who does not know the original seed.

2.1.2 Encryption/Decryption Schemes

An encryption/decryption scheme has three components, the key generation function G, the encryption function E, and the decryption function D. There are two types of encryption/decryption schemes: the public-key scheme (or the asymmetric-key scheme) and the private-key scheme (or the symmetric-key scheme). In a public-key scheme, when queried, the key generation function G generates a key pair (pk, sk), where pk is the public key, and sk is the private key. Any one knowing the public key can encrypt, but only the one who has the corresponding private key can decrypt. In a symmetric-key scheme, the encryption key and the decryption keys are the same and both are secret.

Researchers have studied different levels of security for encryption/decryption schemes. For a good introduction, one can see [33]. In the scope of security protocols that we consider, we require that the encryption/decryption schemes in

use satisfy the indistinguishability property against adaptive chosen-ciphertext attacks, also known as the IND-CCA2 security property.

The IND-CCA2 security of a encryption/decryption scheme is defined using a game that an adversary plays. Let A represent an arbitrary polynomial-time bounded adversary. We first consider the case of a public-key encryption/decryption scheme. To start the game, a public/private key pair (pk, sk) is generated by running the key generation procedure $\mathcal{K}$. Two messages of equal length m_0 and m_1 are chosen and given to A. A value $i \in \{0, 1\}$ is chosen uniformly and randomly. And based on the value i, A is given a challenge y, which is an encryption of message m_i, $y = \mathcal{E}_{pk}(m_i)$. A has access to the decryption procedure $\mathcal{D}_{sk}$ as a blackbox, called the *decrytpion oracle*. A can query the decryption oracle with any input as long as it is not equal to y. The goal of the game is that A tries to guess the bit i. When A stops, A outputs a bit b. A wins the game if $b = i$. We describe the formal definition of the IND-CCA2 security property below.

Definition 2.1.1. *A public-key scheme $(\mathcal{K}, \mathcal{E}, \mathcal{D})$ is (t, q, ϵ)-secure in the IND-CCA2 sense if for all pairs of different messages m_0 and m_1 of the same length, and for every adversary A that runs in time t, and makes at most q queries to the decryption oracle $\mathcal{D}_{sk}$,*

$$\Pr_{(pk,sk)\in\mathcal{K}}[A^{\mathcal{D}_{sk}}(pk, m_0, m_1, \mathcal{E}_{pk}(m_0)) = 1] - \Pr_{(pk,sk)\in\mathcal{K}}[A^{\mathcal{D}_{sk}}(pk, m_0, m_1, \mathcal{E}_{pk}(m_1)) = 1] \leq \epsilon,$$

where the adversary cannot query the decryption oracle with $\mathcal{E}_{pk}(m_i)$ for $i = 0, 1$.

The security property of symmetric-key schemes can be defined similarly.

We will also need a notion of integrity of a symmetric encryption scheme, *INT-CTXT* [4, 5]. We describe the formal definition below, following [5].

Definition 2.1.2. *Let $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be a symmetric encryption scheme. Let η be a security parameter of Π. Let $\mathcal{A}$ be an adversary with access to the encryption and decryption oracles. Consider the following experiements:*

Experiment $\mathrm{Exp}_{\Pi,\mathcal{A}}(\eta)$

$K \xleftarrow{R} \mathcal{K}(\eta)$

$C \xleftarrow{R} \mathcal{A}^{\mathcal{E}_K(\cdot),\mathcal{D}_K^{(}\cdot)}(\eta)$

If $\mathcal{D}_K(C) \neq \perp$, *and* C *was never a response of* $\mathcal{E}_K(\cdot)$

then return 1

else return 0.

We define the advantage functions of the scheme as follows:

$$\mathrm{Adv}_{\Pi}(\eta, t, q_e, q_d, \mu_e, \mu_d) = \max(\Pr[\mathrm{Exp}_{\Pi,\mathcal{A}}(\eta) = 1])$$

where the maximum is over all $\mathcal{A}$ with time-complexity t, each making to the oracle $\mathcal{E}_K(\cdot)$ at most q_e queries the sum of whose lengths is at most μ_s bits, and each making to the $\mathcal{D}_K^{(}\cdot)$ oracle at most q_d queries the sum of whose lengths is at most μ_d bits. The encryption scheme Π is said to be INT-CTXT secure if $\mathrm{Adv}_{\Pi,\mathcal{A}}$ is negligible for any adversary $\mathcal{A}$ whose time-complexity is polynomial in k.

2.2 Formal Methods

With the improvement of engineering and technology, we are building systems that are more complex than ever before. In many areas, these systems have grown so complex that it is extremely to understand the full behavior of the systems and to guarantee the correctness of the systems. In order to solve this problem, many researchers have investigated using mathematical models, abstractions and formal reasoning to analyze the behavior of these systems and to verify whether or not they are correct. Because manual proofs can be tedious and error-prone, a lot of effort has also been spent in automating these analysis methods.

Formal methods research mainly falls into two camps: theorem proving and model checking. In the theorem proving approach, the system being analyzed is described with a set of axioms. 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 prove the desired theorems. The logic and inference system can be as general as first order logic, or can be tailored to the specific problem domain as in the case of the BAN logic [13]. Many tools have been developed for automated deduction. For example, PVS [72, 52], Isabelle [73], and HOL [35] have all enjoyed widespread use.

In the model checking approach, an explicit model of the system is given, as opposed to theorem proving, where the system is described by a set of axioms. The model is often described as a Kripke structure, a finite state transition system

where the states are labeled with atomic propositions. The actual behavior of the system is given by all the possible executions or *traces* of the model. The desired properties are specified as logic formulas, and we need to verify that the given model satisfies the formulas. The model checking problem can then be reduced to state space exploration. One of the first efficient temporal logic model checking procedures can be found in [32]. A main difficulty in model checking is the *sate space explosion* problem. Roughly, this problem refers to the statement that the size of the model of a concurrent system grows exponentially with respect to the number of components in the model. Researchers have proposed several techniques to tackle this problem, including symbolic representation of the transition system and the states explored, symmetry reduction, and partial order reduction. For a good reference for model checking, see [22].

Theorem proving and model checking approaches both have advantages and disadvantages. Model checkers work with a concrete and finite model. Therefore they have the capability to provide *counter-examples*, concrete examples showing that the formula does not hold. On the other hand, the exhausitve state space exploration approach requires the models to be finite. Even though abstraction techniques exist to map many infinite models to finite models, in practice, this is still a limitation. Also the state space explosion problem limits the model checking approach to work on large systems. In contrast, theorem provers can handle infinite system easily. When a property is proven, it means that any system satisfying

the axioms (including an infinite system) also satisfies the specification. The disadvantage is that the procedure may not terminate and there is often little feedback when a proof fails. Also the proof process may be harder to automate and hence it requires more expertise to use theorem provers. Finally, the theorem proving approach usually cannot generate counter-examples.

Chapter 3

Automatic Protocol Verification

In this chapter, we describe the design and implementation of the automatic protocol verifier. We first introduce our formal model for specifying a security protocol in section 3.1, describe our representation of the intruder knowledge in Section 3.2 and our graph representation of protocol executions in Section 3.3. We then describe our logic to specify security properties of security protocols in Section 3.4. We explain our verification algorithm in Section 3.5 and give a concrete example in Section 3.5.7. Finally we descirbe our experiment results in Section 3.6 and discuss about related work in Section 3.7.

3.1 Specifying Security Protocols

3.1.1 Message Representation

We represent messages as symbolic expressions which we call *terms*. Terms are constructed from smaller sub-terms using concatenation and encryption. The smallest terms (i. e. , ones which contain no sub-terms) are called *atomic terms*. We devide the set of atomic terms into three disjoint sets: **Keys**, **IndData**, **Data**. Informally, the symbols in **Keys** represent cryptographic keys generated probabilistically using the key generation function. The symbols in **IndData** represent numbers generated randomly and independently from other atomic terms. They are often used as nonces or principals' protected secrets. The rest of the atomic symbols can be used to represent other data items such as principal names. The terms in **Data** are often public knowledge. The total set of terms, **Terms**, consists of atomic terms and terms constructed using operators Con and Enc:

$$\text{Term} := \text{IndData} \mid \text{Data} \mid \text{Keys} \mid Con(\text{Term}, \text{Term}) \mid Enc(\text{Term}, Keys).$$

Each term can be represented as an abstract syntax tree, with atomic terms as leaves and operators Con and Enc as intermediate nodes. Figure 3.1 shows an example for the message: $Enc(N_A, A, K_B) \equiv Enc(K_B, Con(N_A, A))$.

The notion of terms can also be generalized to **term templates**. A term template can be thought of as a term containing zero or more term variables. We can substitute a term variable with a term (or another term variable). We define the

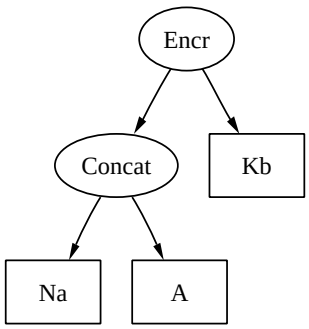


Figure 3.1: Example of a message tree for the message: $\{N_A, A\}_{K_B}$

substitution of term variables in the standard way as follows.

Definition 3.1.1. *A* substitution $\sigma = [\vec{t}/\vec{x}]$ *is a mapping of term variables* $\vec{x}$ *to term templates* $\vec{t}$*. The terms in* $\vec{t}$ *do not have to be closed and may contain free term variables that can later be replaced by other substitutions. We assume that the substitutions are capture-avoiding, i. e. , none of the variables in* $\vec{x}$ *occurs in* $\vec{t}$*.*

Applying a substitution σ *to a particular term template t' yields a new term template* $\sigma(t')$ *obtained from t' by simultaneously replacing all free term variables* $\vec{x}$ *in t' with term templates* $\sigma(\vec{x})$*.*

When the context has no ambiguity, we may use "term " also for "term template". And we use "closed terms" to refer to terms that do not contain any free term variables.

3.1.2 Protocol Specification

A protocol specifies sequences of actions of different parties which we call *roles*. For example, the Needham-Schroeder protocol consists of a client role and a server role. Each role in the protocol specification performs a number of steps. At each step, the role either constructs a message and sends it out, or receives a message. For simplicity, we denote the action of sending a message t as $+t$ and receiving a message t as $-t$. When the role receives a message, it first checks whether the received message is well-formed according to the protocol. If the received message is well-formed, then it saves the message, extracts information from the received message, and then goes to the next step, otherwise aborts the protocol execution. We call a sequence of actions of a role in the protocol specification a *role template*. Each role template has a list of parameters, which are immutable local variables[1]. And note that each role has its own name space for the local variables. A real protocol execution *instantiates* the different role templates by substituting the lists of parameters with concrete values. Some parameters are atomic terms and their values will be assigned by the party itself. For example, the parameter N_a in the Client role template and the parameter N_b in the Server role template in the Needham-Schroeder protocol are nonces generated by the Client and Server respecitively in the protocol execution. The other parameters are term variables whose value will be assigned according to certain values in the received messages.

[1] An immutable variable means that once the variable takes on a value, the value will not change later.

In a role template, each sent or received message is represented as a term constructed with the parameters of the role template. Message terms represent the way the messages are constructed in the sending actions or verified-and-extracted in the receiving actions. For example, the client's first step in the Needham-Schroeder protocol is represented as $+\{N_A, A\}_{K_B}$, meaning that the client encrypts the concatenation of A and N_A with B's public key and then sends out the message. The client's second step in the Needham-Schroeder protocol is represented as $\{N_A, N_B\}_{K_A}$, meaning that the client decrypts the received message using its private key and checks whether the computed N_A in the received message equals to its corresponding local value, N_A in its local parameter list. If this is the case, the client assigns the value of N_B in the received message to its local parameter N_B. The client's and server's three steps in the Needham-Schroeder protocol can be represented as follows:

$\underline{Server(A, B, N_a, N_b)}$	$\underline{Client(A, B, N_a, N_b)}$
$1 : \langle -\{N_a, A\}_{K_b} \rangle$	$1 : \langle +\{\overline{N_a}, A\}_{K_b} \rangle$
$2 : \langle +\{N_a, \overline{N_b}\}_{K_a} \rangle$	$2 : \langle -\{N_a, N_b\}_{K_a} \rangle$
$3 : \langle -\{N_b\}_{K_b} \rangle$	$3 : \langle +\{N_b\}_{K_b} \rangle$

The list (A, B, N_a, N_b) is the client and server's parameter list. Note that the client and the server have seperate name spaces. Note that $\overline{N_a}$ in the client's first

step represents that N_a's value is a freshly generated random number.

Definition 3.1.2. *We refer to each individual step in a role template as a position, denoted as* $[r, i]$*, where* r *represents the role template, and* i *is the index of the position in* r*. Also* $\mathsf{term}([r, i])$ *represents the term in* $[r, i]$*.*

We assume that each concatenation in the protocol implementation contains certain information to specify how a receiver should disassemble the components to prevent ambiguities. A simple way is to prepend to each component some type information such as the length of the component or a type ID. For example, if a legitimate server generates a message $\{N_a, N_b\}_{K_a}$ in the Needham-Schroeder protocol, then when client A decrypts the message, A will know that the decryption is a concatenation of two nonces and extract N_a and N_b correctly. For simplicity, we do not write out the type ID numbers explicitly in the terms.

3.2 The Intruder Knowledge Representation

Let Θ_0 denote the initial knowledge of the intruder before the protocol execution, including its initial public knowledge and its own secret keys. During the protocol execution, the attacker gains knowledge from the messages sent over the network by legitimate parties. Let $M(T)$ denote the set of messages sent by legitmate parties over the network by time T, and let $\Theta(T)$ denote the knowledge the

intruder possesses at time T during the protocol execution. The set $\Theta(T)$ is determined by Θ_0 and $M(T)$. Therefore, we can represent $\Theta(T)$ as a function of Θ_0 and $M(T)$: $\Theta(T) = \mathcal{H}(\Theta_0, M(T))$. Note that all terms we mention in this subsection are closed terms, containing no free term variables.

Let $\alpha(\Theta_0, M(T))$ represent the smallest set of terms that satisfies the following set of rules:

- $x \in \Theta_0 \implies x \in \alpha(\Theta_0, M(T))$;
- $x \in M(T) \implies x \in \alpha(\Theta_0, M(T))$;
- $\text{Con}(x, y) \in \alpha(\Theta_0, M(T)) \implies x \in \alpha(\Theta_0, M(T)) \wedge y \in \alpha(\Theta_0, M(T))$;
- $\{x\}_k \in \alpha(\Theta_0, M(T)) \wedge \hat{k} \in \alpha(\Theta_0, M(T)) \implies x \in \alpha(\Theta_0, M(T))$;

Definition 3.2.1. *We say a term $u \in \Theta(T)$, meaning u can be computed from $\Theta(T)$, if it can be derived from the following rules:*

- $\{x\}_k \in \Theta(T) \Leftrightarrow (\{x\}_k \in \alpha(\Theta_0, M(T)) \vee (x \in \Theta(T) \wedge k \in \Theta(T))$
- $Con(x, y) \in \Theta(T) \Leftrightarrow x \in \Theta(T) \wedge y \in \Theta(T)$.
- $x \in \alpha(\Theta_0, M(T)) \implies x \in \Theta(T)$.

3.3 Graph Model Representation of Protocol Executions

As we described in Chapter 2, a security protocol specifies the communication among the set of processes. A protocol execution is a sequence of state transitions of these processes, which is a product of the local state transitions of each individual process. This representation model is called the *trace model*. The trace model representation requires total ordering of messages. However, total ordering of messages is often unnecssary for checking security properties – many traces with some messages reordered may have the same evaluation result for some security properties. Therefore, the trace model representation causes unnecessary combinatorial search space explosion, and the problem is more severe as the number of participants in the protocol execution increases. We use a different representation, the *graph model* representation. The graph model representation does not require total ordering of messages and captures the exact causal relationship of the messages. The graph model representation enables us to analyze protocols more efficiently. Our graph model was inspired by the Strand Space Model, proposed by Thayer, Guttman, and Herzog in [91]. In this section, we describe our graph model representation for protocol executions.

Definition 3.3.1. *Let $+t$ represent the action of sending a message t, and $-t$ represent the action of receiving the message t. Let M denote the message space. A strand is a list of*

actions of a party, and $\mathrm{length}(s)$ *denotes the length of the list.*

A strand representing a legitimate party's actions is called a regular strand, and can be represented as $r[\sigma]$ *where* r *is the role of the party in the protocol execution and* σ *is a substitution list.*

A strand representing the attacker's actions is called an attacker strand or an intruder strand.

Definition 3.3.2. *A* node *is a pair* $\langle s, i \rangle$*, where* s *is a strand and* i *is an integer satisfying* $1 \leq i \leq \mathrm{length}(s)$*. We say that* $n = \langle s, i \rangle$ belongs *to the strand* s*, denoted by* $n \in s$*. Clearly, every node belongs to a unique strand. If a node belongs to a regular strand, then we call the node a regular node; similarly, if a node belongs to an attacker strand, we call the node an attacker node.*

Given a node $n = \langle s, i \rangle$*, we define* $\mathsf{index}(n) = i$ *and* $\mathsf{strand}(n) = s$*. If the* i*-th element in* s *is* $+t$ *(or* $-t$*), then we define* $\mathsf{term}(n) = t$ *and* $\mathsf{sign}(n) = +$ *(or* $-$*). In other words, a node is a particular action in a given strand.*

A strand s *is a substrand of* s' *(denoted as* $s \subseteq s'$*) iff* s' *contains all the nodes of* s *in the same order. Let* $r[\sigma]_j$ *represent the substrand of* $r[\sigma]$ *which only includes the nodes from number one to number* j *in strand* $r[\sigma]$*.*

For example, a strand representing a server role instantiated with parameters $[A_0, B_0, N_{a0}, N_{b0}]$ in the Needham-Schroeder protocol is the following :

$$\underline{Server[A_0, B_0, N_{a0}, N_{b0}]}$$

$$\langle -\{N_{a0}, A_0\}_{K_{b0}} \rangle$$
$$\Downarrow$$
$$\langle +\{N_{a0}, \overline{N_{b0}}\}_{K_{a0}} \rangle$$
$$\Downarrow$$
$$\langle -\{N_{b0}\}_{K_{b0}} \rangle$$

Definition 3.3.3. *A p-graph G is a triple $(\Sigma, \rightarrow, \Rightarrow)$, where Σ is a set of strands, $\rightarrow$ and $\Rightarrow$ are two different types of relations on pairs of nodes in Σ. Let n_1, n_2 be two nodes in a p-graph G.*

- *If $n_1 \rightarrow n_2$, then $n_1 = +t$ and $n_2 = -t$ for some term t. This represents sending a message t from n_1 to n_2.*
- *$n_1 \Rightarrow n_2$ iff n_1 and n_2 belong to the same strand and $\mathsf{index}(n_2) = \mathsf{index}(n_1) + 1$. This represents the action n_2 follows n_1.*

Definition 3.3.4. *In a p-graph G, we define the relation $\preceq$ to be the reflective and transitive closure of the relation $\rightarrow$ and relation $\Rightarrow$.*

In other words, the relation $\preceq$ is a reflexive and transitive closure of the relation $\rightarrow$ and $\Rightarrow$. Intuitively, the relation $\preceq$ expresses a *causal precedence*: if $n_1 \preceq n_2$, then the action of n_1 must have happened before n_2.

Definition 3.3.5. *Given a node n in p-graph G, let $M(n) = \{\mathsf{term}(n')|\mathsf{sign}(n') = +, n' \in G, \mathrm{and} n' \preceq n\}$. In other words, $M(n)$ denote the set of messages that were sent by legitimate parties before node n. Let $\Theta(n) = \mathcal{H}(\Theta_0, M(n))$, which denotes the attacker's knowledge at the time of execution of node n.*

Although the attacker can perform arbitrary actions, we only represent the sending actions of the attacker in a p-graph. Each sending action of the attacker is represented by a node, and labled as $+t$ where t is the corresponding sent term. The computations performed by the attacker are represented implicitly using the property *attacker-soundness*:

Definition 3.3.6. *We say a p-graph is attacker-sound iff for every attacker's sending-action node n, $\mathsf{term}(n) \in \Theta(n)$.*

Definition 3.3.7. *A p-graph C is a bundle iff C satisfies the following properties:*

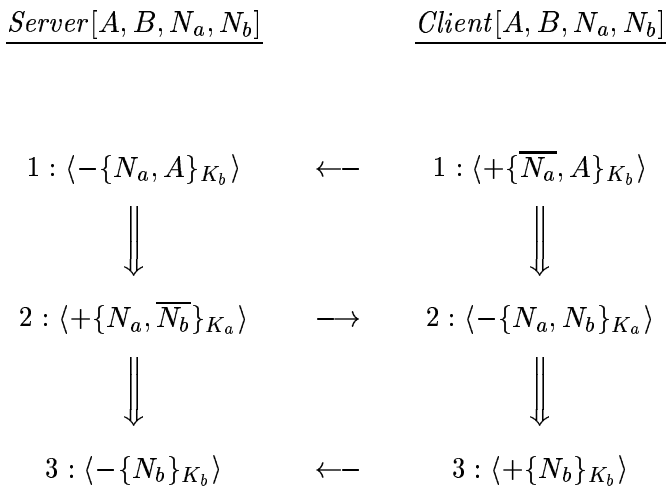
- *C is finite and non-empty;*
- *Backwards closed under "$\rightarrow$": If $n_1 \in C$ is a regular node and $\mathsf{sign}(n_1) = -$, then there is a unique node $n_2 \in C$ such that $n_2 \rightarrow n_1$ is in C;*
- *Backwards closed under "$\Rightarrow$": If $n_1 \in C$ is a regular node and $n_2 \Rightarrow n_1$, then there exists a node $n_2 \in C$ such that $n_2 \Rightarrow n_1$ is in C;*
- *Acyclic: The causal precedence relation $\preceq$ is anti-symmetric, meaning if $n_1 \preceq n_2$ and $n_2 \preceq n_1$ then $n_1 = n_2$;*

- *C is attacker-sound.*

From the definition we can see that the relation $\preceq$ in a bundle is a partial order:

Lemma 3.3.8. *In a bundle, the relation $\preceq$ is a partial order, i.e. a reflexive, antisymmetric, transitive relation. In addition, every non-empty subset of the nodes in a bundle has $\preceq$-minimal members.*

Intuitively, a bundle represents a set of protocol execution traces, and any protocol execution can be represented as a bundle. Below is an example of a bundle representing a concrete protocol run of the Needham-Schroeder protocol.



3.4 Security Property Specification

In this section, we introduce a logic to reason about various security properties of security protocols using our graph model representation. We show that by an-

alyzing statements in the graph model, we can validate properties of real protocol executions.

3.4.1 The Logic

The formulas in the logic state properties about bundles. Let C be a bundle, s be a strand, n_1 and n_2 be two nodes, the logic contains atomic formulas:

- $C \models s$ *iff* s is a subgraph of C.
- $C \models n_1 \preceq n_2$ *iff* $n_1 \preceq n_2$ in the graph C.
- $C \models t \in \Theta(n)$ *iff* $t \in \Theta(n)$ in bundle C. And we say $C \models t \in \Theta$ when n is implicit, meaning for some $n \in C$, $t \in \Theta(n)$ in bundle C.

We also include the standard conjunction and negation operators:

- $C \models f_1 \wedge f_2$ *iff* $C \models f_1$ and $C \models f_2$.
- $C \models \neg f$ *iff* it is not the case that $C \models f$.

Above are formulas about statements of a bundle. When we analyze properties of a protocol, we need to analyze statements of all bundles satisfying the protocol specification. Let C_P be a bundle variable representing a bundle satisfying protocol specification P, then $\forall C_P.\ f$ *iff* $C \models f$ holds for all bundle C satisfying the protocol specification P. Because any real complete protocol execution can be represented as a bundle, $\forall C_P.\ f$ *iff* f holds for all protocol execution satisfying the protocol specification.

3.4.2 Specifying Security Properties in the Logic

Our logic can express many safety properties. In this section, we give two examples: how to use our logic to specify authentication and secrecy properties in the Needham-Schroeder protocol.

Authentication

For example, in the Needham-Schroeder protocol, the property "the server authenticates the client" means that whenever a server s finishes its protocol steps supposedly with a client c and with nonce values N_a and N_b, then there must be a corresponding client c in the protocol run supposedly with s and with nonce values N_a and N_b. This property can be expressed by the following formula:

$$\forall C_P.C_P \models (\text{server}[s, c, N_a, N_b] \implies \text{client}[s, c, N_a, N_b]).$$

Note that the formula is implicitly universally quantified over all s, c, N_a, N_b. This implicit universal quantification holds throughout the document.

A more strict definition of the authentication property also requires that the order of the actions match. For example, a more strict specification of the property "the server authenticates the client" in the Needham-Schroeder protocol is as follows:

$$\begin{aligned}\forall C_P.C_P \models \quad & (\text{server}[s, c, N_a, N_b] \implies \text{client}[s, c, N_a, N_b]) \\ & \wedge(\text{client}[s, c, N_a, N_b][1] \preceq \text{server}[s, c, N_a, N_b][1]) \\ & \wedge(\text{server}[s, c, N_a, N_b][2] \preceq \text{client}[s, c, N_a, N_b][2]) \\ & \wedge(\text{client}[s, c, N_a, N_b][3] \preceq \text{server}[s, c, N_a, N_b][3]).\end{aligned}$$

In addition to the authentication property, there is often also a requirement of the *uniqueness* property. For example, in the Needham-Schroeder protocol, the property "the server uniquely authenticates the client" means whenever a server s finishes its protocol steps supposedly with a client c and with nonce values N_a and N_b, then there must be a *unique* corresponding client c in the protocol run supposedly with s and with nonce values N_a and N_b. Because of the freshness of the nonces generated in the protocol execution, usually the uniqueness property can be proven after the authentication property is proven, with the argument that there cannot be two strands of the form client$[s, c, N_a, N_b] \in C$, since the nonces in client$[s, c, N_a, N_b] \in C$ are uniquely generated by only one strand.

Secrecy

We say a value v in a protocol run is secret to the attacker if $v \notin \Theta$. For example, the property that the nonce N_b generated by a legitimate server s stays as a secret to the intruder in the Needham-Schroeder protocol is expressed by the following formula:

$$\forall C_P.C_P \models (\text{server}[s, c, N_a, N_b] \implies N_b \notin \Theta).$$

3.5 The Verification Algorithm

In this section, we describe the verification algorithm for checking the validity of formulas in our logic. Consider a formula $\forall C_P.\ C_P \models f$. Since any propositional formula f can be put in a *conjunctive normal form* (CNF), we assume that f has the form

$$f \equiv \bigwedge_i (\bigvee_j \neg\gamma_{ij} \vee \bigvee_k \delta_{ik}),$$

where γ_{ij} and δ_{ik} are atomic formulas. Therefore, we only need to discuss the verification procedure for a formula of the form

$$\forall C_P.\ C_P \models (\Gamma \implies \Delta), \tag{3.5.1}$$

where $\Gamma = \bigwedge_j \gamma_j$ and $\Delta = \bigvee_k \delta_k$, and γ_j and $delta_k$ are atomic formulas. If Γ is empty, then $\forall C_P.\ C_P \models \Delta$, and the entire formula is trivially false, because clearly Δ is not in the empty bundle. Hence, we can assume that the antecedent of the implication is never empty. If Δ is empty, then our formula becomes $\forall C_P.\ C_P \models (\bigwedge \Gamma \implies \mathsf{false})$, which is a particular case of Formula 3.5.1 and is covered by our decision procedure. For simplicity, we first only consider the case where Δ contains a single atomic formula denoted as δ, and the atomic formulas in Γ and δ are all of the first type in the definition of the logc, whether a bundle contains a strand. Other cases where Γ and δ contain other types of atomic formulas are simple extentions to the first case.

For simple notation, we use a sequent form to represent formulas. For example,

$\forall C_P.\ C_P \models (\Gamma \implies \delta)$ can be represented as $P;\ \Gamma \vdash \delta$.

3.5.1 Overview

In order to prove the sequent $P;\ \Gamma \vdash \delta$, we need to show that any bundle containing Γ and satisfying protocol specification P must contain strand δ. Because the set of bundles that contain Γ and satisfy protocol P may be infinite, we cannot simply check the validity of the formula by enumerating this set of bundles. We propose a new method to analyze this set of infinite bundles efficiently.

From a high level, our analysis starts with a p-graph containing only Γ and infers step by step what properties a bundle must have if it contains Γ. Recall that a bundle is a p-graph that is attacker-sound and backwards closed under "$\Rightarrow$" and "$\rightarrow$". Assume a p-graph G is contained in a bunlde c. For any regular node $n = \langle s, i\rangle \in G$ where $i > 0$, $n' = \langle s, i-1\rangle$ must be in c. Thus, we can make a p-graph backwards closed under "$\Rightarrow$" by adding the relevant nodes. We call a p-graph that is backwards closed under "$\Rightarrow$" a *semi-bundle*.

We call a received message of a regular node a *regular-goal*. For every regular goal in a bundle c, there must be a node in c that sends out the goal message. Moreover, due to the minimal-membership argument (Lemma 3.3.8), there must be a node in c that is the earliest node that sends out the goal message, called the *first-sender* of the goal message. The relation of a regular-goal and its first sender is called a *regular-goal binding*.

Similarly, we call the knowledge constraints of the attacker posed by the attacker-soundness property *attacker-goals*. For every attacker goal message in a bundle c, the attacker must have learned the message at some earlier point. The relation representing how the attacker learns the attacker-goal message for the first time is called *attacker-goal binding*.

In our analysis, we use a special compact representation called *states* to represent sets of bundles. A state may represent an infinite set of bundles. We show that by analyzing properties of a finite number of states, we analyze properties of the corresponding sets of infinite number of bundles.

A state is a semi-bundle including its regular-goal binding and attacker-goal binding relations. If every regular-goal in a state S has its first-sender node in S, and the state is attacker-sound, then the state can be converted to a bundle containing exactly the same set of legitimate strands. Given a state, we apply a *next state function* that returns a set of states. In particular, the next state function selects an unbound regular-goal or attacker-goal in the given state and enumerates all possible ways that the goal can be bound and outputs the set of next states.

The analysis starts with an initial state containing the smallest semi-bundle that contains Γ. Then at each step, the analysis applies the next state function to a state in the current set of states. Thus the analysis procedure forms a proof tree. If at a certain stage, every regular-goal in a state has its first-sender in the state, and the state is attacker-sound and does not contain strand δ, then the formula is false and

the corresponding bundle is a counterexample. If at a certain stage, all the current states in the proof tree contain strand δ, then the formula is true. We explain the details in the rest of this section.

3.5.2 Compact State Representation

We define two relations over terms: 1) *subterm relation* $\sqsubseteq$: a term a_1 is a *subterm* of term a_2 if a_1 appears as a subtree of a_2; 2) *interm relation* $\Subset$: a term a_1 is an *interm* of a_2 if a_2 is a concatenation of terms including a_1. The formal definition of the two relations is as follows.

- *subterm relation* $\sqsubseteq$ is the smallest relation such that:
  - $a_1 = a_2 \implies a_1 \sqsubseteq a_2$;
  - $a \sqsubseteq g \implies a \sqsubseteq Enc(g, k)$;
  - $a \sqsubseteq g \vee a \sqsubseteq h \implies a \sqsubseteq Con(g, h)$.
- *interm relation* $\Subset$ is the smallest relation such that:
  - $a_1 = a_2 \implies a_1 \Subset a_2$;
  - $a \Subset g \vee a \Subset h \implies a \Subset Con(g, h)$.

Definition 3.5.1. *A semi-bundle is a p-graph that is backwards closed under "$\Rightarrow$".*

It is clear that a semi-bundle which is also attacker-sound and backwards closed under "$\rightarrow$" becomes a bundle.

Definition 3.5.2. *A regular-goal is a pair (t, n), where n is a regular receiving node, $t \Subset \mathsf{term}(n)$, and t is not a concatenation of terms.*

Definition 3.5.3. *An attacker-goal is a pair (t, n), where n is an attacker sending node, $t \Subset \mathsf{term}(n)$, and t is not a concatenation of terms.*

Regular-goal Binding. Given a bundle C, for every regular-goal term, there must be a node sending out the message. Given a regular-goal (t, n) , consider the set of nodes that send out t in the bundle C, $\psi = \{m \in C \mid t \Subset \mathsf{term}(m)$, m is a sending node$\}$. Since ψ is non-empty, we know that there must be at least one minimal-member of ψ, denoted as n'. We call n' the *first-sender* of term t, and the relationship *regular-goal binding*, denoted as $n' \overset{t}{\twoheadrightarrow} n$. When t is clear from the context, we write $n' \twoheadrightarrow n$ to denote $n' \overset{t}{\twoheadrightarrow} n$ for simplicity.

Attacker-goal Binding. Given a bundle C, an attacker goal (t, n) means that $t \in \Theta(n)$. The intruder could have constructed t from other terms that it knows, or extracted t from a bigger term sent out by a regular node. In the latter case, because the attacker's knowledge is monotonically increasing as the protocol execution continues, there must be a regular node n' from which the attacker first learned term t, i.e. $t \sqsubseteq \mathsf{term}(n')$, and $t \notin \Theta(n')$, and the attacker knows the necessary information to extract t from node n' before node n. We call n' the *first informer* of term t, and the relationship *attacker-goal binding*, denoted as $n' \overset{t}{\hookrightarrow} n$. When t is clear from the context, we write $n' \hookrightarrow n$ to denote $n' \overset{t}{\hookrightarrow} n$ for simplicity. We denote

by $\xi(n' \overset{t}{\hookrightarrow} n)$ the set of terms the intruder needs to know in order to extract t from node n', so we need $\xi(n' \overset{t}{\hookrightarrow} n) \in \Theta(n)$ for the attacker-soundness as well.

Definition 3.5.4. *A state is a tuple $\langle S, \pi, G\rangle$, where*

- *S is a semi-bundle;*
- *π is the relation for the regular-goal bindings and attacker-goal bindings;*
- *G is the set of regular-goals and attacker-goals that are not yet bound using the regular-goal bindings.*

Note that G can be computed from S and π. We keep it in the state structure for the convenience of illustration.

We say that a strand s is in a state $l = \langle S, \pi, G\rangle$, denoted as $s \in l$, whenever $s \in S$. We also extend the notion of substitution from Section 3.1.1 to states: For a substitution σ, define $\sigma(\langle S, \pi, G\rangle) = \langle \sigma(S), \sigma(\pi), \sigma(G)\rangle$.

Definition 3.5.5. *Given a state $l = \langle S, \pi, G\rangle$, a set of bundles is called the* bundle set *of l, denoted as $\Psi(l)$, if it satisfies the following property. A bundle $c \in \Psi(l)$ iff*

1. *S is a subgraph of c;*
2. *The relation π in l is* consistent *with the relation $\rightarrow$ and $\Rightarrow$ in c; that is, for every $n' \overset{t}{\twoheadrightarrow}_l n$ in S there is a path $n' = n_1, n_2, \ldots, n_k = n$ in c such that*
   (a) *n' sends a message m where $t \Subset m$, and*

(b) There is no n'' in c such that $t \Subset \mathsf{term}(n'')$ and $n'' \preceq n'$.

The consistency with $\hookrightarrow$ relationship is defined similarly.

Proposition 3.5.6. *If G is empty in a state $l = \langle S, \pi, G \rangle$, then there exists a bundle C such that $C \in \Psi(l)$, and C contain exactly the same set of legitimate strands as in S.*

Proof sketch. Let h be the graph which contains S and the edges defined by the relation π. For each $n_1 \twoheadrightarrow n_2$ in h, we can add new intruder sending nodes and the corresponding "$\rightarrow$" edges, such that we can find a path from n_1 to n_2 connected by a sequence of edges of "$\rightarrow$" and "$\Rightarrow$". We then eliminate the edge $n_1 \twoheadrightarrow n_2$. After we remove all the "$\twoheadrightarrow$" and "$\hookrightarrow$" edges in this way, we get a graph h' which only contains edges of"$\rightarrow$" and "$\Rightarrow$". It is easy to see that h' is a bundle. Since we only add attacker sending nodes in the transformation, h' and S have the same regular strands. □

3.5.3 The Proof System

To simplify notation, we define a new sequent $l \vdash_P \delta$, representing the formula $\forall C \in \Psi(l).\ (C \models \delta)$, and we omit the subscript P when P is clear from the context. To analyze a formula $P; \Gamma \vdash \delta$, the analysis first computes an *initial state* $l_0(P, \Gamma)$, such that $\Psi(l_0)$ represents the set of all bundles that satisfy the protocol P and contain Γ (rule init). In order to prove the formula $P; \Gamma \vdash \delta$, it is sufficient to prove that $l_0(P, \Gamma) \vdash_P \delta$. A sequent $l \vdash_P \delta$ can either be solved by a decision procedure

directly if it applies (rule final), or split into multiple sequents $l_1 \vdash_P \delta, \ldots, l_n \vdash_P \delta$ which are then proven separately (rule split). Thus, the overall verification procedure is a proof search with these three rules, refered to as *inference rules*.

In the rest of this section, we explain the three inference rules and the proof procedure in detail. We also show that the proof procedure is sound and can generate couterexamples by showing that each inference rule is *sound* and *invertible*.

Soundness and Invertibility. An inference rule is *sound* if whenever the premises are provable then the conclusion is provable. An inference rule is *invertible* if whenever the conclusion is provable, all of the premises are provable; in other words, whenever one of the premisses of a rule is proven false, its conclusion is also false. The invertibility property is useful for disproving a sequent and providing a *counterexample* — a successful attack on the protocol. It also allows us to search for a proof without backtracking. In the following, we prove that each inference rule is sound and invertible.

Init rule. To analyze the sequent $P;\ \Gamma \vdash \delta$, the `init` rule converts the sequent to another form:

$$\frac{l_0(P,\ \Gamma) \vdash \delta}{P;\ \Gamma \vdash \delta}\ \mathsf{init}, \tag{3.5.2}$$

where $l_0(P,\ \Gamma)$ is the initial state computed as follows. Let S_Γ be the smallest semi-bundle that contains Γ; that is, S_Γ is the backward closure of Γ over the $\Rightarrow$ relation. Then the initial state is $l_0(P,\ \Gamma) = \langle S_\Gamma,\ \emptyset, G\rangle$, where G is the set of unbound regular-

goals and attacker-goals computed from S_Γ.

By construction, the initial state $l_0 = l_0(P, \Gamma)$ satisfies the following properties:

$$\Gamma \subseteq l_0$$

$$\forall C.\, \Gamma \subseteq C \implies C \in \Psi(l_0).$$

The above property shows that the init rule is sound and invertible.

Final rule. A sequent $l \vdash \delta$ is called a *leaf sequent* if $\delta \in l$. When $l \vdash \delta$ is a leaf sequent, we apply the final inference rule:

$$\frac{}{l \vdash \delta}\ \text{final, if} \delta \in l \tag{3.5.3}$$

The final rule is sound because $\delta \in l$ implies that δ is in C for any bundle $C \in \Psi(l)$.

If the final rule does not apply and $G = \emptyset$, then the current sequent $l \vdash \delta$ is considered proven false, or *refuted*. This follows directly from the Proposition 3.5.6. Recall that δ is a regular strand. By Proposition 3.5.6 there exists a bundle $C \in \Psi(l)$ such that for any regular strand s, $s \in S_l$ if and only if $s \in C$. Since $\delta \notin S_l$, we conclude that δ cannot be in C either, which means $\exists C \in \Psi(l).\delta \notin C$. Therefore, $l \vdash \delta$ is false. Since all of our inference rules are invertible, this means that the original sequent is also false, and the refuted sequent represents a *counterexample*, or a *successful attack* on the protocol.

Split rule. When the final rule does not apply and $G \neq \emptyset$, the split rule applies:

$$\frac{l_1 \vdash \delta \quad \cdots \quad l_n \vdash \delta, \quad \text{where } \{l_1, \ldots, l_n\} = \mathcal{F}(l)}{l \vdash \delta}\ \text{split.} \tag{3.5.4}$$

This rule splits the state l in the current sequent into several *next states* $l_1, \ldots, l_n$ using the *next state function* $\mathcal{F}$. This yields a set of new sequents which are then proven separately. The next state function $\mathcal{F}(l)$ is explained in more detail later in Section 3.5.5. Note that $\mathcal{F}(l)$ can be empty; in this case the split rule successfully finishes the proof of the sequent. When $\mathcal{F}(l) = \emptyset$, the state l contains a contradiction, so its bundle set is empty, and, therefore, the sequent is vacuously true. We will return to this special case in Section 3.5.6 when we discuss some optimizations to the proof search algorithm.

Definition 3.5.7. *We say that $\mathcal{F}$ is* complete-inclusive *if for any state l it satisfies the following properties:*

1. *$\mathcal{F}(l)$ is finite,*
2. *$\Psi(L') = \Psi(l)$, where $L' = \mathcal{F}(l)$ and $\Psi(L') = \bigcup_{l' \in L'} \Psi(l')$.*

Lemma 3.5.8. *The* split *rule is sound and invertible when $\mathcal{F}$ is* complete-inclusive.

Proof. By definition, the split rule is sound when the function $\mathcal{F}$ is complete-inclusive. We now show that the split rule is also *invertible* when $\mathcal{F}$ is complete-inclusive. That is, if any of the assumption sequents $l_i \vdash \delta$ is proven false, then the conclusion $l \vdash \delta$ is also false. A sequent $l_i \vdash \delta$ is false implies that there exists a bundle C from $\Psi(l_i)$ which does not contain δ. Since $\Psi(l_i) \subseteq \Psi(l)$, then $C \in \Psi(l)$, and therefore, $l \vdash \delta$ is also false. □

The Proof Search Procedure

To prove a sequent $P; \Gamma \vdash \delta$, the proof procedure first computes l_0 and applies the init rule to convert the sequent to $l_0 \vdash \delta$. Then the proof procedure iteratively applies the following procedure to the current sequent. First, if the final rule applies, the current sequent is proven. Otherwise, if the set of unbound goals in the current sequent is empty, then the current sequent is proven false. Since all the rules in our proof system are invertible, the original sequent is also false, and the proof search terminates with a failure, returning the current sequent as a counterexample. If the final rule does not apply, and the set of unbound goals in the sequent is not empty, then the split rule applies which may generate a set of new sequents. The proof procedure repeats for each of the new sequents. Thus, the proof search procedure effectively constructs a *proof DAG*.

Theorem 3.5.9. *Let $P; \Gamma \vdash \delta$ be an initial sequent, and $\mathcal{F}$ be a complete-inclusive next state function. If the proof search procedure described in this section terminates with a complete proof, then the initial sequent is true (soundness); if it refutes a sequent anywhere in the proof, then the initial sequent is false (invertibility).*

The theorem holds from the fact that each inference rule is sound and invertible as shown before.

3.5.4 Unification

Given a state $l = \langle S, \pi, G \rangle$, the next state function first selects an unbound goal g from the set of unbound regular-goals and attacker-goals in state l using a priority ranking function. If g is a regular goal, the next-state function will enumerate all possible ways that the goal message could have been sent out for the first time; if g is an attacker goal, the next-state function will enumerate all possible ways that the attacker could have learned the goal message for the first time. The enumeration of all possible ways for binding a goal is done using a *unification* procedure. In particular, for a regular goal term m, we use the *interm-unification* procedure to check whether another regular node n could have sent out the term m; and for an attacker goal term m, we use the *subterm-unification* procedure to check whether the attacker could have learned m from another regular node n. We introduce the interm-unification and subterm-unification procedure below.

Definition 3.5.10. *A substitution σ is called a* unifier *of term t and t' if $\sigma(t) = \sigma(t')$.*

A substitution σ is a most general unifier *(MGU) of t and t' if it is a unifier, and for any other unifier σ' there exists a substitution γ such that $\sigma' = \sigma \circ \gamma$.*

Definition 3.5.11. *A substitution σ is called a* interm-unifier *of a term t and a t' if $\sigma(t) \Subset \sigma(t')$.*

A set of substitutions Υ is a most general interm-unifier *(MGIU) of t and t' if every substitution $\sigma \in \Upsilon$ is a interm-unifier, and for any other interm-unifier $\sigma' \notin \Upsilon$ there exists*

a substitution γ and $\sigma \in \Upsilon$ such that $\sigma' = \sigma \circ \gamma$.

Definition 3.5.12. *A substitution σ is called a* subterm-unifier *of a term t and a t' if $\sigma(t) \sqsubseteq \sigma(t')$. And $\Omega(t, t', \sigma)$ represents the set of terms needed to extract t from t', i.e. the set of keys needed to extract t from t'.*

A set of substitutions Υ is a most general subterm-unifier *(MGSU) of t and t' if every substitution $\sigma \in \Upsilon$ is a subterm-unifier, and for any other subterm-unifier $\sigma' \notin \Upsilon$ there exists a substitution γ and $\sigma \in \Upsilon$ such that $\sigma' = \sigma \circ \gamma$.*

Definition 3.5.13. *For a protocol P and a term t we define a set of* position interm-unifiers $U_P(t)$ *and a set of* position subterm-unifiers $U_P^c(t)$ *as follows:*

$$U_P(t) = \left\{ \; ([r,i],\, \sigma) \;\middle|\; \sigma \text{ is a MGIU of } t \text{ and } \mathsf{term}([r,i]) \text{ for some } [r,i] \in P \; \right\}.$$

$$U_P^c(t) = \left\{ \; ([r,i],\, \sigma, \Omega) \;\middle|\; \begin{array}{l} \sigma \text{ is a MGSU of } t \text{ and } \mathsf{term}([r,i]) \text{ for some } [r,i] \in P \text{ and} \\ \Omega = \Omega(t, \mathsf{term}([r,i]), \sigma) \end{array} \right\}.$$

3.5.5 The Next-State Function

Given a state $l = \langle S_l, \pi_l, G_l \rangle$, the next state function $\mathcal{F}$ first selects a goal $g \in G_l$ according to a priority ranking function and then binds the goal in all possible ways. Intuitively, the priority ranking function gives goal terms that are more constrained higher priority. If a goal term is not constrained enough, the possible ways to bind it may be too many or even infinite. For example, if a goal term is simply a variable with no constraint, then it could have been bound to any sent

message. On the other hand, if the goal term is sufficiently constrained, then there may be fewer ways to bind the goal term and hence yeild a smaller set of next states. In particular, let $x \in \mathrm{IndData}$ be a new atomic term created by a regular node such as a nonce, and $x \notin \Theta(l)$. Let $g = (t, n)$ be a goal in l and $x \sqsubseteq t$, then t has high priority in the goal ordering. We show that we can enumerate all possible ways of binding goal g. Let $\mathcal{F}_1$ denote the procedure of goal binding when g is a regular goal, and $\mathcal{F}_2$ denote the procedure of goal binding when g is an attacker goal. We describe $\mathcal{F}_1$ and $\mathcal{F}_2$ separately in this subsection.

Computing $\mathcal{F}_1(l)$

There are two possibilities to bind a regular goal term t: either to a regular node or to an attacker node. To check all possible ways that the goal term could have been sent out on a regular node, the analysis procedure first interm-unifies the goal term with each position specified by the protocol to see whether the goal term could have been sent out from that position. The procedure then considers all the possibilities that the position could be instantiated as a node in the state: a node of the position either could already be one of the strands in the state or on a new strand added to the state. In the case that the goal term is bound to an attacker node, the procedure simply adds an attacker node tagging as the first sender of the goal term and then adds to the goal list that the attacker needs to know the goal term before sending it out. We describe the details of the algorithm below.

We compute the set $U_P(t)$ for the goal term t. For each element $u = ([r, i],\ \gamma) \in U_P(t)$ we construct the set of next states L'_u as follows. Let $s_u = r[\gamma]_i$.

- For every strand $s \in S_l$ such that $s = r[\gamma']_j$, if there exists a substitution σ such that $\sigma(\gamma') = \sigma(\gamma)$, let $s' = r[\sigma']_k$ where $k = \max i, j$ and $\sigma' = \sigma \circ \gamma$. We construct a new state $l' = \langle S',\ \pi', G' \rangle$ and include it into L'_u, where

  1. $S' = \sigma'(S \setminus s) \cup s'\}$;
  2. $\pi' = \sigma'(\pi) \cup \{\langle s',\ i\rangle \overset{\sigma'(t)}{\twoheadrightarrow} \sigma'(g)\}))$

     (the goal g is bound to the i-th node in the strand s')
  3. Update G to include new regular-goals introduced by the added nodes in strand s'.

- We also construct an additional new state $l'' = \langle S'',\ \pi'', G'' \rangle$ by adding s_u as a new strand:

  1. $S'' = \gamma(S) \cup s_u$;
  2. $\pi'' = \gamma(\pi) \cup \{\langle s_u,\ i\rangle \overset{\gamma(t)}{\twoheadrightarrow} \gamma(g)\})$

     (the goal g is bound to the i-th node of s_u)
  3. Update G to include new regular-goals introduced by the added nodes in strand s'.

Finally we construct the next state for the case that the goal is bound to an attacker node. We simply compute the next state $l''' = \langle S''',\ \pi''', G''' \rangle$ as follows:

1. $S''' = S \cup n'$, where $n' =< +t >$;

2. $\pi''' = \pi \cup \{n' \overset{t}{\twoheadrightarrow} g\}\}$;

   (the goal g is bound to the node n')

3. Update G to include the attacker-goal (t, n').

Finally, we have $\mathcal{F}_1(l) = (\cup_{u \in U_P(t)} L'_u) \cup l'' \cup l'''$.

Computing $\mathcal{F}_2(l)$

In order to satisfy $t \in \Theta(n)$, either the attacker has learned term t by extracting the term t from a bigger term sent by a regular node, or the attacker has constructed the term t itself.

We first describe the case where the attacker has learned the goal term t from a bigger term sent by a regular node. We compute the set $U^c_P(t)$ for the term t. For each element $u = ([r, i], \gamma, \Omega) \in U^c_P(t)$, we construct the set of next states L'_u as follows. Let $s_u = r[\gamma]_i$.

- For every strand $s \in S_l$ such that $s = r[\gamma']_j$, if there exists a substitution σ such that $\sigma(\gamma') = \sigma(\gamma)$, let $s' = r[\sigma']_k$ where $k = \max i, j$ and $\sigma' = \sigma \circ \gamma$. We construct a new state $l' = \langle S', \pi', G' \rangle$ and include it into L'_u, where

  1. $S' = \sigma'(S \setminus s) \cup s'\}$;

  2. $\pi' = \sigma'(\pi) \cup \{\langle s', i \rangle \overset{\sigma'(t)}{\hookrightarrow} \sigma'(g)\}))$

     (the goal g is bound to the i-th node in the strand s')

3. Update G to include new regular-goals introduced by the added nodes in strand s'.

4. For each $\omega \in \Omega$, we add attacker node $n_\omega = \langle +\omega \rangle$ to S', and add relation $n_\omega \Rightarrow n$ to π'.

- We also construct an additional new state $l'' = \langle S'',\ \pi'', G'' \rangle$ by adding s_u as a new strand:

  1. $S'' = \gamma(S) \cup s_u$;

  2. $\pi'' = \gamma(\pi) \cup \{\langle s_u,\ i \rangle \overset{\gamma(t)}{\hookrightarrow} \gamma(g)\})$

     (the goal g is bound to the i-th node of s_u)

  3. Update G to include new regular-goals introduced by the added nodes in strand s'.

  4. For each $\omega \in \Omega$, we add attacker node $n_\omega = \langle +\omega \rangle$ to S'', and add relation $n_\omega \Rightarrow n$ to π''.

Finally we compute the case where the attacker has constructed the goal term itself. This case occrs when the goal term t is of the form $\{m\}_k$. To calculate the next state l'''', we add attacker node $n' = \langle +m \rangle$ and $n'' = \langle +k \rangle$ to S and relation $n' \Rightarrow n$ and $n'' \Rightarrow n$ to π and update G to include new attacker-goals. The result is the next state l'''.

Finally $\mathcal{F}_2(l) = (\cup_{u \in U_P^c(t)} L'_u) \cup l'''$.

The states in $\mathcal{F}_1(l)$ and $\mathcal{F}_2(l)$ will then be checked with a validity procedure as we describe in the next section.

3.5.6 Pruning Invalid States

Each state in the output of $\mathcal{F}_1(l)$ and $\mathcal{F}_2(l)$ is analyzed for *validity*. The final result of $\mathcal{F}(l)$ is the set of all valid next states in $\mathcal{F}_1(l)$ and $\mathcal{F}_2(l)$.

Definition 3.5.14. *A state l is invalid if $\Psi(l) = \emptyset$. Intuitively, an invalid state is a state which is inconsistent such that it cannot be extended to a real protocol execution.*

The analysis algorithm automatically marks a state invalid if one of the following conditions holds:

- The $\preceq$-minimal relationship is not satisfied. For example, there exists a sending node n_1 where t binds to, and another node n_2, where $t \Subset \mathsf{term}(n_2)$ and $n_2 \preceq n_1$.
- There is a cycle in the resulting state over the relationship π';
- The "nonce" property is not satisfied. For example, when a nonce N_a is generated on a legitimate node n_1, and if $N_a \sqsubseteq \mathsf{term}(n_2)$ and $n_2 \preceq n_1$ in l, then the state is considered invalid.

If every state in $\mathcal{F}(l)$ is invalid, then $\mathcal{F}(l) = \emptyset$ and the split rule takes its extreme form and completes the proof of the sequent:

$$\frac{\mathcal{F}(l) = \emptyset}{l \vdash \delta}\ \mathsf{split}.$$

The analysis algorithm can also systematically incorporate results from theorem proving by using *pruning theorems*. These pruning theorems can prove that a state is invalid, thus, "prune" the search space at an early stage. In our framework, pruning theorems are expressed as computable predicates, denoted as τ: if $\tau(l)$ is true, then the state l is invalid. This allows us to introduce a new inference rule:

$$\frac{}{l \vdash \delta}\ \mathsf{prune}, \text{ if } \tau(l) \text{ is true.} \tag{3.5.5}$$

The pruning theorems can be specific to a particular protocol, or can be general theorems. The latter can be proven once and for all and included in the core of the tool. The tool APV also provides an easy interface for the expert users to add their own protocol-specific pruning theorems. Although in general, users do not need to input their own pruning theorems, we provide this easy interface to achieve extensibility of the tool, which enables APV to be used for more complicated protocols and reduce the search space even further. So far, APV has three built-in pruning theorems, described below as Proposition 3.5.15, 3.5.17 and 3.5.18. These three theorems are shown to be effective and sufficient in our experiments.

Proposition 3.5.15. *Let k be a secret key of a legitimate party and $k \notin \Theta_0$. If $k \not\sqsubseteq \mathsf{term}(n)$ for any position n in the protocol specification, then $k \notin \Theta$. Therefore, if k occurs in an attacker sending node in a state l, then l is invalid.*

Definition 3.5.16. *Let t be a received term on node n in a bundle C. Consider the set of nodes $\phi = \{m \in C \mid t \sqsubseteq \mathsf{term}(m), \mathsf{sign}(m) = +\}$. Because ϕ is non-empty, there must be*

at least a minimal-member, denoted as n'. We call n' the creator of term t, and the relation creator-binding, denoted as $n' \overset{t}{\mapsto} n$. When t is clear from the context, we omit t and write $n' \mapsto n$ to denote $n' \overset{t}{\mapsto} n$ for simplicity.

Note that the creator and the first-sender of a goal term may be the same node.

Proposition 3.5.17. *Assume an attacker-goal binding relation $n_1 \overset{t}{\hookrightarrow} n_2$ in a state l, where n_2 is a regular node and $n_3 \Rightarrow n_2$. If there exists a term $t' \sqsubseteq \mathsf{term}(n_3)$ that is not a concatenation form and $t \sqsubseteq t'$, then the creator of term t' must be a regular node. If there cannot be a creator of term t' from a regular node, then state l is invalid.*

Proposition 3.5.18. *Assume x is a nonce first created in node n', and $x \Subset t'$, $\{t'\}_{k_2} \sqsubseteq \mathsf{term}(n')$. Let m be a goal term on node n, where $x \Subset t$ and $\{t\}_{k_1} \sqsubseteq m$. If $\overline{k_2} \notin \Theta$, then there must be a regular node that decrypts $\{t\}_{k_2}$.*

This proposition is an extension of the authentication test proposed in [36].

Since we only add (and never delete) nodes and strands to the next states, and since the next state transition covers all possibilities of binding a goal, and L' is finite, we can see that $\mathcal{F}$ is *complete-inclusive*.

Notice that due to the most general substitution in goal binding, the nodes and strands in a state may contain free variables. This often allows us to prove a sequent with the minimal possible instantiation of the terms in each state and greatly helps to reduce the search space.

3.5.7 Example: Needham-Schroeder Protocol Analysis

In this section, we illustrate our verification algorithm with a concrete example, analyzing the authentiaction property in the Needham-Schroeder protocol:

$$\forall C_P.C_P \models (\text{server}[C^\circ, S^\circ, N^\circ, M^\circ] \implies \text{client}[C^\circ, S^\circ, N^\circ, M^\circ]).$$

We illustrate the states with figures, where an ellipse represents a regular node that receives a message, a hexagon represents a regular node that sends a message, and a box represents an attacker node. The solid-head arrows represent the goal-binding relation. The empty-head arrows represent the "⇒" relation. A dotted node represents an unbound goal. A filled node represents the goal selected by the ranking function to bind in the next step. For simplicity, we illustrate the following analysis without using the purning theorem 3.5.18.

We first apply the $Init$ rule and obtain an initial state containing only strand $S0 = \text{server}[C^\circ, S^\circ, N^\circ, M^\circ]$, as shown in Figure 3.2.

The initial state has two regular-goals as indicated by the dotted ellipse. Because the `Final` rule does not apply, we apply the `Split` rule. The next state function first selects a goal, as indicated with the filled ellipse.

To bind this regular-goal, there are two possibilities: this regular goal term was either first sent out by a regular node or an intruder node. In the first case, the next state function interm-unifies the goal term with positions on the regular strands. In this case, the only possibility is that a client strand sends out the message in

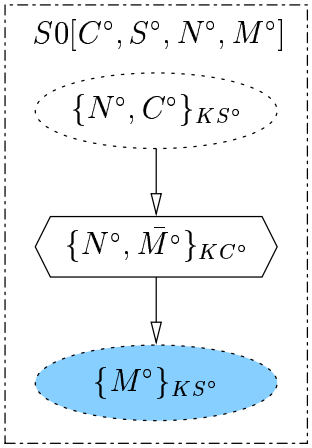


Figure 3.2: State 0

its third step. There is no client strand present in the State 0. So we add a client strand $C0$ with the computed substitution to form a new state, State 1, shown in Figure 3.3. Note that in the variable list of $C0$, C_{C0} and N_{C0} remain as free variables, and hence the strand $C0$ effectively represents a set of infinite number of strands with different instantiations. Adding the strand $C0$ also introduces new regular-goals as indicated by the dotted ellipse nodes. The filled ellipse denotes the goal that will be selected next when this state is split.

In the second case of the regular-goal binding in State 0, the goal term was first sent out by an intruder node. The algorithm simply add the intruder node and obtain a new state, State 2, shown in Figure 3.4. Adding the intruder node also introduces a new intruder-goal indicated by the dotted box. The filled node represents the goal that will be selected next when this state is split.

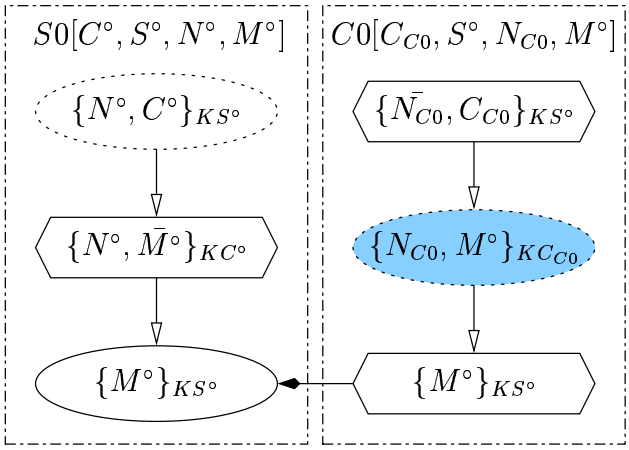


Figure 3.3: State 1

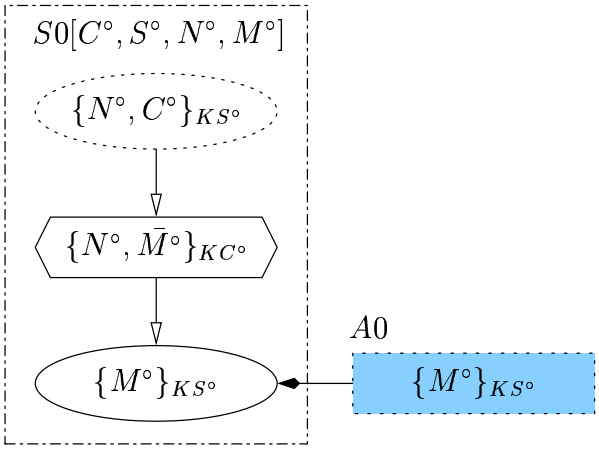


Figure 3.4: State 2

The current active state set contains State 1 and State 2. The procedure then analyze State 1. To bind the regular-goal indicated by the filled node in State 1, there are two possibilities: this regular goal term was either first sent out by a regular node or an intruder node. In the former case, the only possibility is that a server strand sends out the message in its second step. There is one server strand present in State 1 already. So the goal term can either bind to node $\langle S0, 2\rangle$ or to a node in a new server strand. Figure 3.5 shows the new state where the goal term is bound to $\langle S0, 2\rangle$. The algorithm automatically finds out that the case where the goal term binds to a new server strand is invalid, because this new server strand would need to generate nonce M° in the second step which is equal to the nonce generated by $S0$ and hence reach a conflict. Note that even though State 3 still contains unbound goals, the strand $C0[C^\circ, S^\circ, N^\circ, M^\circ]$ is already in the state. Therefore, the algorithm automatically applies the `Final` rule to State 3 and ends this branch of the search tree.

In the second case of the regular-goal binding in State 1, the goal term was first sent out by an intruder node. The analysis adds an intruder node and obtains a new state, State 4, shown in Figure 3.6. Adding the intruder node also introduces a new attacker-goal indicated by the dotted box. The filled node represents the goal that will be selected next when this state is split.

The current active state set contains State 2 and State 4. We repeatedly apply the `Split` rule and `Final` rule and generate a proof search tree. This analysis

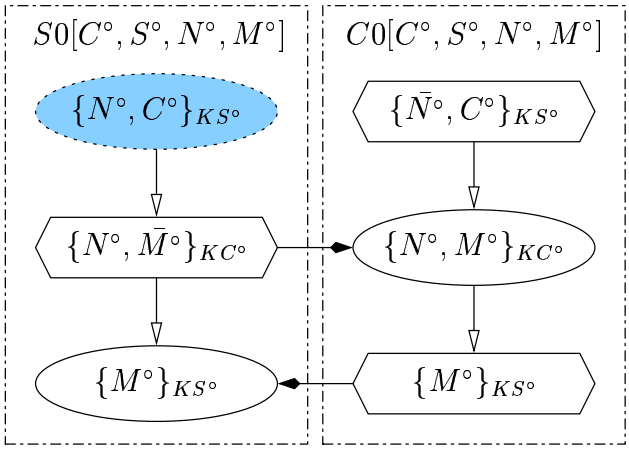


Figure 3.5: State 3

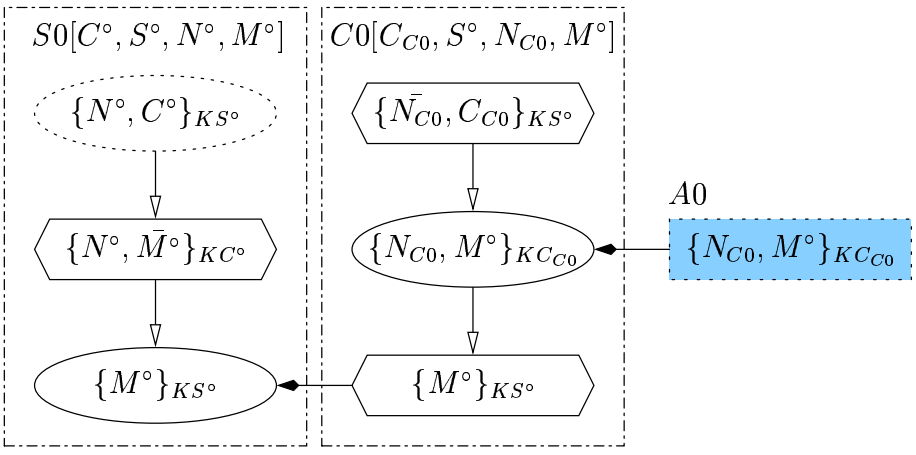


Figure 3.6: State 4

eventually stops with a counterexample after analyzing 20 states.

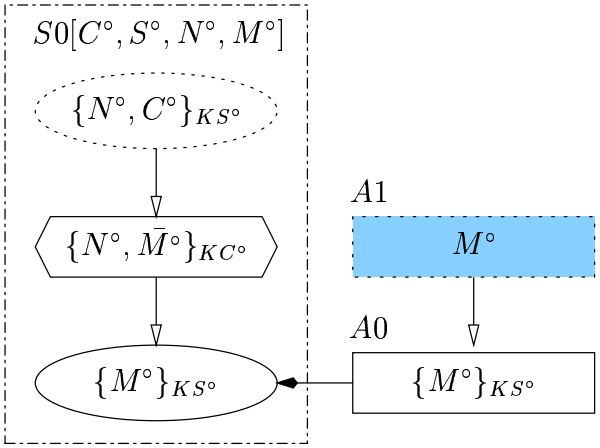


Figure 3.7: State 5

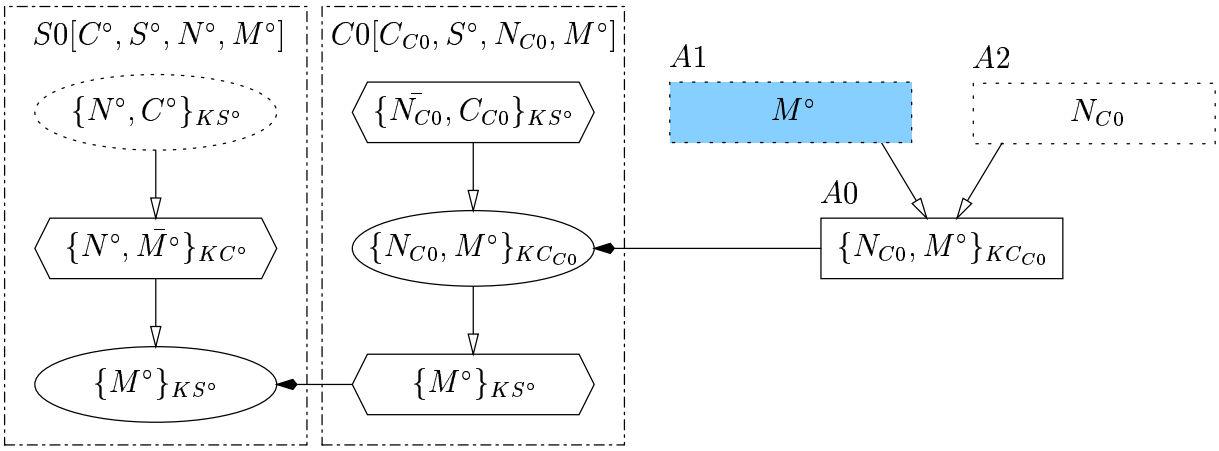


Figure 3.8: State 6

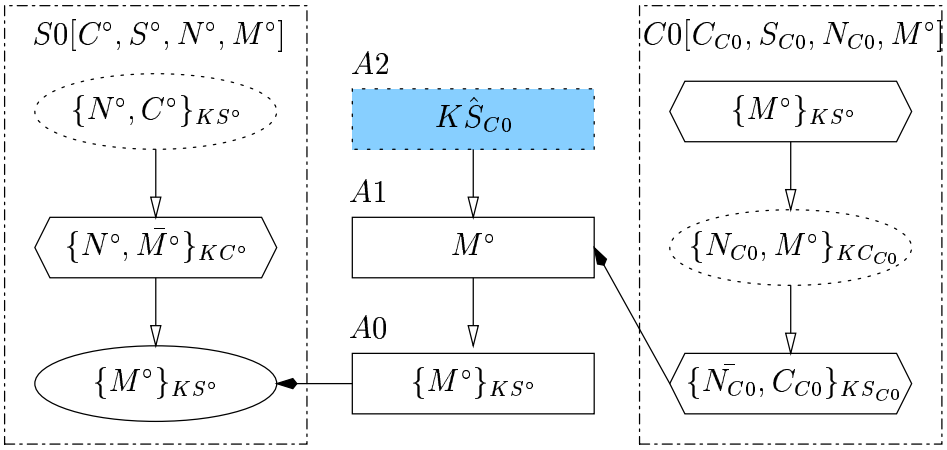


Figure 3.9: State 7

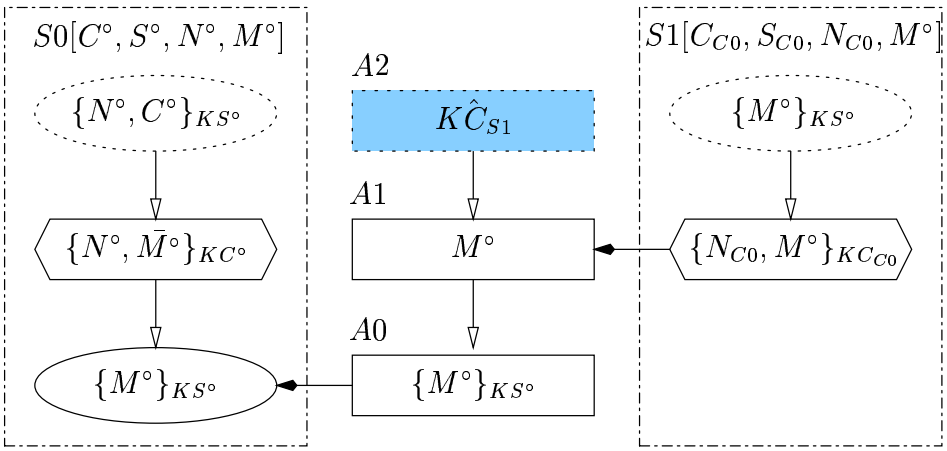


Figure 3.10: State 8

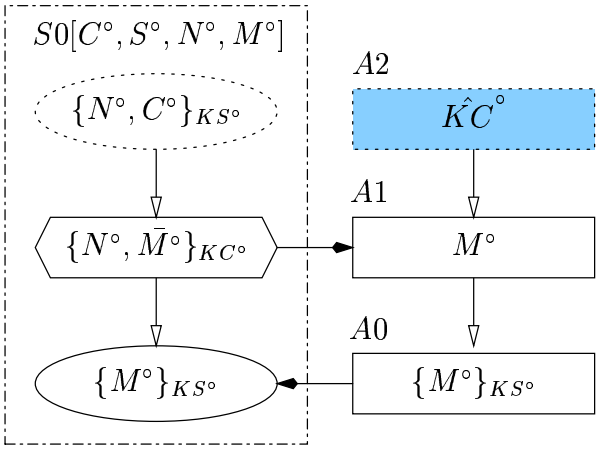


Figure 3.11: State 9

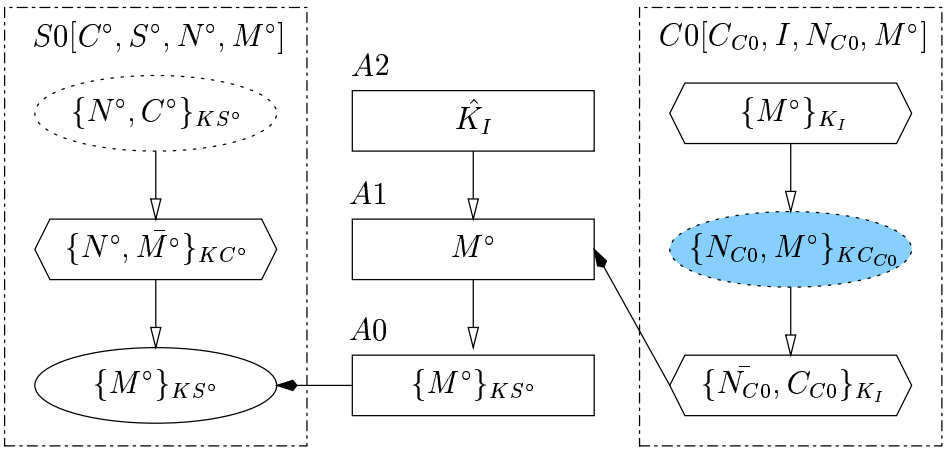


Figure 3.12: State 10

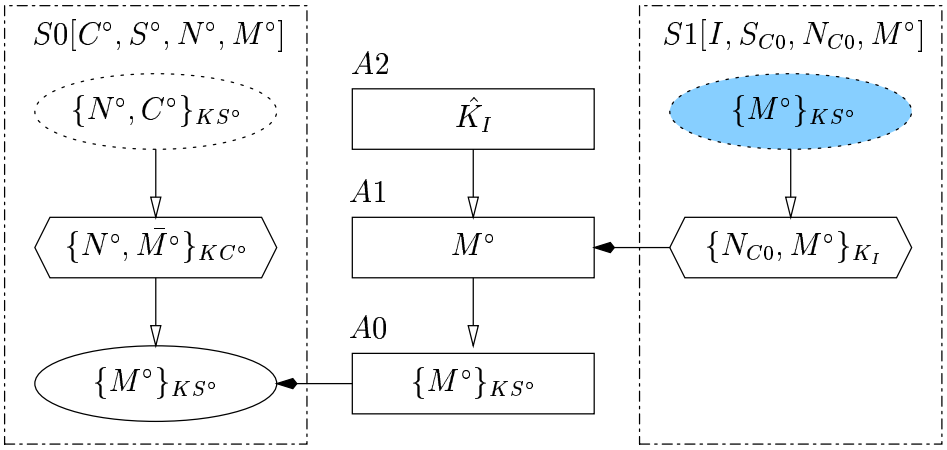


Figure 3.13: State 11

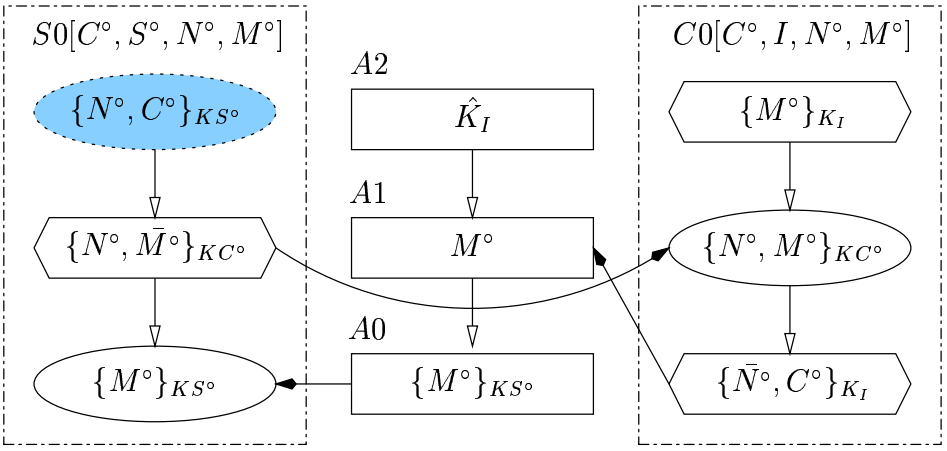


Figure 3.14: State 12

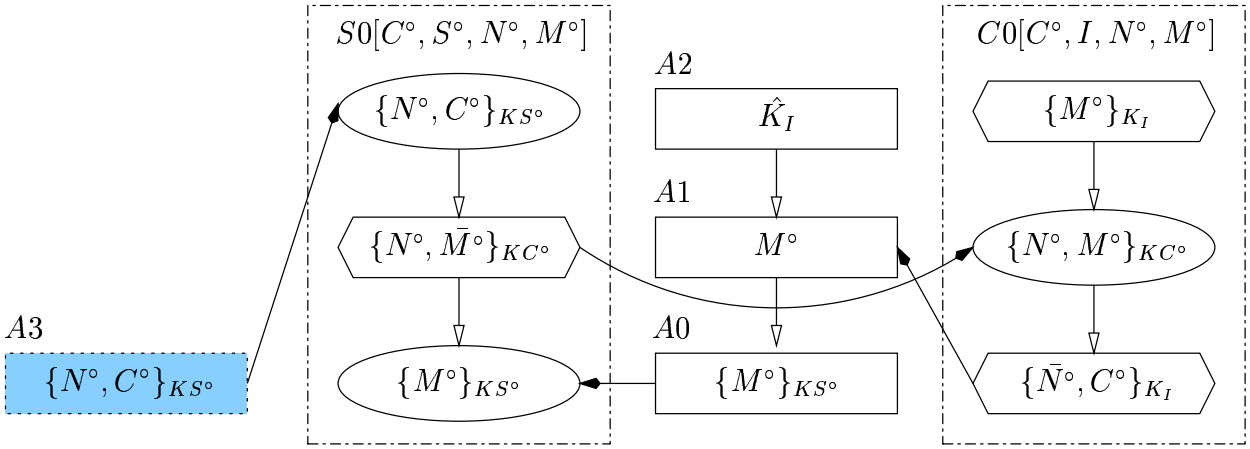


Figure 3.15: State 13

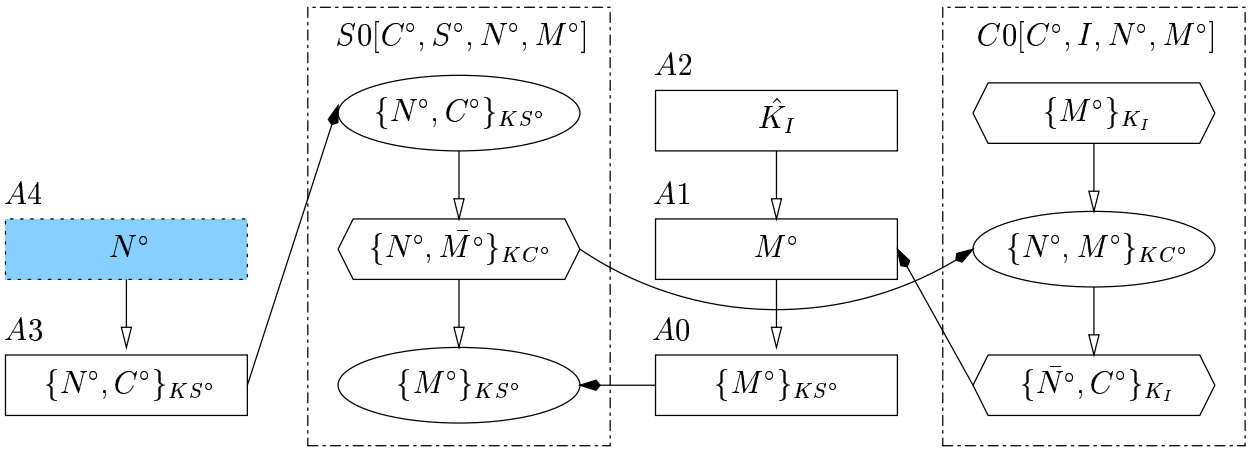


Figure 3.16: State 14

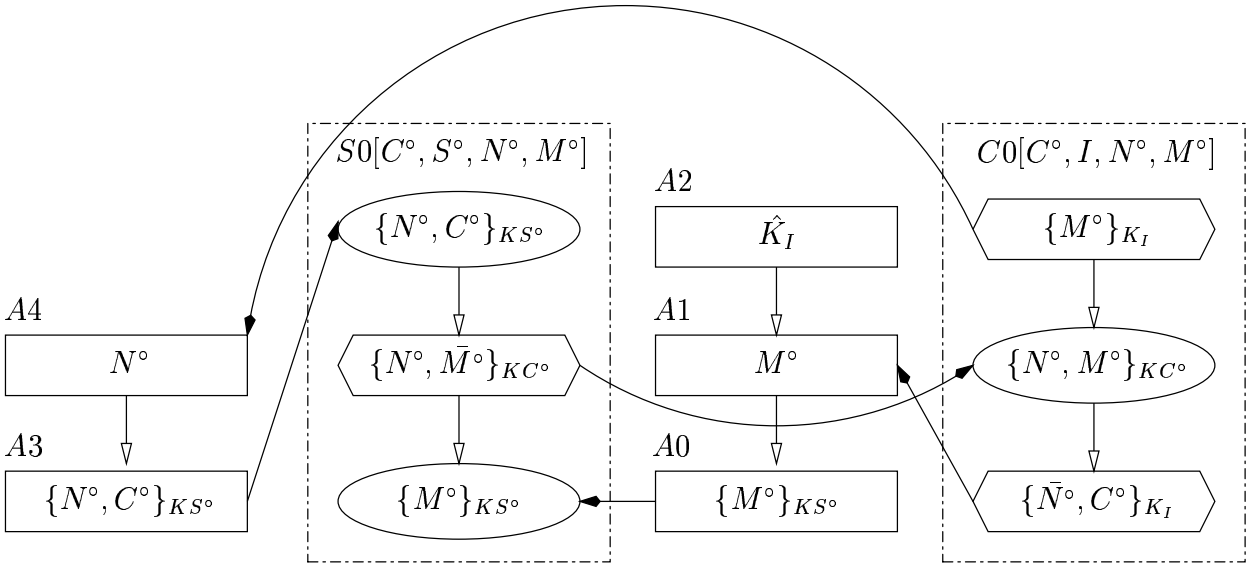


Figure 3.17: State 15

3.6 Experimental Results

We have implemented our verification technique as a research prototype using SML [39]. An input to APV consists of a protocol description and a set of security properties in an input language described in Appendix A. Given the input, APV runs automatically. When it terminates, it either generates a proof if the protocol satisfies the desired properties, or generates a counterexample if the protocol does not. In the latter case, a successful attack is constructed from the counterexample, and illustrated graphically using a graph drawing package *Dot* [1].

We have run several batches of experiments on a Pentium III PC with 450MHz CPU and 256MB RAM. The first batch is the analysis of various security properties

of protocols from the collection by Clark and Jacob [20]. We have used APV to check more than 20 protocols from [20], and discovered previously known flaws in many of these protocols including the ISO Symmetric Key Two-Pass Mutual Authentication Protocol, the Andrew RPC Protocol, the Otway-Rees Protocol, the Woo-and-Lam Authentication Protocol versions Π, Π_1, Π_2, and Π_3. We also discovered new anomalies in the Yahalom Protocol and the Carlsen's Secret Key Initiator Protocol as described in Section 4.7. All these experiments took less than a second for each security property checked. We describe some examples in Appendix A.

Another batch consists of 1641 asymmetric two party authentication protocols constructed by an automatic protocol generator (more details are described later in Chapter 4). These protocols are already filtered by an attacker filter which eliminates flawed protocols suffering from simple impersonation and replay attacks as described in Chapter 4). APV analyzes all of the 1641 protocols in about 100 seconds, with the average of around 60 milliseconds per protocol, and reports 120 protocols to be correct in terms of mutual authentication. We have also experimented with protocols generated by the automatic protocol generator for symmetric two party authentication and key distribution as described in Chapter 4. The protocol generator output more than $11,000$ candidate protocols. APV took less than two hours to analyze all of them.

3.7 Related Work

A number of different approaches have been proposed for security protocol analysis. Almost all of them use the same adversary model originally proposed by Dolev and Yao [29]. We describe these different approaches below.

Logics of Authentication. One of the earliest approaches for formally reasoning about security protocols is the development of new logics to express and deduce security properties. One of the earliest examples is the Logic of Authentication proposed by Burrows, Abadi, and Needham [13], commonly referred to as the BAN logic. Other logics for reasoning about security protocols include the GNY Logic [34] and a logic for accountability [47]. Kindred and Wing have built an automatic tool, Revere, to perform the protocol analysis using different logics [51, 50]. Brackin also developed a system for using the logics to analyze security protocols [11, 12]. These approaches have been successfully applied to find attacks to some protocols. However, the approach of using authentication logics only analyzes an abstraction of the protocol and does not have a corresponding concrete operational model for how the protocol executes [71].

Model Checkers. Researchers have applied general purpose model checkers as well as built special purpose model checkers for security protocol analysis. Gavin Lowe proposed to use FDR to analyze CSP [44] models of security protocols [54,

42]. CSP is a language for modeling asynchronous communication of processes. A tool Casper has been developed to automatically generate the adversary model in FDR [55]. Mitchell et. al. proposed to use a general purpose model checking tool, Murφ [2], to analyze a wide range of security protocols [68, 86, 69, 85]. Heintze et. al. used model checking techniques to analyze electronic commerce protocols [43].

Millen et. al. proposed one of the earliest special-purpose model checkers for security protocol analysis, Interrogator [67, 66]. Clarke, Marrero and Jha have proposed a special-purpose model checker, Brutus [25]. This tool uses symmetry reduction and partial order reduction techniques to reduce the search space [26].

These tools all start with a chosen protocol execution configuration, e.g. the number of parties of each role involved in the protocol execution. The analysis begins with the initial state of the chosen protocol execution configuration and then exhaustively search through all possible sequences of actions of the legitimate parties and the attacker to see whether an attack could happen. All these tools have been successfully applied to find attacks on protocols. But they all suffer from two main problems:

1. Bounded number of principles. All of these tools have to specify in advance the maximum number of principals that can participate in the protocol, which means that they can only check whether a protocol is correct for a limited number of participants. Even if they do not find an attack on the protocol, it is still possible that the protocol might have an attack with a higher

number of participants. In some cases researchers have manually proved that it is sufficient to show the correctness of a protocol execution for arbitrary number of parties by showing that the protocol is correct when constrained to a small number of participants [54, 56]. Unfortunately this kind of proof has to be manually worked out for every protocol and is often tedious and lengthy and hence is time-consuming and prone to error.

2. **State space explosion problem.** Although these tools explore a number of state reduction techniques, they still suffer from the state space explosion problem. They often can only analyze protocol executions containing only a small number of participants, e.g. three or five, which send and receive a small number of messages in each protocol run.

The above techniques are based on the traditional *trace-based model* (TBM), where each principal is modeled by a *process*. The global state space of the protocol is the Cartesian product of local state spaces of individual processes. The state transition of each local process is based on sending or receiving a message. The processes are composed *asynchronously*, and the global transition relation is an interleaving of local transitions. Due to such parallel composition and the interleaving semantics, the number of states and transitions to be explored grows exponentially with the number of participants involved in the protocol execution [76, 77]. Hence, these approaches suffer severely from the state space explosion problem, and are often only able to verify protocol executions with small number of par-

ticipants, e.g. three or five. This problem is especially acute for the explicit state enumeration techniques.

Multiple concurrent runs of a security protocol may yield many principals with identical roles. In this case the transition graph often possesses a lot of symmetries. Although some reduction techniques have been used to reduce the search space, including partial order and symmetry reductions [24], a large number of states and transitions still have to be checked unnecessarily.

Theorem Provers. Paulson proposed to use Isabelle [73], a general purpose theorem prover, for security protocol analysis. This approach has been successfully used to prove the security of certain protocols [74, 75]. However, this approach of using theorem provers requires expertise with theorem provers and human interaction, and has the disadvantage that it cannot generate counterexamples directly. Cohen has built an tool to reduce the amount of human interaction needed in using theorem proving techniques for security protocol analysis [27]. But his tool still has the problem that it cannot generate counterexamples.

Meadows proposed the NRL Protocol Analyzer, a special-purpose tool that uses a theorem proving approach for security protocol analysis [60]. It starts from an insecure state and performs a backward search trying to prove that this insecure state is unreachable. It can use many theorem proving techniques such as inductive methods. The advantage of this approach is that it can prove a protocol correct for

arbitrary number of participants. However, it often requires non-trivial amount of human interaction and expertise, and the running time could be much slower than in the model checking approach [62]. The NRL Protocol Analyzer reduces symmetry redundancy by using symbolic variables. But it also uses the traditional TBM and asynchronous composition, and hence still suffer from the state space explosion problem. The NRL Protocol Analyzer has recently been extended to handle other classes of security protocols [63, 64].

Chapter 4

Automatic Protocol Generation

This chapter describes the automatic protocol generator APG. We first give the motivation in Section 4.1 and an overview of the system in Section 4.2. We then describe the details about the design and implementation of the system, including the input specification in Section 4.3 and the automatic protocol generation procedure in Section 4.4. We explain our experiment results in Sections 4.5 and 4.6. Finally we summarize the related work in Section 4.8.

4.1 Motivation

The current protocol design process is manual and ad-hoc, and as a result, has many disadvantages:

- Manually-designed protocols are error-prone. Experience shows that secu-

rity protocols are hard to get right. Even when security protocols are carefully designed and examined by experts, subtle flaws may still remain undiscovered for years. Without proofs of security, these protocols cannot give any guarantee about the required security properties. In addition, due to the lack of formalism and mechanical assistance, manually designed protocols often contain undocumented assumptions. The implemented system may be flawed if the implementation is not consistent with these hidden assumptions.

- Manually-designed protocols may not be optimal in terms of efficiency. Some proposed protocols may be over-engineered and contain unnecessary operations. Since protocol designers can only consider a limited number of candidate protocols, they may not find the most efficient protocol for the given system requirements.

- The manual design process is slow and expensive. It may take months or years to design and evaluate a security protocol. Also, flawed protocols may cause substantial economic loss when exploited in real attacks.

With the rapid growth of the Internet, more and more new applications and systems need to be designed and developed. Many of these new applications and systems require new security protocols. Unfortunately, the current manual design process cannot satisfy the need for designing new security protocols. We propose

a new approach for security protocol design: *Automatic Protocol Generation* (APG). With APG, the protocol designer specifies the desired security properties (such as authentication and secrecy), and system requirements. A *protocol generator* generates *candidate* security protocols which satisfy the system requirements. In the final step, a *protocol screener* analyzes the candidate protocols, discards the flawed protocols, and outputs the correct protocols that satisfy the desired security properties. The approach of APG has many advantages:

- The protocol design process is automatic and efficient. The protocol designer specifies the security properties and the system requirements. The remaining process is fully automatic.

- The designed protocols are provably secure. The protocol screener has a powerful engine which can automatically generate a proof if a protocol is correct, or an attack example otherwise. In addition, since the input specifications are written in a well-defined specification language and the automatic protocol generation is fully mechanical, this approach largely eliminates hidden assumptions.

- The designed protocols are efficient. The user-defined system requirements include a metric function which specifies the cost or overhead of a protocol. APG tries to generate correct protocols with minimal cost with respect to the metric function.

4.2 Overview

APG is composed of an automatic protocol generator and an automatic protocol screener (see Figure 4.1).

Figure 4.1: APG overview

Figure 4.2 illustrates the data flow:

1. **Input specification**. The protocol designer specifies the desired security properties and system requirements as input.

2. **Protocol generation**. The protocol generator searches through the protocol space and generates candidate protocols that satisfy the system requirements.

3. **Protocol screener**. The protocol screener analyzes the candidate protocols,

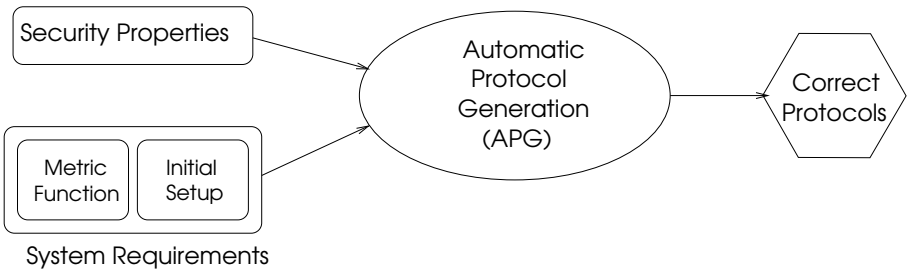


Figure 4.2: APG data flow

discards flawed protocols, and outputs the correct protocols that satisfy the desired security properties.

The automatic protocol screener needs to be sound and efficient. The protocol screener is sound if when it claims that a protocol satisfies certain security properties, it is true that the protocol really does satisfy these properties. Since the protocol generator generates thousands of candidate protocols, the protocol screener needs to be highly efficient for the APG approach to be viable. In our system, we use the Athena protocol verifier APV as the protocol screener.

We describe the input specification and the protocol generator in more detail in Section 4.3 and 4.4.

4.3 Input Specification

The input specification includes the specification of desired security properties and system requirements. Security properties are specified using a formal language, e.g., the logic in Section 3.4, so that the security requirements can be formally verified. The system requirements include a specification of the initial setup and a metric function:

- The **initial system configuration** defines the cryptographic primitives that each principal can perform and the keys each principal initially possesses. For example, in a public-key infrastructure environment, each party knows

its own private key and the public keys of the other principals, whereas in a symmetric-key environment the principals have shared secret keys.

- The **metric function** specifies the cost or overhead of a protocol in the given system. The protocol designer can specify the metric in terms of bandwidth, latency, memory, power consumption, or some combination of these factors.

For example, the metric function can be the time overhead of the protocol. In a smart-card system, symmetric encryption can be fast while the bandwidth between the card and the card reader may be low. Therefore the metric function specifies a low cost for encryption, and a high cost for sending and receiving messages.

The metric function needs to increase monotonically as the protocol complexity increases. This requirement is necessary during the protocol generation phase, where the protocols are generated by increasing the complexity until the protocol reaches the maximum cost threshold.[1]

The metric function defines an ordering among the protocols generated. The protocol screener analyzes the protocols in the order of increasing cost,[2] hence the first correct protocol has a minimal cost-value with respect to the metric function.

Definition 4.3.1. *Given a specification of security properties and system requirements,*

[1] A non monotonically increasing metric function would risk that some low-cost protocols may not be generated.

[2] All protocols are sorted after the generation step, since the generation might not generate them in strictly increasing order.

we say that a protocol is optimal if it has the lowest cost-value with respect to the metric function among protocols that satisfy the security properties and the system requirements.

4.4 The Protocol Generator

The protocol generator generates candidate protocols that satisfy the specified system requirements. From a high level, the protocol generator searches through the protocol space using iterative deepening, a standard search technique [84]. In each iteration, the protocol generator sets a *maximum cost threshold* and searches through the protocol space to generate all the protocols below the threshold. After sorting the protocols, the protocol screener tests them in the order of increasing cost. If one protocol satisfies the desired properties, it is minimal with respect to the cost metric and the generation process can stop. Otherwise, the protocol generator increases the cost threshold and does another iteration.

4.4.1 Enumerating the Protocol Space

A protocol defines a sequence of sending and receiving messages among the participating parties. We call each sending of a message a *step* of the protocol. The protocol space is composed of the message space and the message flow space. The message flow space contains all possible sequences of protocol steps, i. e. the sender and receiver of each step. The message space contains all possible messages

within the protocol scope. The protocol generator enumerates possible protocols by going through the message space and the message flow space.

4.4.2 Pruning Techniques

The search space increases drastically as the cost threshold increases. Even though an efficient protocol verification takes only a fraction of a second per protocol, verifying all protocols below a maximum cost threshold may still take too long to be practical. To make APG viable, we develop efficient reduction techniques and heuristics to prune candidate protocols early and reduce the number of candidate protocols passed to the protocol screener. In the rest of this section, we describe our pruning techniques in the protocol generator.

Our pruning algorithm can either operate at the *message level* (by inspecting a message in a single protocol step) or at the *protocol level* (by analyzing the entire protocol with all steps). Different security properties require different pruning techniques. For example, a secrecy property allows a pruning mechanism at the message level, since the secret value cannot be disclosed in any message. An authentication property, however, requires the pruning algorithm to work at the protocol level, since the authentication property is about relationship of multiple steps in a protocol.

Before we discuss our pruning techniques in more detail, we introduce some terminology and notions:

- **Early-pruning / late-pruning reduction**. An *early-pruning* reduction prunes a protocol after inspecting only its initial steps, e.g.the first and second step. On the contrary, *late-pruning* reductions have to inspect almost all the steps of a protocol to decide whether to prune the protocol or not. Clearly, early-pruning reductions are much more efficient than late-pruning techniques since they prune the search space at an early stage.

- **Minimal correct protocol / decorated protocol**. A *minimal correct protocol* is a correct protocol, which becomes flawed as soon as any message component is removed. A *decorated protocol* is a correct protocol which remains correct even after some message components are removed.

- **Lossless / quasi-lossless / lossy reduction**. A reduction technique is *lossless* if it does not prune away any correct protocol. A *quasi-lossless reduction* may prune some decorated protocols, but never prunes away any minimal correct protocol. A *lossy* reduction may prune away the optimal protocols. The quasi-lossless reductions are often early-pruning reductions and more aggressive than lossless reductions. In order to find optimal protocols, we have to use only lossless and quasi-lossless reduction techniques.

The reduction techniques consist of simple syntactical restriction rules and more sophisticated pruning algorithms. We assume that messages are well-typed, meaning that we do not consider type flaws, since they are easy to prevent [40].

Furthermore, during the protocol generation process, we do not consider permutations of message components of a concatenated message. In other words, two concatenated messages will be considered equivalent if they only differ in the ordering of their components. This highly reduces the protocol space although it is a lossy reduction. In Section 4.6 we show that this reduction does lose the optimal protocol in one of our experiments and we also show how we fix this problem.

Syntactical Restriction Rules

We now describe some of the syntactical restriction rules we use in the protocol generator.

1. Message-level restrictions. The following reductions operate on the message level, and prune messages in each step.
    - In any concatenated message, there are no redundant message components, e.g., N_A, N_A.
    - No long-term secret keys are sent in a message.
    - No useless encryptions. For example, never use nested encryptions that use the same key twice in a row, e.g., $\{\{X\}_{K_A}\}_{K_A}$.
2. Protocol-level restrictions.
    - Nonce-flow requirements for freshness. If a principal wants to authenticate another principal or check the freshness of a session key, it has to

generate a nonce and receive it back later in the protocol.

- The final message in a protocol should not contain any part which the recipient cannot decrypt.

Early-pruning by Simple Impersonation Attack

In authentication protocols, we introduce a simple *impersonation attack* pruning technique. This attack tests whether an adversary M, who eavesdrops all the messages, can generate the same messages as a legitimate principal P, in which case M can impersonate P. Although this technique is already quite powerful, it is a late-pruning reduction. We improve this reduction to be early-pruning by using the following observation: a principal can only save itself from an impersonation attack by sending a message which its impersonator cannot generate. Therefore, if the number of principals which can be impersonated is larger than the number of steps remaining, we can prune the current branch of the search tree. In the case of three-party and four-step authentication protocols, this reduction already starts pruning messages in the second step, and is therefore much more efficient than before.

Preventing Simple Replay Attacks

We also introduce a simple *replay attack* pruning technique. After a protocol session of an initiator A and a responder B, an intruder module stores all the mes-

sages sent in the session. Then the intruder module tries to impersonate A by resending A's packets to B. If the intruder can trick B to finish its session believing it is talking to A, then the protocol is flawed and is discarded. Similarly, the intruder can attempt to impersonate B and launch a replay attack against A. The purpose for this attack is to check whether nonces are used in a correct way. The intruder does not try to encrypt or decrypt messages or alter the received messages and the replay verification is hence very efficient.

4.5 Experiment I: Two-Party Authentication Protocols

Our first APG experiment is the automatic generation of two-party mutual authentication protocols. Authentication protocols are among the most widely used and intensely studied security protocols. Their complexity is suitable for an initial case study, and they are known to be difficult to design correctly [13, 54]. We now summarize the experimental results and our findings.

4.5.1 Input Specification

The property we are interested in is the mutual authentication property for a two-party authentication protocol. It can be represented similarly to the authentication property specification of the Needham-Schroeder protocol:

$$\forall C_P.C_P \models (\text{server}[s, c, N_a, N_b] \implies \text{client}[s, c, N_a, N_b]);$$

$$\forall C_P.C_P \models (\text{client}[s, c, N_a, N_b] \implies \text{server}[s, c, N_a, N_b]).$$

We perform two experiments: one uses symmetric key encryption schemes, the other one uses asymmetric key encryption schemes. In both experiments, we use a simple, linear metric function. Each operation has a unit-cost. The cost value of a protocol is the sum of the costs of all the protocol operations. In our example, we choose the cost to generate a new nonce to be one unit, the cost of a message encryption and decryption to be three units, and the cost to send a unit length to be one unit.

4.5.2 Running Time and Effectiveness of the Reduction Techniques

The automatic protocol generation generates approximately 2000 protocols per second including the pruning (this number is based on our Java implementation running on a 400 MHz Pentium II Linux workstation). Table 4.1 shows the statistics for the protocol generation in our experiment. The column labeled "Generated" shows how many protocols were initially generated with the corresponding cost threshold without applying the intruder reduction. The column marked "I.A." shows the number of protocols that are eliminated by the impersonation attack. Similarly, the "R.A." column depicts the number of protocols that are eliminated by the replay attack. The combination of the two attacks is efficient (shown in the "Combined" column) and filters out about 98% of candidate protocols for the sym-

Type	Generated	I.A.	R.A.	Combined	Candidates	Correct
Symmetric	19856	12098	18770	19449	407	2
Asymmetric	46518	46378	40687	46408	110	1

Table 4.1: Experiment Statistics for protocol generation. I.A. stands for impersonation attack and R.A. for replay attack

metric case and 99.8% for the asymmetric case (shown in the "Candidate" column).

4.5.3 Output Protocols

APV analyzed the candidate protocols and reported two correct symmetric-key protocols and one correct asymmetric-key protocol. The three protocols are listed below:

- Symmetric-key mutual authentication protocols.

$$\begin{aligned} \textsf{Protocol}: \quad & A \rightarrow B : N_A, A \\ & B \rightarrow A : \{N_A, N_B, A\}_{K_{AB}} \\ & A \rightarrow B : N_B \end{aligned}$$

$$\begin{aligned} \textsf{Protocol}: \quad & A \rightarrow B : N_A, A \\ & B \rightarrow A : \{N_A, N_B, B\}_{K_{AB}} \\ & A \rightarrow B : N_B \end{aligned}$$

The standard symmetric key mutual authentication protocol using nonces is

documented in ISO/IEC 9798 [46] as *ISO Symmetric-Key Three-Pass Mutual Authentication Protocol*:

$$\begin{aligned} \textsf{Protocol}: \quad & A \rightarrow B : N_A, A \\ & B \rightarrow A : \{N_A, N_B, B\}_{K_{AB}} \\ & A \rightarrow B : \{N_A, N_B\}_{K_{AB}} \end{aligned}$$

Our automatically-generated protocols are clearly simpler than the one listed in ISO standard for the same mutual authentication property.

- Asymmetric-key mutual authentication protocols.

$$\begin{aligned} \textsf{Protocol}: \quad & A \rightarrow B : \{N_A, A\}_{K_B} \\ & B \rightarrow A : \{N_A, N_B, B\}_{K_A} \\ & A \rightarrow B : N_B \end{aligned}$$

This protocol happens to be the same as the fixed version of Needham-Schroeder protocol [54], except that the last message is not encrypted. This is because we do not have the secrecy requirement in the security property specification.

Although intuitive, another interesting result from the statistics is that the ratio of the number of correct protocols versus the number of protocols that have the same cost is very low. For example, in this case study, the ratio of the number of generated protocols to the number of correct protocols is around 10^{-4}. And this ratio decreases as the cost threshold increases.

4.5.4 Experiment with Different Metric Functions

In another experiment, we generate two-party mutual authentication protocols with two extreme cases of the metric function and show how we benefit from the automatic protocol generation to generate near-optimal protocols.

In the first case we generate protocols for a bandwidth-constrained environment. Consider a smart-card with a built-in cryptographic accelerator, which can quickly perform encryption/decryption operations. But the smart-card has a low-bandwidth link to the card reader. In this case, we set the cost of encryption much lower than the bandwidth cost. With this metric function, APG finds one symmetric-key authentication protocol with minimum cost:

$$\begin{aligned} \text{Protocol}: \quad & A \rightarrow B : \{N_A, A\}_{K_{AB}} \\ & B \rightarrow A : \{N_A, N_B\}_{K_{AB}} \\ & A \rightarrow B : N_B \end{aligned}$$

In the second case we generate protocols for a computation-constrained environment. Consider a machine with a slow processor but a fast network link, so the cryptographic operations are the bottleneck. In this case, we set the bandwidth cost much lower than the encryption cost in the metric function. APG generates

the following two symmetric-key protocols.

$$\begin{aligned} \textsf{Protocol}: \quad & A \rightarrow B : N_A, A \\ & B \rightarrow A : \{N_A, N_B, A\}_{K_{AB}} \\ & A \rightarrow B : N_B \end{aligned}$$

$$\begin{aligned} \textsf{Protocol}: \quad & A \rightarrow B : N_A, A \\ & B \rightarrow A : \{N_A, N_B, B\}_{K_{AB}} \\ & A \rightarrow B : N_B \end{aligned}$$

Interestingly, the two protocols in the first case use one more encryption than the two protocols in the second case, but shorter messages. This experiment illustrates the benefit of automatic protocol generation.

For the asymmetric-key protocol, in both cases, the automatic protocol generation finds the same protocol as the asymmetric-key protocol we discuss in the previous subsection.

4.6 Experiment II: Automatic Generation of Protocols for Two-Party Authentication with Trusted Third Party

In this section, we present our results in automatic generation of protocols for two-party authentication with a trusted third party (TTP) using symmetric-key encryption schemes.

The specification of the authentication properties are below. Let S represent the trusted third party.

If A completes the protocol run with B in a session with N_a and N_b, then there must be a unique responder protocol run with the same binding of parameters. The same holds for B also.

1.

$$\forall C_P.C_P \models (Init[A, B, S, N_a, N_b] \implies Resp[A, B, S, N_a, N_b])$$

The uniqueness of the protocol run of B is achieved by the freshness of N_b generated in the protocol run.

2.

$$\forall C_P.C_P \models (Resp[A, B, S, N_a, N_b] \implies Init[A, B, S, N_a, N_b,])$$

The uniqueness of the protocol run of A is achieved by the freshness of N_a generated in the protocol run.

After analyzing $140,275$ protocols, the protocol generator generated 473 candidate protocols. Using less than 10 minutes, APV found two protocols:

1.
$$A \rightarrow B : N_A, A$$
$$B \rightarrow S : \{N_A, N_B, A\}_{K_{BS}}, B$$
$$S \rightarrow A : \{N_A, N_B, A, B\}_{K_{AS}}$$
$$A \rightarrow B : N_B$$

2.
$$A \rightarrow B : N_A, A$$
$$B \rightarrow S : \{N_A, N_B, A, B\}_{K_{BS}}, B$$
$$S \rightarrow A : \{N_A, N_B, B\}_{K_{AS}}$$
$$A \rightarrow B : N_B$$

Interestingly, APG initially missed the following protocol that is more efficient:

$$A \rightarrow B : N_A, A$$
$$B \rightarrow S : \{A, N_A, N_B\}_{K_{BS}}, B$$
$$S \rightarrow A : \{N_A, N_B, B\}_{K_{AS}}$$
$$A \rightarrow B : N_B$$

The reason that APG missed the above protocol is that APG ignores the permutation of components in a concatenated message. APG generated the following protocol instead and the protocol verifier pruned the protocol because it is flawed.

$A \rightarrow B : N_A, A$

$B \rightarrow S : \{N_A, N_B, A\}_{K_{BS}}, B$

$S \rightarrow A : \{N_A, N_B, B\}_{K_{AS}}$

$A \rightarrow B : N_B$

Obviously, ignoring the permutation of components in a concatenated message is a lossy reduction. But it is such a powerful reduction that we cannot remove it. Instead, we add a new rule to APG to solve this problem. The protocol generator would detect when the protocol contains two messages with the same format. When this occurs, the protocol generator will change the order of the concatenated components in one of the two messages to make it a different format and produce a new protocol. After this fix, APG was able to find the more efficient protocol.

4.7 Experiment III: Authentication and Key Distribution with Trusted Third Party

In this section, we show our results in automatic generation of authentication and key distribution protocols with trusted third party using symmetric-key encryption schemes.

The security properties for key agreement are much more complex than previous examples, including key freshness, key confirmation and key secrecy. We first give a complete list of the specifications of different security properties. We then

describe the protocols found by APG with different sets of desired security properties. To make APG easy to use, we developed a GUI for specifying the inputs, as shown in Figure 4.3.

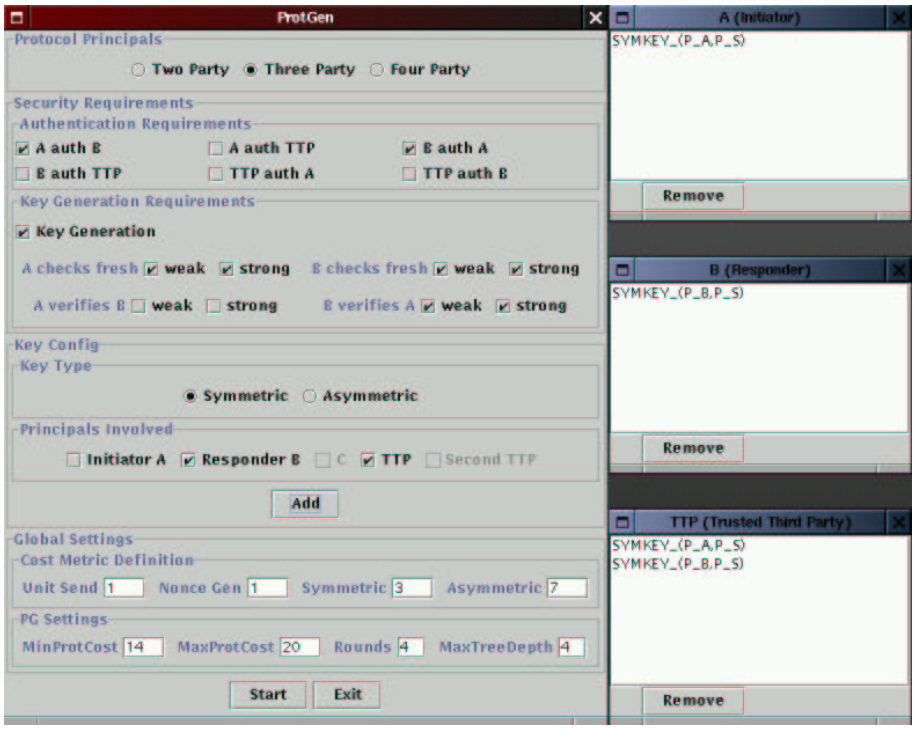


Figure 4.3: APG GUI

Our list of formal specifications of the security properties is more comprehensive than the one in [90]. For example, we distinguish between weak key freshness and strong key freshness, while [90] does not consider strong key freshness. In fact, the Yahalom protocol [20] was previously proven to provide key freshness [31]. However, the Athena protocol verifier proved that the Yahalom protocol only provides weak key freshness.

Specification of security properties

1. Authentication between the initiator and the responder

   If A completes the protocol run with B in a session with N_a and N_b, then there must be a unique responder protocol run with the same binding of parameters. The same holds for B also.

$$\forall C_P.C_P \models Init[A, B, S, N_a, N_b, *] \implies Resp[A, B, S, N_a, N_b, *]$$

   The uniqueness of the protocol run of B is achieved by the freshness of N_b generated in the protocol run.

$$\forall C_P.C_P \models Resp[A, B, S, N_a, N_b, *] \implies Init[A, B, S, N_a, N_b, *]$$

   The uniqueness of the protocol run of A is achieved by the freshness of N_a generated in the protocol run.

2. Weak key freshness with server authentication

   If A completes a protocol run and accepts a session key K_{ab} in a session with B and nonce N_a, then there must be a unique protocol run of server S with K_{ab} for the session between A and B with nonce N_a. The same holds for B also.

$$\forall C_P.C_P \models Init[A, B, S, N_a, *, K_{ab}] \implies Srv[A, B, S, N_a, *, K_{ab}]$$

$$\forall C_P.C_P \models Resp[A, B, S, *, N_b, K_{ab}] \implies Srv[A, B, S, *, N_b, K_{ab}]$$

The uniqueness of the protocol run of S is achieved by the freshness of K_{ab} generated in the protocol run.

3. Strong key freshness with server authentication

   If A completes a protocol run and accepts a session key K_{ab} in a session with I and nonce N_a, then there must be a unique protocol run of server S with K_{ab} for the session between A and I with nonce N_a. The same holds for B also.

$$\forall C_P.C_P \models Init[A, I, S, N_a, *, K_{ab}] \implies Srv[A, I, S, N_a, *, K_{ab}]$$

$$\forall C_P.C_P \models Resp[I, B, S, *, N_b, K_{ab}] \implies Srv[I, B, S, *, N_b, K_{ab}]$$

   The uniqueness of the protocol run of S is achieved by the freshness of K_{ab} generated in the protocol run.

   This property is stronger than the weak key freshness property, because in this property even when A (or B) is having a session with I, I cannot trick A (or B) to accept a key that is not freshly generated by S for the session.

4. Weak key confirmation between the initiator and the responder

If A completes the protocol run and accepts the session key K_{ab} with B in a session with N_a, then there must be a responder protocol run with the same binding of parameters. The same holds for B also.

$$\forall C_P.C_P \models Init[A, B, S, N_a, *, K_{ab}] \implies Resp[A, B, S, N_a, *, K_{ab}]$$

$$\forall C_P.C_P \models Resp[A, B, S, *, N_b, K_{ab}] \implies Init[A, B, S, *, N_b, K_{ab}]$$

5. Strong key confirmation between the initiator and the responder

   If A completes the protocol run and accepts the session key K_{ab} with B in a session with N_a, N_b, then there must be a unique responder protocol run with the same binding of parameters. The same holds for B also.

$$\forall C_P.C_P \models Init[A, B, S, N_a, N_b, K_{ab}] \implies Resp[A, B, S, N_a, N_b, K_{ab}]$$

   The uniqueness of the protocol run of B is achieved by the freshness of N_b generated in the protocol run.

$$\forall C_P.C_P \models Resp[A, B, S, N_a, N_b, K_{ab}] \implies Init[A, B, S, N_a, N_b, K_{ab}]$$

   The uniqueness of the protocol run of A is achieved by the freshness of N_a generated in the protocol run.

6. Key secrecy

If A completes the protocol run and accepts the session key K_{ab} with B, then the intruder cannot learn K_{ab}. The same holds for B also.

$$\forall C_P.C_P \models Init[A, B, S, N_a, N_b, K_{ab}] \implies K_{ab} \notin \Theta$$

$$\forall C_P.C_P \models Resp[A, B, S, N_a, N_b, K_{ab}] \implies K_{ab} \notin \Theta$$

New protocols

We did experiments using APG to find near-optimal protocols which satisfy three different sets of the security properties as shown in Table 4.2. The protocol generator output more than $11,000$ candidate protocols. APV took less than two hours to analyze them. We list our findings below. Property-set 1 is the strongest among the three. Protocol-set S2 and S3 have the same costs and S1 has a higher cost than both. Interestingly, Yahalom protocol, shown in Figure 4.4, has the same costs as protocol-set S1 but only achieves the same security properties as protocol-set S2. In fact, the Yahalom protocol was generated by the protocol generator but Athena found a counterexample of the strong key freshness property in Yahalom. The counterexample simply shows that if B is a responder in a protocol run with the intruder I as the initiator, I can easily make B to accept an old key as the "fresh" session key generated by S. In some applications, if the session key is also used for some other purposes, e.g., the hash of the session key is used as a transaction ID (of course, such a usage of session keys is not recommended in the first place), the strong key freshness may be important.

	A auth B	B auth A	Weak key freshness (A&B)	Strong key freshness (A&B)	Weak key conf (A to B)	Key secrecy	Protocols
1	X	X	X	X	X	X	S1
2	X	X	X		X	X	S2
3	X	X	X	X		X	S3

Table 4.2: Three sets of security properties

- Protocol-Set S1 satisfies the first set of the security properties:

  $A \rightarrow B : N_A, A$

  $B \rightarrow S : \{N_A, N_B, A\}_{K_{BS}}, B$

  $S \rightarrow A : \{SK_{AB}, N_B\}_{K_{BS}}, \{SK_{AB}, N_A, N_B, B\}_{K_{AS}}$

  $A \rightarrow B : \{SK_{AB}, N_B\}_{K_{BS}}, \{N_B\}_{SK_{AB}}$

- Protocol-Set S2 satisfies the second set of the security properties:

1.
   $A \rightarrow B : N_A, A$

   $B \rightarrow S : \{N_A, N_B, A\}_{K_{BS}}, B$

   $S \rightarrow A : \{SK_{AB}\}_{K_{BS}}, \{SK_{AB}, N_A, N_B, B\}_{K_{AS}}$

   $A \rightarrow B : \{N_B\}_{SK_{AB}}, \{SK_{AB}\}_{K_{BS}}$

Yahalom :

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

$$B \rightarrow S : \{N_A, N_B, A\}_{K_{BS}}, B$$

$$S \rightarrow A : \{SK_{AB}, A\}_{K_{BS}}, \{SK_{AB}, N_A, N_B, B\}_{K_{AS}}$$

$$A \rightarrow B : \{SK_{AB}, A\}_{K_{BS}}, \{N_B\}_{SK_{AB}}$$

Figure 4.4: Yahalom Protocol [20]

2.

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

$$B \rightarrow S : \{N_A, N_B, A\}_{K_{BS}}, B$$

$$S \rightarrow A : \{SK_{AB}, N_A, N_B, B, \{SK_{AB}\}_{K_{BS}}\}_{K_{AS}}$$

$$A \rightarrow B : \{N_B\}_{SK_{AB}}, \{SK_{AB}\}_{K_{BS}}$$

- Protocol-Set S3 satisfies the third set of the security properties:

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

$$B \rightarrow S : \{N_A, N_B, A\}_{K_{BS}}, B$$

$$S \rightarrow A : \{SK_{AB}, N_B\}_{K_{BS}}, \{SK_{AB}, N_A, N_B, B\}_{K_{AS}}$$

$$A \rightarrow B : N_B, \{SK_{AB}, N_B\}_{K_{BS}}$$

4.8 Related Work

APG is the first approach to automatically generate security protocols [79, 81]. Clark and Jacob independently investigated this problem using AI techniques [21]. But their approach does not generate real protocols, only an abstract specification in the BAN logic, and does not support the concept of metric functions and optimal protocols.

Chapter 5

Automatic Code Generation

5.1 Motivation

Even when the design is correct, implementation errors can still break the security of a system. Unfortunately, the current implementation process is through manual programming, and as a result, error-prone and expensive:

- Human errors are inevitable. Therefore, programming bugs are inevitable in a manual implementation process.
- The manual implementation process introduces a gap between the protocol specification and the final implementation, with no guarantee that the implementation realizes the protocol specification.
- A manual implementation process consumes programmers' time, an expen-

sive commodity. Moreover, implementation bugs are often expensive to fix and patch, and can cause substantial economic loss when exploited.

To address these problems, we propose an automatic approach for security protocol implementation. In particular, we have designed and developed a tool, Automatic Code Generator (ACG). Given a security protocol specified in a protocol specification language, ACG can automatically generate Java source code implementing the protocol. Our approach is fast, fully automatic, and eliminates human programming errors. For example, our approach prevents several popular attacks:

- **Buffer overruns** account for more than half of all recent security vulnerabilities [92]. Since we use Java as our implementation language, our automatically generated code is secure against this class of attack.

- **False input attacks** result from unchecked input parameters or unchecked conditions. A recently discovered vulnerability in PGP was due to such an implementation flaw [18]. Our automatic implementation ensures that all possible ranges of each input parameter are carefully checked.

- **Type flaws** are a source of many protocol attacks [20]. We eliminate these type flaws by automatically include type information in the constructed messages in the automatically generated implementation, and the receiver always verifies the correctness of the type information [40].

- Many implementation flaws are caused by programmers misusing cryptographic primitives. In our implementation, we ensure that correct cryptographic primitives are used.

5.2 Automatic Code Generation

ACG takes as input a protocol description in a formal protocol specification language used in specifying the input to APV and the output of APG. The automatic code generation process encompasses two steps: translation and generation. Translation is the process of converting the protocol specifications into an internal data structure. Generation is the process of translating the internal data structure and producing source code that follows the given specification. Although this can be done by a single module, separating the process into translation and generation allows us the possibility of changing the protocol representation without having to rewrite the generation code. This is helpful if we expand the message grammar.

For each party in the protocol, the program generates a Java class file, which implements the party's actions. For each step X that the party participates in, the class provides a method of the form step$X$. These methods may take as input a byte array representing a message from another party, and may also produce as output a byte array representing that party's response. These methods are called in turn when executing a protocol run.

5.3 Experimental Results

We demonstrate the code generated by ACG using a symmetric-key three-pass mutual authentication protocol generated by APG as described in Section 4.5. The high-level protocol description is as follows:

$$\begin{aligned} \text{Protocol}: \quad & A \rightarrow B : N_A, A \\ & B \rightarrow A : \{N_A, N_B, A\}_{K_{AB}} \\ & A \rightarrow B : N_B \end{aligned}$$

5.3.1 The Generated Code

From this description, ACG generates the following classes and method signatures:

```
class PrincipalA {
  byte[] step0();
  byte[] step1(byte[] message);
}

class PrincipalB {
  byte[] step0(byte[] message);
  void step1(byte[] message);
}
```

In an application that uses this generated code, the protocol execution would be carried out with the following series of calls:

```
byte[] passAround;
PrincipalA pa = new PrincipalA();
PrincipalB pb = new PrincipalB();
passAround = pa.step0();
passAround = pb.step0(passAround);
passAround = pa.step1(passAround);
pb.step1(passAround);

/*
 Protocol: VAR: P_B,P_A,NONCE_B1,NONCE_A1;
 P_A->P_B:C[NONCE_A1,P_A] | New(NONCE_A1);
 P_B->P_A:E{C[NONCE_A1,NONCE_B1,P_A],SYMKEY_(P_A,P_B)} | New(NONCE_B1);
 P_A->P_B:NONCE_B1;
*/
import acg.blocks.*;
import java.security.*;
import javax.crypto.*;
import java.util.Vector;
public class sym3mapP_A
```

The variables named in uppercase are specific to Principal A in the protocol. The code knows that at some point, it will have to send and/or receive these variables. The variables randGen, kgen, and myCipher are necessary components in the generation of nonces, session keys, as well as the encryption and decryption of messages.

```
private PrincipalAO _P_A;
private NonceAO _NONCE_A1;
private NonceAO _NONCE_B1;
private KeyAO _SYMKEY__P_A_P_B_;
private SecureRandom randGen;
```

```
private KeyGenerator kgen;
private Cipher myCipher;
```

In the constructor, we take as input a symmetric key K_{AB} and a name P_A standing for principal A. In this example, we use Blowfish for our default key generator and block cipher, however, other key generators or block ciphers can also be used.

```
public _3_19_tempP_A(KeyAO _SYMKEY__P_A_P_B_) throws ACGException {
  this._SYMKEY__P_A_P_B_ = _SYMKEY__P_A_P_B_;
  try {
    randGen = new SecureRandom();
    kgen = KeyGenerator.getInstance("Blowfish");
    myCipher = Cipher.getInstance("Blowfish");
  } catch(Exception e) {
     throw new ACGException(e.toString());
  }
}
```

From the input protocol description, ACG knows that Principal A initiates the protocol, so the `round0` method takes no input. Below is the code for `round0` of principal A. This round corresponds to the first protocol round, where principal A sends message $A \rightarrow B : N_A, A$.

```
public byte[] round0() throws ACGException {
  /*Processing output section.*/
  Vector outVector0 = new Vector();
  _NONCE_A1 = new NonceAO(randGen);
  outVector0.addElement(_NONCE_A1);
  return ACGUtils.objectsToBytes(outVector0);
}
```

Below is the code for `round1` of principal A. This round corresponds to the second and third protocol rounds, where principal A receives message $B \rightarrow A : \{N_A, N_B, A\}_{K_{AB}}$ and sends message $A \rightarrow B : N_B$.

```
public byte[] round1(byte[] input) throws ACGException {
  /*Processing input section.*/
  Object obj;
  Vector inVector0 = ACGUtils.bytesToObjects(input);
  if (inVector0.size() != 1)
    throw new ACGException("Input message is of size "
                           +inVector0.size()
                           +", but it should be of size 1");
  obj = inVector0.elementAt(0);
  if (!(obj instanceof SymEncryptedAO))
    throw new ACGException("obj not instanceof SymEncryptedAO");
  SymEncryptedAO receiveEncrypt1 = (SymEncryptedAO) obj;
  receiveEncrypt1.initializeCrypto(myCipher, _SYMKEY__P_A_P_B_.getKey(), randGen);
  receiveEncrypt1.decryptAndVerify();
  Vector inVector1 = ACGUtils.bytesToObjects(receiveEncrypt1.getMessage());
  if (inVector1.size() != 3)
    throw new ACGException("Input message is of size "
                           +inVector1.size()
                           +", but it should be of size 3");
  obj = inVector1.elementAt(0);
  if (!(obj instanceof NonceAO))
    throw new ACGException("obj not instanceof NonceAO");
  /*We received an old variable. Let's check if it is still equal.*/
  if (!(_NONCE_A1.equals((NonceAO) obj)))
    throw new ACGException("_NONCE_A1 is not equal to itself.");

  obj = inVector1.elementAt(1);
  if (!(obj instanceof NonceAO))
```

```
    throw new ACGException("obj not instanceof NonceAO");
  /*We received this new variable. Need to assign it a value.*/
  _NONCE_B1 = (NonceAO) obj;

  obj = inVector1.elementAt(2);
  if (!(obj instanceof PrincipalAO))
    throw new ACGException("obj not instanceof PrincipalAO");
  /*We received this new variable. Need to assign it a value.*/
  _P_A = (PrincipalAO) obj;

  /*Processing output section.*/
  Vector outVector0 = new Vector();
  outVector0.addElement(_NONCE_B1);
  return ACGUtils.objectsToBytes(outVector0);
}
```

The code for Principal B is included for completeness. Note the data flow from `A.round0()` → `B.round0()` → `A.round1()` → `B.round1()`.

```
/*
 Protocol: VAR: P_B,P_A,NONCE_B1,NONCE_A1;
 P_A->P_B:C[NONCE_A1,P_A] | New(NONCE_A1);
 P_B->P_A:E{C[NONCE_A1,NONCE_B1,P_A],SYMKEY_(P_A,P_B)} | New(NONCE_B1);
 P_A->P_B:NONCE_B1;
*/
import acg.blocks.*;
import java.security.*;
import javax.crypto.*;
import java.util.Vector;
public class _3_19_tempP_B {
  private PrincipalAO _P_A;
  private NonceAO _NONCE_A1;
```

```
private NonceAO _NONCE_B1;
private KeyAO _SYMKEY__P_A_P_B_;
private SecureRandom randGen;
private KeyGenerator kgen;
private Cipher myCipher;

public _3_19_tempP_B(PrincipalAO _P_A, KeyAO _SYMKEY__P_A_P_B_) throws ACGException {
  this._P_A = _P_A;
  this._SYMKEY__P_A_P_B_ = _SYMKEY__P_A_P_B_;
  try {
    randGen = new SecureRandom();
    kgen = KeyGenerator.getInstance("AuthEncCipher");
    myCipher = Cipher.getInstance("AuthEncCipher");
  } catch(Exception e) {
    throw new ACGException(e.toString());
  }
}

public byte[] round0(byte[] input) throws ACGException {
  /*Processing input section.*/
  Object obj;
  Vector inVector0 = ACGUtils.bytesToObjects(input);
  if (inVector0.size() != 1)
    throw new ACGException("Input message is of size "
                           +inVector0.size()
                           +", but it should be of size 1");
  obj = inVector0.elementAt(0);
  if (!(obj instanceof NonceAO))
    throw new ACGException("obj not instanceof NonceAO");
  /*We received this new variable. Need to assign it a value.*/
  _NONCE_A1 = (NonceAO) obj;
```

```
  /*Processing output section.*/
  Vector outVector0 = new Vector();

  Vector outVector1 = new Vector();
  outVector1.addElement(_NONCE_A1);
  _NONCE_B1 = new NonceAO(randGen);
  outVector1.addElement(_NONCE_B1);
  outVector1.addElement(_P_A);
  SymEncryptedAO sendEncrypt1 =
      new SymEncryptedAO(ACGUtils.objectsToBytes(outVector1),
                         myCipher,
                         _SYMKEY__P_A_P_B_.getKey(),
                         randGen);
  outVector0.addElement(sendEncrypt1);

  return ACGUtils.objectsToBytes(outVector0);
}

public void round1(byte[] input) throws ACGException {
  /*Processing input section.*/
  Object obj;
  Vector inVector0 = ACGUtils.bytesToObjects(input);
  if (inVector0.size() != 1)
    throw new ACGException("Input message is of size "+
                           inVector0.size()
                           +", but it should be of size 1");
  obj = inVector0.elementAt(0);
  if (!(obj instanceof NonceAO))
    throw new ACGException("obj not instanceof NonceAO");
  /*We received an old variable. Let's check if it is still equal.*/
  if (!(_NONCE_B1.equals((NonceAO) obj)))
    throw new ACGException("_NONCE_B1 is not equal to itself.");
```

```
    }
}
```

Chapter 6

Conclusions and Future Work

In this chapter, we conclude by summarizing our results and highlighting promising directions for future work.

6.1 Summary of Results

In this thesis, we have described the Athena system for automatic protocol generation, verification and implementation. The system contains three components: the automatic protocol generator (APG), the automatic protocol verifier (APV), and the automatic code generator (ACG). To use Athena to automatically design and implement a security protocol, the user specifies desired security properties and system requirements and then the tool automatically runs. When the tool terminates, APG and APV together find an efficient protocol that satisfies the desired security properties and ACG produces the Java source code for the generated pro-

tocol. We have overcome many challenges in designing and developing such a system. We were the first group to propose and realize such an automatic approach for security protocol design and development. Our experiments demonstrate that such an automatic approach is an important improvement over previous manual approaches in building secure systems. Our approach can help to make the system design and development process more productive and effective, making the systems more secure.

In Chapter 3, we described the automatic protocol verification tool, APV. The efficiency and capabilities of APV make it an important improvement over previous approaches for security protocol analysis. Previous approaches for security protocol analysis mainly fall into two camps: theorem-proving and model-checking. Previous theorem-proving approaches are not automatic and require a lot of human interaction [74, 75]. They also have the disadvantage that they cannot find counterexamples for flawed protocols [74, 75]. Previous model-checking approaches can automatically test whether a protocol is correct for some small test cases, but cannot provide a proof of correctness of the protocol for all cases. They also suffer severely from the state space explosion problem that prevents them from analyzing complicated protocols or bigger test cases. Given a security protocol and desired security requirements, APV analyzes the protocol automatically. When APV terminates, it can either generate a proof of correctness if the protocol satisfies the given requirements, or generate a counterexample if the pro-

tocol is flawed. APV is the first automatic approach that has both the capability of providing a proof of correctness when the protocol is correct and the capability of generating a counterexample when the protocol is flawed. APV achieves this by combining techniques from both theorem-proving and model-checking, while leveraging a new compact state representation to reduce the search space. The compact state representation captures the exact causal relationships in protocol executions. One state in APV actually represents an infinite set of protocol executions. The proof search procedure is designed in such a way that by analyzing a finite number of states, it can analyze an infinite number of all possible protocol executions. The proof search procedure exploits several reduction techniques and analyzes infinite sets of protocol executions efficiently—APV often analyzes a protocol within fractions of a second. In addition, APV also enables the approach of automatic protocol generation, which was not achieved before partially because no efficient automatic protocol verifier existed.

In Chapter 4, we described the automatic protocol generation tool, APG. APG uses powerful pruning techniques to drastically reduce the protocol search space. It is the first approach for automatic generation of security protocols. In our experiments, APG was able to generate desired security protocols within hours. APG has generated new protocols that are more efficient than previous protocols proposed in the literature for different system settings.

We have implemented our algorithms for automatic protocol verification, gen-

eration, and implementation as a toolkit, Athena. The toolkit Athena is flexible – the user can either use each component separately as a stand-alone tool, or link all the components as an end-to-end system. Each component is easy to use. We describe the input language of the APV system in Appendix A. An input file specifying a security protocol often takes less than a page. To make the tool APG easy to use, we have developed a graphical user interface for a general class of key distribution protocols as shown in Section 4.6. However, Athena is still at a stage of a research prototype and the implementation of Athena can be improved in many ways. The implementation of APV can be optimized by using more efficient data structures. For example, the role templates in a protocol are implemented as lists which only allow sequential access not random access. A data structure such as an array or a hash table can be more efficient. This implementation choice was mainly because lists are easy to implement in ML and can take shorter time to implement than other data structures such as arrays and hash tables. The implementation of APG can be improved with more generalization. The current implementation has many search and pruning techniques hard-coded. A more general implementation should be modularized and allow different techniques to be plugged in and out freely.

Our Athena system proposes a new approach for security protocol design and development: the automatic approach for security protocol generation, verification, and implementation. Our toolbox provides an end-to-end system for auto-

matic protocol design and implementation. The user simply needs to input the desired security properties and system requirements. Then Athena will automatically find an efficient protocol satisfying the requirements and output an implementation in Java source code. Our experiments demonstrate that this approach is efficient and viable, yielding protocols and implementations of higher security and efficiency.

6.2 Future Work

The Athena system has demonstrated an automatic approach for security protocol generation, verification and implementation. However, the scope of security protocols that the Athena system currently considers is limited. It will be valuable to explore how to expand the scope, including extending the set of cryptographic primitives currently covered in the Athena system as well as considering other families of security protocols. For example, we could extend the set of cryptographic primitives to include hash functions and message authentication codes. Such an extension can be done by extending the proof system with new proof rules in the Athena protocol verifier and expanding the protocol search space with the new primitives in the Athena protocol generator.

One interesting family of security protocols to consider is group communication protocols. In group communication protocols, the number of participants in

a protocol run is not fixed in the protocol specification, but rather a parameter that is instantiated during the protocol execution. Therefore, the analysis needs an extra step of abstraction or induction to handle the parameterized number of participants. For some recent work in this direction, see [64].

Another interesting family of security protocols to consider is zero-knowledge proof protocols. These protocols are generally based on number theoretic assumptions and often enjoy a modular design—complicated zero-knowledge proof protocols are often constructed using simpler zero-knowledge proof protocols as building blocks. It would be interesting to formalize the number theoretic assumptions and analyze zero-knowledge proof protocols automatically. The modular design of complex zero-knowledge proof protocols also suggests that it may be possible to automatically generate zero-knowledge proof protocols using simpler zero-knowledge proof protocols as building blocks. However, the automatic analysis and generation of zero-knowledge proof protocols may require quite a different framework from previous approaches.

Another natural extension of the Athena system is to analyze the application programming interfaces (APIs) used by secure co-processors. A *secure co-processor* is a computing environment that is tamper resistant. A recent family of attacks on secure co-processors shows that by presenting valid commands to the secure co-processor, but in an unexpected sequence, it is possible to obtain results that break the security policy envisioned by its designer [10]. Such attacks are economically

important, as secure co-processors are used to support a wide range of services: automatic teller machines, pay-TV, prepayment utility metering with low-end secure co-processors, and piracy suppression and distributed commerce applications with high-end secure co-processors [93, 94, 96, 97]. Designing APIs that resist such attacks is difficult, as a typical secure co-processor needs a substantial command set with several dozen commands that allow it to service a number of external and internal protocols. Even if each individual function is implemented correctly, it can only guarantee the local security property of a single function. But the interactions of different functions may still break the global security property. It would be interesting to extend the Athena system to analyze some of these complex APIs.

Most of the formal analysis of security protocols are based on the Dolev-Yao model [29], where cryptographic primitives are abstracted away and represented in idealistic forms. A separate line of work, initiated by Bellare and Rogaway, analyzes the way specific cryptographic primitives are used in protocols with concrete computational complexity assumptions on the security of the primitives. This approach gives asymptotic bounds on the risk of failures of secrecy or authentication [7, 6]. Unfortunately this analysis so far is done through manual analysis and proofs. An interesting challenge is to investigate how to combine the formal methods view, which has rich experience in automatic analysis techniques, and the concrete computational complexity security view, which reasons about security under computational complexity assumptions. For examples of recent research in this

direction, see [37, 53, 3, 82].

The Athena protocol generator uses powerful pruning techniques to reduce the protocol search space. However, currently it is a one-pass process: the protocol generator generates candidate protocols and passes them to the protocol verifier, without a feedback loop from the protocol verifier to the protocol generator. An interesting approach to explore is for the protocol verifier to pass hints to the protocol generator to extend the pruning rules or to even try to automatically fix flawed protocols. Another interesting approach to speed up protocol generation is to generate smaller sub-protocols, satisfying a subset of the requirements, and combining correct sub-protocols to form the full protocol.

Appendix A

Description on How to Use the Athena System

In this chapter, we describe how to use the APV, APG, and ACG system. The whole software package can be downloaded from `http://www.ece.cmu.edu/~dawnsong/athena`.

A.1 How to Use the APV System

To use the APV system, the user first starts SML, and then runs the command "use shell.sml", and then the command "use athena.sml". Before running athena.sml, the user needs to write the correct input file name in the last line of the file athena.sml. In the rest of this section, we first describe the input specification

language of the APV system, and then the output of the APV system.

A.1.1 Input Language of the APV System

To use the APV system, the user first specifies in an input file the security protocols and desired security properties in the APV specification language. The user then executes the APV program with the input file. When the program terminates, it will either generate a proof of correctness if the protocol satisfies the given security requirements, or generate a counter-example otherwise. In this section, we describe the APV specification language. Most of the syntax constraints are due to the limitation of the parser and can be eliminated by extending the parser. The syntax of the input language is shown in Table A.1 and Table A.2. Note that by convention of the BNF form, optional items are enclosed in meta symbols "`[`" and "`]`", and repetitive items (zero or more times) are enclosed in meta symbols "{" and "}".

The input file starts with the string "`BEGIN`" and ends with the string "`END`". Between the strings "`BEGIN`" and "`END`", the user can specify one or more protocols – the specification language supports batch processing mode. Each protocol specification starts with a *Header* whose syntax is described in Table A.2. The two *Ints* in the *Header* do not affect the verification process in any way. They are present only as a convenience for the user in batch mode. For example, they can be used to denote the numbering of the protocols in batch mode. The *String* in the *Header*

Digit	::=	0 | . . . | 9
Int	::=	*Digit*$^+$
Letter	::=	A | . . . | Z | a | . . . | z
Dletter	::=	*Digit* | *Letter*
String	::=	*Letter*$^+$
Dstring	::=	*Dletter*$^+$
Principal	::=	"`P_`"*String*
Nonce	::=	"`NONCE_`"*String*
PublicKey	::=	"`PUBKEY_`"*Principal*
PrivateKey	::=	"`PRIVKEY_`"*Principal*
SymmetricKey	::=	"`SYMKEY_(`" *Principal* "`,`" *Principal* "`)`"
SessionKey	::=	"`SESKEY_`"*String*
Key	::=	*PublicKey* | *PrivateKey* | *SymmetricKey* | *SessionKey*
Encryption	::=	"`E{`" *Message* "`,`" *Key* "`}`"
Concatenation	::=	"`C[`" *Message* {"`,`" *Message*} "`]`"
Message	::=	*Principal* | *Nonce* | *Key* | *Encryption* | *Concatenation*

Table A.1: Syntax of the APV Input Language (Part I)

NewAtom ::= *Nonce* | *SessionKey*

NewClause ::= "`New(`" *NewAtom* {, *NewAtom*} "`)`"

Send ::= "`->`" *Message* ["`|`" *NewClause*] "`;`"

Recv ::= "`<-`" *Message* ["`|`" *NewClause*] "`;`"

Action ::= *Send* | *Recv*

Var ::= *Nonce* | *Principal* | *SessionKey*

VarDeclare ::= `VAR:` *Var* {"`,`" *Var*} "`;`"

PartyHeader ::= *Principal* "`(`" *Int* "`,`" *Int* "`)`"

PartyClause ::= *PartyHeader* "`{`" *VarDeclare*
{*Action*} "`}`"

Node ::= "`Strand(`" *Int* "`,`" *Int* "`)`"

Binding ::= "`(`" *Var* "`,`" *Dstring* "`)`"

LegiStrand ::= *Node* "`[`"*Binding* {"`,`" *Binding*} "`]`"

SecrecyStrand ::= "`Strand(24,1)[(ANY_secretX,`" *Dstring* "`]`"

AuthFormula ::= "`VC.` {" *LegiStrand* {"`,`" *LegiStrand*} "`}`"
"`=>`" "`{`" *LegiStrand* {"`,`" *LegiStrand*} "`}`"

SecrecyFormula ::= "`VC.` {" *LegiStrand* {"`,`" *LegiStrand*} "`,`" *SecrecyStrand*"`}`" "`=>`" "`|`"

Formula ::= *AuthFormula* | *SecrecyFormula*

Header ::= *Int* "`:`" *Int* "`Protocol:`" *String*

PrtclSpec ::= {*PartyClause*} "`End`"

PropertySpec ::= "`Spec`" "`{`" {*Formula*} "`}`"

SinglePrtcl ::= "`BEGIN`" *Header*
PrtclSpec
PropertySpec
"`END`"

is used for annotation, for example, to specify the name of the protocol. Each protocol specification contains two parts: the protocol description and the security property description. We first describe the syntax for the protocol description, *PrtclSpec*, and then the syntax for the property description, *PropertySpec*.

The protocol description *PrtclSpec* specifies the sequence of actions of each party in a separate clause, called a *PartyClause*. For each party, the clause begins with a *PartyHeader*. In the *PartyHeader*, the *String* specifies the name of the party, and the first *Int* is called the Role ID, which starts from 0 and increments by 1 after each PartyClause. For example, for the first party, this number will be 0, and for the second party, this number will be 1, and so on. The second *Int* is the total number of actions (or the total number of messages sent and received) by the party in the protocol. After the *PartyHeader* in a PartyClause is the *VarDeclare*, which specifies the list of local variables of the party for the protocol execution. After the variable declaration *VarDeclare* is a list of actions, each in the form *Action*. In a *Action*, "$->$" means sending a message, and "$<-$" means receiving a message. If a message contains a freshly generated nonce or session key, we add to the end of the message a structure *NewClause* to indicate that the nonce or the session key in the message is freshly generated.

A message can either be an atomic term or a term constructed using the concatenation or encryption operation. The representation for an atomic term is of the form "*String1_String2*". String1 is the type of the term; String2 is the name of the

variable, which is unique in the list of variables within a PartyClause. For example, `''P''` represents the type *Principal Name*, and `''NONCE''` represents the type *Nonce*; `P_A` represents the atomic term Principal Name A, and `NONCE_B` represents Nonce B. Note that the type information will be prepended to the value of the variables in the implementation of the protocol to prevent ambiguities and type flaws as described in Section 3.1.2. Currently, the keys are represented as uninterpreted functions of principal names. For example, `''PUBKEY_P_A''` represents the public key of the party P_A, `''PRIVKEY_P_A''` represents the private key of the party P_A, and `''SYMKEY(P_A,P_B)''` represents the symmetric key shared between P_A and P_B.[1] The syntax for encryption and concatenation is shown as *Encryption* and *Concatenation* in Table A.1.

The current parsing restriction is that the clause for the initiator needs to be the first clause and the principal name has to be P_A, the principal name for the second clause has to be P_B, and the principal name for the third clause has to be P_S. This restriction is only an artifact of a quickly-written parser and can be easily eliminated by extending the parser.

After the protocol description, *PrtclSpec*, is the property description, *PropertySpec*. The *PropertySpec* starts with the line "`Spec{`" and ends with the corresponding closing bracket "}". Within the clause, each security property is specified using an individual formula in the form of *Formula*. A *Formula* can be a *Au-*

[1] This approach does limit the number of keys that each principal can have. We can simply extend this to handle the case where each principal can have arbitrary number of keys by adding another parameter in the key name to distinguish different keys of the same principal.

thFormula which specifies an authentication property, or a *SecrecyFormula* which specifies a secrecy property. The syntax for each formula is similar to the syntax of formulas described in Section 3.4. Each formula starts with the string "`VC.`", which stands for $\forall C$. The form $u \Rightarrow v$ represents the formula $C \models u \Rightarrow v$, where u and v are lists of strands. Each strand is specified in the form `Strand`(*Int*,*Int*)[*Binding List*] where the first integer specifies the Role ID of the strand, the second integer specifies the length of the strand, and the *Binding List* is the list of parameter bindings. A parameter binding binds a local variable to a specified value. For example, `Strand(1,3)` represents a strand of Role ID 1 with length 3. We use "|" to represent the empty set. The secrecy property is specified in a slightly different form, SecrecyFormula. For example, the property that the attacker does not know the value y is specified in the form `Strand(24,1)[(ANY_secretX, y)] => |`.

We give the input file of the fixed version of the Needham-Schroeder protocol below as an example. The numbers at the beginning of each line indicate the line numbers for the convenience of explanation. They are not in the actual file.

```
1.  BEGIN
2.  1:1 Protocol: NeedhamSchroederAsym
3.    P_A(0,3){
4.    VAR: P_A, P_B, NONCE_Na, NONCE_Nb;

5.    -> : E{C[P_A,NONCE_Na],PUBKEY_P_B} | New(NONCE_Na);
```

```
6.    <- : E{C[P_B,NONCE_Na,NONCE_Nb],PUBKEY_P_A};
7.    -> : E{NONCE_Nb,PUBKEY_P_B};
8.   }

9.   P_B(1,3){
10.   VAR: P_A, P_B, NONCE_Nb, NONCE_Na;

11.   <- : E{C[P_A,NONCE_Na],PUBKEY_P_B};
12.   -> : E{C[P_B,NONCE_Na,NONCE_Nb],PUBKEY_P_A} | New(NONCE_Nb);
13.   <- : E{NONCE_Nb,PUBKEY_P_B};
14.   }

15.  End

16.  Spec {
17. VC.{Strand(1,3)[(P_A,A0),(P_B,B0),(NONCE_Nb,Nb0),(NONCE_Na,Na0)]}
18. =>{Strand(0,3)[(P_A,A0),(P_B,B0),(NONCE_Nb,Nb0),(NONCE_Na,Na0)]},
19. VC.{Strand(0,3)[(P_A,A0),(P_B,B0),(NONCE_Nb,Nb0),(NONCE_Na,Na0)]}
20.  =>{Strand(1,2)[(P_A,A0),(P_B,B0),(NONCE_Nb,Nb0),(NONCE_Na,Na0)]}
21.       }
22.  END
```

To remind the reader about the protocol, we give the description of the Needham-Schroeder protocol is below:

$$\begin{aligned} \textsf{Protocol}: \quad & A \rightarrow B : \{A, N_A\}_{K_B} \\ & B \rightarrow A : \{B, N_A, N_B\}_{K_A} \\ & A \rightarrow B : \{N_B\}_{K_B} \end{aligned}$$

We now go through each line of the example of the input file of the fixed version of the Needham-Schroeder protocol to explain the syntax:

- Each APV input file starts with the string "`BEGIN`".
- Line 2 is the *Header*, where the string "1:1" and "NeedhamSchroederAsym" are for the user's convenience and ignored by the APV system. "`Protocol:`" is a required token.
- Line 3 is the *PartyHeader* and starts the *PartyClause*, where "P_A" is the principal name of this *PartyClause*. The number 0 is the Role ID of P_A. The number 3 is the number of steps of P_A in the protocol specification.
- Line 4 is the *VarDeclare*. "`VAR:`" is the required symbol. P_A, P_B, NONCE_Na, and NONCE_Nb are the local variables.
- Line 5 is the action of sending the message E{C[P_A,NONCE_Na],PUBKEY_P_B}, where PUBKEY_P_B is the public key of P_B. Because NONCE_Na is a freshly generated nonce in

this message, the *NewClause* structure is appended after the message. Similarly, Line 6 represents the action of receiving the message E{C[P_B,NONCE_Na,NONCE_Nb],PUBKEY_P_A}, and Line 7 represents the action of sending the message E{NONCE_Nb,PUBKEY_P_B}.

- Line 9 is similar to line 3. "P_B" is the principal name of this *PartyClause*. The number 1 is the Role ID of P_B. The number 3 is the number of steps of P_B in the protocol specification.

- Line 10 is the *VarDeclare*, which declares the local variables of P_B. This line is similar to line 4.

- Line 11 through 13 represent the actions of P_B, similar to line 5 through 7.

- In line 15, "`End`" is a required symbol that marks the end of the *PrtclSpec*.

- In line 16, "`Spec`" is a required symbol that marks the beginning of the "*PropertySpec*".

- Line 17 and 18 is a *Formula* specifying the authentication property that for all possible protocol execution, if a strand of Role ID 1 has reached its third step with the parameter bindings that bind the local variable P_A to A0, P_B to B0, NONCE_Nb to Nb0, and NONCE_Na to Na0, then there must have existed a strand of Role ID 0 in the protocol execution which has reached its third step with the parameter bindings that bind the local variable P_A to A0,

P_B to B0, NONCE_Nb to Nb0, and NONCE_Na to Na0. Note that the values that these variables bind to can be any arbitrary unique strings except for the string "`I`".

- Line 19 and 20 is another formula, similar to line 17 and 18.
- In line 21, the *PropertySpec* is closed with "}".
- In line 22, "`END`" marks the end of the description of the Needham-Schroeder protocol specification including both the protocol specification and the property specification.

We give the input file of another example, the Yahalom protocol [20]. The protocol description is as follows:

$$\begin{aligned} \text{Yahalom}: \quad & A \rightarrow B : N_A, A \\ & B \rightarrow S : \{N_A, N_B, A\}_{K_{BS}}, B \\ & S \rightarrow A : \{SK_{AB}, A\}_{K_{BS}}, \{SK_{AB}, N_A, N_B, B\}_{K_{AS}} \\ & A \rightarrow B : \{SK_{AB}, A\}_{K_{BS}}, \{N_B\}_{SK_{AB}} \end{aligned}$$

The input file of the Yahalom Protocol is listed below.

```
1:1 Protocol: Yahalom
  P_A(0,3){
  VAR: P_A, P_B, P_S, NONCE_Na, ANY_X,  SESKEY_Kab, NONCE_Nb;
```

```
-> : C[P_A, NONCE_Na] | New(NONCE_Na);
<- : C[E{C[P_B,SESKEY_Kab,NONCE_Na,NONCE_Nb],
     SYMKEY_(P_A,P_S)},ANY_X];
-> : C[ANY_X,E{NONCE_Nb, SESKEY_Kab}];
}

P_B(1,3){
VAR: P_A, P_B, P_S,  SESKEY_Kab, NONCE_Nb, NONCE_Na;

<- : C[P_A, NONCE_Na];
-> : C[P_B,E{C[P_A,NONCE_Na,NONCE_Nb], SYMKEY_(P_B,P_S)}]
     | New(NONCE_Nb);
<- : C[E{C[P_A, SESKEY_Kab], SYMKEY_(P_B,P_S)},
     E{NONCE_Nb, SESKEY_Kab}];
}

P_S(2,2){
VAR: P_A, P_B, P_S,  SESKEY_Kab, NONCE_Nb, NONCE_Na;

<- : C[P_B, E{C[P_A,NONCE_Na,NONCE_Nb], SYMKEY_(P_B,P_S)}];
```

```
  -> : C[E{C[P_B,SESKEY_Kab,NONCE_Na, NONCE_Nb],
       SYMKEY_(P_A,P_S)},E{C[P_A,SESKEY_Kab],SYMKEY_(P_B,P_S)}]
       | New(SESKEY_Kab);
  }

End

Spec {
  VC. { Strand(0,3)[(P_S,S0),(P_B,B0),(P_A,A0),
                    (SESKEY_Kab,Q3),(NONCE_Na,Q5)] } => {
  Strand(2,2)[(P_S,S0),(P_B,B0),(P_A,A0),(SESKEY_Kab,Q3),
              (NONCE_Na,Q5)] },
  VC. { Strand(0,3)[(P_S,S0),(P_B,I),(P_A,A0),(SESKEY_Kab,Q3),
                    (NONCE_Na,Q5)] } => {
  Strand(2,2)[(P_S,S0),(P_B,I),(P_A,A0),(SESKEY_Kab,Q3),
              (NONCE_Na,Q5)] }
  ,
  VC. { Strand(1,3)[(P_S,S0),(P_B,B0),(P_A,A0),
                    (SESKEY_Kab,Q3),(NONCE_Nb,Q4)] } => {
  Strand(2,2)[(P_S,S0),(P_B,B0),(P_A,A0),
              (SESKEY_Kab,Q3),(NONCE_Nb,Q4)] }
```

```
}
END
```

A.1.2 Output of the APV System

In this section, we describe the output of the APV system. If we run the APV system with the above input file of the Needham-Schroeder protocol example, the APV system outputs the following lines indicating that the protocol satisfies both two specified security properties:

```
 checking spec < 1 > :
*******true
checking spec < 2 > :
*******true
```

As the above example shows, when a protocol is verified to be correct by APV, the output is short. But what happens when the protocol is flawed? Then the output is far lengthier. When we analyze the original Needham-Schroeder protocol, APV outputs the following data showing the counterexample:

```
 checking spec < 1 > :


----false
```

```
inStrand
state( sb(6, Strand (Sname 5, len: 3, Role 0)
::Strand (Sname 4, len: 3, Role 12)
::Strand (Sname 3, len: 3, Role 12)
::Strand (Sname 2, len: 3, Role 14)
::Strand (Sname 1, len: 3, Role 14)
::Strand (Sname 0, len: 3, Role 1)
),
goalset: ,

env: (Principals Pvar Strand(5).A, Principals PrincipalC A0)
:: (AnyV Strand(4).SSN#11, Concat []):::
(AnyV Strand(4).SSN#12, Concat []):::
(Nonce NonceV Strand(5).Nb, Nonce NonceC Nb0)::
(Principals Pvar Strand(5).B, Principals PrincipalC I)
:: (AnyV Strand(3).SSN#5, Principals PrincipalC A0)
:: (Nonce NonceV Strand(5).Na, Nonce NonceC Na0)
:: (AnyV Strand(3).SSN#6, Concat []):::
(AnyV Strand(4).Hd, Concat [AnyV Strand(4).SSN#11,
 Nonce NonceC Nb0, AnyV Strand(4).SSN#12])::
(Principals Pvar Strand(4).Ppub, Principals PrincipalC I)
```

```
:: (AnyV Strand(3).Hd, Concat [AnyV Strand(3).SSN#5,
 Nonce NonceC Na0, AnyV Strand(3).SSN#6])::
 (Principals Pvar Strand(3).Ppub, Principals
PrincipalC I):: (AnyV Strand(2).Hde, Nonce NonceC Nb0)
:: (Principals Pvar Strand(2).Ppube, Principals PrincipalC B0)
:: (AnyV Strand(1).Hde, Concat [Principals PrincipalC A0,
 Nonce NonceC Na0]):: (Principals Pvar Strand(1).Ppube,
 Principals PrincipalC B0):: (Nonce NonceV
Strand(0).Na, Nonce NonceC Na0):: (Nonce NonceV Strand(0).Nb,
Nonce NonceC Nb0):: (Principals Pvar Strand(0).B,
Principals PrincipalC B0):: (Principals Pvar Strand(0).A,
 Principals PrincipalC A0),
porderset: pset(Interm({Concat [Nonce NonceC Na0,
Nonce NonceC Nb0]}_(Pubkey (Pvar Strand(5).A)),
 Node (1, Sname 0)) ::Interm({Concat
[AnyV Strand(4).SSN#11, Nonce NonceC Nb0, AnyV
Strand(4).SSN#12]}_(Pubkey (PrincipalC I)), Node (2, Sname 5))
::Interm({Concat [AnyV Strand(3).SSN#5, Nonce NonceC Na0, AnyV
Strand(3).SSN#6]}_(Pubkey (PrincipalC I)), Node (0, Sname 5))
::Interm(Nonce NonceC Nb0, Node (2, Sname 4))
::Interm(Nonce NonceC Na0, Node (2, Sname 3))
```

```
::Interm({Nonce NonceC Nb0}_(Pubkey
(PrincipalC B0)), Node (2, Sname 2))
::Interm({Concat [Principals PrincipalC A0,
Nonce NonceC Na0]}_(Pubkey (PrincipalC B0)), Node (2,
Sname 1)),
Subterm(Nonce NonceV TempName (Na), Node (0, Sname 5))
::Subterm(Nonce NonceV TempName (Nb), Node (1, Sname 0)),
p->c(Node (1, Sname 0), Node (1, Sname 5))
::p->c(Node (2, Sname 5), Node (1, Sname 4))
::p->c(Node (1, Sname 5), Node (2, Sname 5))
::p->c(Node (0, Sname 5), Node (1, Sname 5))
::p->c(Node (0, Sname 5), Node (1, Sname 3))
::p->c(Node (2, Sname 4), Node (1, Sname 2))
::p->c(Node (1, Sname 4), Node (2, Sname 4))
::p->c(Node (0, Sname 4), Node (1, Sname 4))
::p->c(Node (1, Sname 0), Node (1, Sname 2))
::p->c(Node (2, Sname 3), Node (1, Sname 1))
::p->c(Node (1, Sname 3), Node (2, Sname 3))
::p->c(Node (0, Sname 3), Node (1, Sname 3))
::p->c(Node (2, Sname 2), Node (2, Sname 0))
::p->c(Node (1, Sname 2), Node (2, Sname 2))
```

```
::p->c(Node (0, Sname 2), Node (1, Sname 2))
::p->c(Node (2, Sname 1), Node (0, Sname 0))
::p->c(Node (1, Sname 1), Node (2, Sname 1))
::p->c(Node (0, Sname 1), Node (1, Sname 1))
::p->c(Node (1, Sname 0), Node (2, Sname 0))
::p->c(Node (0, Sname 0), Node (1, Sname 0))))
= false
a counter example!
```

Most of the output above gives internal information from the APV program and is used for debugging APV. We describe the part that users can use to find a counterexample to the specification. The output shows the information about the state where the specified security property failed. The information about the state is shown in the data structure starting with the string "`state(`", called *StateInfo*. As described in Chapter 3, a state contains a list of regular strands[2] that represent the actions of legitimate parties in the protocol execution. The information about these strands is listed at the beginning of the *StateInfo*. Each strand is listed in the form "`Strand (Sname` *Int*, `len:` *Int*, `Role` *Int*)". The first *Int* gives a unique ID to the strand, called the *Strand ID*. The second *Int* is the length of the strand, i.e., the number of actions in the strand. The third *Int* is the Role ID described earlier in the APV input language. The user only needs to look at the strands with the Role

[2] A strand is a sequence of a party's actions.

ID below 10, which are the regular strands representing legitimate parties. Using the Role ID, the user can get the sequence of actions of the party from the protocol specification in the input file. For example, in the first line in the *StateInfo* in the above output example, "`Strand (Sname 5, len: 3, Role 0)`" is a strand with Strand ID 5, length 3, and Role ID 0. From the input file described earlier, we can get the *PartyClause* of Role ID 0 shown below, which specifies the sequence of actions of a party of Role ID 0:

```
P_A(0,3){
VAR: P_A, P_B, NONCE_Na, NONCE_Nb;

-> : E{C[P_A,NONCE_Na],PUBKEY_P_B} | New(NONCE_Na);
<- : E{C[P_B,NONCE_Na,NONCE_Nb],PUBKEY_P_A};
-> : E{NONCE_Nb,PUBKEY_P_B};
}
```

The last strand in the strand list is "`Strand (Sname 0, len: 3, Role 1)`", which is a strand of Strand ID 0, length 3, and Role ID 1. From the input file described earlier, we can get the *PartyClause* of Role ID 1 shown below, which specifies the sequence of actions of a party of Role ID 1:

```
P_B(1,3){
VAR: P_A, P_B, NONCE_Nb, NONCE_Na;
```

```
<- : E{C[P_A,NONCE_Na],PUBKEY_P_B};
-> : E{C[P_B,NONCE_Na,NONCE_Nb],PUBKEY_P_A} | New(NONCE_Nb);
<- : E{NONCE_Nb,PUBKEY_P_B};
}
```

The only information the user needs to reconstruct the counterexample is the parameter bindings of these regular strands. The parameter binding information is listed in the structure starting with the string "`env`", called *EnvInfo*, which is output after the list of strands in the *StateInfo*. Each parameter binding is represented as a pair consisting of a variable and the value the variable is bound to. Note that as we described in Chapter 3, each variable name is composed of the name of the strand to which the variable belongs and the local name of the variable. The strand name is in the form "Strand(*Int*)" where the *Int* represents the Strand ID. So to find the binding of a strand, the user can simply go through the parameter binding list in the *EnvInfo* data structure and get the parameter bindings of the corresponding Strand ID. For example, to get the parameter bindings of the strand of Strand ID 5, the user go through the *EnvInfo* to get all the pairs that contain variables belong to `Strand(5)`: (Principals Pvar Strand(5).A, Principals PrincipalC A0), (Nonce NonceV Strand(5).Nb, Nonce NonceC Nb0), (Principals Pvar Strand(5).B, Principals PrincipalC I), and (Nonce NonceV Strand(5).Na, Nonce NonceC Na0). Note that the string "`I`" is the attacker

name. From these parameter bindings, the user can construct the strand with Strand ID 5 as Strand(1,3)[(P_A,A0),(P_B,I),(NONCE_Nb,Nb0),(NONCE_Na,Na0)]. Similarly, the user can go through the *EnvInfo* to get the binding of the strand with Strand ID 0: Strand(0,3)[(P_A,A0),(P_B,B0),(NONCE_Nb,Nb0),(NONCE_Na,Na0)], The user can then construct the counterexample as described in Chapter refchapter:apv. This particular counterexample is explained earlier in Chapter 1. This counterexample shows that when the client A0 tries to open a session with a server `I` who is malicious, server `I` can then impersonate as A0 to another server B0.

A.2 How to Use the APG System

APG has a graphical user interface shown in Figure A.1 and does not require any command line arguments. To run it, simply use command "`java APG`".

APG can generate two-party and three-party security protocols. The user first selects the number of parties involved in the protocol by selecting the button "`Two Party`" or "`Three Party`". The button "`Four Party`" is for future extension of the tool and is currently not working. After selection of the number of parties, one "knowledge window" opens for each principal and for the adversary, listing all known terms of that principal.

The "Security Requirements" area of the APG GUI allows specification of the security requirements of the protocol to be generated. For authentication, APG

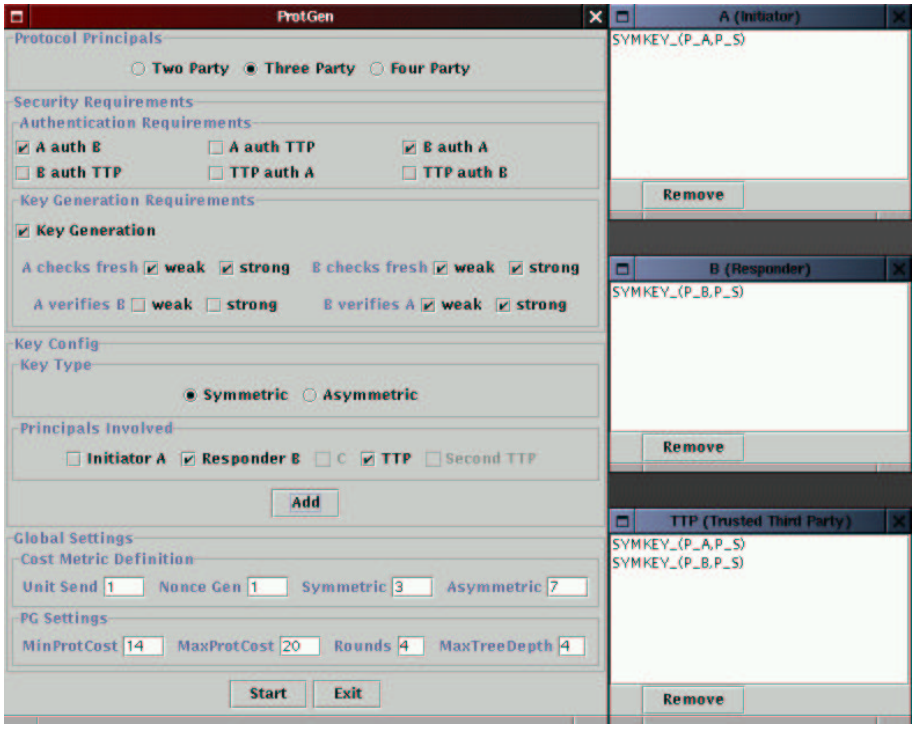


Figure A.1: APG GUI

supports ”A auth B”, meaning A authenticates B, ”B auth A”, ”A auth TTP”, meaning A authenticates TTP (Trusted Third Party), ”TTP auth A”, ”B auth TTP”, and ”TTP auth B”. In the two party case, only security requirements involving A and B will be valid.

APG supports key generation as well as authentication. When the user select the box for “Key Generation”, APG will generate candidate protocols that will perform session key distribution. It is assumed that the key is always generated by the TTP and then distributed to A and B in the three party case. In case key generation is enabled, the box ”Key Generation Requirements” lists the security requirements

for the generated key. "A checks fresh" means that after the protocol, principal A can verify that the agreed session key is indeed freshly generated. Similarly, "B checks fresh" is the same for principal B. "A verifies B" and "B verifies A" are for key confirmation, where the former means that principal A is assured after the protocol that principal B knows the correct session key that A has agreed to.

The "Key Config" box is used to set up the initial symmetric or asymmetric keys of the principals. For symmetric keys, it is assumed that always two parties share one key. To set up an initial symmetric key between two parties, the user first selects the two principals and then click on the "Add" button, a shared key will then be added to the two principal's knowledge windows. To remove the key, one can simply select the key in the principal's knowledge window and click on the "Remove" button. For adding asymmetric keys, only one principal should be selected. After selecting a principal, the user clicks on the "Add" button and a public/private key pair is added to the principal's knowledge window, and the public key will be added to all other principals's knowledge windows. Note that the attacker also gets the public key.

The "Global Settings" box is used to set various parameters of APG. In the "Cost Metric Definition" box, one can specify the cost metric of each atomic action, such as sending or receiving a message, nonce generation, symmetric-key and asymmetric-key encryption/decryption. The "MinProtCost" and "MaxProtCost" settings specify the cost range of the generated protocols. APG will not output

any protocols cheaper than MinProtCost, and will not output any protocols more expensive than MaxProtCost. Attention: the number of protocols APG needs to analyze is exponential in the value of MaxProtCost, using too large values risks non-termination within a reasonable time period. The ”Rounds” setting specifies how many rounds the generated protocol will have. Finally, the ”MaxTreeDepth” setting specifies the maximum depth for a message tree. As I discuss in my thesis, each message can be represented as a tree, this setting allows limiting the message complexity by limiting the message tree depth.

Once everything is specified, use ”Start” to begin protocol generation. APG will output the generated protocols in a file with the name Out<MinProtCost>_<MaxProtCost> _ <Rounds> _ <MaxTreeDepth> _ <time>.prot. The statistics about the protocol generation are output to the file Out<MinProtCost> _ <MaxProtCost> _ <Rounds> _ <MaxTreeDepth> _ <time>.data. For the settings MinProtCost=9, MaxProtCost=15, Rounds=3, MaxTreeDepth=4, the output files at time 949816487337 would be Out9_15_4_4_949816487337.prot and Out9_15_4_4_949816487337.data.

A.3 How to Use the ACG System

ACG requires two files as input (file1 and file2), both containing a description of the protocols to be generated. To use ACG, simply run with the two files as

command-line arguments: "`java acg.test.GenerateTest` file1 file2".

File1 contains the protocol in the format of the Automatic Protocol Verifier (APV), for example:

```
3:19 Protocol: temp
 P_A(0,3) { VAR: P_B,P_A,NONCE_B1,NONCE_A1;
  -> : NONCE_A1 | New(NONCE_A1);
  <- : E{C[NONCE_A1,NONCE_B1,P_A],SYMKEY_(P_A,P_B)};
  -> : NONCE_B1;
 }
 P_B(1,3) { VAR: P_B,P_A,NONCE_B1,NONCE_A1;
  <- : NONCE_A1;
  -> : E{C[NONCE_A1,NONCE_B1,P_A],SYMKEY_(P_A,P_B)} | New(NONCE_B1);
  <- : NONCE_B1;
 }
End
Spec {
  VC. { Strand(0,3)[(P_S,S0),(P_B,B0),(P_A,A0),
        (SESKEY_KAB,Q3),(NONCE_A3,Q5)] } => {
        Strand(2,2)[(P_S,S0),(P_B,B0),(P_A,A0),
        (SESKEY_KAB,Q3),(NONCE_A3,Q5)] }}
```

```
File2 contains the protocol in condensed description, for example:
\begin{verbatim}
  3:2
  19 Protocol: VAR: P_B,P_A,NONCE_B1,NONCE_A1;
   P_A->P_B:C[NONCE_A1,P_A] | New(NONCE_A1);
   P_B->P_A:E{C[NONCE_A1,NONCE_B1,P_A],SYMKEY_(P_A,P_B)} | New(NONCE_B1
   P_A->P_B:NONCE_B1;
```

ACG creates one java source file for each principal that participates in the protocol. In the above example, if ACG is called with file1 and file2, ACG generates the files _3_19_tempP_A.java and _3_19_tempP_B.java. The former one is for Principal A and the latter for Principal B. For a detailed example on how to use the source code generated by ACG, see Chapter 5.

Bibliography

[1] http://www.research.att.com/sw/tools/graphviz/.

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

[3] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). In *IFIP International Conference on Theoretical Computer Science (IFIP TCS2000)*, Lecture Notes in Computer Science. Springer-Verlag, August 2000.

[4] Jee Hea An and M. Bellare. Does encryption with redundancy provide authenticity? In *Advances in Cryptology – EUROCRYPT '2001*, volume 2045 of *Lecture Notes in Computer Science*, pages 509–524. Springer-Verlag, 2001.

[5] M. Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology – ASIACRYPT '2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 531–545. International Association for Cryptologic Research, Springer-Verlag, 2000.

[6] M. Bellare and P. Rogaway. Entity authentication and key distribution. In *Advances in Cryptology – CRYPTO '93*, volume 773 of *Lecture Notes in Computer Science*, pages 232–249. International Association for Cryptologic Research, Springer-Verlag, 1994.

[7] M. Bellare and P. Rogaway. Provably secure session key distribution — the three party case. In *Proceedings of the 27th Annual Symposium on Theory of Computing (STOC)*, pages 57–66. ACM Press, May 1995.

[8] Dominique Bolignano. An approach to the formal verification of cryptographic protocols. In *Proceedings of the 3rd ACM Conference on Computer and Communications Security*, pages 106–118. ACM Press, March 1996.

[9] Dominique Bolignano. Towards the formal verification of electronic commerce protocols. In *10th IEEE Computer Security Foundations Workshop*, pages 133–146. IEEE Computer Society Press, 1997.

[10] M. Bond and R. Anderson. API-level attacks on embedded systems. *IEEE Computer*, 34(10):67–75, October 2001. Special Issue on Cryptographic Hardware and Embedded Systems.

[11] S. Brackin. Automatic formal analyses of cryptographic protocols. In *Proceedings of the 19th National Conference on Information Systems Security*, 1996.

[12] S. Brackin. Automatic formal analyses of two large commercial protocols.

In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, September 1997.

[13] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. Technical Report 39, DEC System Reearch Center, February 1990. revised version, appeared also as [14].

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

[15] R. Canetti and H. Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In *Advances in Cryptology – EUROCRYPT '2001*, volume 2045 of *Lecture Notes in Computer Science*, pages 451–472. Springer-Verlag, 2001.

[16] CCITT. The Directory-Authentication Framework, Version 7, 1987. Gloucester.

[17] CERT. CERT Advisory CA-2000-18: Flaws in PGP 5.0 key generation. Technical report, Computer Emergency Response Team, May 2000.

[18] CERT. CERT Advisory CA-2000-18: PGP may encrypt data with unauthorized ADKs. Technical report, Computer Emergency Response Team, August 2000.

[19] I. Cervesato, N.A. Durgin, P.D. Lincoln, J.C. Mitchelland, and A. Scedrov. A

meta-notation for protocol analysis. In *12th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, June 1999.

[20] J. Clark and J. L. Jacob. A survey of authentication protocol literature. Version 1.0, University of York, Department of Computer Science, November 1997.

[21] J. Clark and J. L. Jacob. Searching for a solution: Engineering tradeoffs and the evolution of provably secure protocols. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press, May 2000.

[22] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[23] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[24] E. Clarke, S. Jha, and W. Marrero. Using partial order and symmetry reductions in security protocol verification. Unpublished, 1999.

[25] 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.

[26] E. M. Clarke, S. Jha, and W. Marrero. Partial order reductions for security

protocol verification. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2000.

[27] E. Cohen. A first-order verifier for cryptographic protocols. In *Proceedings of the Computer Security Foundation Workshop*. IEEE Computer Society Press, 2000.

[28] Don Davis and R. Swick. Workstation services and Kerberos authentication at project athena. Technical Report Laboratory of Computer Science LCS, Massachusetts Institute of Technology MIT, 545 Technology Sq., Cambridge MA, 02139TM-424, MIT, February 1990.

[29] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, March 1989.

[30] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, March 1989.

[31] B. Donovan, P. Norris, and G. Lowe. Analyzing a library of security protocols using Casper and FDR. In *Workshop on Formal Methods and Security Protocols*, 1999.

[32] E. Emerson. *Handbook of Theoretical Computer Science*, volume B. MIT Press, 1990.

[33] S. Goldwasser and M. Bellare. Lecture notes on cryptography. `http://www.cs.ucsd.edu/users/mihir/papers/gb.html`.

[34] L. Gong, R. Needham, and R. Yahalom. Reasoning about belief in cryptographic protocols. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 234–248, May 1990.

[35] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[36] J. D. Guttman and F. J. Thayer Fábrega. Authentication tests and the structure of bundles. *Theoretical Computer Science*, 2001.

[37] J. D. Guttman, F. J. Thayer Fábrega, and L. D. Zuck. The faithfulness of abstract protocol analysis: Message authentication. In *Proceedings of the 8th ACM Conference on Computer and Communications Security*. ACM Press, November 2001.

[38] J. D. Guttman and F. J. Thayer. Authentication tests. In *Security and Privacy Symposium'00*, 2000.

[39] M. R. Hansen and H. Rischel. *Introduction to Programming using SML*. Addison-Wesley, 1999.

[40] J. Heather, G. Lowe, and S. Schneider. How to prevent type flaw attacks on

security protocols. In *Computer Security Foundations Workshop, CSFW 13*, July 2000.

[41] N. Heintze and J. D. Tygar. A model for secure protocols and their compositions. *IEEE Transactions on Software Engineering*, 22(1):16–30, January 1996.

[42] N. Heintze, J. D. Tygar, J. Wing, and H. Wong. Model checking electronic commerce protocols. In *Proceedings of the USENIX 1996 Workshop on Electronic Commerce*, pages 146–164, 1996.

[43] N. Heintze, J.D. Tygar, J. Wing, and H.C. Wong. Model checking electronic commerce protocols. In *Proceedings of the 2nd USENIX Workshop on Electronic Commerce*, pages 147–164. USENIX, November 1996.

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

[45] IEEE Computer Society, Technical Committee on Security and Privacy. *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, CA, April 1987. IEEE Computer Society Press.

[46] International Standards Organization. *Information Technology - Security techniques — Entity Authentication Mechanisms Part 3: Entity authentication using symmetric techniques*, 1993. ISO/IEC 9798.

[47] R. Kailar. Accountability in electronic commerce protocols. *IEEE Transactions on Software Engineering*, 22(5), May 1996.

[48] R. Kemmerer. Analyzing encryption protocols using formal verification techniques. *IEEE Journal on Selected Areas in Communication*, 7(4), 1989.

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

[50] D. Kindred. *Theory Generation for Security Protocols*. PhD thesis, Carnegie Mellon University, 1999.

[51] D. Kindred and J. M. Wing. Fast, automatic checking of security protocols. In *Proceedings of the 2nd USENIX Workshop on Electronic Commerce*, pages 41–52. USENIX, November 1996.

[52] SRI International Computer Science Laboratory. PVS web site. `http://pvs.csl.sri.com`.

[53] P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 112–121. ACM Press, November 1998.

[54] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, 1996.

[55] 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.

[56] G. Lowe. Towards a completeness result for model checking security protocols. *Journal of Computer Security*, 1999.

[57] C. Meadows. Using narrowing in the analysis of key management protocols. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 138–147. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press, May 1989.

[58] C. Meadows. A system for the specification and analysis of key management protocols. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 182–195. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press, May 1991.

[59] C. Meadows. Applying formal methods to the analysis of a key management protocol. *Journal of Computer Security*, 1(1):5–35, 1992.

[60] C. Meadows. A model of computation for the NRL protocol analyzer. In *Proceedings of the 1994 Computer Security Foundations Workshop*, June 1994.

[61] C. Meadows. Formal verification of cryptographic protocols: A survey. In

Advances in Cryptology - Asiacrypt '94, volume 917 of *Lecture Notes in Computer Science*, pages 133–150. Springer-Verlag, 1995.

[62] C. Meadows. Analyzing the Needham-Schroeder public key protocol: A comparison of two approaches. In *Proceedings of ESORICS*. Springer-Verlag, 1996.

[63] C. Meadows. Analysis of the Internet Key Exchange protocol using the NRL protocol analyzer. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 48–61. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press, May 1999.

[64] C. Meadows. Extending formal cryptographic protocol analysis techniques for group protocols and low-level cryptographic primitives. In *Workshop on Issues in the Theory of Security (WITS'00)*, July 2000.

[65] D. Micciancio and B. Warinschi. Completeness theorems for the abadi-rogaway logic of encrypted expressions. In *Workshop on Issues in the Theory of Security*, 2002.

[66] J. Millen. The Interrogator model. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 251–260, 1995.

[67] Jonathan K. Millen, Sidney C. Clark, and Sheryl B. Freedman. The interrogator: Protocol security analysis. *IEEE Transactions on Software Engineering*, 13(2):274–288, February 1987.

[68] 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*, 1997.

[69] J.C. Mitchell, V. Shmatikov, and U. Stern. Finite-state analysis of SSL 3.0. In *Proceedings of the 7th USENIX Security Symposium*. USENIX, January 1998.

[70] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978. Also Xerox Research Report, CSL-78-4, PARC.

[71] D. Nessett. A critique of the burrows, abadi, and needham logic. *ACM Operating Systems Review*, 24(2):35–38, 1990.

[72] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *Eleventh International Conference on AUtomated Deduction (CADE)*, volume 607, pages 748–752. Springer-Verlag, 1993.

[73] L. Paulson. Isabelle: A generic theorem prover. In *Lecture Notes in Computer Science*, volume 828. Springer-Verlag, 1994.

[74] 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.

[75] Lawrence C. Paulson. Inductive analysis of the internet protocol TLS. *ACM Transactions on Information and System Security*, 2(3):332–351, August 1999.

[76] D. Peled. All from one, one for all: on model checking using representatives. In *Proceedings of the Fifth Workshop on Computer-Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag, June 1993.

[77] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Proceedings of the Sixth Workshop on Computer-Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 377–390. Springer-Verlag, June 1994.

[78] A. Perrig and D. Song. A first step towards the automatic generation of security protocols. In *Proceedings of Symposium on Network and Distributed Systems Security (NDSS '00)*, pages 73–83, February 2000.

[79] A. Perrig and D. Song. A first step towards the automatic generation of security protocols. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS '00)*, pages 73–83, February 2000.

[80] A. Perrig and D. Song. Looking for diamonds in the desert — extending automatic protocol generation to three-party authentication and key agreement protocols. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 64–76, July 2000.

[81] A. Perrig and D. Song. Looking for diamonds in the desert — extending automatic protocol generation to three-party authentication and key agreement protocols. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 64–76, July 2000.

[82] B. Pfitzmann and M. Waidner. Composition and integrity preservation of secure reactive systems. In *7th ACM Conference on Computer and Communications Security*, number 245-254. ACM Press, 2000.

[83] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[84] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence, 1995.

[85] Vitaly Shmatikov and J. Mitchell. Analysis of a fair exchange protocol. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS '00)*, pages 119–128. Internet Society, February 2000.

[86] Vitaly Shmatikov and Ulrich Stern. Efficient finite-state analysis for large security protocols. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, 1998.

[87] D. Song, S. Berezin, and A. Perrig. Athena, a novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9(1):47–74, 2001.

[88] D. Song, A. Perrig, and D. Phan. Agvi — automatic generation, verification, and implementation of security protocols. In *Proceedings of 13th Conference on Computer Aided Verification CAV 2001*, pages 241–245, July 2001.

[89] Dawn Song. Athena: An automatic checker for security protocol analysis. In *Proceedings of the 12th Computer Science Foundation Workshop*, 1999.

[90] P. Syverson and C. Meadows. Formal requirements for key distribution protocols. In *Advances in Cryptology – EUROCRYPT '94, Lecture Notes in Computer Science*, volume 950, 1995.

[91] F. Javier Thayer Fábrega, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(2,3):191–230, 1999.

[92] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS '00)*, pages 3–17, February 2000.

[93] S.H. Weingart. Physical security for the microabyss system. In *Proceedings of the IEEE Symposium on Research in Security and Privacy* [45].

[94] S.R. White and L. Comerford. Abyss: A trusted architecture for software protection. In *Proceedings of the IEEE Symposium on Research in Security and Privacy* [45].

[95] 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.

[96] B. Yee. *Using Secure Coprocessors*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1994. CMU-CS-94-149.

[97] B. Yee and J. D. Tygar. Secure coprocessors in electronic commerce applications. In *Proceedings of the First USENIX Workshop on Electronic Commerce*. USENIX, July 1995.

[98] P. Zimmermann. *The Official PGP User's Guide*. MIT Press, 1995. ISBN 0-262-74017-6.