Doctoral Dissertation

Formal verification with algebraic techniques and its application

Tran Dinh Duong

Supervisor: Kazuhiro Ogata

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
[Information Science]

September, 2023

Abstract

Formal verification has been extensively used to analyze various kinds of systems, such as verifying cryptographic protocols with security properties and mutual exclusion protocols with mutex properties. It is known as a unique approach in guaranteeing the absence of bugs (undesirable properties) in such systems. This approach formally describes the system under verification as a mathematical model using a dedicated language. The obtained result is called the formal specification of the system. Once the desired properties are specified with respect to the specification, formal verification that the system satisfies the properties can be conducted. There are two complementary approaches in formal verification: model checking and theorem proving. The former can be automatically conducted but cannot be used for systems that have an infinite number of states (infinite-state systems) in general due to the state explosion problem. The latter can deal with infinite-state systems but it requires human creativity, especially in lemma conjecture.

This thesis presents a formal verification approach with the employment of an algebraic specification language, namely CafeOBJ, equipped with an interactive theorem proving system, applied to verify the requirement properties of systems. We propose an approach and implement a supporting tool, namely IPSG, that can automatically generate formal proofs, the so-called proof scores, for formal verification of invariant properties. The algebraic specification language CafeOBJ is equipped with a rich specification syntax and many useful features for formal specifications of even complex systems, such as concurrent systems and distributed systems. It can be used as a powerful interactive theorem proving system, where humans can write a proof score to verify a desired property. However, writing proof scores is time- and effort-consuming, especially with complicated systems or specifications, and proof scores manually written are subject to human errors because they are user-defined, while CafeOBJ does not check their correctness. That is the reason motivating us to automate the proof score writing process and implement the tool. To demonstrate the efficiency and the practicability of the tool, experiments with various systems/protocols are conducted, ranging from a classical key distribution protocol to authentication protocols, from a real-time system to mutual exclusion protocols, and from a distributed protocol to real cryptographic protocols currently in use.

In recent years, advanced research in the field of quantum computing and quantum information theory has brought a credible threat to cryptosystems currently in use. The most popular public-key (or asymmetric) primitives used today will no longer be secure under sufficient strong quantum computers because they can be efficiently broken by Shor's algorithm. That motivates cryptographers and security researchers to construct a new class of cryptographic protocols that

are resistant to quantum attacks, called post-quantum cryptographic protocols (PQCPs), and verify the security of those PQCPs. Therefore, it would be very useful and meaningful to apply formal verification techniques to PQCP security analysis. This thesis presents two security verification case studies with: (1) the Hybrid Post-Quantum Transport Layer Security Protocol (PQ TLS) and (2) the Hybrid Post-Quantum Secure Shell Transport Layer Protocol (PQ SSH). PQ TLS has been proposed by Amazon Web Services (AWS) as a quantum-resistant version of the TLS 1.2 protocol, which is one of the most crucial and extensively used cryptographic protocols. PQ SSH has been proposed as a quantum-resistant version of the SSH Transport Layer protocol, where AWS is also one of the authors. We formally verify that the two protocols enjoy the desired security properties claimed in their design specifications, such as *session key secrecy* and *forward secrecy*, by using IPSG to generate their proof scores. The formal verifications are achieved under a threat model with the presence of an active attacker who can control the network, with respect to an unbounded number of protocol participants and protocol executions. The attacker can break the security of classical key exchange algorithms presuming by utilizing the power of large quantum computers. Moreover, the threat model also assumes the compromises of all secret types, such as ephemeral secret keys and long-term private keys of honest principals.

In the PQ SSH verification case study, in addition to the formal verification of three properties, we point out a counterexample showing that the protocol does not enjoy the *authentication* property, although what we found does not affect the confidentiality of session keys shared between honest participants. We then propose to slightly revise the protocol by adding the identifiers of the client and the server into the exchange hash. After revising the CafeOBJ formal specification accordingly, we can formally verify that the improved protocol enjoys the *authentication* property as well as the three other properties.

Keywords: formal verification, theorem proving, proof scores, post-quantum cryptographic protocols, proof score generation, algebraic language, CafeOBJ, Maude, IPSG.

Acknowledgments

I would like to express my deep gratitude and appreciation to my super-nice supervisor, Professor Kazuhiro Ogata, for his constant guidance and support. Needless to say, without his kind instruction, it would have been impossible for me to complete my PhD. It is very lucky for me to have had such a good opportunity to work under his guidance. Besides scientific knowledge, in some other aspects, I also learned from him an ideal model of researchers that I want to pursue.

I got many useful comments from Associate Professor Nakamura Masaki, Associate Professor Ishii Daisuke, Professor Toshiaki Aoki, and Professor Tatsuhiro Tsuchiya to improve this study and to revise the thesis. Thus, I would like to sincerely thank them. I would like to express my deep appreciation to Associate Professor Santiago Escobar from Spain, the supervisor of my minor research project, for his comments to complete this study as well as his support in working with Maude and Maude-NPA. I also wish to say grateful thanks to Associate Professor Pham Ngoc Hung from Vietnam, my former advisor, for his useful advice to me despite that we are physically far away from each other.

I would like to express my sincere thanks to Shiba-sensei, who has been doing her best to teach me Japanese for the last three years despite my slow progress. There was a lot of fun, and I enjoyed the time we studied. Despite my slow progress, my Japanese is indeed improved after a long time of learning, from which I could enjoy my life in Japan better.

Since the first day when I came to Japan, the members of Ogata's lab gave me plenty of support in both scientific and non-scientific aspects. Besides, I also received a lot of help from many Vietnamese friends during my daily life in Japan, in particular, Van Anh, Minh Quan, and Dang Duy. With all of them, life here became much more fun. For those reasons, I want to sincerely thank them.

I am deeply indebted to my lovely wife, Le Thi Theu, and my family in Vietnam. Thanks so much to Theu for her love, encouraging me through the difficult times, and tolerating my fouls, to mention just a few. Thanks so much to my parents and my sister for all of the best things they have done for me, especially, the value of learning that my parents have always tried to teach me. I hope you understand that not only on this occasion but also at any time, I always want to say that I love all of you. I would also like to take this opportunity to thank one of my cousins, Tran Lien, for plenty of things she has done for me since I was a young boy. There are other family members that I am indebted to, but I cannot mention all of them here. I would not make it here and this study could not be completed without you. To Parents: I hope you are proud of me.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Formal verification has been extensively used to analyze various kinds of systems, such as verifying cryptographic protocols with security properties and mutual exclusion protocols with mutex properties. It is known as a unique approach in guaranteeing the absence of bugs (undesirable properties) in such systems. Making software systems trustworthy is indeed a crucial issue; otherwise, serious accidents could happen. We can name several serious accidents in history caused by flaws in software systems. For instance, on August 14, 2003, wide parts of the United States and Ontario province of Canada were blackouts because of a system failure of FirstEnergy, the company providing electricity in those regions. This accident led to a series of other interrupted services, such as telephone networks, water supply, and flight landing. Estimates of the total costs of that blackout accident in the United States ranged between 4 billion and 10 billion US dollars [63]. Tracking the blackout bug, a race condition bug in the software system of FirstEnergy was found as the original cause of the accident[1][2]. Another example is the ARIANE 5 space failure on June 4, 1996. The rocket veered off its flight path 37 seconds after launch and broke up right after that, resulting in a loss of about 370 million US dollars. A technical board was established to investigate the cause and concluded that the failure was due to inadequate protection against integer overflow errors in the flight control software [92]:

> This loss of information was due to specification and design errors in the software of the inertial reference system. The extensive reviews and tests carried out during the Ariane 5 Development Programme did not include adequate analysis and testing of the inertial reference system or of the complete flight control system, which could have detected the potential failure.

Program testing, although has long been used as one of the most successful approaches to bug detection in systems, is not applicable in this case to prove systems contain no bugs. That

[1] https://web.archive.org/web/20110610163731/http://www.securityfocus.com/news/8412
[2] https://web.archive.org/web/20110610224942/http://www.securityfocus.com/news/8016

statement was proclaimed by Dikjtra [54]:

> Program testing can be used to show the presence of bugs, but never to show their absence!

Formal verification, on the other hand, provides formal reasoning based on mathematics to show that a system acts as its design, i.e., all the desired properties are satisfied and there are no undesirable ones (bugs). The ability of formal verification in guaranteeing the absence of bugs in software systems and the importance of making software systems trustworthy are the general motivations of this study, which applies formal verification to enhance the reliability of software systems. Formal verification describes the system under verification as a mathematical model using a dedicated language. The obtained result is called the formal specification of the system. Once the desired properties are specified with respect to the specification, formal verification that the system satisfies the properties can be conducted. There are two complementary approaches in formal verification: model checking and theorem proving. The former can be automatically conducted but is inapplicable to systems that have an infinite number of states (infinite-state systems) in general. The latter can deal with infinite-state systems but it generally requires human creativity, especially in lemma conjecture.

The general objective of this study is a formal verification technique that can be used to verify the required properties of a large class of systems with less human effort. To this end, we propose an approach and implement a supporting tool that can automatically generate formal proofs for the formal verification of invariant properties. The verification process employs an algebraic specification language. The efficiency and practicability of the tool are demonstrated through various systems/protocols, such as classical security protocols, mutual exclusion protocols, a real-time system, and real cryptographic protocols currently in use. However, in the thesis, we choose to report in detail the application of the tool in the verifications of a class of systems, namely post-quantum cryptographic protocols, with their security properties. Post-quantum cryptographic protocols refer to those replacements of classical cryptographic protocols as an early precaution against future attacks from quantum computers. This has been motivated by the fact that the public-key cryptosystems used today will be no longer secure under large-scale quantum computers, which are promisingly becoming available in the near future. Two formal verification case studies are presented in detail with (1) the Hybrid Post-Quantum Transport Layer Security Protocol [36] and (2) the Hybrid Post-Quantum Secure Shell Transport Layer Protocol [84].

1.1 Formal verification

Model checking and theorem proving are two complementary approaches in formal verification. In the model checking approach, a model checker exhaustedly travels the whole state space to check whether or not a desired property is satisfied [13]. The verification approach is automatic. Moreover, in the case when the property does not hold, the model checker returns a counterexample, pointing out a system state in which the property is violated, which is helpful to find the corresponding bug. However, the exhaustive searching makes the approach in general cannot be used when the number of states in the state space is very huge or even infinite. The issue is known as the *state space explosion problem* [42]. Even though many techniques have been proposed to mitigate the problem in the past few decades, such as abstraction [40, 39, 78] and partial order reduction [41], the problem remains a challenge, restricting the scope of model checking application.

On the other hand, theorem proving can deal with infinite-state systems. In the theorem proving approach, the system under verification is modeled as a logical set of definitions. A property then can be verified if the theorem prover can derive the theorem specifying the property from those logical definitions. Theorem provers are classified into two types: *automated theorem provers* (ATPs) and *interactive theorem provers* (ITPs), or also called *proof assistants*. Mathematical theorems can be automatically proved by computer programs with ATPs, while ITPs often require human-computer collaboration to derive formal proofs. Some well-known ATPs are Vampire [89], SPASS [148], and E [132], while various ITPs can be listed, such as Isabelle/HOL [108], Coq [23], PVS [130], and ACL2 [85]. The applications of ATPs are restricted to only small classes of systems and theorems, in most cases, the involvement of human interactions is necessary as stated in [77]:

> Interactive proof is likely to be the only way to formalize most non-trivial theorems in mathematics or computer system correctness.

Theorem proving is suitable for complex systems, such as concurrent and distributed systems, where the nondeterministic behaviors make the number of states in the state space generated very huge. However, in most non-trivial proofs, human creativity is required, especially in conjecturing some auxiliary lemmas to complete the proof of the main theorem. Another drawback is that when the theorem prover cannot derive the proof, counterexamples are not automatically produced as in model checking. In that case, either the property does not hold or the human conducting the verification is not intelligent enough, for example, some more lemmas are needed.

Lamport has classified properties of an execution of a program, particularly concurrent and distributed programs, into two types: *safety properties* and *liveness properties* [91]. A safety property asserts that something bad does not happen during execution, while a liveness property

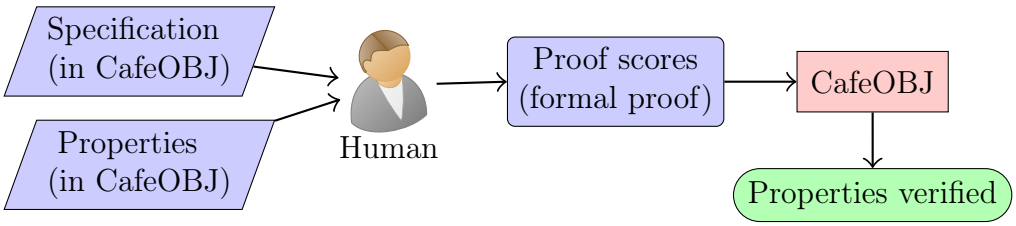


Figure 1.1: Formal verification by writing proof scores

asserts that something good eventually happens. It has been proved that all properties can be constructed using *safety properties* and *liveness properties* [7]. This thesis focuses on *invariant properties*, the most important class of *safety properties*. It is typically necessary to use *invariant properties* to prove *safety properties* or even *liveness properties*. An invariant property is a safety property but only takes the current state into its definition [13].

CafeOBJ [52] is an advanced algebraic language for writing formal specifications for a wide variety of systems and can be used as a powerful interactive theorem proving system. CafeOBJ is equipped with a rich specification syntax and many useful features for formal specifications of even complex systems, such as concurrent systems and distributed systems. Such features include module expressions, modules instantiated parameters using views, and the flexible mix-fix syntax, among others. It supports order-sorted equational logic with various equational theory attributes such as associative, commutativity, identity, and idempotency. CafeOBJ can be used as a powerful interactive theorem proving system, where humans are supposed to write a formal proof, a so-called proof score [116, 109], for an invariant property under verification. That proof score is executable, and the formal verification is done by executing it with CafeOBJ. The approach is visualized in Figure 1.1. The usefulness of the approach essentially comes from the power of the CafeOBJ language in specifying systems and its flexibility in writing proof scores. Using the approach, various formal verification case studies have been conducted analyzing many systems/protocols, such as the Mondex payment system [87], the *i*KP electronic payment protocol [111, 112], the OMA license choice algorithm [146], the Transport Layer Security 1.0 protocol [110], and the electronic commerce protocols [115].

The flexibility of writing proof scores, however, comes at a cost, that is the approach is subject to human errors. Proof scores consist of many user-defined open-close fragments (also called proof fragments), where each of them has one reduction command, which generally gives a term to be reduced to true or false. If each reduction command reduces to true as expected, which is supposed to be checked by human users, the formal verification concerned is done. Therefore, human users are responsible for the correctness of the proof. In particular, human users need to make sure that the proof covers all base/induction cases, the proof uses proper case splittings, and each open-close fragment uses proper premises of implications and/or lemmas. However, because an open-close fragment is user-defined, human users can, for example, unintentionally

add an extra equation, incorrectly write some equations, or overlook some open-close fragments. That is the reason why proof scores are said to be prone to human errors. Moreover, the task of writing proof scores is really time- and effort-consuming, especially with complicated systems or specifications. During the verification, there are many trivial sub-cases that are tedious to write the proof again and again.

This thesis proposes an approach to automation of the proof score writing process for formal verifications of invariant properties. A supporting tool, called IPSG (Invariant Proof Score Generator), is implemented, which can automatically generate the proof score of a given invariant property. By using the tool, human users only need to focus on solving non-trivial sub-cases, which normally require additional lemmas, but trivial sub-cases are already discharged by the tool. To demonstrate the efficiency and the practicability of the tool, we conduct experiments with various systems/protocols, ranging from a classical key distribution protocol to authentication protocols, from a real-time system to mutual exclusion protocols, and from a distributed protocol to real cryptographic protocols currently in use.

1.2 Post-quantum cryptographic protocols

Cryptographic protocols and security analysis

Cryptographic protocols are designed to provide information security, such as confidentiality and integrity, enabling two parties to securely communicate over insecure networks, such as the Internet. In our daily life, cryptographic protocols are used to protect bank transactions, online credit card payment systems, and radio-frequency identification, among others. They are crucial, but it is extremely hard to design a secure cryptographic protocol and it is challenging to detect flaws lurking in the design [58]. A very famous example illustrating this difficulty is the Needham-Schroeder public key authentication protocol (NSPK) [107]. Even though its authors, namely Needham and Schroeder, are security experts, the protocol was uncovered against an attack by Lowe [98] 17 years after its publication. The same thing has been happening with other cryptographic protocols, even very important ones used in practice, such as the Secure Sockets Layer protocol (SSL) and its successor Transport Layer Security protocol (TLS) [104]. Program testing unfortunately cannot be used to detect security flaws in cryptographic protocols because that technique lacks the presence of malicious participants. Therefore, techniques and supporting tools for the security analysis of cryptographic protocols are very crucial.

There are two complementary approaches to the security analysis of cryptographic protocols:

- Computational approach: This approach treats messages as bit strings and cryptographic primitives as functions from bit strings to bit strings. The attacker is an arbitrary probabilistic polynomial-time Turing machine. Computational security verification was studied

since the early 1980s [75, 150]. A security proof in the computational approach requires a definition of secure cryptographic construction (of a primitive or a protocol) and some assumptions about the computationally infeasible problem. The proof can be regarded as a mathematical reduction, such that it makes sure that the only chance to violate the security of such a construction is to solve the infeasible problem.

- Symbolic approach: This approach models messages as terms and cryptographic primitives as functions from terms to terms. A term is either a constant, a variable, or an application of a function symbol to a list of argument terms. Symbolic security verification was originally proposed by Dolev and Yao [56]. Since then, the threat model proposed in that work has been widely used as a de facto standard to model the attacker's capabilities, and it is often called the Dolev-Yao model. The Dolev-Yao attacker can completely control the network, including intercepting, deleting, and modifying messages in the network, gleaning information from such messages, and synthesizing information to build messages to send to others. Those capabilities are specified by manipulating terms representing messages exchanged in the network. Symbolic verification typically assumes perfect cryptography, treating cryptographic primitives as black boxes. Equational theory may be used to specify algebraic properties of cryptographic primitives.

The computational approach is widely used by cryptographers as a standard way to verify the security of cryptographic protocols, while formal method researchers typically prefer the symbolic approach. Blanchet [31, 30] has surveyed various techniques and supporting tools for security protocol verification in both symbolic and computational approaches. In comparison between the two approaches, a computational proof gives a tighter security guarantee because it takes probability and complexity into account. However, a computational proof is complicated in general and not easy to understand for non-experts in cryptography. Flawed proofs often happen and it is not so easy to check the correctness of a proof. Although it is possible to mechanize security proofs in the computational approach to some extent with some supporting tools, such as CryptoVerif [29] and EasyCrypt [17, 16], the techniques are still not mature and the proofs are not fully automated. On the other hand, a symbolic verification is easier to understand, computer-verified, and suitable for automation. Since the 1990s, symbolic verification of cryptographic protocols has been an attractive research direction in the applied formal method field. Based on different theory foundations, such as applied pi calculus [1], multiset rewriting [103], and narrowing & rewriting logic [68], many tools supporting for symbolic verification of cryptographic protocols have been developed, such as Syther [47], Tamarin [101], ProVerif [28], DEEPSEC [38], and Maude-NPA [64]. Some tools support an unbounded number of executions of the protocol. Whereas, some others limit the number of executions of the protocol and/or the size of messages in order to make the state space finite so that the standard model checking techniques can be

applied. DEEPSEC, for instance, is classified into this class.

Post-quantum cryptographic protocols

In recent years, advanced research in the field of quantum computing and quantum information theory has brought a credible threat to cryptosystems currently in use. The most popular public-key (or asymmetric) primitives used today will become insecure under sufficient strong quantum computers because they can be efficiently broken by Shor's algorithm [135]. The security of these primitives relies on one of the following three hard mathematical problems:

- The *integer factorization* problem: given a composite number K, find two integers M and N such that $M \cdot N = K$. When the composite number K is sufficiently large, no efficient integer factorization algorithm on classical computers is known. RSA public-key encryption and RSA digital signature [128] were invented based on the assumed difficulty of this problem.

- The *discrete logarithm* problem: given a multiplicative cyclic group G with the base b and an element a in the group, find the discrete logarithm x to the base b of a in the group G, i.e., $b^x = a$ with respect to G. With some selected group, no efficient algorithm on classical computers is known to solve the problem. Based on the presumed difficulty of the *discrete logarithm* problem, some cryptographic protocols are built, such as Diffie-Hellman key exchange protocol [53] and ElGamal public-key encryption scheme [74].

- The *elliptic curve discrete logarithm* problem: given an elliptic curve E and two points P and Q in E, find a number k such that $kP = Q$. This is a special case of the *discrete logarithm* problem. With some specific curves, no efficient algorithm to solve the problem on classical computers is known. Elliptic Curve Diffie-Hellman protocol [14] bases its security on the difficulty of this problem.

Unfortunately, although these three problems are hard under conventional computers, they can be efficiently solved by sufficiently large quantum computers. On the other hand, symmetric primitives can be said to be secure against quantum attackers. The most well-known quantum algorithm, namely Grover's algorithm [76], can reduce the complexity to break symmetric primitives to some extent, however, doubling the key size can efficiently ignore these attacks. For example, we can say that AES-256 would be as hard to break by a quantum computer as AES-128 is by a classical computer. Although right now there is no quantum computer with enough power to break the real cryptosystems currently used, with a huge research and development investment recently from many tech giants, such as Intel, IBM, and Google, large-scale quantum computers are promisingly becoming available in the near future. Besides, attackers can record the encrypted information from now and later decrypt it when large-scale quantum

computers become available, which is known as the *harvest now and decrypt later* attack. These are motivations for the early construction of cryptographic algorithms that are resistant to quantum attackers, the so-called post-quantum cryptographic algorithms, and security verification of them. Historically, it took some decades to deploy the modern public-key cryptosystems today. Therefore, it is crucial to start research on post-quantum cryptographic protocols now.

This thesis presents two security verification case studies with two protocols: (1) the Hybrid Post-Quantum Transport Layer Security Protocol [36] (PQ TLS) and (2) the Hybrid Post-Quantum Secure Shell Transport Layer Protocol [84] (PQ SSH). PQ TLS has been proposed by Amazon Web Services (AWS) as a quantum-resistant version of the TLS 1.2 protocol [123], which is one of the most crucial and extensively used cryptographic protocols. PQ SSH has been proposed as a quantum-resistant version of the SSH Transport Layer protocol [96], where AWS is also one of the authors. For each protocol, we conduct a formal analysis of the security properties claimed in its design specifications.

1.3 Contributions

In summary, in this study, we automate the proof score writing process with a supporting tool to formally verify the desired properties of systems. The tool is demonstrated to be applicable to various systems/protocols. In this thesis, we mainly focus on reporting the application of the tool in the verifications of post-quantum cryptographic protocols with security properties.

For the automation of the proof score writing process, our contributions are summarized as follows:

- An approach to the generation of proof scores for formal verifications of invariant properties.
- A supporting tool called IPSG. Given a CafeOBJ formal specification and an invariant property list, the tool can automatically produce the proof scores of those properties. By using the tool, not only human efforts are saved, but also potential human errors lurking in the proof scores can be avoided.
- To demonstrate the efficiency and the practicability of the tool, experiments are conducted with various systems/protocols, including classical security protocols, mutual exclusion protocols, a real-time system, a distributed protocol, and especially, a cryptographic protocol currently used in practice. For each verification experiment, we confirm the correctness of the generated proof scores by using the CafeOBJ proof assistant CiMPA and proof generator CiMPG [126].

- A lemma conjecture technique called Lemma Weakening, which can make the verification attempt of a protocol converge that otherwise does not seem to converge in a reasonable amount of time.

The source code of the tool is publicly available on the webpage[3] and the verifications of the protocols and systems are publicly available on the webpage[4].

With the PQ TLS verification case study, our contributions are summarized as follows:

- A comprehensive symbolic model of the protocol that is formally specified in CafeOBJ as two separate specifications: one for the case when client authentication is not requested and the other for the case when it does request client authentication. Each specification faithfully captures what is specified in the Internet Engineering Task Force (IETF) Draft [36] and covers both the full and abbreviated handshake modes.

- We formally prove that the protocol enjoys three desired security properties including *session key secrecy*, *forward secrecy*, and *authentication*, which are either claimed in the IETF Draft [36], or inherited from the original TLS 1.2 [123], but have not yet been formally proved. The formal verification is achieved under a threat model with the presence of an active attacker who can control the network. The attacker can break the classical key exchange algorithms, i.e., ECDH, by utilizing the power of large quantum computers. Moreover, the threat model also assumes the compromises of (1) symmetric handshake keys, (2) ECDHE secret keys, (3) PQ KEM secret keys, and (4) long-term private keys of honest principals. The verification is done by using IPSG to generate proof scores with respect to an unbounded number of protocol executions.

The protocol formal specification with detailed clarification, the proof scores, and the input requirements as well as the detailed guideline on how to generate the proof scores again are publicly available on the webpage[5].

With the PQ SSH verification case study, our contributions are summarized as follows:

- A symbolic model of the protocol that is formally specified in CafeOBJ as a specification, faithfully captures what is specified in the IETF Draft [84].

- Formal verifications that the protocol enjoys three desired security properties including (1) *session key secrecy*, (2) *forward secrecy*, and (3) *session identifier uniqueness*, where IPSG is used to generate the proof scores. The verifications are achieved with respect to an unbounded number of protocol participants and session executions.

[3]https://github.com/duongtd23/IPSG-tool
[4]https://github.com/duongtd23/IPSG-TLS
[5]https://github.com/duongtd23/PQTLS

- We consider another property, namely *authentication*, which we find a counterexample against the property. We propose to slightly revise the protocol by adding the identifiers of the client and the server into the exchange hash. We revise the CafeOBJ formal specification accordingly so that we can formally verify that the improved protocol enjoys the *authentication* property as well as (1), (2), and (3).

- For the threat model, more general transitions are used (than those of the PQ TLS case study) to specify the intruder's capabilities of learning information and forging messages, making sure that the intruder is given the full capability of forging an arbitrary message synthesized from the information that has been learned. Moreover, efforts can be saved when specifying other cryptographic protocols since those transitions may be reused.

The protocol formal specification, the proof scores, and other related materials used in this case study are available on the webpage[6].

1.4 Thesis organization

In this first chapter, we have briefly introduced formal verification with two major approaches, cryptographic protocols with two major security analysis approaches, and the necessitate of constructing post-quantum cryptographic protocols as well as verifying their security. The remaining of this thesis consists of six chapters, and each chapter is summarized as follows:

- **Chapter 2** gives some preliminaries, which are background requirements to understand the main content of this thesis, including the simultaneous induction proof method, the syntax of the CafeOBJ language in a nutshell, and the proof score writing approach to formal verification.

- **Chapter 3** presents how to automatically generate proof scores and the implementation of the tool IPSG. To demonstrate the efficiency and practicability of the tool, experiments with various systems/protocols are conducted and their experimental results are reported in this chapter.

- **Chapter 4** presents how to model and specify the PQ TLS protocol in CafeOBJ and the formal verification of the three desired properties of the protocol. IPSG is employed to automatically produce the proof scores to complete the verification and its experimental results are reported.

- **Chapter 5** presents the CafeOBJ formal specification of the PQ SSH protocol and the verification that the protocol enjoys the three desired properties with the employment of

[6]https://github.com/duongtd23/PQSSH

IPSG. This chapter also shows the counterexample that the protocol does not enjoy the *authentication* property, the improved protocol we proposed, and the formal verifications of the improved version.

- **Chapter 6** discusses some work closely related to ours, particularly some techniques and existing tools supporting formal verification with CafeOBJ, several state-of-the-art tools supporting cryptographic protocol analysis, and some case studies on security verification of post-quantum cryptographic protocols.
- **Chapter 7** summarizes our main contributions and mentions several lines of our future work.

Chapter 2

Preliminaries

This chapter first gives the definition of OTS and then describes the simultaneous induction proof method and the syntax of the CafeOBJ language in a nutshell. After that, with a simple protocol serving as a running example, we describe how to use CafeOBJ to specify the protocol and how to write proof scores based on the simultaneous induction proof method to formally verify a desired property of the protocol.

2.1 Observational Transition System (OTS)

We suppose that there exists a universal state space denoted by Υ, where each data type used in OTSs is provided. The data types include Bool for Boolean values. A data type is denoted by D, possibly with a subscript, such as D_{o1} and D_o.

Definition 1. An OTS $\mathcal{S}$ is a tuple $\langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$ in which:

- $\mathcal{O}$: A finite set of observers. Each *observer* $o : \Upsilon\ D_{o1}\ \ldots\ D_{om} \rightarrow D_o$ takes one state and m ($m \geq 0$) data values and returns one data value. The equivalence relation ($\upsilon_1 =_{\mathcal{S}} \upsilon_2$) between two states υ_1 and υ_2 is defined as $(\forall o \in \mathcal{O})(\forall x_1 \in D_{o1}) \ldots (\forall x_m \in D_{om}).(o(\upsilon_1, x_1, \ldots, x_m) = o(\upsilon_2, x_1, \ldots, x_m))$.
- $\mathcal{I}$: The set of initial states, where $\mathcal{I} \subseteq \Upsilon$.
- $\mathcal{T}$: A finite set of transitions. Each *transition* $t : \Upsilon\ D_{t1}\ \ldots\ D_{tn} \rightarrow \Upsilon$ takes one state and n ($n \geq 0$) data values, and returns one state. Each transition t has the *effective condition* $c\text{-}t : \Upsilon\ D_{t1}\ \ldots\ D_{tn} \rightarrow$ Bool. If $c\text{-}t(\upsilon, x_1, \ldots, x_n)$ does not hold, then $t(\upsilon, x_1, \ldots, x_n) =_{\mathcal{S}} \upsilon$ for $x_1 \in D_{t1}, \ldots, x_n \in D_{tn}$.

A pair (υ, υ') of states is called a *transition instance* if there exists $t \in \mathcal{T}$ such that $\upsilon' =_{\mathcal{S}} t(\upsilon, x_1, \ldots, x_n)$ for some $x_i \in D_{ti}$ for $i = 1, \ldots, n$. Such a pair (υ, υ') may be denoted by $\upsilon \rightarrow_{\mathcal{S}} \upsilon'$ (or $\upsilon \rightarrow \upsilon'$) to emphasize that υ directly goes to υ' by one step.

Each state that is reachable from an initial state through transitions is called a reachable state.

Definition 2. Given an OTS $\mathcal{S}$, *reachable states* with respect to (wrt) $\mathcal{S}$ are inductively defined:

- Each $\upsilon \in \mathcal{I}$ is reachable wrt $\mathcal{S}$.
- For each $t \in \mathcal{T}$ and each $x_k \in D_{tk}$ for $k = 1, \ldots, n$, $t(\upsilon, x_1, \ldots, x_n)$ is reachable wrt $\mathcal{S}$ if $\upsilon \in \Upsilon$ is reachable wrt $\mathcal{S}$.

Let $\mathcal{R}_{\mathcal{S}}$ be the set of all reachable states wrt $\mathcal{S}$.

Predicates whose types are $\Upsilon\ D_1\ \ldots\ D_n \rightarrow$ Bool are called *state predicates*. A state predicate that holds in all reachable states is called an *invariant*. For example, predicate $\rho : \Upsilon\ D_1\ \ldots\ D_n \rightarrow$ Bool, where $(\forall \upsilon \in \mathcal{R}_{\mathcal{S}})(\forall d_1 \in D_1) \ldots (\forall d_n \in D_n).\ \rho(\upsilon, d_1, \ldots, d_n)$, is an invariant wrt $\mathcal{S}$.

2.2 Simultaneous induction proof

This section briefly describes the simultaneous induction proof method (for more detail, readers are referred to [109, 116]). This proof method is used for proving invariant properties. Let us consider a predicate $p_1 : \Upsilon\ D_{p1}\ \ldots\ D_{pl} \rightarrow$ Bool, in which we want to prove $(\forall x_1 \in D_{p1}) \ldots (\forall x_l \in D_{pl})\, p_1(\upsilon, x_1, \ldots, x_l)$ is an invariant wrt an OTS $\mathcal{S}$. To make the formula become easier to see, let us use $\mathbf{D}_1$ to denote $D_{p1}\ \ldots\ D_{pl}$ and $\mathbf{x}_1$ to denote $x_1, \ldots, x_l$. This is also applied with other notations in the rest of this thesis as well, i.e., a bold upper-case letter denotes a list of data types while a bold lower-case letter represents a list of variables. Consequently, what is needed to prove is $(\forall \upsilon \in \mathcal{R}_{\mathcal{S}})(\forall \mathbf{x}_1 \in \mathbf{D}_1)\, p_1(\upsilon, \mathbf{x}_1)$. We prove that by using induction on the argument of states of p_1. For the base case, we need to prove the following:

$$(\forall \upsilon \in \mathcal{I}_{\mathcal{S}})(\forall \mathbf{x}_1 \in \mathbf{D}_1)\, p_1(\upsilon, \mathbf{x}_1) \tag{2.1}$$

Recall that $\mathcal{I}_{\mathcal{S}}$ denotes the set of initial states of $\mathcal{S}$. (2.1) can typically be proved by deduction (or if we use a theorem prover, it can be straightforwardly resolved by the prover).

For each induction case t associated with the transition $t : \Upsilon\ \mathbf{D}_t \rightarrow \Upsilon$, what we need to prove is as follows:

$$\begin{array}{l}(\forall \upsilon \in \mathcal{R}_{\mathcal{S}}) \\ ((\forall \mathbf{x}_1 \in \mathbf{D}_1)\, p_1(\upsilon, \mathbf{x}_1) \Rightarrow (\forall \mathbf{y}_t \in \mathbf{D}_t)(\forall \mathbf{x}_1 \in \mathbf{D}_1)\, p_1(t(\upsilon, \mathbf{y}_t), \mathbf{x}_1))\end{array} \tag{2.2}$$

It suffices to prove $p_1(t(\upsilon, \mathbf{y}'_t), \mathbf{x}'_1)$ for an arbitrary state υ and arbitrary values $\mathbf{y}'_t$ and $\mathbf{x}'_1$ of $\mathbf{D}_t$ and $\mathbf{D}_1$, respectively, under the induction hypotheses $(\forall \mathbf{x}_1 \in \mathbf{D}_1) p_1(\upsilon, \mathbf{x}_1)$. The induction hypothesis

instance $p_1(\upsilon, \mathbf{x}'_1)$ is often used for that proof. We can also use other instances, such $p_1(\upsilon, \mathbf{x}_2)$ and $p_1(\upsilon, \mathbf{x}_3)$. It is, however, typically impossible to prove (2.2) standalone for non-trivial p_1. Instead, we often prove the conjunction of p_1 and $k-1$ other predicates, let's say $p_2, \ldots, p_k$, where $p_i : \Upsilon\ \mathbf{D}_i \rightarrow \text{Bool}$ for $i = 2, \ldots, k$. That is, we prove $(\forall \mathbf{x}_1 \in \mathbf{D}_1) p_1(\upsilon, \mathbf{x}_1) \wedge \ldots \wedge (\forall \mathbf{x}_k \in \mathbf{D}_k) p_k(\upsilon, \mathbf{x}_k)$ is invariant wrt $\mathcal{S}$. Subsequently, for the base case, now we need to prove the following:

$$\begin{aligned} &(\forall \upsilon \in \mathcal{I}_{\mathcal{S}})(\forall \mathbf{x}_1 \in \mathbf{D}_1) \ldots (\forall \mathbf{x}_k \in \mathbf{D}_k) \\ &(p_1(\upsilon, \mathbf{x}_1) \wedge \ldots \wedge p_k(\upsilon, \mathbf{x}_k)) \end{aligned} \tag{2.3}$$

To prove (2.3), we can prove each conjunct $p_i(\upsilon, \mathbf{x}_i)$ separately. As mentioned before, it can typically be proved by deduction for each $i = 1, \ldots, k$.

For the induction case associated with the transition t, (2.2) is now changed to:

$$\begin{aligned} &(\forall \upsilon \in \mathcal{R}_{\mathcal{S}}) \\ &((\forall \mathbf{x}_1 \in \mathbf{D}_1)\, p_1(\upsilon, \mathbf{x}_1) \wedge \ldots \wedge (\forall \mathbf{x}_k \in \mathbf{D}_k)\, p_k(\upsilon, \mathbf{x}_k) \\ \Rightarrow &(\forall \mathbf{y}_t \in \mathbf{D}_t)\, ((\forall \mathbf{x}_1 \in \mathbf{D}_1)\, p_1(t(\upsilon, \mathbf{y}_t), \mathbf{x}_1) \wedge \ldots \wedge (\forall \mathbf{x}_k \in \mathbf{D}_k)\, p_k(t(\upsilon, \mathbf{y}_t), \mathbf{x}_k))) \end{aligned} \tag{2.4}$$

It suffices to prove each conjunct $p_i(t(\upsilon, \mathbf{y}'_t), \mathbf{x}'_i)$ of the conclusion part for an arbitrary state υ and arbitrary values $\mathbf{y}'_t$ and $\mathbf{x}'_i$ of $\mathbf{D}_t$ and $\mathbf{D}_i$, respectively, under the induction hypotheses $(\forall \mathbf{x}_1 \in \mathbf{D}_1)\, p_1(\upsilon, \mathbf{x}_1) \wedge \ldots \wedge (\forall \mathbf{x}_k \in \mathbf{D}_k)\, p_k(\upsilon, \mathbf{x}_k)$. Typically, it suffices to use only $p_i(\upsilon, \mathbf{x}'_i)$ as the induction hypothesis instance:

$$p_i(\upsilon, \mathbf{x}'_i) \Rightarrow p_i(t(\upsilon, \mathbf{y}'_t), \mathbf{x}'_i) \tag{2.5}$$

Sometimes, it is necessary to use some more instances, for example:

$$p_j(\upsilon, \mathbf{x}_j) \Rightarrow p_i(\upsilon, \mathbf{x}'_i) \Rightarrow p_i(t(\upsilon, \mathbf{y}'_t), \mathbf{x}'_i) \tag{2.6}$$

Note that (2.6) is equivalent to $p_j(\upsilon, \mathbf{x}_j) \wedge p_i(\upsilon, \mathbf{x}'_i) \Rightarrow p_i(t(\upsilon, \mathbf{y}'_t), \mathbf{x}'_i)$ (as usual, $\wedge$ has higher precedence than $\Rightarrow$). In this case, we can say that the proof of p_i uses p_j as a lemma. From what has been presented, although k predicates depend on each other, we can prove them compositionally by using induction for each predicate, and in the proof of p_i, p_j can be used to strengthen the induction hypothesis. Therefore, the proof method is called simultaneous induction. It may also be called compositional proofs. An interesting point of the simultaneous induction method is that even if p_j is used to strengthen the induction hypothesis in the proof of p_i and vice versa, i.e., p_i is used to strengthen the induction hypothesis in the proof of p_j, there is no problem. In the next sections, we briefly give the CafeOBJ syntax and then present how

to use CafeOBJ to write proof scores for k invariants individually based on this proof method.

2.3 CafeOBJ in a nutshell

CafeOBJ is an advanced language for writing formal specifications for a wide variety of systems and protocols. It is equipped with a powerful specification syntax and many useful features for both writing formal specifications and specifying required properties of even complex systems, for example, module expressions, modules instantiated parameters using views, and the flexible mix-fix syntax. CafeOBJ supports order-sorted equational logic and can be used as a powerful interactive theorem proving system. This section gives the syntax of CafeOBJ, in a nutshell, to help readers understand the rest of this thesis. For the complete syntax, readers are referred to its user manual [105] and the book [52].

Module

Modules are the most basic building block of CafeOBJ. A module has the syntax: **`module`** M `{` *module_elements* `}`, where M is the module name. **`module`** can be abbreviated as just **`mod`**, and it can be alternatively declared as: (1) **`mod`**$*$ denoting that the module has loose semantics, or (2) **`mod`**`!` denoting that the module has tight semantics. A loose semantic module denotes a class of models, which means many different implementations for the sorts and operators declared in the module satisfy the given axioms [11]. Whereas, a tight semantic module denotes a unique model up to isomorphism. *module_elements* may contain the following declarations:

- declarations of importations of previously defined modules. There are four modes of importation: **`protecting`**, **`extending`**, **`using`**, and **`including`**. For example, **`pr`**`(`M'`)` denotes the module M' is imported under the **`protecting`** mode (**`pr`** is the abbreviation of **`protecting`**).
- sort declarations: `[` $s_1\ s_2\ \ldots\ s_n$ `]`, where $s_1, s_2, \ldots, s_n$ are sort names.
- declarations of ordering sorts relation: `[` s_1 `<` s_2 `]` denotes that s_1 is a subsort of s_2.
- operator declarations: **`op`** f `:` $s_1\ \ldots\ s_n \rightarrow s$ $\{at_1 \ldots at_k\}$ or **`ops`** $f_1\ f_2$ `:` $s_1\ \ldots\ s_n \rightarrow s$ $\{at_1 \ldots at_k\}$ where s is also a sort name, and $at_1, \ldots, at_k$ are equational theory attributes, such as `assoc` (associativity) and `comm` (commutativity).
- variable declarations: **`var`** V `:` s or **`vars`** $V\ V_2\ \ldots$ `:` s, where $V, V_2, \ldots$ are variable names. Note that by convention, variable names in CafeOBJ should be in upper case.
- unconditional equation declarations: **`eq`** t_1 `=` t_2 `.`, where t_1 and t_2 are terms.

- conditional equation declarations: **ceq** t_1 = t_2 **if** *cond* ., where *cond* is a Boolean term, which may be a conjunction of equations, such as c_1 = c_2.

The following is a simple example of a module definition, which specifies natural numbers only with the addition:

```
mod SIMPLE-NAT {
  [ Zero NzNat < Nat ]
  op 0 : -> Zero
  op s : Nat -> NzNat
  op _+_ : Nat Nat -> Nat
  vars N N' : Nat
  eq 0 + N = N .
  eq s(N) + N' = s(N + N') .
}
```

`0` of the sort `Zero` represents zero, and it is called a constant since the operator has empty arity. The two sorts `NzNat` and `Nat` represent non-zero numbers and natural numbers (either zero, or non-zero), respectively. `Zero` and `NzNat` are subsorts of `Nat`, meaning that any terms of the sort `Zero` and the sort `NzNat` also belong to the sort `Nat`. `s` is the successor function of natural numbers, taking as input a natural number and returning as output a non-zero natural number (successor of n is $n + 1$). `0` and `s` are declared in the standard operator declaration, i.e., prefix syntax, while `_+_` is introduced as an infix operator, thanks to the flexible mid-fix syntax of CafeOBJ. Two underscores in `_+_` represent two natural numbers that are inputs of the addition operator (e.g., we write `0 + s(0)`). The last two equations define the semantics of the operators.

Open-close environment

An open-close environment in CafeOBJ provides a temporary copy of a given module, which is particularly useful to do theorem proving. An open-close environment has the following syntax:

```
open M .
  ...
close
```

This is also called the open-close fragment. New operators, equations, etc. may be introduced inside the fragment (`...`). This fragment creates a new temporary module by copying the module M and adding into the new module all the operators, equations, etc. newly introduced. We can use this environment, for example, to check that `s(0) + s(s(0))` is truly `s(s(s(0)))` as our expectation (namely, `1 + 2` is `3`) with respect to the definition of the module `SIMPLE-NAT`:

```
open SIMPLE-NAT .
  red s(0) + s(s(0)) .
close
```

loop { "Remainder Section"
rs : enqueue($queue, i$);
ws : **repeat until** top($queue$) = i;
"Critical Section"
cs : dequeue($queue$); }

Figure 2.1: Qlock protocol

where **red** (standing for **reduce**) reduces the given term to its normal form. Feeding the open-close fragment into CafeOBJ, CafeOBJ returns the result as follows:

```
%SIMPLE-NAT> -- reduce in %SIMPLE-NAT : (s(0) + s(s(0))):Nat
(s(s(s(0)))):NzNat
```

`s(s(s(0)))` is returned as our expectation. This open-close environment is more helpful than what has just been illustrated, for example, when doing theorem proving based on induction, an open-close fragment can be used to solve the base case and several other ones can be used to solve the induction cases, where each of them requires a different declaration of the induction hypothesis instance used. The next section illustrates how to use open-close environments to write a formal proof in CafeOBJ.

2.4 Formal verification by writing proof scores

Based on the simultaneous induction proof method, this section illustrates how to use CafeOBJ to write a computer-verified proof, a so-called *proof score* [116, 109], for a given property.

Definition 3. Given a CafeOBJ equation specifying a theorem, the *proof score* in CafeOBJ proving the theorem is a collection of open-close fragments that describe the *proof tree* leaves of the equation and are discharged by reduction.

If the proof score is executed with CafeOBJ in which each term in each open-close fragment reduces as expected, such as to true, then the theorem undertaking is proved. To illustrate the proof score verification approach and the correspondence between the proof tree leaves and CafeOBJ open-close fragments, let us consider a mutual exclusion protocol, namely Qlock, which has the pseudo-code as in Figure 2.1. In the figure, rs, ws, and cs stand for Remainder Section, Waiting Section, and Critical Section, respectively. *queue* is an atomic queue of process identifiers (IDs) shared by all processes. Initially, *queue* is empty and each process i is located at rs. If i wants to enter cs, it first enqueues its ID i into *queue* and moves to ws. While the top

of *queue* is not i, it needs to wait there. When i leaves cs, it dequeues *queue* and goes back to rs.

With this protocol, we want to formally verify that it enjoys the *mutual exclusion* property.

Definition 4. *Mutual exclusion property is:*
There always exists at most one process located at the Critical Section.

To verify that property, we first formally specify Qlock in CafeOBJ as an OTS $\mathcal{S}_{\text{Qlock}}$. We introduce sort `Sys` representing $\mathcal{R}_{\mathcal{S}_{\text{Qlock}}}$ and sort `Queue` representing the set of process ID queues. We use two observers `pc` and `queue` to observe the location of each process and the queue:

```
op pc    : Sys Pid -> Label
op queue : Sys     -> Queue
```

where `Pid` is the sort of process IDs, and `Label` is the sort of locations such as rs, ws, and cs. Given a state `s` (of the sort `Sys`) and process `p` (of the sort `Pid`), `pc(s,p)` denotes the location at which `p` is located in state `s` and `queue(s)` is the queue in state `s`. Observers and CafeOBJ operators that express observers are interchangeably used in this thesis.

We use three transitions that are expressed as CafeOBJ operators `want`, `try`, and `exit`. An arbitrary initial state of $\mathcal{S}_{\text{Qlock}}$ is represented by constant `init`. They are declared as follows:

```
op init :         -> Sys {constr}
op want : Sys Pid -> Sys {constr}
op try  : Sys Pid -> Sys {constr}
op exit : Sys Pid -> Sys {constr}
```

where the attribute `constr` states that the four operators are constructors of `Sys`. The four operators together with process IDs construct $\mathcal{R}_{\mathcal{S}_{\text{Qlock}}}$. Let `I` and `J` be CafeOBJ variables of the sort `Pid`. `init` is defined in terms of equations as follows:

```
eq pc(init,I) = rs .
eq queue(init) = empty .
```

The equations state that each process `I` is located at `rs` and *queue* is empty in the initial state `init`. We show here how to define the transition `try`, which models the movement of a process from ws to cs, while the transitions `want` and `exit` are defined in a similar way.

```
ceq pc(try(S,I),J) = (if I = J then cs else pc(S,J) fi) if c-try(S,I) .
eq queue(try(S,I)) = queue(S) .
ceq try(S,I) = S if not c-try(S,I) .
eq c-try(S,I) = (pc(S,I) = ws and top(queue(S)) = I) .
```

where `c-try` is the effective condition of the transition; **`if`** c **`then`** a **`else`** b **`fi`** is a if the condition c holds and is b if c does not. The equations say that if process `I` is located at ws and the top of *queue* is `I` in the state denoted by `S`, then `I` moves to cs in the successor state of S, which is denoted by `try(S,I)`; otherwise, nothing changes.

Once the formal specification of Qlock is complete, we can turn to formally verify the *mutual exclusion* property. This property is formally specified in CafeOBJ as the following state predicate `inv1`:

```
op inv1 : Sys Pid Pid -> Bool
eq inv1(S,I,J) = (pc(S,I) = cs and pc(S,J) = cs) implies (I = J) .
```

It says that if two processes `I` and `J` both are located at `cs`, they must be identical. We prove that `QLOCK` $\vdash$ ($\forall$`S` $\in$ `Sys`)($\forall$`I`,`J` $\in$ `Pid`) `inv1(S,I,J)` $= true$, namely, `inv1(S,I,J)` $= true$ can be derived from `QLOCK`, the CafeOBJ module contains the complete specification of the protocol (we may use `inv1(S,I,J)` as an abbreviation of `inv1(S,I,J)` $= true$). In other words, we prove ($\forall i, j \in$ `Pid`) `inv1`(υ, i, j) is an invariant wrt $\mathcal{S}_{\text{Qlock}}$ (we may also shortly call that `inv1` is an invariant). The predicate is proved by (simultaneous) induction on the argument of the sort `Sys`. There are one base case and three induction cases. For the base case, we need to prove: `QLOCK` $\vdash$ ($\forall$`I`,`J` $\in$ `Pid`) `inv1(init,I,J)`. By applying the theorem of constants, the free variables `I` and `J` can be replaced by two CafeOBJ fresh constants `i` and `j`, and then the proof of the base case is done by the following open-close fragment, which is referred to as fragment (`init`):

```
open QLOCK .
  ops i j : -> Pid .
  red inv1(init,i,j) .
close
```

The fresh constants `i` and `j` denote arbitrary process identifiers. CafeOBJ returns `true` for the open-close fragment, meaning that the base case is discharged. The open-close fragment describes a leaf of the proof tree of `QLOCK` $\vdash$ ($\forall$`S` $\in$ `Sys`)($\forall$`I`,`J` $\in$ `Pid`) `inv1(S,I,J)`, which is partially depicted in Figure 2.2. What we need to prove is written in the tree root. By using induction on variable `S`, an induction case and three induction cases are produced, where the two induction cases associated with the transitions `want` and `exit` are omitted in the figure. For the base case (the leftmost branch from the root), after applying the theorem of constants, `inv1(init,i,j)` can be derived from the specification denoted by (`QLOCK`, **`ops`** `i j : -> Pid`), that is, an extended module of `QLOCK` by adding two fresh constants `i` and `j`. Thus, the open-close fragment (`init`) shown above exactly describes the leftmost leaf of the proof tree. The proof score of `inv1` consists of a collection of open-close fragments, where each one describes a leaf in the proof tree.

Turning to the three induction cases, for the induction case associated with the transition `try`, we need to prove: `QLOCK` $\vdash$ ($\forall$`I`,`J` $\in$ `Pid`) `inv1(try(s,k),I,J)` under the induction hypotheses ($\forall$`I`, `J` $\in$ `Pid`) `inv1(s,I,J)`, where `s` and `k` are fresh constants of `Sys` and `Pid`, respectively. Again, the free variables `I` and `J` are replaced by CafeOBJ fresh constants `i` and `j` (step "By theorem of constants" in Figure 2.2). After that, the most typical induction hypothesis instance `inv1(s,i,j)` is used, and the proof attempt of this induction case is written as the following open-close fragment:

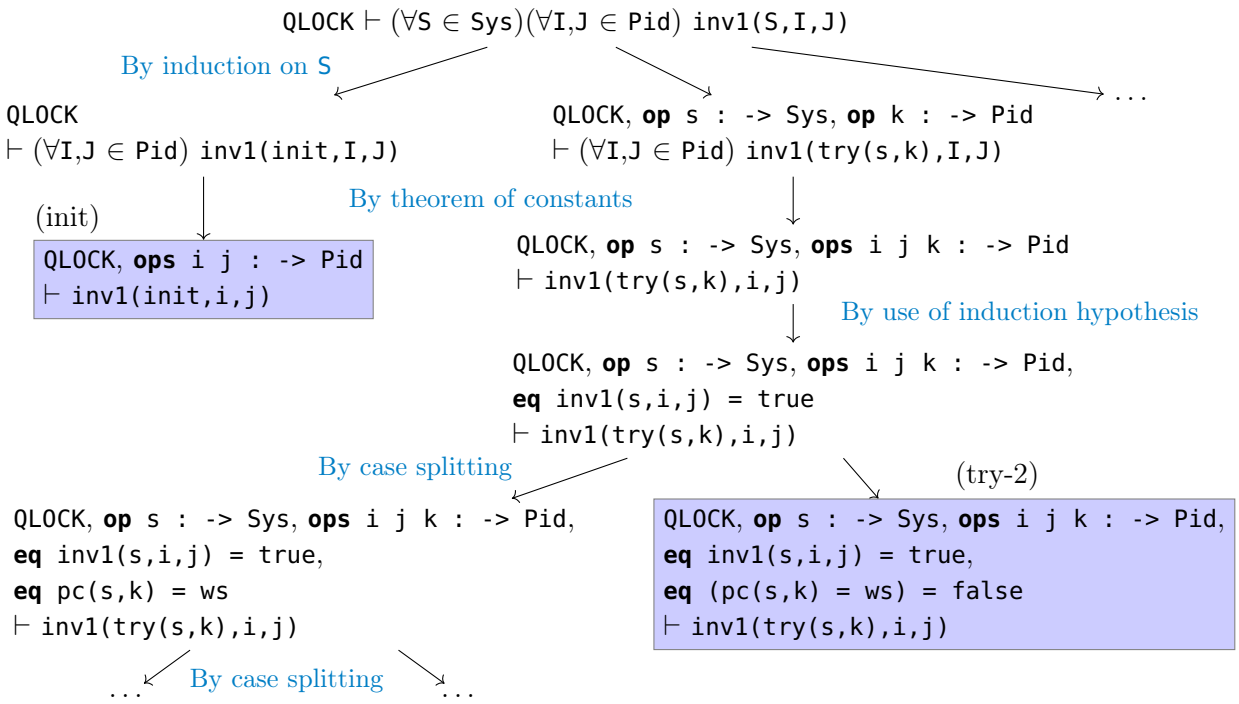


Figure 2.2: Proof tree of `inv1`

```
open QLOCK .
  op s : -> Sys .   ops i j k : -> Pid .
  eq inv1(s,i,j) = true .
  red inv1(try(s,k),i,j) .
close
```

Or alternatively, we can declare the induction hypothesis instance as the premise of the **red** command:

```
open QLOCK .
  op s : -> Sys .   ops i j k : -> Pid .
  red inv1(s,i,j) implies inv1(try(s,k),i,j) .
close
```

Let us use the name (`try`) to refer to this open-close fragment. However, feeding this open-close fragment into CafeOBJ, the returned result is neither `true` nor `false`, but instead a complicated term as follows:

```
(((pc(try(s,k),j) = cs) and ((i = j) and (pc(try(s,k),i) = cs))) xor
(((pc(try(s,k),i) = cs) and ...)))
```

where `...` stands for terms that are omitted. The term cannot be reduced because the module lacks information on the processes `k`, `i`, and `j`. Case splitting is used to overcome this situation. The case is first split into two sub-cases: (`try-1`) `pc(s,k) = ws` and (`try-2`) `(pc(s,k) = ws) = false`. The open-close fragment (`try-2`) for the latter is as follows:

```
open QLOCK .
  op s : -> Sys .   ops i j k : -> Pid .
  eq (pc(s,k) = ws) = false .
  red inv1(s,i,j) implies inv1(try(s,k),i,j) .
close
```

The equation characterizes the sub-case. Feeding the fragment into CafeOBJ, CafeOBJ returns `true`, indicating that the sub-case is discharged. The open-close fragment corresponds to the leaf tagged with (try-2) in the proof tree in Figure 2.2. For the sub-case (`try-1`), as indicated in Figure 2.2, it is necessary to conduct case splitting several more times. Let us consider a sub-case of (`try-1`), which has the following open-close fragment:

```
open QLOCK .
  op s : -> Sys .   ops i j k : -> Pid .
  eq pc(s,k) = ws .
  eq top(queue(s)) = k .
  eq i = k .
  eq (j = k) = false .
  eq pc(s,j) = cs .
  red inv1(s,i,j) implies inv1(try(s,k),i,j) .
close
```

`false` is returned for the open-close fragment. It means that we need to conjecture a lemma to discharge the sub-case. Provided that `inv1` is truly invariant wrt $\mathcal{S}_{\mathrm{Qlock}}$ as our expectation, what can be deduced is that states denoted by the fresh constant `s` must be unreachable wrt $\mathcal{S}_{\mathrm{Qlock}}$. There should exist a contradiction among the equations characterizing the sub-case. From our comprehension of the protocol, we strongly believe that if a process is located at the Critical Section, the process must be the top of *queue*. Consequently, it turns out that the following three equations have a contradiction:

```
  eq top(queue(s)) = k .
  eq (j = k) = false .
  eq pc(s,j) = cs .
```

because process `j` is located at `cs`, but the top the queue is `k`, which is different from `j`. Based on that deduction, a lemma candidate, namely `inv2`, is conjectured as follows:

```
op inv2 : Sys Pid -> Bool
eq inv2(S,I) = (pc(S,I) = cs implies top(queue(S)) = I) .
```

It says that when process `I` is located at the Critical Section, it must be the top of *queue*. Then, in the open-close fragment above, we can use `inv2` as a lemma to discharge the sub-case as follows:

```
red inv2(s,j) implies inv1(s,i,j) implies inv1(try(s,k),i,j) .
```

CafeOBJ now returns `true` for the open-close fragment. The formal verification now changes to prove the conjunction of `inv1` and `inv2`. In a similar way, we can complete the proof score of `inv1`, which is a collection of open-close fragments like the three ones shown above. After that, to complete the formal verification, we need to write a proof score to prove that `inv2` is also an invariant wrt $\mathcal{S}_{\text{Qlock}}$. In the proof of `inv2`, `inv1` is used as a lemma. The proof of `inv1` uses `inv2` as a lemma and vice versa, however, as proved in Section 2.2, there is no circular error in the verification.

Chapter 3

IPSG: Invariant Proof Score Generator

In this chapter, we propose an approach and implement a tool (called IPSG) that can automatically generate proof scores for formal invariant property verification. This is motivated by the fact that writing proof scores by hand are prone to human errors and time-consuming even though the approach is powerful and flexible, which has been demonstrated through many case studies.

3.1 The drawbacks of writing proof score

CafeOBJ [52], as mentioned in Sections 2.3 and 2.4, is a powerful language for writing formal specifications of systems and for verifying system requirements by writing proof scores. In the proof score approach [116, 118, 109], OTSs are used as transition systems to model a system, and then proof scores are written essentially by applying simultaneous induction on a state variable, which has been presented through a running example in Section 2.4. The usefulness of the approach essentially comes from the power of the CafeOBJ language in specifying systems and its flexibility in writing proof scores. Using the approach, various formal verification case studies have been conducted analyzing many systems/protocols, such as the Mondex payment system [87], the *i*KP electronic payment protocol [111, 112], the OMA license choice algorithm [146], the Transport Layer Security 1.0 protocol [110], and the electronic commerce protocols [115].

The flexibility of writing proof scores, however, comes at a cost, that is the approach is subject to human errors. Proof scores consist of many user-defined open-close fragments (also called proof fragments), where each of them has one reduction command, which generally gives a term to be reduced to true or false. If each reduction command reduces to true as expected, which is supposed to be checked by human users, the formal verification concerned is done. Therefore, human users are responsible for the correctness of the proof. In particular, human users need to make sure that the proof covers all base/induction cases, the proof uses proper

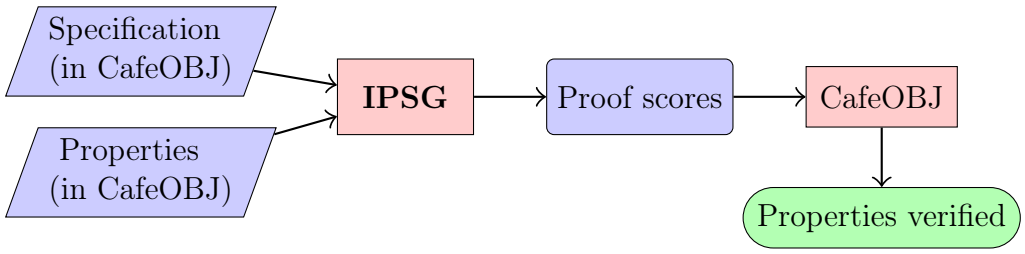


Figure 3.1: Formal verification by using IPSG

case splittings, and each open-close fragment uses proper induction hypotheses and/or lemmas. However, because an open-close fragment is user-defined, human users can, for example, add an extra equation, incorrectly write some equations, or overlook some open-close fragments or sub-cases. That is the reason why proof scores are said to be prone to human errors. Moreover, the task of writing proof scores is really time- and effort-consuming, especially with complicated systems or specifications. During the verification, there are many trivial sub-cases that are tedious to write the proof again and again.

To address the above-mentioned drawbacks, we propose an approach and implement a tool, called IPSG (Invariant Proof Score Generator), that can automatically generate proof scores for formal invariant property verification. By using the tool, human users only need to focus on solving non-trivial sub-cases, which typically require additional lemmas, but trivial sub-cases are already discharged by the tool. The formal verification approach by using IPSG is visualized in Figure 3.1. To demonstrate the efficiency and the practicability of the tool, we conduct experiments with various systems/protocols, ranging from a classical key distribution protocol to authentication protocols, from a cloud synchronization protocol to mutual exclusion protocols, and from a distributed protocol to real cryptographic protocols currently in use. The source code of the tool is publicly available on the webpage[1], while the specifications, the proof scores, and other related materials of all experiments are available on the webpage[2].

3.2 CafeInMaude and Maude meta-level functionalities

The tool is implemented in Maude [43, 62]. It uses CafeInMaude [127], which is the second major implementation of CafeOBJ in the Maude environment. Providing a formal specification written in CafeOBJ, CafeInMaude makes it possible to convert the specification into the Maude environment. After that, we can utilize Maude functionalities, such as meta-level representations of modules and terms, to parse the specification and the properties under verification. Such functionalities are not available in CafeOBJ, which is the reason why we need to employ CafeInMaude and Maude. Figure 3.2 visualizes how CafeInMaude and Maude are used in the

[1]https://github.com/duongtd23/IPSG-tool
[2]https://github.com/duongtd23/IPSG-TLS

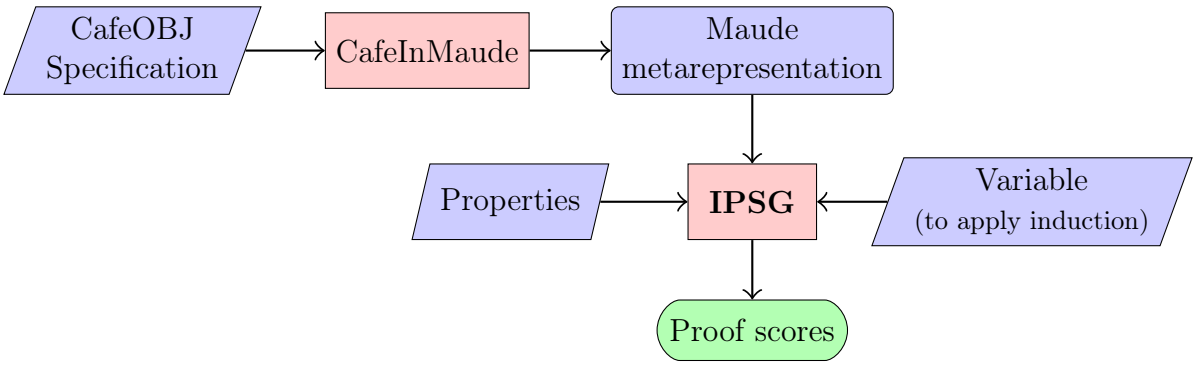


Figure 3.2: The uses of CafeInMaude and Maude

implementation of IPSG to generate proof scores. Before going into the tool implementation, this section first briefly describes some Maude meta-level functionalities.

Maude is a declarative language and high-performance tool that focuses on simplicity, expressiveness, and performance to support the formal specification and analysis of concurrent programs/systems in rewriting logic. Rewriting logic is a reflective logic, which means that the logic can be faithfully interpreted in itself [43]. This section borrows some descriptions from Chapter 14 of the book [43] to briefly describe reflection and metaprogramming in Maude. Rewriting logic is reflective in a precise mathematical way, namely, there is a finitely presented rewrite theory $\mathcal{U}$ that is *universal* in the sense that we can represent in $\mathcal{U}$ any finitely presented rewrite theory $\mathcal{M}$ (including $\mathcal{U}$ itself) as a term $\overline{\mathcal{M}}$, any terms t, t' in $\mathcal{M}$ as terms $\overline{t}, \overline{t'}$, and any pair $(\mathcal{M}, t)$ as a term $\langle \overline{\mathcal{M}}, \overline{t} \rangle$, in such a way that the rewrite $t \to t'$ with respect to $\mathcal{M}$ are equivalence to the rewrite $\langle \overline{\mathcal{M}}, \overline{t} \rangle \to \langle \overline{\mathcal{M}}, \overline{t'} \rangle$ with respect to $\mathcal{U}$:

$$\mathcal{M} \vdash t \to t' \iff \mathcal{U} \vdash \langle \overline{\mathcal{M}}, \overline{t} \rangle \to \langle \overline{\mathcal{M}}, \overline{t'} \rangle$$

Because $\mathcal{U}$ is representable in itself, we can achieve a "reflective tower" with an arbitrary number of levels of reflection:

$$\mathcal{M} \vdash t \to t' \iff \mathcal{U} \vdash \langle \overline{\mathcal{M}}, \overline{t} \rangle \to \langle \overline{\mathcal{M}}, \overline{t'} \rangle \iff \mathcal{U} \vdash \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{M}}, \overline{t} \rangle} \rangle \to \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{M}}, \overline{t'} \rangle} \rangle \ldots$$

In this chain of equivalences we say that the first rewriting computation takes place at level 0, the second at level 1, and so on. In Maude, the key functionality of the universal theory $\mathcal{U}$ has been efficiently implemented in the functional module `META-LEVEL`. In this module:

- any Maude term can be reified as an element of the data type `Term` in the module `META-TERM`.
- any Maude module can be reified as a term of the data type `Module` in the module `META-MODULE`.

- two functions `upModule` and `upTerm` allow for moving modules and terms, respectively, from object level to meta-level.
- function `downTerm` allows for moving terms from meta-level to object level.
- reducing a term to canonical form using Maude **`reduce`** command is meta represented by the built-in function `metaReduce`, whose signature is as follows:

  ```
  op metaReduce : Module Term ~> ResultPair [special (...)] .
  ```

  The function takes as inputs the metarepresentation of a module $\mathcal{M}$ and the metarepresentation of a term t, and returns as output the metarepresentation of the canonical form of t by using the equations in $\mathcal{M}$, together with the metarepresentation of its corresponding sort or kind, paired in a term of sort `ResultPair`. Note that this function is partial (denoted by `~>`) because it is possible that the term does not make sense in the module.
- rewrite a term using the command `rewrite` is meta represented by the built-in function `metaRewrite`, whose signature is as follows:

  ```
  op metaRewrite : Module Term Bound ~> ResultPair [special (...)] .
  ```

  The function takes as inputs the metarepresentation of a module $\mathcal{M}$, i.e., $\overline{\mathcal{M}}$, the metarepresentation of a term t, i.e., $\overline{t}$, and a bound b, which can be either a natural number or the constant `unbounded`. When b is a number, `metaRewrite`$(\overline{\mathcal{M}}, \overline{t}, b)$ returns as output the metarepresentation of the term obtained from t after at most b rewriting steps by using both equations and rewrite rules in $\mathcal{M}$. When b is `unbounded`, there is no limitation about the number of rewriting steps.

Metaprogramming is a programming technique of writing computer programs that manipulate other programs or even themself. A metaprogram can read another program or even itself, analyze and modify that program and return the modified program. Therefore, metaprogramming is very powerful and helpful to manipulate programs on the fly. The ability of a programming language to be its own metalanguage is called reflection. Maude is a high-performance reflective language that supports human users to write metaprograms. In Maude, a term in the object level has the corresponding representation in the meta-level and vice versa. We can write metaprograms in Maude simply by importing the module `META-LEVEL`, and then we can use the built-in functions mentioned above, such as `metaReduce` and `metaRewrite`.

3.3 Invariant Proof Score Generator

The key idea to automatically generate proof scores is briefly described as follows: When we feed an open-close fragment into CafeOBJ and it returns a term t, which is neither `true` nor `false`, we will select a sub-term of t, say t', and split the current open-close fragment (or case) into two sub-cases: one when t' holds, and the other when it does not. For each sub-case, the same procedure is applied. We start running the procedure with a list of initial open-close fragments, where all of them do not contain any equation. Some of them (typically only one) are for base cases, such as the open-close fragment (`init`) shown in Section 2.4, while the remains are for induction cases, such as the open-close fragment (`try`) shown in Section 2.4. Eventually, we get the proof that consists of a list of open-close fragments in which the reduction commands return either `true` or `false`. If `true` is returned, the current case is discharged and we do not need to do anymore. If `false` is returned, the tool tried to find a lemma from a collection of all possible lemmas provided by human users that can be used to discharge the current case. We must emphasize that this is a concise and abstract description of how to automate the proof score writing process. To make the idea work and to implement an efficient tool supporting that idea, many other detailed procedures and algorithms are required, such as how should we choose the sub-term t' and how to find a lemma that can be used. We present them in the next sections.

3.3.1 Proof score generation algorithm

The algorithm for generating proof scores is described in Algorithm 1. The algorithm takes as inputs (1) CafeOBJ module $\mathcal{M}$ describing an OTS, (2) a state predicate $inv : \Upsilon\, D_1 \ldots D_n \rightarrow$ Bool, in which we want to generate a proof score (i.e., a collection of open-close fragments) to prove that $(\forall d_1 \in D_1) \ldots (\forall d_n \in D_n)\, inv(v, d_1, \ldots, d_n)$ is an invariant of the OTS, (3) the argument of inv on which induction is used Υ, and (4) a lemma list lms, which might be used in the proof of inv. The algorithm starts by extracting CafeOBJ operators that represent initial states ($inits$) and transitions (ts) from the module $\mathcal{M}$ and the argument on which induction is used Υ, i.e., the argument of states. Returning to our verification example with the Qlock protocol presented in Section 2.4, $inits$ would be the single operator `init`, while ts would be the set of `want`, `try`, and `exit`:

```
op init :         -> Sys {constr}
op want : Sys Pid -> Sys {constr}
op try  : Sys Pid -> Sys {constr}
op exit : Sys Pid -> Sys {constr}
```

In this case, Υ is `Sys`. The module $\mathcal{M}$ in the object level is converted to its metarepresentation by using the Maude built-in function `upModule`. Then, the algorithm extracts from $\mathcal{M}$'s metarepresentation the operators attached with the `constr` attribute whose coarity is Υ, among them,

the ones with empty arity form *inits*, while the ones with `Sys` in their arity form *ts*. Hereinafter, let us use initial states (respectively, transitions) and CafeOBJ operators representing them interchangeably in this thesis even though precisely, multiple instances of the former are typically represented by one instance of the latter.

The algorithm essentially consists of two parts: one for generating the proof score of base cases (lines 5-8), and the other for generating the proof score of induction cases (lines 9-13). With the former, the algorithm enumerates each state i in the set of the initial states extracted from the specification (line 5), while with the latter, the algorithm enumerates each transition $v \rightarrow v'$ in the set of the transitions extracted (line 9). The variable re is initially set to empty (by the function empty-list()). Then, it is respectively appended the proof score of the base cases & the induction cases (by the function append()). Finally, it is returned as the output. Note that re receives as value a list of open-close fragments and append() is the concatenation function of lists. The function empty-prsc() at line 6 and line 10 initializes an open-close fragment, which does not contain any equation, but consists of only fresh constant (operator) declarations and a **`reduce`** command in which the most typical induction hypothesis instance is used if it is an induction case. Mapping to the Qlock verification example presented in Section 2.4, p at line 6 would receive the open-close fragments (`init`) as a value:

```
open QLOCK .
  ops i j : -> Pid .
  red inv1(init,i,j) .
close
```

while p at line 10 would receive the open-close fragments (`try`) as a value:

```
open QLOCK .
  op s : -> Sys .   ops i j k : -> Pid .
  red inv1(s,i,j) implies inv1(try(s,k),i,j) .
close
```

The two parts share the same function GenPrsc, which takes as inputs the module $\mathcal{M}$, the current open-close fragment p, and the target term te that we try to reduce in p, and returns as output a list of open-close fragments. Mapping to the Qlock verification example, te is the argument of the **`red`** command in each of the two open-close fragments shown above, i.e., the metarepresentation of the term `inv1(init,i,j)` if we want to generate the proof of the base case, and the metarepresentation of the term `inv1(s,i,j) implies inv1(try(s,k),i,j)` if the induction case `try` is taken into account.

In line 16, the function tries to reduce the target term wrt the current open-close fragment, thanks to the Maude built-in function `metaReduce`. Note that the notation $\mathcal{M}||p$ denotes the metarepresentation of the module in which $\mathcal{M}$ and all operators & equations in the open-close fragment p are available. The obtained result (t) falls into one of the following three cases:

1. The result is true. The current case of the proof is discharged, and then the algorithm simply returns the current open-close fragment.

2. The result is false. This is a non-trivial case of the proof. To discharge this case, we need to find a possible lemma that can be used for this case. The function HandleFalseCase is invoked, which tries to find and use some suitable lemma from the lemma list provided by human users. We leave the description of this function in Section 3.3.2.

3. The result is neither `true` nor `false`, i.e., a composite term. The function ChooseATermCS extracts and chooses one sub-term in the result to apply case splitting (line 22). Which sub-term is chosen may affect the efficiency of the whole algorithm. In writing proof scores, if we choose a good order of equations used for case splitting, then the number of fragments will be reasonably compact. In contrast, if the order is not good, the size of the proof scores will become larger. The function ChooseATermCS implements several techniques for selecting an appropriate sub-term to conduct case splitting at this step. For example, if the result contains a sub-term **if** c **then** a **else** b **endif**, the sub-terms inside the condition c will be given the highest priority for choosing.

   The function then recursively calls itself twice in which the second parameters are updated by adding two equations, i.e., the selected sub-term equals true and it equals false, into itself. The obtained results of the two recursive calls are then concatenated together and returned as output (line 23). Returning to the Qlock verification example in the previous chapter, when the induction case `try` is taken into account, firstly, the Boolean term `pc(s,k) = ws` is used to split the case into two sub-cases: (`try-1`) it is true and (`try-2`) it is false. In the sub-case (`try-2`), the **`eq`** `(pc(s,k) = ws) = false .` is inserted into the associated open-close fragment.

3.3.2 Finding and using suitable lemmas

For each sub-case in the induction cases such that false is returned for the reduction command in the associated open-close fragment, IPSG tries to find and use a suitable lemma from the lemma list provided by human users. The function HandleFalseCase is in charge of performing this task, and its algorithm is shown in Algorithm 2.

The function first collects all terms and sub-terms (sts) that exist in the current open-close fragment (line 2). For each possible lemma, it tries to find a list of terms that can be used as parameters to instantiate the lemma. Precisely, for each lemma $lm : \Upsilon\ \bar{D}_1\ \ldots\ \bar{D}_m \rightarrow \text{Bool}$, it tries to find a term list $\bar{d}_1, \ldots, \bar{d}_m$ whose each entry belongs to sts and the term list can be instantiated as parameters to lm (i.e., $\bar{d}_1 \in \bar{D}_1, \ldots, \bar{d}_m \in \bar{D}_m$). The function then tries to reduce the instantiated lemma $lm(\upsilon, \bar{d}_1, \ldots, \bar{d}_m)$ in the context of the open-close fragment p (line 5). If

the obtained result is false (line 6), we can use this lemma to strengthen the induction hypothesis to discharge the current proof (because when $lm(\upsilon, \bar{d}_1, \ldots, \bar{d}_m)$ is false, then "$lm(\upsilon, \bar{d}_1, \ldots, \bar{d}_m)$ **implies** te" will be true). If that is the case, the states (denoted by υ) are not reachable wrt the OTS concerned, and then we do not need to consider the states provided that the lemma is invariant wrt the OTS (which we also need to prove). Note that we suppose that the lemma lm is also proved by simultaneous induction on the same state variable as of inv. Therefore, the state parameter υ is fixed when instantiating each possible lemma.

If the function finishes enumerating every lemma but cannot find any suitable lemma that can be used to discharge the current proof, it simply returns the original open-close fragment. This case is marked as a "false case" and left for human users to resolve manually.

3.3.3 Handling conditional equations

This section presents in detail the algorithm for handling conditional equations in the input CafeOBJ specification. Recall from Section 2.4, when we feed the open-close fragment (`try`) into CafeOBJ, the following result is returned:

```
(((pc(try(s,k),j) = cs) and ((i = j) and (pc(try(s,k),i) = cs))) xor
 (((pc(try(s,k),i) = cs) and ...)))
```

If we select, for example, the sub-term `pc(try(s,k),j) = cs` for conducting case splitting, it is not correct because the sub-term contains the successor state of `s`, i.e., `try(s,k)`. The term `pc(try(s,k),j)` could be simplified more depending on whether the value of `c-try(s,k)` is true or false. Indeed, the specification defines the change of the value observed by the observer `pc` through the transition `try` by the following two conditional equations:

```
ceq pc(try(S,I),J) = (if I = J then cs else pc(S,J) fi) if c-try(S,I) .
ceq try(S,I) = S if not c-try(S,I) .
```

There exists a pattern match $\sigma_1 = \{\texttt{S} \mapsto \texttt{s}, \texttt{I} \mapsto \texttt{k}, \texttt{J} \mapsto \texttt{j}\}$ between the left handside of the first equation and the term `pc(try(s,k),j)`, and a pattern match $\sigma_2 = \{\texttt{S} \mapsto \texttt{s}, \texttt{I} \mapsto \texttt{k}\}$ between the left handside of the second equation and the term `try(s,k)`. Then, `pc(try(s,k),j)` is rewritten to `if k = j then cs else pc(s,j) fi` based on the first equation if `c-try(s,k)` is true, and it is rewritten to `pc(s,j)` based on the second equation if `c-try(s,k)` is false. In both cases, `try(s,k)` no longer appears.

However, CafeOBJ (or CafeInMaude) cannot rewrite the term `pc(try(s,k),j)` based on the two equations above in the context of the open-close fragment (`try`). The reason is that the value of `c-try(s,k)` cannot be determined whether it is true or false in the context of the open-close fragment. It can only be reduced to `pc(s,k) = ws and top(queue(s)) = k`, which is neither true nor false, based on the equation defining `c-try`. Recall from Section 2.4, the open-close fragment (`try`) does not contain any equation, thus, it lacks information on `s` and `k`, making the term

unable to be rewritten more.

The IPSG implementation addresses this problem, making it possible to handle conditional equations. Inside the function ChooseATermCS($\mathcal{M}, p, t$) (at line 22 of Algorithm 1), after extracting a sub-term from t, say tcs', the procedure is as follows. If there exists a conditional equation, say **ceq** $l = r$ **if** c, in the module $\mathcal{M}$ such that there exists pattern match σ between l and tcs', and $\sigma(c)$ is reduced to term c', which is neither true nor false, in the module denoted by $\mathcal{M}||p$ (i.e., the extended module of $\mathcal{M}$ where all operators & equations in the p are available), then a sub-term of c' is returned as output. Otherwise, if there does not exist such an equation or such a pattern match, tcs' is returned as output. The algorithm can deal with not only conditional equations dedicated to specifying transition effective conditions but also all conditional equations in general.

3.3.4 Case splitting is used first before reduction

The practicability and efficiency of the tool should be evaluated with large CafeOBJ specifications and complex properties. The property under verification may cause the tool to lose its performance in terms of running time if the property consists of many logical connectives, such as **and** & **or**. This is due to the time taken to reduce the property to canonical form, i.e., the time taken by the function `metaReduce`, probably becomes very long. Because this function is invoked many times during the algorithm execution, it should not be expensive; otherwise, the running time of the tool will become very long. To avoid that situation, at the beginning of the function GenPrsc in Algorithm 1, if the total number of logical connectives exceeds a specific threshold, case splitting is used first before the function `metaReduce` at line 16 is invoked to reduce the term te. The technique, therefore, is called "case splitting is used first before reduction." By using case splitting in advance, the property will be rewritten to a term with fewer logical connectives in each sub-case so that `metaReduce` can reduce the term in a reasonable amount of time.

Algorithm 3 shows the precise algorithm dealing with this problem, in which the function GenPrsc is revised. We define: (1) a flag, namely $csFirst$, to enable/disable the use of this technique, and (2) a threshold of logical connectives, namely $maxLogConn$, such that if the number of logical connectives in the given term exceeds $maxLogConn$, case splitting is used; otherwise, `metaReduce` is invoked as usual. If the flag is enabled, the algorithm first tries to rewrite the induction hypothesis $inv(v, d_1, \ldots, d_n)$ within only one rewriting step (line 3), thanks to the Maude built-in function `metaRewrite`. Because only one rewriting step is allowed, this invocation is much less expensive than the one with the function `metaReduce`, which tries to reduce the given term to canonical form. If the number of logical connectives in the obtained result t is greater than the threshold (line 4), then the algorithm extracts and selects a sub-term from t by the function ChooseATermCS to apply case splitting. The procedure is recursively recalled until the number of logical connectives in t is smaller than the threshold. The technique, however,

will make the number of open-close fragments in the generated proof score increase, which is understandable because case splitting is extensively used.

In the experiment with the MCS protocol which will be shown later on (Section 3.4), by choosing a suitable value of $maxLogConn$, this technique can reduce by nearly 65% the time taken by IPSG in generating the proof scores. What value should be chosen for the threshold is a subtle job. Generally speaking, a small value should be chosen so that the cost of invoking the function `metaReduce` would not be expensive. However, choosing a too-small value may backfire because a huge number of open-close fragments are generated, making the handling time in total longer than our expectation. We will soon come back to discuss this problem in the section describing the MCS experiment.

3.3.5 Other features

In addition to what has been described above, we mention in this section two other helpful features that are implemented in the tool.

Keywords `only` and `exclude`

The tool provides two useful keywords: `only` and `exclude`. The former supports generating proof for only induction cases specified. Whereas, the latter supports generating proof excluding some induction cases specified. They are helpful when conducting formal verification with a large CafeOBJ specification. In practice, when conducting formal verification, we start with a property that we aim to verify that it is actually invariant. The proof attempt produced by the tool typically contains several fragments/cases where false is returned for the reduction command. To discharge those cases, we first need to conjecture certain auxiliary lemmas and then rerun the tool again with the uses of these conjectured lemmas. However, it is redundant to ask the tool to generate the proof score for all induction cases again. Instead, it is enough to re-tackle only the induction cases in which false is returned for some open-close fragments. For this purpose, we can use the keyword `only` to specify such induction cases or use the keyword `exclude` to ignore other induction cases.

Use more than one lemma

The algorithm for finding and using suitable lemmas presented in Section 3.3.2 does not mention the case in which we need to use more than one lemma to discharge a sub-case. Even though, the tool does support such a situation. The tool has a configuration variable, namely $depthTryUseLm$, which is set by human users and receives 0 as its default value. For each open-close fragment in which false is returned, $depthTryUseLm$ specifies how many times the tool tries to split the case and find lemmas that can be used to discharge the sub-cases. For

Table 3.1: Experimental results

Name	**No. lines**	**No. properties**	**No. lemmas**	**Time (s)**	**No. open-close fragments**
TAS	36	1	1	0.02	22
Qlock	78	1	1	0.05	32
ASDS-RT	141	1	3	0.2	38
Cloud	95	1	6	0.24	132
IFF	176	1	1	0.17	50
A-Anderson	103	1	8	0.48	190
SDS	358	1	8	2.69	403
Suzuki-Kasami	386	1	5	4.18	374
MCS	288	1	7	110.66	665
NSLPK†	441	3	20	27.05	1534
TLS 1.2†	1296	3	17	497.4	2537

†: Invariants and lemmas are split into some smaller groups and IPSG is asked to generate proof scores for each group one by one.

instance, when $depthTryUseLm$ is 1, for each case in which false is returned, if IPSG finishes running Algorithm 2 but cannot find a suitable lemma to discharge the proof, the tool will try to split the current case into two sub-cases by some equation/term such that one of the sub-cases can be discharged by some lemma. For the other sub-case, the tool executes Algorithm 2 once again to find some other lemma to discharge the sub-case.

From our experience, for most cases, $depthTryUseLm$'s default value, i.e., 0, is enough to complete a formal verification problem when suitable additional lemmas are provided. However, with the Suzuki-Kasami protocol, which will be discussed later in Section 3.4, to generate proof scores in which all fragments are reduced to true, we need to set $depthTryUseLm$'s value to 1.

3.4 Experimental evaluations

The complete implementation of IPSG consists of about 3250 lines of Maude code, excluding the source code of CafeInMaude. Let us illustrate how to use the tool to generate the proof scores to verify the *mutual exclusion* property of the Qlock protocol described in Section 2.4. Recall that the *mutual exclusion* property is proved with one additional lemma. The following script is used to generate the proof scores:

```
load qlock.cafe .
ipsgopen QLOCK .
  inv inv1(S:Sys, P:Pid, Q:Pid) .
  inv inv2(S:Sys, P:Pid) .
  generate inv1(S:Sys, P:Pid, Q:Pid) induction on S:Sys .
  generate inv2(S:Sys, P:Pid) induction on S:Sys .
```

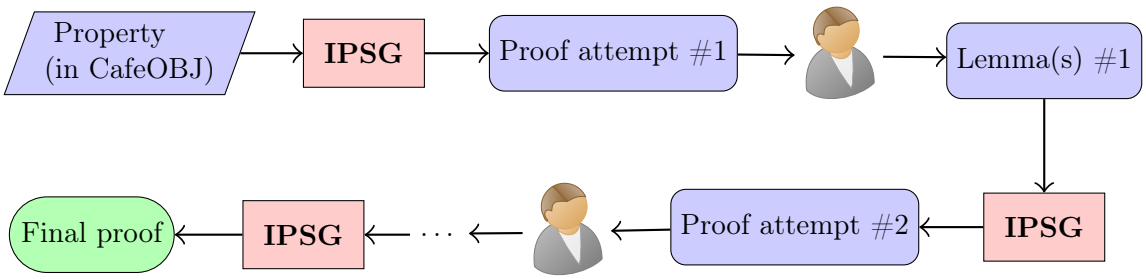


Figure 3.3: A typical verification process

```
close
set-output proof-scores.cafe .
:save-proof .
```

Suppose that the CafeOBJ specification of the protocol is saved in the file named `qlock.cafe`, the **`load`** command reads the file and loads the specification. The two **`inv`** commands declare the two invariants to be used as lemmas during the simultaneous induction proof. The two `generate` commands ask the tool to generate the proof scores of the two invariants, where simultaneous induction is used on the argument `S:Sys`. The generated proof scores are then saved to the file `proof-scores.cafe` by the last two commands. IPSG successfully generates the proof scores after about 50 milliseconds on a MacBook Pro i7 2.3 GHz, 32 GB memory. The result consists of 32 open-close fragments in total.

To demonstrate the practicability of IPSG, in addition to Qlock, we have conducted experiments with ten other systems/protocols including mutual exclusion protocols, classical security protocol, a real-time system, and a real cryptographic protocol widely used every day. The experimental results of those protocols are reported in Table 3.1. The experiments have been conducted on a MacBook Pro i7 2.3 GHz with 32 GB of memory. In the table, the second column shows the number of lines of CafeOBJ code of each formal specification (including invariants and lemmas specification). The third and fourth columns indicate the number of properties to be verified and the number of lemmas needed to complete the verification of each protocol (the lemmas are also invariant). The time taken by IPSG to generate the proof scores is shown in the fifth column (in seconds). The last column indicates the number of open-close fragments in the generated proof scores of each experiment.

There are two things that we need to emphasize. Firstly, all auxiliary lemmas are supposed to be provided by human users, in which IPSG provides good hints for conjecturing them. Secondly, even if auxiliary lemmas are missing, IPSG will still produce a list of open-close fragments, but there may exist some fragments in which `false` is returned (they will be indicated by the tool). If all auxiliary lemmas are provided in advance, we can simply ask IPSG to generate the proof scores for the property under verification as well as those lemmas. In fact, however, such auxiliary lemmas are not available in advance when we try to verify a given property. In that case, the

verification process normally follows Figure 3.3. The user conducting verification first uses IPSG to generate the proof score attempt of the property. In that generated proof, there may exist some sub-cases in which `false` is returned, and then the user is supposed to conjecture some additional lemma(s) to discharge the sub-cases. The property and the conjectured lemmas are fed into IPSG, asking it to produce the proof again for the property as well as the proof attempt for those new lemmas. The second proof attempt produced may require the user to conjecture some other lemmas. The process is repeated until the final proof contains no sub-case in which `false` is returned.

In summary, IPSG can automatically conduct case splitting such that for each sub-case either true or false is returned. Each case in which true is returned has been discharged or proved. For each case in which false is returned, users are supposed to conjecture lemma(s) to discharge the case. Thus, IPSG allows human users to concentrate on the most difficult task in interactive theorem proving, that is, lemma conjecture. Nevertheless, the case produced by IPSG (in which false is returned) does provide good hints for conjecturing lemmas. By looking into the equations characterizing the case, the users are supposed to find out a contradiction among them and based on that contradiction conjecture a lemma. This has been illustrated in Section 2.4, where we explained how to conjecture lemma `inv2` to solve a sub-case of `inv1` in the verification of the Qlock protocol. By using the tool, more importantly, lots of manual effort can be saved. For example, with the verification of the TLS 1.2 protocol, there are 20 proof scores, where each of them consists of around 100-200 open-close fragments, around several thousand lines of CafeOBJ code. It would cost lots of time and effort to manually write such proof scores.

In the following, we give a brief description for each verification experiment in Table 3.1.

Test and Set (TAS)

loop { "Remainder Section"
  rs : **repeat while** test&set(*locked*);
  "Critical Section"
  cs : *locked* := false; }

Figure 3.4: TAS protocol

This is a mutual exclusion protocol that uses the atomic instruction test&set. The pseudo-code of the protocol is depicted in Figure 3.4, where *locked* is a Boolean variable shared by all processes and initially is false. Each process is located at either rs (Remainder Section) or cs (Critical Section) and initially at rs. test&set(*locked*) atomically does the following: if *locked* is false, then it sets *locked* to true and returns false; otherwise, it just returns true.

A desired property of TAS that we want to verify is the *mutual exclusion* property whose informal description is that there is always at most one process located at the Critical Section. To complete the formal verification of the *mutual exclusion* property, one additional lemma is used. The complete specification of the protocol consists of 36 lines of CafeOBJ code, which is a quite simple mutual exclusion protocol.

ASDS-RT

This is an Asynchronous Data Sending system, a real time system [114] (Asynchronous Data Sending - Real Time). In this system, there exist a sender and a receiver. The sender repeatedly sends natural numbers to the receiver one by one from zero in ascending order via a cell. The sender puts a natural number into the cell and the receiver gets a natural number from the cell if the cell is not empty. The receiver on reception of a number puts it into a list owned by his/her. If the sender puts a natural number in the cell that is not empty, some natural numbers will be lost, i.e., such numbers will not received by the receiver. The sender, however, is not able to check whether the cell is empty or not. Instead, the system uses timing constraints to guarantee that no natural numbers sent by the sender will be lost, that is, the sender should not put a new natural number into the cell before the receiver gets a natural number from the cell provided that the cell is not empty. In other words, the receiver should get a natural number from the cell before the sender puts a new natural number into the cell provided that the cell is not empty. To this end, two time units d_{rec}^{max} and d_{send}^{min} are used, where:

- d_{rec}^{max}: the receiver is forced to get a natural number from the cell within this time units after the number is put into the cell by the sender.
- d_{send}^{min}: After putting a natural number into the cell, the sender must not send a new number within this time units.
- $d_{rec}^{max} < d_{send}^{min}$.

The challenging task is to describe real numbers in the CafeOBJ specification of this system. The verified property is described as follows: no natural number sent by the sender is lost. The complete specification of the protocol consists of 141 lines of CafeOBJ code. To complete the formal verification of that property, three auxiliary lemmas with proofs and a trivial lemma without proof on real numbers are used. The trivial lemma is as follows:

```
eq lemma1(T,T1,T2) = (T <= T1 and T1 <= T2)
    implies T <= T2 .
```

It states that if a real number `T` is not greater than `T1`, and `T1` is not greater than `T2`, then `T` is not greater than `T2`. We do not prove it due to its obviousness. Equations cannot be used

loop { "Idling"
gotval : $status_p := \text{gotval};\ status_c := \text{busy};\ temp_p := value_c;$
updated : $status_p := \text{updated};$
if $temp_p \leq value_p$
$value_c := value_p;\ temp_p := value_p;$
else
$value_p := temp_p;$
finished : $status_p := \text{idle};\ status_c := \text{idle};\ temp_p := 0;$ }

Figure 3.5: Cloud protocol: synchronization between a client (p) and the cloud (c)

to define that lemma, but we need to use the operator `lemma1` because CafeOBJ does not allow equations with variables on the right-hand side do not appear on the left-hand side.

Cloud

This is a simplified cloud synchronization protocol [125]. In the protocol, there is a cloud computer together with many PCs (clients), and they try to keep a value synchronized, i.e., new values appearing in the PCs must be uploaded to the cloud and, similarly, PCs must retrieve new values from the cloud. Figure 3.5 shows how synchronization happens between a client (p) and the cloud (c). The cloud c is represented by its current *value*, which is a natural number, and its *status*, which is either idle or busy. Each client p is represented by its temporal value fetched from the cloud denoted by $temp_p$, its current value denoted by $value_p$, and its status denoted by $status_p$, which is either idle, gotval, or updated. Initially, the *status* of the cloud and each client is idle. A client can increment its $value_p$ at any time, which is not shown in Figure 3.5. Synchronization between a client and a server depicted in the figure cannot be started unless the statuses of both the client and the cloud are idle. The actions at each step gotval, updated, and finished are atomically executed.

The protocol is formally verified that it enjoys the *synchronization* property, which states that if a client has its status updated, then the client has the same value as the cloud. Six auxiliary lemmas are used to complete the formal verification. The complete specification of the protocol including the seven invariants consists of 95 lines of CafeOBJ code.

Identify Friend or Foe (IFF)

This is an authentication protocol [8] that verifies whether a principal is a member of a group. There are some different groups, where each group is given a unique symmetric key in advance

and each principal belongs to only one group. Let A and B denote two principals. The protocol consists of two messages exchanged as follows:

$$\begin{aligned} &\text{Check} \quad A \rightarrow B \ : \ r \\ &\text{Reply} \quad B \rightarrow A \ : \ \text{senc}(k, r\,; B) \end{aligned}$$

where *senc* denotes symmetric encryption and ; is the concatenation operator. Whenever A wants to determine whether B is also a part of A's group, A first selects a fresh random r and sends it to B via a Check message. Upon receiving that message, B replies to A with a Reply message whose content is the received random and the ID of B encrypted by the symmetric key of B's group, i.e., k. Upon receiving that Reply message, A will know that B also belongs to the same group if A successfully decrypts the ciphertext using their group's symmetric key and the plaintext contains r and B.

A desired property of IFF that we want to verify is the *identifiable* property, whose informal description is as follows: if principal A receives a valid Reply message and A believes that the message was sent by B, B belongs to the same group with A. This property is proved with the use of an auxiliary lemma. The complete specification of the protocol consists of 179 lines of CafeOBJ code.

A-Anderson

loop { "Remainder Section"
rs : $place[i] = \text{fetch\&inc}(next)$;
ws : **repeat until** $array[place[i]]$;
"Critical Section"
cs : $array[place[i+1]] := \text{true}$; }

Figure 3.6: A-Anderson protocol

This protocol [145] is a revised and abstract version of the Anderson mutual exclusion protocol [9]. We suppose that there are N processes participating in the protocol. The pseudo-code of the protocol for each process i is depicted in Figure 3.6. Each process is located at rs, ws or cs and initially located at rs. *place* is an array of size N whose each element stores one from $\{0, 1, \ldots, N-1\}$. Initially, each element of *place* can be any from $\{0, 1, \ldots, N-1\}$ but is 0 in this thesis. Although *place* is an array, each process i only uses $place[i]$ and then we can regard $place[i]$ as a local variable to each process i. *array* is an infinite Boolean array. Initially, $array[0]$ is true and $array[j]$ is false for any non-zero natural number j. *next* is a natural number variable

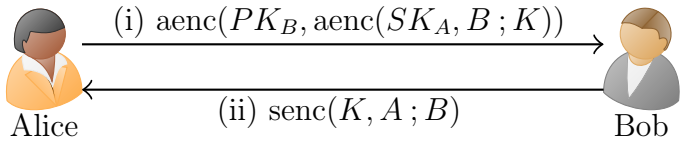


Figure 3.7: Two messages exchanged in the SDS protocol

and initially set to 0. fetch&inc(*next*) atomically does the following: setting *next* to *next* + 1 and returning the old value of *next*.

The *mutual exclusion* property of this protocol is verified with eight additional lemmas. The complete specification of the protocol consists of 103 lines of CafeOBJ code.

SDS

This protocol is a simplified version of the classical key distribution protocol proposed by Denning and Sacco [51]. The protocol is called SDS (Simplified Denning-Sacco) in this thesis. Digital certificates and timestamps are excluded in our simplified version. The purpose of the protocol is to securely distribute a secret key between principals. Let A and B denote two principals, the message exchanged in the protocol is depicted in Figure 3.7. aenc and senc denote asymmetric encryption and symmetric encryption, respectively. Each principal A has a pair of public and private keys denoted by PK_A and SK_A, respectively, that can be used for asymmetric encryption and decryption. As usual, the public key of a principal is known by everyone, whereas, nobody knows the private key of others. K denotes a secret key, which is unguessable, and ; is the concatenation operator. The two messages exchanged between A and B can be explained as follows. A first selects a secret key K, and encrypts it together with the identifier of B under the private key of A, obtaining a ciphertext. The ciphertext is once more encrypted by the public key of B, and then A sends the obtained result to B. When B receives the message from A, B consecutively decrypts the content received twice respectively with his/her private key and the public key of A. If the two decryptions are successful and the final obtained plaintext consists of his/her identifier and a key K, then B responds back to A with the identifiers of A and B symmetrically encrypted by the key K in order to prove the possession of the secret key.

We formally verify the *key secrecy* property, which states that a secret key can be securely distributed to principals, or in other words, two principals can establish a key that cannot be compromised even by an active attacker placed in the middle of the connection. The property is proved with eight auxiliary lemmas. The complete specification of the protocol consists of 403 lines of CafeOBJ code.

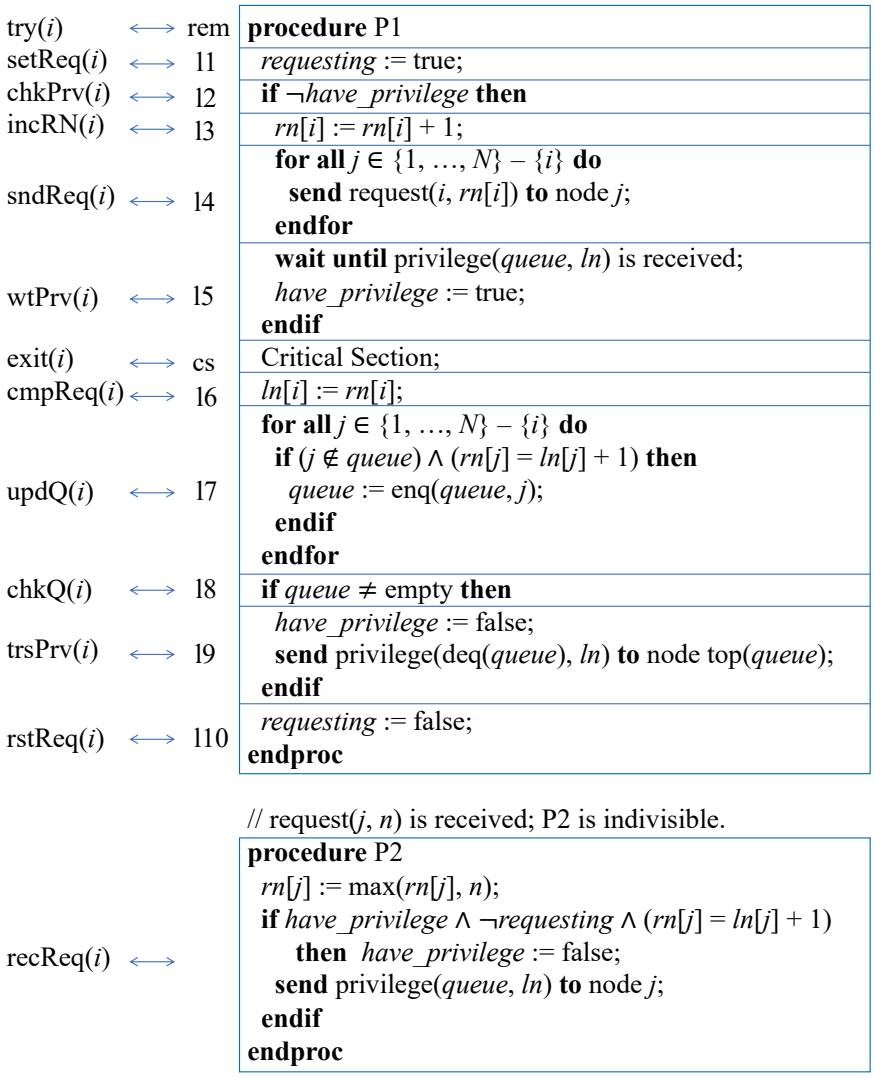


Figure 3.8: Suzuki-Kasami protocol

Suzuki-Kasami

Suzuki-Kasami is a distributed mutual exclusion protocol [141]. The name Suzuki-Kasami came from its authors' names, namely Ichiro Suzuki and Tadao Kasami. The protocol is designed to work over the network with the participation of multiple nodes. The key idea of the protocol is a shared privilege, in which a node cannot enter the critical section unless it owns the privilege, and the privilege can be transferred between nodes in the network. Suppose that there are N nodes participating in the protocol, where $1, \ldots, N$ are used as their identifiers. The pseudo-code of the protocol for each node i is shown in Figure 3.8. A node i can send a request message, which is in the form of request(i, n), to another node to request for the privilege, where n is a natural number that identifies the request number. A node can send a privilege message, which is in the form of privilege(q, a), to another node to transfer the privilege after it exits the critical

section, where q is a queue of node IDs and a is an N-array of natural numbers.

Each node i maintains the following local variables:

- *requesting*: a Boolean variable, which is true if the node wants to enter the critical section; otherwise, it is false.
- *have_privilege*: a Boolean variable, which is true if the node currently owns the privilege; otherwise, it is false.
- *queue*: a queue of node IDs that are requesting to enter the critical section.
- ln: an N-array of natural numbers, where $ln[j]$ is the request number of the request of node j granted most recently.
- rn: an N-array of natural numbers recording the largest request number ever received from each other node.

Figure 3.8 consists of two procedures, namely P1 and P2. The former is invoked when a node i attempts to enter the critical section. First, *requesting* is set to true. If i owns the privilege, it directly moves to the critical section. Otherwise, it increments $rn[i]$ and sends the request message, i.e., request$(i, rn[i])$, to all other nodes in the network. Then, i waits for the privilege. Once the privilege is received, it updates its queue and ln by the ones received in that privilege message, sets *have_privilege* to true, and moves to the critical section. When i leaves the critical section, it updates $ln[i]$ by $rn[i]$. After that, i checks for each node j if j is waiting to enter the critical section ($rn[j] = ln[j]+1$) and j is not in the queue maintained by i ($j \notin queue$). If that is the case, j is put into the queue. After that, if the queue is empty, i sets *requesting* to false and keeps the privilege. Otherwise, *have_privilege* is set to false and i transfers the privilege to the node at the top of the queue by sending the message privilege(deq($queue$), ln) to it.

Procedure P2 is invoked when node i receives a request message in the form of request(j, n) from node j. Note that the procedure is atomically executed.

The *mutual exclusion* property is formally verified, i.e., two different nodes cannot simultaneously access the critical section. The property is proved with five more lemmas. During the process of proof score generation, there exist some open-close fragments such that the tool needs to use more than one lemma to discharge each of the cases as discussed in Section 3.3.5. Precisely, we need to set the value of $depthTryUseLm$ to 1 so that all open-close fragments in the generated proof scores are reduced to true. The complete specification of the protocol consists of 386 lines of CafeOBJ code.

```
rs  : "Remainder Section"
l1  : next_p := nop;
l2  : prede_p := fetch&store(glock, p);
l3  : if prede_p ≠ nop {
l4  :    lock_p := true;
l5  :    next_{prede_p} := p;
l6  :    repeat while lock_p; }
cs  : "Critical Section"
l7  : if next_p = nop {
l8  :    if comp&swap(glock, p, nop)
l9  :       goto rs;
l10 :    repeat while next_p = nop; }
l11 : lock_{next_p} := false;
l12 : goto rs;
```

Figure 3.9: MCS protocol

MCS

MCS is a mutual exclusion protocol invented by Mellor-Crummey and Scott [102]. Variants of MCS have been used in Java VMs and therefore the 2006 Edsger W. Dijkstra Prize in Distributed Computing went to their paper [102]. The algorithm inside the MCS protocol is a scalable algorithm for spin locks that generates $O(1)$ remote references per lock acquisition, independent of the number of processes attempting to acquire the lock. Figure 3.9 shows the pseudo-code of the protocol for each process p. MCS uses one global variable *glock* and three local variables $next_p$, $prede_p$ and $lock_p$ for each process p. Process IDs are stored in *glock*, $next_p$, and $prede_p$, while a Boolean value is stored in $lock_p$. There is one special (dummy) process ID, i.e., nop, that is different from any real process IDs. Initially, each of *glock*, $next_p$ and $prede_p$ is set to nop and $lock_p$ is set to false. We suppose that each process is located at one of the labels, such as rs, l1, and cs. Initially, each process is located at rs. When a process wants to enter "Critical Section," it first moves to l1 from rs.

MCS uses two non-trivial atomic instructions: fetch&store and comp&swap. For a variable x and a value a, fetch&store(x, a) atomically does the following: x is set to a and the old value of x is returned. For a variable x and values a, b, comp&swap(x, a, b) atomically does the following: if x equals a, then x is set to b and true is returned; otherwise, false is just returned.

Figure 3.10 graphically visualizes the change of state of MCS when a process p moves to l3 from l2. In the state υ, which is represented by Figure 3.10 (a), processes p, q, and r, located at

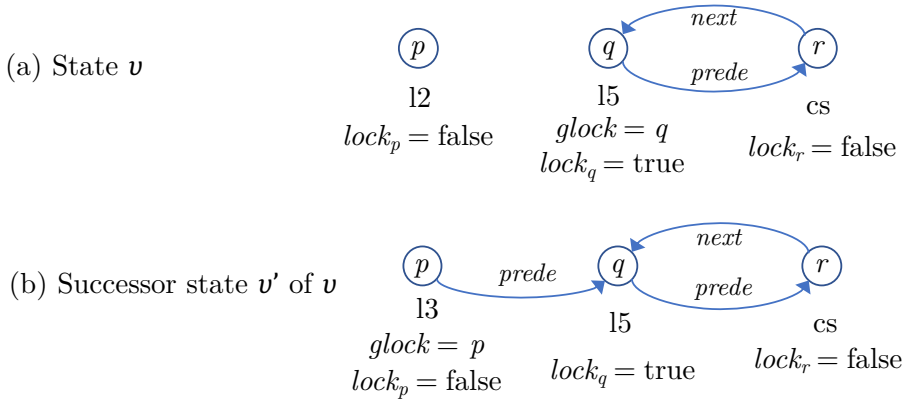


Figure 3.10: The change of state of MCS when a process p moves to l3 from l2

l2, l5, and cs, respectively; *glock* is q; *next* of r is q; and *prede* of q is r. When process p moves to $l3$, *glock* is set to itself, and its *prede* is set to q (Figure 3.10 (b)).

Seven auxiliary lemmas are used to prove that the protocol enjoys the *mutual exclusion* property and IPSG took 110.66 seconds to generate the proof scores of all invariants. Four of these consist of many **and** & **or** logical connectives, for example, one of which is as follows:

```
eq inv7(S,P,Q) = (((pc(S,Q) = l11 or pc(S,Q) = l10 or pc(S,Q) = l8 or pc(S,Q) = l7 or
    pc(S,Q) = cs) and (P = Q) = false) implies ((pc(S,P) = cs or pc(S,P) = l7 or pc(S,P)
    = l8 or pc(S,P) = l10 or pc(S,P) = l11 or (pc(S,P) = l6 and lock(S,P) = false)) =
    false)) .
```

where `pc(S,P)` and `lock(S,P)` denote the location and *lock* of process `P`, respectively, in state `S`. As explained in Section 3.3.4, the time taken to generate the proof score of this kind of property can be reduced by using the technique "case splitting is used first before reduction." Therefore, with this MCS protocol, we conducted some more experiments to check the performance of the tool when enabling that technique with different thresholds of the number of logical connectives. The experimental results are shown in Table 3.2 and graphically visualized in Figure 3.11. In the table, the second column denotes the threshold values used, i.e., 20, 15, 10, 7, and 5. The symbol ∞ denotes that the technique is not used (we can say that the threshold, in this case, is infinity). The third column shows the time taken by IPSG to generate the proof scores with each threshold. The fourth column shows how much improvement in the time taken for each experiment is in comparison to when the technique is not used. The last column shows the number of open-close fragments in the generated proof scores for each experiment. When the technique is not enabled, IPSG took about 110.66 seconds to generate the proof scores consisting of 665 open-close fragments in total. When the technique is enabled with the threshold is 20, the time taken by the tool was significantly reduced by 54.14% down to 50.75 seconds. In contrast, the number of open-close fragments increased to 1788, which is understandable because case

Table 3.2: Experimental results of MCS when "case splitting is used first before reduction" with different thresholds

Protocol	Threshold	Time (s)	Percentage of improvement	No. open-close fragments
MCS	∞	110.66	-	665
	20	50.75	54.14%	1788
	15	48.31	56.34%	1837
	10	39.67	64.15%	2030
	7	38.87	64.87%	2355
	5	40.08	63.78%	2601

splitting is used much more extensively in this experiment. With the thresholds 15, 10, and 7, the time taken was gradually reduced when the threshold is decreased, but there is not a significant improvement when decreasing the threshold from 10 to 7, namely the improvement is less than 1 second. When the threshold is decreased to 5, the time taken was even increased. That is likely because too many case splittings are used and many sub-cases are generated in this experiment (i.e., 2601), hence, although the time taken to handle each sub-case is small enough, the total time to handle all the sub-cases could not become smaller.

The changes in the time taken and the number of open-close fragments generated are graphically visualized in the line chart in Figure 3.11. The red line (with circle points) denotes the change in the time taken to generate the proof scores, while the blue line (with cross points) denotes the change in the number of open-close fragments in the generated proof scores. The experiments give us a suggestion about choosing a suitable value of the threshold of the logical connective: we should choose a small enough value, but it should not be too small to keep the balance with the number of open-close fragments in the generated proof scores, for example, 7 is a good option.

Needham-Schroeder-Lowe Public Key (NSLPK)

The Needham-Schroeder Public Key (NSPK) protocol [107] is a classical authentication protocol, which has the following three messages exchanged:

$$
\begin{aligned}
A \rightarrow B :&\ \mathrm{aenc}(PK_B, A\,; N_A)\\
B \rightarrow A :&\ \mathrm{aenc}(PK_A, N_A\,; N_B)\\
A \rightarrow B :&\ \mathrm{aenc}(PK_B, N_B)
\end{aligned}
$$

where A and B denote Alice and Bob principal identifiers. N_A and N_B are nonces, which are unique and unguessable values, generated by A and B, respectively. Recall that $\mathrm{aenc}(PK_A, m)$

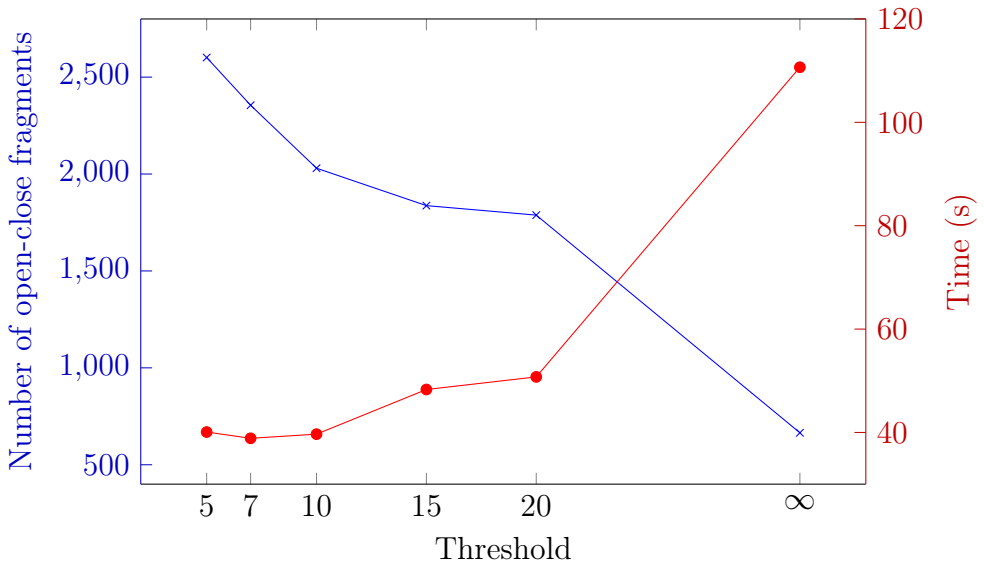


Figure 3.11: Experimental results of MCS when "case splitting is used first before reduction" with different thresholds

denotes asymmetric encryption of message m by the public key of A. The three messages can be explained as follows. A first generates a nonce N_A and sends it together with their ID encrypted by B's public key to B (note that the semicolon denotes the concatenation). Upon receiving that message, B decrypts it and obtains a nonce. The nonce and a newly generated nonce N_B are encrypted by A's public key and then sent back to A. When receiving the message, A decrypts it, getting two nonces, and checking if the first one is exactly the one that A has sent in this session. A finishes the communication by sending to B the other nonce encrypted under B's public key. Lowe found a man-in-the-middle attack on NSPK, and then he proposed a modified version of the protocol, called the Needham-Schroeder-Lowe Public Key (NSLPK) protocol. NSLPK has the following three messages exchanged:

$$\begin{aligned} A \to B &: \text{aenc}(PK_B, A\,;N_A) \\ B \to A &: \text{aenc}(PK_A, N_A\,;N_B\,;B) \\ A \to B &: \text{aenc}(PK_B, N_B) \end{aligned}$$

The first, second, and third messages are respectively called the Challenge, Response, and Confirmation messages. There are three properties are verified:

- the *nonce secrecy* property: all nonces available to the intruder are those created by the intruder or those created for the intruder.
- the *one-to-many correspondence* property from the initiator point of view: whenever A has sent a Challenge message to B and receives a valid Response message apparently from

B, the principal that A is communicating with is indeed B.

- the *one-to-many correspondence* property from the responder point of view: whenever B has sent a Response message to A and receives a valid Confirmation message apparently from A, A has indeed sent that Confirmation message.

20 additional lemmas are used to complete the verification of the three above-mentioned properties. The complete specification of the protocol consists of 441 lines of CafeOBJ code. In this verification experiment, we did not ask IPSG to generate the proof scores at once, but we split the invariants into some smaller groups, and then run the tool with each group one by one.

Transport Layer Security (TLS) 1.2

Transport Layer Security (TLS) is one of the most widely deployed cryptographic protocols in practice, protecting numerous internet communications every day. Although the newest version of TLS is 1.3 [122], its predecessor, namely TLS 1.2 [123], still plays an important role because only a modest number of endpoints support TLS 1.3 so far [137]. That is the reason why the protocol version 1.2 is chosen to conduct formal verification. The protocol consists of two layers (or subprotocols): the TLS Record Protocol and the TLS Handshake Protocol. The TLS Handshake Protocol allows a server and a client to authenticate each other and to negotiate an encryption algorithm and cryptographic keys that will be used later in the Record Protocol. The TLS Record Protocol provides confidentiality and integrity for communication between two parties. That is all messages exchanged are encrypted by using symmetric keys established before in the Handshake Protocol, and each message also includes message integrity check. Hereinafter, if we say TLS without any more specific information, we are talking about the TLS 1.2 Handshake Protocol.

With the handshake, we consider only the most common case, i.e., server authentication is mandatory, while client authentication is not requested. Concretely, a server always sends his/her certificate through a Certificate message to a client in a full handshake but never sends a CertificateRequest message, whereas the client sends neither Certificate nor CertificateVerify messages. Furthermore, ServerHelloDone and ChangeCipherSpec messages are omitted. Those assumptions help the ease of the verification, but the protocol is still much more complex than classical security protocols, such as SDS. This can be seen through the experimental results reported in Table 3.1.

Let C and S denote a client and a server, messages exchanged between them are shown in Figure 3.12. The first seven messages are for a full handshake, while the remaining four messages are for an abbreviated handshake, which is a session resumption of a previously established session. Rand_X denotes a unique random generated by principal X. PK_X and SK_X denote the

$$
\begin{array}{rrcl}
\text{ClientHello} & C \to S & : & \text{Rand}_C, \text{CipherSuites} \\
\text{ServerHello} & S \to C & : & \text{Rand}_S, \text{CipherSuite}, \text{SID} \\
\text{ServerCert} & S \to C & : & \text{Certificate}_S \\
\text{ServerKeyEx} & S \to C & : & \text{PublicKeyShare}_S, \text{SignParams} \\
\text{ClientKeyEx} & C \to S & : & \text{PublicKeyShare}_C \\
\text{ClientFinished} & C \to S & : & \varepsilon_{\text{HS}_C}(\text{CFin}) \\
\text{ServerFinished} & S \to C & : & \varepsilon_{\text{HS}_S}(\text{SFin}) \\
 & & & \\
\text{ClientHello2} & C \to S & : & \text{Rand}_C, \text{SID}, \text{CipherSuites} \\
\text{ServerHello2} & S \to C & : & \text{Rand}_S, \text{SID}, \text{CipherSuite} \\
\text{ServerFinished2} & S \to C & : & \varepsilon_{\text{HS}_S}(\text{SFin2}) \\
\text{ClientFinished2} & C \to S & : & \varepsilon_{\text{HS}_C}(\text{CFin2})
\end{array}
$$

Figure 3.12: Messages exchanged in the TLS Handshake Protocol

long-term public and private keys of X, respectively. $\varepsilon_K(\text{PlainText})$ denotes the ciphertext obtained by encrypting the PlainText with key K (either symmetric or asymmetric). CipherSuites denotes a list of cipher suites offered by the client, while CipherSuite denotes a cipher suite selected by the server. Composite data used in the protocol are as follows:

- Certificate_X: $\{X, \text{PK}_X\}_{CA}$ - the certificate of X signed by the trustable certificate authority CA.
- SignParams: $\varepsilon_{\text{SK}_S}(\text{Rand}_C, \text{Rand}_S, \text{PublicKeyShare}_S)$ - the signature over the server's key exchange parameters signed under the server's long-term private key, where PublicKeyShare_S is an ephemeral public key of S.
- Context: $\{C, S, \text{Rand}_C, \text{CipherSuites}, \text{Rand}_S, \text{CipherSuite}, \text{SID}, \text{Certificate}_S, \text{PublicKeyShare}_S, \text{SignParams}, \text{PublicKeyShare}_C\}$ - the concatenation of all messages exchanged from ClientHello to ClientKeyEx.
- CFin: $PRF(\text{MS}, \text{“client finished”}, H(\text{Context}))$ - where H and PRF are a hash function and a pseudorandom function, respectively, MS is the master secret key, which is computed as explained below.
- SFin: $PRF(\text{MS}, \text{“server finished”}, H(\text{Context}))$.
- Context2: $\{C, S, \text{Rand}_C, \text{SID}, \text{CipherSuites}, \text{Rand}_S, \text{CipherSuite}\}$ - the concatenation of two messages exchanged ClientHello2 and ServerHello2.

- SFin2: PRF(MS, "server finished", H(Context2)).
- CFin2: PRF(MS, "client finished", H(Context2)).

Note that CipherSuites contains not only the cryptographic options supported by the client but also other necessary information, such as the version of the protocol by which the client wishes to communicate during this session. CipherSuite is understood in a similar fashion. We assume that CA is the only trustable certificate authority. There are three types of secret keys, which are calculated as follows:

- PMS - pre-master secret, computed from the ephemeral public key PublicKeyShare_S (or PublicKeyShare_C) and the corresponding ephemeral secret key of C (or S).
- MS - master secret, computed by $\text{MS} = PRF(\text{PMS}, \text{"master secret"}, \text{Rand}_C, \text{Rand}_S)$.
- HS_C and HS_S - symmetric handshake keys of client and server, respectively, computed by $(\text{HS}_C, \text{HS}_S) = PRF(\text{MS}, \text{"key expansion"}, \text{Rand}_\text{C}, \text{Rand}_\text{S})$.

The following three properties are verified:

- the *key secrecy* property: the negotiation of handshake keys between two honest principals is secure, that means nobody except for the honest client and the honest server who established the handshake keys can learn the shared keys.
- the *authentication* property with respect to a full handshake: when honest client C has received a valid Finished message in a full handshake apparently from server S, then S has indeed sent that message.
- the *authentication* property with respect to an abbreviated handshake: the property has a similar description to the previous one, except for now, the property is stated with respect to an abbreviated handshake.

TLS can be regarded as the most complicated one among the protocols used in our experiments. Indeed, the complete specification of TLS consists of 1296 lines of CafeOBJ code, and the proof scores to verify the three above-mentioned properties consist of 2537 open-close fragments, which are larger than any other protocols.

3.5 Confirming correctness of proof scores with CiMPG and CiMPA

We confirm the correctness of the generated proof scores in our experiments by employing the CafeInMaude Proof Assistant (CiMPA) and Proof Generator (CiMPG) [126]. The proof

Table 3.3: Time taken by CiMPG to generate proof scripts from proof scores

Protocol	**No. invariants**	**No. open-close fragments**	**Time (h:m:s)**
TAS	2	22	0:00:03
Qlock	2	32	0:00:06
Cloud	7	132	0:00:33
IFF	2	50	0:00:27
A-Anderson	9	190	0:03:06
SDS	9	403	0:10:59
Suzuki-Kasami	6	374	1:08:37
MCS	8	665	1:05:00
NSLPK	23	1534	2:11:00
TLS 1.2	20	2537	9:45:00

generator CiMPG takes proof scores as input and generates another kind of formal proof, called proof script, that can be checked by the proof assistant CiMPA. Traditional, with hand-written proof scores, if CiMPG successfully generates the proof scripts from those proof scores, and the generated proof scripts are successfully checked by CiMPA, we can confirm that there is no human error lurking in the hand-written proof scores. In this section, we employ the two tools to confirm the correctness of the proof scores generated by IPSG, and subsequently, can confirm the correctness of IPSG to some extent as well as there is no flaw in the tool implementation. The theory foundations of CiMPA and CiMPG are given in the article [126]. We also refer readers to that article for the syntax of proof scripts. The two kinds of formal proofs in comparison, proof scores are easier for human users to comprehend and/or to analyze a proof than proof scripts, for example, to conjecture a new lemma candidate when encountering a case that requires an additional lemma. Therefore, our tool, IPSG, generates proof scores but not proof scripts.

To use CiMPG, all we need to do is to annotate each open-close fragment in the input proof scores with an identifier and then feed the annotated proof scores into CiMPG. For example, the proof fragment (`try-2`) in Section 2.4, by annotating identifier `qlock`, becomes as follows:

```
open QLOCK .
  :id(qlock)
  op s : -> Sys .   ops i j k : -> Pid .
  eq (pc(s,k) = ws) = false .
  red inv1(s,i,j) implies inv1(try(s,k),i,j) .
close
```

Table 3.3 shows the time taken by CiMPG to generate the proof scripts from the proof scores of each verification experiment. All of the generated proof scripts are successfully checked by CiMPA, confirming the correctness of the proof scores generated by IPSG. The second and third columns of the table recall the number of invariants (properties and lemmas) and the number of

open-close fragments in the proof scores of each verification. With the MCS protocol, we use the most compact proof scores, namely the one with 665 open-close fragments. It can be seen from the table that with simple systems/protocols like TAS and Qlock, CiMPG only took several seconds to produce the proof scripts from the proof scores. However, with the more complicated protocols and properties, the time taken quickly increases as the number of open-close fragments increases. With the TLS 1.2 protocol, CiMPG even took nearly 10 hours to complete the job. CiMPA may take up to several minutes to check the generated proof scripts of each experiment.

3.6 Lemma weakening - a technique for lemma conjecture in invariant proofs

Lemma conjecture is the creativity task that IPSG leaves to the users conducting formal verification. In the simultaneous induction proof method, to prove that a state predicate p_1 is an invariant wrt an OTS $\mathcal{S}$, we typically prove a stronger version of it, namely p, in the form of $p_1 \wedge p_2 \wedge \ldots \wedge p_k$, where $p_2, \ldots, p_k$ are the lemmas of the proof. We propose a lemma conjecture technique called Lemma Weakening, which replaces some lemma p_i with a weaker version of it. The technique is used to conjecture the lemmas for the MCS's formal verification.

Definition 5. A state predicate $p : \Upsilon\ \mathbf{D}_p \rightarrow \text{Bool}$ is called an *inductive invariant* wrt an OTS $\mathcal{S}$ if it satisfies the following two conditions:

(i) $(\forall \upsilon \in \mathcal{I})(\forall \mathbf{x} \in \mathbf{D}_p).\ p(\upsilon, \mathbf{x})$

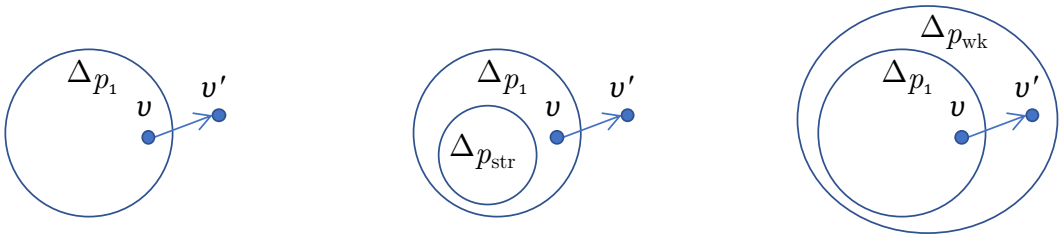
(ii) For each $t \in \mathcal{T}$, where $t : \Upsilon\ \mathbf{D}_t \rightarrow \Upsilon$,
$(\forall \upsilon \in \Upsilon)((\forall \mathbf{x} \in \mathbf{D}_p).\ p(\upsilon, \mathbf{x}) \Rightarrow (\forall \mathbf{y}_t \in \mathbf{D}_t)(\forall \mathbf{x} \in \mathbf{D}_p).\ p(t(\upsilon, \mathbf{y}_t), \mathbf{x}))$

As usual, $\mathbf{D}_p$ and $\mathbf{D}_t$ denote lists of data types. Informally, inductive invariants are invariants preserved by all transitions.

Proposition 1. Inductive invariants wrt $\mathcal{S}$ are invariants wrt $\mathcal{S}$.

Proof. The proof is directly derived from the definition. (Q.E.D.)

However, the vice versa is not correct, namely, an invariant is not always inductive. Let us show a toy example reflecting this case. An OTS is defined by a single observer, namely $m \in \mathbb{Z}$, a single initial state in which $m = 3$, and a single transition, which updates m by $2m - 2$. The predicate $p \triangleq m > 0$ is an invariant wts the OTS, but not an inductive invariant. Indeed, given $m > 0$, it is impossible to prove that $2m - 2 > 0$ ($m = 1$ would falsify that), and thus the condition (ii) of Definition 5 is not satisfied. The predicate $p' \triangleq m > 1$ is an inductive invariant. p' is stronger than p, namely $p' \Rightarrow p$.



(a) The source making the proof non-trivial

(b) p_{str} obtained by Strengthening p_1

(c) p_{wk} obtained by Weakening p_1

Figure 3.13: The reason why invariant proofs become non-trivial and two approaches to tackling the non-trivial situation

3.6.1 Lemma Strengthening

In general, when proving that a state predicate p_1 is an invariant wrt an OTS $\mathcal{S}$, it is often the case that p_1 is not inductive, which means that there exists a transition instance $v \rightarrow v'$ that does not preserve p_1, i.e., $p_1(v)$ holds but $p_1(v')$ does not. This situation is depicted in Figure 3.13 (a), where $\Delta_{p_1} \triangleq \{v \in \Upsilon \mid p_1(v)\}$. This is the reason why invariant proofs become non-trivial or even very hard. If we can show that the source v is unreachable wrt $\mathcal{S}$ (like the case $m = 1$ in the toy example above), then we will not need to consider the transition instance anymore. One possible way to do so is by finding p_{str} such that: (1) it is stronger than p_1, (2) $p_{\text{str}}(v)$ does not hold, and (3) p_{str} is an invariant wrt $\mathcal{S}$. This technique is depicted in Figure 3.13 (b). If p_{str} is inductive wrt $\mathcal{S}$, the verification is done without any more lemmas. This technique has been summarized as the proof rule Inv by Manna and Pnueli [100].

As described in Section 2.2, p_{str} is typically in the form $p_1 \wedge p_2 \wedge \ldots \wedge p_k$. $p_2, \ldots, p_k$ are the lemmas of the proof that p_1 is an invariant of $\mathcal{S}$. However, when doing proof, we do not know any of $p_2, \ldots, p_k$ in advance. Instead, we need to gradually conjecture $p_2, \ldots, p_k$ one by one when we encounter the situation depicted in Figure 3.13 (a). For example, while proving that p_1 is an invariant wrt $\mathcal{S}$, we may conjecture p_2, and when proving p_2, we need to conjecture p_3. Therefore, invariant proofs can be regarded as strengthening lemmas, which is called Lemma Strengthening (LS) in this thesis.

We also need to use LS when conjecturing each individual lemma candidate p_i. For instance, suppose that we need to prove $\text{rev}(\text{rev}(L)) = L$ for all lists L by induction on L, where rev is the reverse function of lists. The equation says that two consecutive reverses of a list L returns L itself. In the induction case, we need to use a lemma, for which the most straightforward lemma would be:

$$\text{rev}(\text{rev}(L) \mathbin{@} (E \mid \text{nil})) = E \mid L \tag{3.1}$$

for all lists L and all elements E, where @ is the concatenation function of lists, | is the constructor of lists, and nil is the empty list. The proof of the lemma 3.1 requires us to use another lemma:

$$\mathrm{rev}(\mathrm{rev}(L) \ @ \ (E_1 \mid E_2 \mid \mathrm{nil})) = E_2 \mid E_1 \mid L \tag{3.2}$$

for all lists L and all elements E_1, E_2. If we only use the most straightforward lemmas, we may move toward the direction in which our proof attempt never converges. To make the proof converge, we need to strengthen such lemma candidates. One possible lemma obtained by strengthening the lemmas 3.1 and 3.2 is:

$$\mathrm{rev}(\mathrm{rev}(L_1) \ @ \ L_2) = \mathrm{rev}(L_2) \ @ \ L_1 \tag{3.3}$$

for all lists L_1, L_2. 3.3 is stronger and more generic than 3.1 and 3.2. By using that lemma, the proof is done.

3.6.2 Lemma Weakening

The reason why invariant proofs become non-trivial or even can become very hard is because there exists a transition instance $\upsilon \rightarrow \upsilon'$ as shown in Figure 3.13 (a). The proof rule Inv gets rid of such a transition instance as shown in Figure 3.13 (b). Another possible way to get rid of such a transition instance is to find p_{wk} that is weaker than p_1 such that $p_{\mathrm{wk}}(\upsilon')$ holds and to prove that p_{wk} is an invariant wrt $\mathcal{S}$. This technique is depicted in Figure 3.13 (c). Even though p_{wk} is an invariant wrt $\mathcal{S}$, it does not guarantee that p_1 is an invariant wrt $\mathcal{S}$. This is because Δ_{p_1} may not contain all reachable states in $\mathcal{R}_{\mathcal{S}}$ even though $\Delta_{p_{\mathrm{wk}}}$ does contain all reachable states in $\mathcal{R}_{\mathcal{S}}$. Therefore, the technique is not suitable for the proof that p_1 is an invariant wrt $\mathcal{S}$. The technique, however, may be useful for some p_i, a lemma of the proof that p_1 is an invariant wrt $\mathcal{S}$. In this thesis, weakening lemmas p_i is called Lemma Weakening (LW). While proving that MCS enjoys the mutual exclusion property, we realized that LW could make the proof attempt to converge that otherwise did not seem to converge in a reasonable amount of time. In the next section, we describe in which way LW is used to complete the verification of MCS.

3.6.3 Use of Lemma Weakening in the MCS's formal verification

We do not go into detail on how to specify the MCS protocol in CafeOBJ as OTS $\mathcal{S}_{\mathrm{MCS}}$. We refer readers to the webpage mentioned at the beginning of this chapter as well as the description of the Qlock protocol specification presented in Section 2.4 because both are mutual exclusion protocols and they share some common parts. Sorts `Sys` and `Pid` represent state space and process IDs, respectively. Given a state `s` and a process `p`, `glock(s)`, `pc(s,p)`, `next(s,p)`, and `prede(s,p)` denote *glock*, the location of `p`, $next_{\mathtt{p}}$, and $prede_{\mathtt{p}}$, respectively.

There are mainly two cases in which LW is used.

Case 1

Let us consider the following predicate, which is a lemma candidate to prove the mutual exclusion property:

```
eq inv4-0(S,P) = ((pc(S,P) = l3 or pc(S,P) = l4 or pc(S,P) = l5 or pc(S,P) = l6 or
  pc(S,P) = cs or pc(S,P) = l7 or pc(S,P) = l8 or pc(S,P) = l10 or pc(S,P) = l11)
implies not(glock(S) = nop)) .
```

where `S` and `P` are variables of states and process IDs. The predicate states that if there exists a process `P` located at `l3` or `l4` or `l5` or `l6` or `cs` or `l7` or `l8` or `l10` or `l11`, then *glock* is different from `nop`, i.e., it is not null. Intuitively, *glock* is the tail of the virtual queue, i.e., the last process requesting to enter the critical section. Thus, if process `P` is located at one of the aforementioned locations, the virtual queue cannot be empty, and subsequently, *glock* - the tail of the virtual queue, cannot be null. From that justification, we strongly believe that `inv4-0` is an invariant wrt $\mathcal{S}_{\text{MCS}}$. However, it seems very tough to prove that. Let us consider a sub-case of an induction case for the proof attempt of `inv4-0`. The open-close fragment of the sub-case is as follows:

```
open INV .
op s : -> Sys .     ops p r : -> Pid .
eq pc(s,r) = l8 .   eq (p = r) = false .
eq glock(s) = r .   eq pc(s,p) = l3 .
red inv4-0(s,p) implies inv4-0(chglk(s,r),p) .
close
```

`chglk` defines the transition in which a process moves to `l9` or `l10` (depending on whether *glock* equals that process or not) from `l8`. CafeOBJ returns `false` for the fragment. Let υ_{40} be an arbitrary state in which the four equations used in the fragment hold. In such a state, there exist two processes `p` and `r` located at `l3` and `l8`, respectively, and *glock* is `r`. Because *glock* is `r`, `r` must be the tail of the virtual queue, so its request to enter the critical section must happen after `p`'s request. Consequently, `r` cannot enter (and exit) the critical section before `p`, which is contradicted by `p` and `r` located at `l3` and `l8`. Therefore, the state υ_{40} would be unreachable, and we need to conjecture another lemma to complete the proof of `inv4-0`. From the four equations used in the open-close fragment, by using LS, we conjecture the following lemma:

```
eq inv4-1(S,P,Q) = ((pc(S,P) = l3 or pc(S,P) = l4 or pc(S,P) = l5 or pc(S,P) = l6 or
  pc(S,P) = cs or pc(S,P) = l7 or pc(S,P) = l8 or pc(S,P) = l10 or pc(S,P) = l11) and
  glock(S) = Q and not(P = Q))
implies not(pc(S,Q) = cs or pc(S,Q) = l7 or pc(S,Q) = l8 or pc(S,Q) = l10 or
  pc(S,Q) = l11 or (pc(S,Q) = l6 and lock(S,Q) = false)) .
```

When attempting to prove `inv4-1`, we encounter a sub-case in which `false` is returned:

```
open INV .
op s : -> Sys .          ops p q r : -> Pid .
eq pc(s,r)  = l11 .      eq next(s,r) = q .
eq glock(s) = q .        eq pc(s,p) = l3 .
eq pc(s,q)  = l6 .       eq lock(s,q) = true .
eq (p = r)  = false .    eq (q = r) = false .
eq (p = q)  = false .
red inv4-1(s,p,q) implies inv4-1(stlnx(s,r),p,q) .
close
```

`stlnx` defines the transition in which a process moves to `l12` from `l11`. There exist other two sub-cases that CafeOBJ also returns `false`. Those two sub-cases differ from the sub-case shown above only in `pc(s,p) = l4` and `pc(s,p) = l5` are respectively used instead of `pc(s,p) = l3`. Let υ_{41} be an arbitrary source state of the three sub-cases. The state is partially visualized in Figure 3.14 (a), saying that `q` is located at `l6`, `r` is located at `l11`, $next_{\mathtt{r}}$ is `q`, *glock* is `q` and $lock_{\mathtt{q}}$ is true. After the transition proceeded, in the successor state, `r` moves to `l12` and `q` has permission to enter the critical section ($lock_{\mathtt{q}}$ is false). Because `p` is located at `l3` (or `l4` or `l5`), it must have requested to enter the critical section and it must be in the virtual queue. `q` is the tail of the queue, but it is granted access to the critical section while `p` is waiting, which is a contradiction. Therefore, υ_{41} would be unreachable. We conjecture another lemma to discharge the sub-cases:

```
eq inv4-2(S,P,Q,R) = (glock(S) = Q and next(S,R) = Q and not(P = R or Q = R or P = Q)
  and (pc(S,R) = cs or pc(S,R) = l7 or pc(S,R) = l8 or pc(S,R) = l10 or
       pc(S,R) = l11 or (pc(S,R) = l6 and lock(S,R) = false)))
implies not(pc(S,P) = l3 or pc(S,P) = l4 or pc(S,P) = l5 or pc(S,P) = l6) .
```

When attempting to prove `inv4-2`, we again encounter a sub-case in which `false` is returned:

```
open INV .
op s : -> Sys .          ops p q r t : -> Pid .
eq pc(s,t)  = l11 .      eq next(s,t) = r .
eq next(s,r) = q .       eq glock(s) = q .
eq pc(s,q)  = l6 .       eq lock(s,q) = true .
eq pc(s,r)  = l6 .       eq lock(s,r) = true .
eq pc(s,p) = l3 .
eq (p = t)  = false .    eq (q = t) = false .
eq (r = t)  = false .    eq (p = r) = false .
eq (q = r)  = false .    eq (p = q) = false .
red inv4-2(s,p,q,r) implies inv4-2(stlnx(s,t),p,q,r) .
close
```

There also exist other two sub-cases that CafeOBJ returns `false`. Those two sub-cases differ from the sub-case shown above only in `pc(s,p) = l4` and `pc(s,p) = l5` are respectively used instead of `pc(s,p) = l3`. Let υ_{42} be an arbitrary source state of the three sub-cases. The state

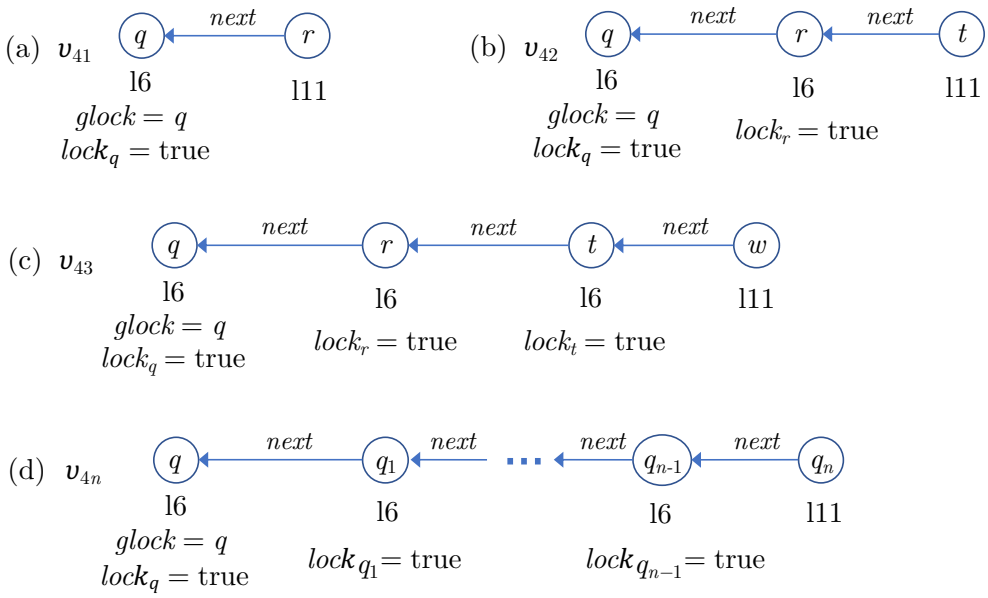


Figure 3.14: States υ_{41}, υ_{42}, υ_{43}, and υ_{4n}

is partially visualized in Figure 3.14 (b), saying that `q` and `r` are located at `l6`, `t` is located at `l11`, $next_{\texttt{r}}$ is `q`, $next_{\texttt{t}}$ is `r`, *glock* is `q`, $lock_{\texttt{q}}$ and $lock_{\texttt{r}}$ are true.

The similarity and difference between υ_{41} and υ_{42} can be visually observed from Figures 3.14 (a) and (b). One process located at l6 is inserted between the two processes in Figure (a) and its *lock* is true, although `t` is used in Figure (b) instead of `r` in Figure (a). What if we keep on doing the proof attempt as we did? If we conjecture a new lemma, say `inv4-3`, to discharge the three sub-cases of `inv4-2` as we did with `inv4-1` and `inv4-2`, we will encounter some sub-cases in which

```
inv4-3(s,p,q,r,t) implies inv4-3(stlnx(s,w),p,q,r,t)
```

reduces to `false` while proving `inv4-3`. Let υ_{43} be an arbitrary source state of such sub-cases, which is partially visualized in Figure 3.14 (c). The difference between Figures (b) and (c) is essentially the same as that of Figures (a) and (b). One more process located at l6 such that its *lock* is true is inserted into the structure constructed with *next* variables. The structure virtually forms the queue in which processes requesting to enter the critical section wait. If we repeat what we did, we will encounter the situation that can be partially visualized as shown in Figure 3.14 (d), which suggests that this way to conjecture lemmas never converges.

There must be a generic lemma that is stronger than all of `inv4-1`, `inv4-2`, etc. similar to $\text{rev}(\text{rev}(L_1) \mathbin{@} L_2) = \text{rev}(L_2) \mathbin{@} L_1$ for the proof of $\text{rev}(\text{rev}(L)) = L$, but we could not construct such a generic one. Instead, we make `inv4-0` weaker, constructing `inv4`:

```
eq inv4(S,P) = (pc(S,P) = cs or pc(S,P) = l7 or pc(S,P) = l8 or pc(S,P) = l10 or
  pc(S,P) = l11 or (pc(S,P) = l6 and lock(S,P) = false) or
  (pc(S,P) = l3 and prede(S,P) = nop))
```

```
implies not (glock(S) = nop) .
```

inv4 is made weaker than inv4-0 by attaching two conditions in the premise part: (1) lock(S,P) = false when pc(S,P) = l6 and (2) prede(S,P) = nop when pc(S,P) = l3. Subsequently, the sub-case of inv4-0 shown above now becomes as follows:

```
open INV .
op s : -> Sys .     ops p r : -> Pid .
eq pc(s,r) = l8 .   eq (p = r) = false .
eq glock(s) = r .   eq pc(s,p) = l3 .
eq prede(s,p) = nop .
red inv4(s,p) implies inv4(chglk(s,r),p) .
close
```

The only difference between the two sub-cases is that the equation prede(s,p) = nop is added in the sub-case of inv4. This sub-case can be discharged by using a lemma, namely inv1, which essentially states that if there exists a process located at l3 and its *prede* is not null, then there does not exist any process located at cs or l7 or l8 or l10 or l11. inv1 is successfully proved with some additional lemmas.

We strongly believe that inv4-0 as well as inv4-1 and inv4-2 are invariants wrt $\mathcal{S}_{\mathrm{MCS}}$. We were, however, not able to construct any generic lemma that is stronger than all of inv4-1, inv4-2, etc., and subsequently, we have not successfully completed the proof of inv4-0. Consequently, we cannot guarantee that inv4-0 is actually an invariant wrt $\mathcal{S}_{\mathrm{MCS}}$. However, by using LW, inv4, a weaker version of inv4-0, has been successfully proved. Note that the proof of the mutual exclusion property requires the use of inv4 (or inv4-0).

Case 2

During the verification, the following lemma is conjectured to complete the verification:

```
eq inv5-0(S,P,Q) = (not(pc(S,Q) = l12 or pc(S,Q) = l1 or pc(S,Q) = rs) and
  next(S,Q) = P)
implies (pc(S,P) = l6 and lock(S,P) = true and prede(S,P) = Q) .
```

A weaker version of it is made by adding the condition not(P = Q) to its premise:

```
eq inv5(S,P,Q) = (not(pc(S,Q) = l12 or pc(S,Q) = l1 or pc(S,Q) = rs) and
  next(S,Q) = P and not(P = Q))
implies (pc(S,P) = l6 and lock(S,P) = true and prede(S,P) = Q) .
```

Intuitively, *next* variables are used to virtually construct the queue of process IDs requesting to enter the critical section. $next_p$ refers to the process that has requested to enter the critical section right after p. Thus, we strongly believe that $next_p$ never receives p as its value, and then the condition not(P = Q) (or not(next(S,Q) = Q)) in the premise of inv5 is unnecessary. In other

words, `inv5-0` is an invariant wrt $\mathcal{S}_{\mathrm{MCS}}$ if `inv5` is an invariant wrt $\mathcal{S}_{\mathrm{MCS}}$. Initially, we conjectured and used `inv5-0`. We realized, however, that the proofs of the two lemmas are totally different.

The proof of `inv5` uses only `inv3` as a lemma and the proof of `inv3` uses only `inv5` as a lemma. On the other hand, the proof of `inv5-0` requires two more lemmas `inv5-1` and `inv5-3` in addition to `inv3`. The proof of `inv5-1` requires `inv3`, `inv5-0`, `inv5-2`, and `inv5-3` as lemmas. The proof of `inv5-2` uses `inv5-3` as a lemma. `inv5-1`, `inv5-2` and `inv5-3` are as follows:

```
eq inv5-1(S,P,Q) = (not(pc(S,P) = l12 or pc(S,P) = l1 or pc(S,P) = rs) and
  not(pc(S,Q) = l12 or pc(S,Q) = l1 or pc(S,Q) = rs) and
  not(next(S,Q) = nop) and not(P = Q))
implies not(next(S,P) = next(S,Q)) .
eq inv5-2(S,P) = not(next(S,P) = P) .
eq inv5-3(S,P) = not(prede(S,P) = P) .
```

`inv5-0` is obtained by removing the seemingly unnecessary assertion `not(next(S,Q) = Q)` from `inv5` to make it stronger. However, to complete the verification of `inv5-0`, eventually, we need to use and prove that assertion, which is in form of the lemma `inv5-2`. Similar to $next_p$, $prede_p$ would never receive p as its value, and then `inv5-3` must be invariant. We realized, however, that it is not straightforward to prove `inv5-3` despite its triviality. Precisely, the proof of `inv5-3` requires six new lemmas, namely `inv5-4`, `inv5-5`, `inv5-6`, `inv5-7`, `inv5-8`, and `inv5-9`. We again refer readers to the webpage mentioned at the beginning of this chapter for their definition and their proofs. In summary, even we could prove `inv5-0`, the proof required nine new lemmas. Whereas, a weaker version of it obtained by using LW, i.e., `inv5`, could be proved without introducing any new lemma. `inv5` is enough for the proof of the mutual exclusion property.

3.7 Limitations

Invariant properties are the only ones that IPSG can produce proofs of. The tool cannot work with (1) liveness properties and (2) safety properties that are not invariant properties (non-invariant safety properties). The simultaneous induction proof method is not feasible to apply to liveness properties because they take future states into account. For a non-invariant safety property, although the tool cannot deal with it, we may overcome this by choosing a suitable way to formalize the system under verification as an OTS so that the property can be specified as an invariant property instead. If the OTS is formalized such that the observers save all necessary information that happened during past state transitions, then the values of the observers in the current state contain all necessary information related to the past states, and consequently, we can specify the property as an invariant property because it is no longer necessary to query the past states. To make it clearer, let us recall the *authentication property* of the TLS 1.2 protocol, which asserts (with respect to a full handshake): when honest client C has received a valid Finished message in a full handshake apparently from server S, then S has indeed sent that

message. Given a state s in which C has just received a valid Finished message, the assertion of the property refers to some past state of s when S has sent the message. However, because the formal specification defines an observer observing the network (messages exchanged), namely nw, where nw in state s saves all messages exchanged by all principals so far, the property can be specified without taking past states into account. Precisely, the assertion "S has indeed sent that message" can be specified as "$msg \in \mathsf{nw}(s)$", where msg denotes the Finished message concerned.

3.8 Summary

Doing formal verification by writing proof scores in CafeOBJ, although has been demonstrated through many conducted case studies in the past that it is powerful and flexible, the process is time-, effort- consuming, and human errors can be lurking in the hand written proofs. To help human users no longer waste time writing proof scores manually and to avoid subtle errors during writing them, this chapter has presented an approach to automation of the proof score writing process and the implementation of the tool supporting it. Given a CafeOBJ formal specification of a protocol and an invariant property specified as a CafeOBJ equation, IPSG can automatically generate the proof score proving that the protocol enjoys the property. The tool has been implemented in Maude, with the use of CafeInMaude, the second major implementation of CafeOBJ in the Maude environment. The experiments have been conducted with various protocols, demonstrating the efficiency and the practicability of the tool. Among them, including classical security protocols, mutual exclusion protocols, and especially, a complicated cryptographic protocol currently in use, i.e., TLS. For each experiment, we have verified the correctness of the generated proof scores with the use of a proof generator and a proof assistant, guaranteeing that there is no subtle error in the generated proof scores.

We have also proposed the Lemma Weakening technique for conjecturing lemmas. A non-trivial invariant typically cannot be proved standalone, instead, we often prove a stronger version of it, which is in the form of a conjunction of that invariant and some auxiliary lemmas. Lemma Strengthening is typically used to make each of such lemmas generic enough, otherwise, its proof may be tough or even impossible. We found, however, that Lemma Weakening, which replaces a lemma with a weaker version of it, was an effective way to make the verification attempt of the MCS protocol converge. Without the use of LW, the MCS verification did not seem to converge in a reasonable amount of time.

In the next chapters, IPSG will be employed to conduct formal verification of a class of protocols, namely post-quantum cryptographic protocols. They are designed as replacements for classical cryptosystems as a precaution against future attacks from quantum computers. Practical quantum computers are promised to become available in near future as a result of advanced research in the field of quantum computing and significant investment from industry

giants in recent years. Therefore, it would be very useful and meaningful to apply the formal verification technique to post-quantum cryptographic protocols.

Algorithm 1 Algorithm for generating proof scores

Require: Module $\mathcal{M}$,
predicate inv : Υ D_1 ... D_n $\rightarrow$ Bool,
argument where induction is used Υ,
possible lemma list lms.

1: $inits \leftarrow$ extract-initial-states($\mathcal{M}, \Upsilon$)
2: $ts \leftarrow$ extract-transitions($\mathcal{M}, \Upsilon$)
3: $d_1, \ldots, d_n \in D_1, \ldots, D_n$ ▷*arbitrary* $d_1, \ldots, d_n$
4: $re \leftarrow$ empty-list()
5: **for each** i **in** $inits$ **do**
6: $p \leftarrow$ empty-prsc() ▷*create an empty fragment*
7: $re \leftarrow$ append(re, GenPrsc($\mathcal{M}, p, inv(i, d_1, \ldots, d_n)$))
8: **end for**
9: **for each** $(\upsilon \rightarrow \upsilon')$ **in** ts **do** ▷υ' *is a successor state of* υ
10: $p \leftarrow$ empty-prsc() ▷*create an empty fragment*
11: $te \leftarrow (inv(\upsilon, d_1, \ldots, d_n) \Rightarrow inv(\upsilon', d_1, \ldots, d_n))$
12: $re \leftarrow$ append(re, GenPrsc($\mathcal{M}, p, te$))
13: **end for**
14: **return** re

15: **function** GenPrsc($\mathcal{M}, p, te$)
16: $t \leftarrow$ `metaReduce`($\mathcal{M}||p, te$) ▷*try to reduce the term*
17: **if** $t =$ true **then return** p
18: **else**
19: **if** $t =$ false **then**
20: **return** HandleFalseCase($\mathcal{M}, p, te, lms$) ▷*try to find a lemma*
21: **else**
22: $tcs \leftarrow$ ChooseATermCS($\mathcal{M}, p, t$) ▷*extract a sub-term for case splitting*
23: **return** append(
GenPrsc($\mathcal{M}$, addEq(p, **eq** tcs = true), te),
GenPrsc($\mathcal{M}$, addEq(p, **eq** tcs = false), te))
24: **end if**
25: **end if**
26: **end function**

Algorithm 2 Finding a suitable lemma for a sub-case of induction cases

Require: Module $\mathcal{M}$,
current open-close fragment p,
term $te \leftarrow (inv(\upsilon, d_1, \ldots, d_n) \Rightarrow inv(\upsilon', d_1, \ldots, d_n))$,
possible lemma list lms.

1: **function** HANDLEFALSECASE($\mathcal{M}, p, te, lms$)
2: $sts \leftarrow$ extract-sub-terms(p) ▷*collect all terms and sub-terms in p*
3: **for each** $lm : \Upsilon\ \bar{D}_1\ \ldots\ \bar{D}_m \rightarrow$ Bool **in** lms **do**
4: **for each** $(\bar{d}_1, \ldots, \bar{d}_m) \in sts \mid \bar{d}_1 \in \bar{D}_1, \ldots, \bar{d}_m \in \bar{D}_m$ **do**
▷*for each term list $\bar{d}_1, \ldots, \bar{d}_m$ in sts such that it can be instantiated to lm*
5: $t \leftarrow$ `metaReduce`$(\mathcal{M}||p, lm(\upsilon, \bar{d}_1, \ldots, \bar{d}_m))$ ▷*note that υ is the source state*
6: **if** $t =$ false **then**
7: **return** p using lemma lm ▷*use lm as a lemma*
8: **end if**
9: **end for**
10: **end for**
11: **return** p ▷*cannot find any suitable lemma*
12: **end function**

Algorithm 3 Case splitting is used first before reduction for induction cases

Require: Module $\mathcal{M}$,
predicate $inv : \Upsilon\ D_1 \ldots D_n \rightarrow \text{Bool}$,
argument where induction is used Υ,
transition $\upsilon \rightarrow \upsilon'$,
term $te \leftarrow (inv(\upsilon, d_1, \ldots, d_n) \Rightarrow inv(\upsilon', d_1, \ldots, d_n))$,
possible lemma list lms,
flag $csFirst$,
logical connective threshold $maxLogConn$.

1: **function** GenPrsc($\mathcal{M}, p, te$)
2: **if** $csFirst$ **then**
3: $t \leftarrow$ `metaRewrite`$(\mathcal{M}||p, inv(\upsilon, d_1, \ldots, d_n), 1)$ ▷*rewrite only one step*
4: **if** NoLogConn(t) $> maxLogConn$ **then**
5: $tcs \leftarrow$ ChooseATermCS($\mathcal{M}, p, t$) ▷*extract a sub-term for case splitting*
6: **return** append(
GenPrsc($\mathcal{M}$, addEq(p, **eq** tcs = true), te),
GenPrsc($\mathcal{M}$, addEq(p, **eq** tcs = false), te))
7: **else** GenPrsc'($\mathcal{M}, p, te$)
8: **end if**
9: **else** GenPrsc'($\mathcal{M}, p, te$)
10: **end if**
11: **end function**

12: **function** GenPrsc'($\mathcal{M}, p, te$)
13: $t \leftarrow$ `metaReduce`$(\mathcal{M}||p, te)$ ▷*try to reduce the term*
14: **if** t = true **then return** p
15: **else**
16: **if** t = false **then**
17: **return** HandleFalseCase($\mathcal{M}, p, te, lms$) ▷*try to find a lemma*
18: **else**
19: $tcs \leftarrow$ ChooseATermCS($\mathcal{M}, p, t$) ▷*extract a sub-term for case splitting*
20: **return** append(
GenPrsc'($\mathcal{M}$, addEq(p, **eq** tcs = true), te),
GenPrsc'($\mathcal{M}$, addEq(p, **eq** tcs = false), te))
21: **end if**
22: **end if**
23: **end function**

Chapter 4

Security verification of Hybrid Post-Quantum TLS Handshake Protocol

Transport Layer Security (TLS) Protocol plays an important role in providing safe communication between two peers over insecure networks, which makes it one of the most extensively used cryptographic protocols. Unfortunately, once large-scale quantum computers become available, the traditional key exchange schemes in TLS will be vulnerable to quantum attacks, which necessitates the development of a quantum-resistant version of the protocol. Taking into account that potential threat, the Amazon Web Services team has proposed the Hybrid Post-Quantum TLS Protocol [36], a quantum-resistant version of the TLS 1.2 protocol. In this chapter, we construct a comprehensive symbolic model of the proposed protocol, specify it in CafeOBJ, and formally prove the claimed security properties with the employment of the tool IPSG.

4.1 Key Encapsulation Mechanism (KEM)

In 2017, The National Institute of Standards and Technology (NIST) started the Post-Quantum Cryptography Standardization Project [138], calling for proposals of public-key algorithms that are secure against both conventional and quantum computers. In the first round of this standardization project, there were 69 submissions. After two rounds of evaluation, the seven most promising candidates have been selected as the finalists for the first track and eight alternative candidates have been selected for the second track [139]. The submissions are classified into two types of public-key algorithms, namely public-key encryption (and key exchange) algorithms and digital signatures. According to the NIST standardization project, public-key encryption (and key exchange) algorithms are formulated as key encapsulation mechanisms. The following is the definition of a key encapsulation mechanism.

Definition 6. A key encapsulation mechanism (KEM) is a tuple of algorithms (`KeyGen`, `Encaps`,

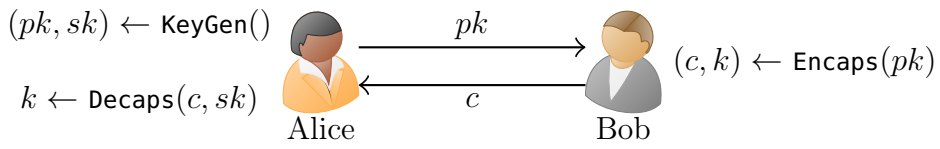


Figure 4.1: KEM visualization

Decaps) along with a finite key space $\mathcal{K}$:

- KeyGen() $\rightarrow (pk, sk)$: A probabilistic *key generation* algorithm that outputs a public key pk and a secret key sk.
- Encaps$(pk) \rightarrow (c, k)$: A probabilistic *encapsulation* algorithm that takes as input a public key pk, and outputs an encapsulation (or ciphertext) c and a shared secret $k \in \mathcal{K}$.
- Decaps$(c, sk) \rightarrow k$: A (usually deterministic) *decapsulation* algorithm that takes as inputs a ciphertext c and a secret key sk, and outputs a shared secret $k \in \mathcal{K}$.

Figure 4.1 visualizes how KEM is used by Alice and Bob to establish a shared secret k. There are different approaches to post-quantum public-key cryptographic algorithm construction, such as lattice-based, hash-based, and code-based. The latticed-based approach tends to be the most promising way to construct future post-quantum KEMs as the number of latticed-based submissions is the most in the NIST standardization competition. In this approach, a post-quantum algorithm can base its security on the difficulty of learning with errors, ring learning with errors, and learning with rounding, among others. For example, a KEM based on learning with errors generally computes the public key $pk = As + e$, where A is a public matrix, s is a vector serving as a secret key, and e is a small error vector acting as a noise. This public key is sent to Bob with the expectation that Eve is unable to derive As (and subsequently, s) even though A and pk are given. As can be interpreted as a vector in the lattice $\mathcal{L}\{a_1, \ldots, a_k\}$. Because e is small, pk is close to As. Thus, recovering As corresponds to finding the closest vector problem in lattices. No efficient algorithm on either classical computers or quantum computers is known to break the closest vector problem, so it is believed hard even for quantum computers. Figure 4.2 illustrates this problem in a 2-dimensional lattice generated by two vectors $\vec{a}_1 = (2, 3)$ and $\vec{a}_2 = (2, -1)$. Each blue point in the figure denotes a vector of the lattice, which would be a linear combination of $\vec{a}_1$ and $\vec{a}_2$, i.e., $m\vec{a}_1 + n\vec{a}_2$, with $m, n \in \mathbb{Z}$. Given an external vector denoted by the red point in the figure, the closest vector problem is to find the closest vector that belongs to the lattice. In this case, the closest vector should be $\vec{b}$. That external vector (in red) may be made by adding a noise (or an error) to $\vec{b}$. When the dimension becomes large, the problem to find the closest lattice vector becomes intractable to solve. There exist some other lattice problems that are believed hard even for quantum computers, such as the shortest

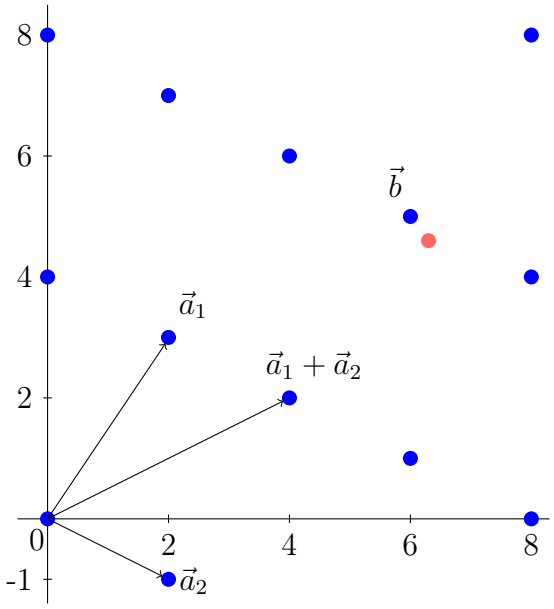


Figure 4.2: An illustration of the closest vector problem in 2-dimensional lattice $\mathcal{L}\{\vec{a}_1, \vec{a}_2\}$, where $\vec{a}_1 = (2, 3)$ and $\vec{a}_2 = (2, -1)$

vector problem and shortest independent vectors problem. Various latticed-based KEMs can be named, such as CRYSTALS-Kyber [35], Saber [48], FrodoKEM [5], NTRU [80], and NTRU Prime [22].

4.2 Hybrid Post-Quantum TLS Handshake Protocol

Various kinds of communications over the Internet every day are secured by the TLS protocol [122, 123], the successor of Secure Sockets Layer protocol (SSL) [121, 15]. This importance has led to numerous security analysis studies of the protocol in both the computational model, such as [60, 59, 70, 90, 82], and the symbolic model, such as [44, 25, 142, 45]. Although TLS is safe under classical computers today, traditional key exchange schemes currently used by the protocol, such as Diffie-Hellman (DH) and Elliptic Curve Diffie-Hellman (ECDH), are vulnerable to new generation attacks from quantum computers. As mentioned in Chapter 1, it is well known that the public-key primitives in use today will lose their security under the presence of sufficiently large quantum computers. The reason is that the computationally hard mathematical problems on which they are relying can be efficiently solved by Shor's algorithm [135] running on a sufficiently large quantum computer. In particular, the elliptic curve discrete logarithm problem will be no longer hard with large quantum computers, and one of the assumptions in our threat model in this case study verification - the intruder can break the ECDH key exchange scheme, makes sense. Because there is a substantial amount of advanced research in quantum computing and quantum information theory in recent years, quantum attacks are really a threat

that security research groups need to seriously pay attention to.

As an effort against the quantum attack threat, an Internet Engineering Task Force (IETF) Working Group established by the Amazon Web Services (AWS) has designed a quantum-resistant version of the TLS 1.2 protocol, namely the Hybrid Post-Quantum TLS Protocol [36], which is shortly called PQ TLS in this thesis. The hybrid key exchange scheme used in the proposal attempts to enable two concurrent key exchanges, one is a classical key exchange algorithm, which is fixed to ECDH, and the other is a quantum-safe key encapsulation mechanism (KEM), such as BIKE [10] and CRYSTALS-Kyber [35, 12]. In that way, a shared secret is expected to be secure as strong as ECDH against a classical attacker and as strong as the selected post-quantum KEM (PQ KEM) against a quantum attacker. In 2020, Amazon announced that the AWS Key Management Service (AWS KMS) supported the PQ TLS protocol on their cloud service, which means that their customers can freely enable the use of the protocol. Although TLS 1.3 was released in 2018, TLS 1.2 still keeps an important role because not every endpoint supports the latest version so far [137], especially with a big service like AWS, upgrading software for all endpoints immediately is almost impossible. This is one of the reasons why they proposed the quantum-resistant version for TLS 1.2 but not TLS 1.3.

We present a formal verification of PQ TLS in this chapter. Verification and analysis of security protocols, such as SDS, NSLPK, and TLS 1.2, require the assumption of the presence of malicious participants in addition to honest participants, which is an essential difference from verification of other systems/protocols. The Dolev-Yao generic intruder model [56] is used for this purpose as a de facto standard. To tackle security verification of post-quantum cryptographic protocols, such as PQ TLS, however, we need to extend the Dolev-Yao intruder model because the availability assumption of large-scale quantum computers gives the intruder some new capabilities. To come up with a reasonable and strong threat model for formal verification of post-quantum cryptographic protocols like PQ TLS is a creative task. Many quantum algorithms have been proposed. Among them, Shor's algorithm [135] and Grover's algorithm [76] are considered potential threats to public-key primitives and symmetric primitives, respectively, used today. We can work around Grover's algorithm by doubling the symmetric key length. Thus, Shor's algorithm is the only one for which we need to come up with new public-key cryptographic primitives to make cyberspace in the quantum era secure. To this end, NIST has been then launching the competition to standardize new public-key primitives, such as KEMs. We need to comprehend such post-quantum primitives as well as to come up with a way to model them for verification of higher-level protocols, such as PQ TLS in this chapter.

In the previous chapter, to demonstrate the efficiency and practicability of IPSG, we have presented a verification case study with the TLS 1.2 protocol (but not PQ TLS), but with several simplifications:

1. Client authentication has not been considered. In contrast, in this PQ TLS case study, we

cover both cases when client authentication is not requested and when it is requested.

2. With the threat model, the compromise of any secret keys has not been taken into account. In contrast, in this PQ TLS case study, we consider the compromises of symmetric handshake keys, ECDH secret keys, PQ KEM secret keys, and long-term private keys of honest principals.

3. Only the *session key secrecy* and *authentication* properties have been verified. With the PQ TLS case study presented in this chapter, we furthermore verify the *forward secrecy* property.

4. Some kinds of messages, such as ServerHelloDone and ChangeCipherSpec, have been excluded for ease of verification. With the PQ TLS case study presented in this chapter, no message is excluded, we try to model the protocol in CafeOBJ faithfully capturing what is specified in the IETF Draft [36].

Besides, the hybrid key exchange mechanism obviously makes the PQ TLS protocol different from the original one, namely different formats of messages exchanged between principals. In summary, all the above-mentioned differences make our verification presented in this chapter superior to the previous one.

We provide the webpage[1], from which readers can find the comprehensive clarification for the CafeOBJ formal specifications, the proof scores from which readers can execute them with CafeOBJ to re-check our proofs, the input requirement as well as the detailed guideline on how to generate those proof scores again, and other related materials used in this chapter.

Messages exchanged in a Full handshake

A full handshake of the protocol consists of several messages as depicted in Figure 4.3, where * indicates that the message is sent only in the case when client authentication is requested, and [ ] indicates that the message actually belongs to the change cipher spec protocol. The hybrid key exchange mechanism directly impacts on ClientHello, ServerHello, ServerKeyExchange, and ClientKeyExchange messages.

Suppose that client C wants to initiate a new connection with server S. C starts by sending a ClientHello message to S, which consists of the protocol version, a random number, an empty session ID, a cipher suite list, and a set of post-quantum KEM parameters (including the name of KEM and its parameters) supported by C. In response, S sends back to C a ServerHello message, which consists of the protocol version, a random number, a non-empty session ID, and a selected cipher suite. Next, S sends their digital certificate followed by a ServerKeyExchange

[1] https://github.com/duongtd23/PQTLS

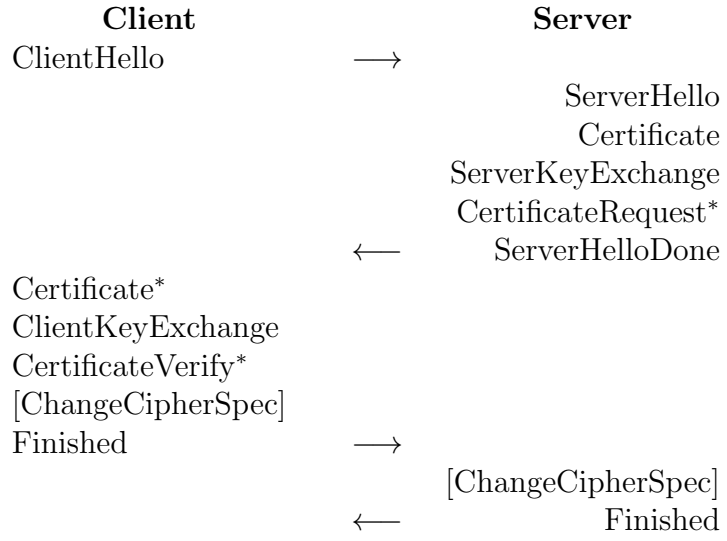
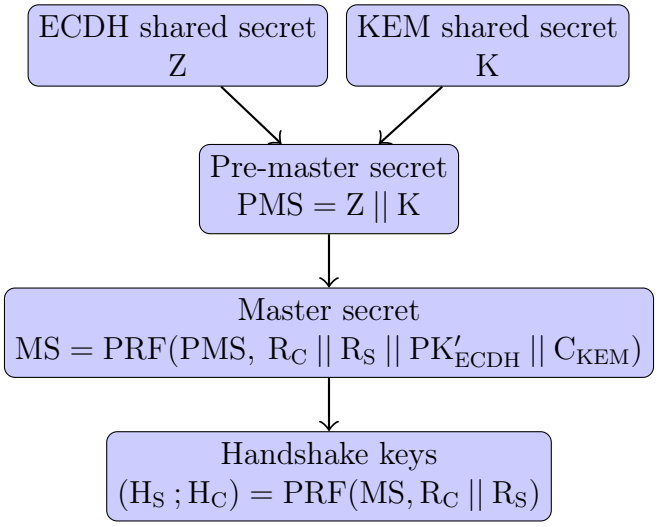


Figure 4.3: Messages exchanged in a full handshake of PQ TLS

message, which consists of S's ECDH & PQ KEM public keys and a signature over the two public keys together with the two random numbers in the ClientHello and ServerHello messages (Hello messages) signed under S's long-term private key. S can optionally send a CertificateRequest message if client authentication is requested. After that, S sends a ServerHelloDone message, informing C that the hello handshake phase on the server side is complete.

Upon receiving the ServerHelloDone message, C first sends their digital certificate if they have received a CertificateRequest message from S (indicating that client authentication is requested). In either case with or without client authentication, C always sends a ClientKeyExchange message, which consists of C's ECDH public key and the PQ KEM ciphertext value. If client authentication is requested, next, the client will send a CertificateVerify message, whose content is a digital signature over all messages exchanged so far signed by C's long-term private key. After that, C sends a ChangeCipherSpec message (which actually belongs to the change cipher spec protocol), indicating that subsequent messages will be secured by the symmetric handshake keys. The procedure to compute these keys for both C and S is depicted in Figure 4.4. They first compute the pre-master secret, which is the concatenation of the ECDH shared secret Z and the KEM shared secret K. The master secret is then computed by using the pseudorandom function taking as inputs the pre-master secret and a seed, which consists of the two random numbers (sent in the Hello messages), the ECDH public key, and the PQ KEM ciphertext sent by C in the ClientKeyExchange message. From the master secret, the handshake keys are derived by using the pseudorandom function with the two random numbers acting as a seed. A Finished message encrypted by the client handshake key is finally sent by C.

Upon receiving the Finished message from C, S validates it, and respectively sends their own ChangeCipherSpec and Finished messages.



where PRF denotes the pseudorandom function

Figure 4.4: Key calculation in PQ TLS

Client		**Server**
ClientHello	$\longrightarrow$	
		ServerHello
		[ChangeCipherSpec]
	$\longleftarrow$	Finished
[ChangeCipherSpec]		
Finished	$\longrightarrow$	

Figure 4.5: Messages exchanged in an abbreviated handshake of PQ TLS

Messages exchanged in an Abbreviated handshake

When client C and server S want to resume a previously established session, instead of exchanging the complete handshake messages, the two principals can perform an abbreviated handshake, with messages exchanged depicted in Figure 4.5. C first sends a ClientHello using the session ID of the session to be resumed. Upon receiving that message, S sends a ServerHello message with the same session ID value after checking that there exists a matching session ID. Then, S jumps to sending a ChangeCipherSpec message followed by a Finished message. Similarly, C sends their ChangeCipherSpec message followed by a Finished message.

4.3 Modeling the protocol

This section first describes the threat model used and then presents how to specify the protocol in CafeOBJ under that threat model. We focus on presenting the specification in the case when client authentication is not requested, while only briefly describing it in the case when client authentication is requested.

4.3.1 Threat model

When modeling a mutual exclusion protocol, such as Qlock presented in Section 2.4, it suffices to model the behavior of participants following the protocol algorithm. Modeling a cryptographic protocol like PQ TLS, however, requires us to additionally model the presence of malicious participants, who do not follow the protocol execution. In this case study, the combination and cooperation of malicious participants is modeled as a generic intruder based on the Dolev-Yao model [56]. The intruder can intercept and modify messages sent in the network, glean information from such messages, synthesize information to construct new messages, and impersonate some participants to send the messages to others.

Moreover, we also consider:

- the intruder can break the classical key exchange algorithm (ECDH) (this is because the intruder can utilize quantum computers, running Shor's algorithms [135]);
- an ECDH secret key can be leaked to the intruder;
- a PQ KEM secret key can be leaked to the intruder;
- a handshake key established between two principals can be leaked to the intruder;
- a long-term private key of an honest principal can be leaked to the intruder.

We assume some perfect cryptographic assumptions, in which the intruder is unable to: decrypt a ciphertext without the appropriate decryption key, sign some data without knowing the appropriate key, and reverse the hash function to compute the preimages of a hash. Note that the digital signature algorithms used in the protocol are resistant to quantum computers. There is a number of quantum-resistant digital signature algorithms have been proposed, such as CRYSTALS-Dilithium [61] and Rainbow [55].

4.3.2 Modeling hybrid key exchange and key calculation

Classical key exchange algorithms

We first model classical key exchange algorithms, or precisely, ECDH, because only ECDH is nominated for the classical key exchange algorithm in the protocol proposal [36]. Sorts `ClPriKeyEx`, `ClPubKeyEx`, and `ClassicKey` are introduced representing ECDH secret keys, public keys, and shared keys, respectively. We declare the following operators and equation:

```
-- the associated ECDH public key is derived from a secret key
op clPubKeyEx : ClPriKeyEx -> ClPubKeyEx {constr}
-- a shared key is computed from a public and a secret keys
op classicKey : ClPubKeyEx ClPriKeyEx -> ClassicKey
```

```
-- constructor of a shared key is a secret key pair
op _&_        : ClPriKeyEx ClPriKeyEx -> ClassicKey {constr comm}
eq classicKey(PK,K) = (priClKey(PK) & K) .
```

where `PK` and `K` are variables of the corresponding sorts. The first and second operators reflect that an ECDH public key is derived from a secret one and a principal calculates an ECDH shared key from a public/secret key pair, respectively. `comm` attribute says that the binary infix operator `_&_` is commutative. This is necessary to make the rewriting of the ECDH shared keys on the client side and server side result in the same value. `priClKey` is the projection operator taking as input an ECDH public key and returning as output its associated secret key.

Key encapsulation mechanisms

We choose to model KEMs based on their general definition, i.e., Definition 6. The CafeOBJ formal specification of the protocol does not take into account how KEMs like CRYSTALS-Kyber [35] and BIKE [10] are implemented, that is, we omit to specify their implementation components in detail, such as vectors and matrices. The three algorithms `KeyGen`, `Encaps`, and `Decaps` are regarded as three black boxes taking some inputs and returning the outputs. As mentioned previously, there are different approaches to post-quantum KEMs construction, such as CRYSTALS-Kyber is based on the learning with errors problem over module lattices, while BIKE relies on error correction codes in coding theory. Because their designs are totally different, there is no way to specify them consistently in a specification unless we use an abstract model that represents all. Using abstract versions of cryptographic primitives to model them like this and assumption of their security are commonly made in the symbolic analysis of cryptographic protocols. For example, to model hash functions, typically, it suffices to use just a function/-operator, which takes any data as input and returns the corresponding hash. The design and implementation aspects of the hash functions, such as block cipher, would be omitted in the formal specification. The hash function is assumed to be secure, namely, given a hash value, attackers are unable to derive its preimage.

To model KEMs in CafeOBJ, we introduce sorts `PqPriKey`, `PqPubKey`, `PqCipher`, and `PqKey` representing KEMs secret keys, public keys, encapsulations, and shared keys, respectively. Similar to modeling ECDH, an operator is declared reflecting the calculation of KEMs public keys from secret keys:

```
op pqPubKeyEn : PqPriKey -> PqPubKey {constr}
```

Actually, the operator represents the algorithm `KeyGen`. Indeed, because `KeyGen` is probabilistic, to model it as a deterministic procedure in CafeOBJ, we need to add an input argument serving as the random parameter. In this case, we set `PqPriKey` as such an input argument.

In the same manner, we should add one more extra argument of the sort `PqPriKey` into the arity of the operator modeling the algorithm `Encaps` in addition to the argument of the sort

PqPubKey because the algorithm is also probabilistic, but not deterministic. Given two secret keys k' and k'_2, then a shared key may be in the form of $(k'$ & $k'_2)$, where _&_ is the constructor of PqKey similar to the case of ECDH. However, in this case, we want to embed one more argument in the constructor of PqKey. Such an argument stores the time when the corresponding shared key is established, which is used to verify the *forward secrecy* property later on. For example, the verification needs to check whether the key concerned is established before the compromise of the long-term private key of the server, who established that key in a session with another client. For that time information, here we simply use a natural number to represent. The idea is basically as follows: initially, time is set to 1, and after some actions of honest principals, such as sending a ClientKeyExchange/ServerKeyExchange message, or compromising some long-term private key, it is incremented. Consequently, the constructor of PqKey and the operators modeling Encaps are defined as follows:

```
op _&_          : PqPriKey PqPriKey -> $PqKey   {constr}
op pqKey        : $PqKey   Nat      -> PqKey    {constr}
op encapsCipher : PqPubKey PqPriKey -> PqCipher {constr}
op encapsKey    : PqPubKey PqPriKey -> $PqKey
eq encapsKey(PK',K2') = (priPqKey(PK') & K2') .
```

where Nat is the sort of natural numbers, and PK' and K2' are variables of the corresponding sorts. Here, we introduce one more sort - $PqKey. priPqKey is the projection operator taking as input a PQ KEM public key and returning as output its secret counterpart. The algorithm Encaps is modeled by two separate operators encapsCipher and encapsKey returning the ciphertext and the shared key, respectively. We can choose to use only one operator, but then we need to define two new projection operators (e.g., getCiphertext and getKey).

The deterministic algorithm Decaps can be straightforwardly modeled as follows:

```
op decaps : PqCipher PqPriKey -> $PqKey
```

Finally, the algebraic property of KEMs is specified as follows:

```
eq (decaps(EN,K') = (K3' & K2')) =
 (K3' = K' and EN = encapsCipher(pqPubKeyEn(K'),K2')) .
```

The equation says that taking as inputs a ciphertext EN and a secret key K', Decaps can be successfully performed only when the ciphertext is obtained by the Encaps procedure taking as input the public key associated with the secret key K'.

Turning to model key calculation, we introduce four more CafeOBJ sorts including Key, Ms, Pms, and Seed representing handshake keys, master secrets, pre-master secrets, and seeds (used for pseudorandom function - PRF), respectively. Four constructor operators of Pms, Ms, and Key are declared as follows:

```
op _||_     : ClassicKey PqKey -> Pms {constr}
op prf-ms   : Pms Seed         -> Ms  {constr}
```

```
op prf-ckey : Ms  Seed         -> Key {constr}
op prf-skey : Ms  Seed         -> Key {constr}
```

where `prf-ckey` and `prf-skey` are used to calculate symmetric handshake keys on the client side and the server side, respectively. We define: (1) the projection operator `getMs` of `prf-ckey` and `prf-skey`, which returns the corresponding master secret of a given handshake key, (2) the projection operator `getPms` of `prf-ms`, which returns the corresponding pre-master secret of a given master secret, and (3) two projection operators `pmsClKey` and `pmsPqKey` of `_||_`, which return the corresponding ECDH shared key and PQ KEM shared key of a given pre-master secret, respectively.

4.3.3 Modeling messages exchanged

In the case when client authentication is not requested, we introduce sort `Msg` with 14 constructor operators to represent all kinds of messages exchanged in the protocol (covers both the full handshake and the abbreviated handshake modes). Among them, we show here three ones including `chM`, `skexM`, and `ckexM`, which denote ClientHello, ServerKeyExchange, and ClientKeyExchange messages in the full handshake mode, respectively. Their declarations are as follows:

```
op chM   : Prin Prin Prin Version Rand CipherSuites PqKemParams -> Msg {constr}
op skexM : Prin Prin Prin ClPubKeyEx PqPubKey Cipher Nat        -> Msg {constr}
op ckexM : Prin Prin Prin ClPubKeyEx PqPubKey Nat               -> Msg {constr}
```

where `Prin`, `Version`, `Rand`, `CipherSuites`, `PqKemParams`, and `Cipher`, are the sorts denoting principals, protocol versions, random numbers, cipher suite lists, PQ KEM cryptographic parameters, and ciphertexts (encrypted by some keys), respectively. Given three principals a, b, a_1, an ECDH public key pk_1, a PQ KEM public key pk_2, and a number t, a ClientKeyExchange message is in the form of `ckexM`$(a_1, a, b, pk_1, pk_2, t)$, where b is the recipient of the message and a is the seeming sender whom b believes that he/she is the principal who sent the message. Furthermore, the first argument a_1 is embedded into the message denoting the real creator of that message. In particular, when a_1 is the intruder, the intruder tries to impersonate a to send the message to b. Note that the first argument is used for modeling and verification purposes only, but it can neither be seen by the receiver nor be controlled by the intruder. Given a term denoting a message, projection operators `crt`, `src`, and `dst` return the first, second, and third arguments of the term, respectively. The argument of the sort `Nat` is embedded to the last of each `skexM` and `ckexM` to store the time when the corresponding message is sent. Recall that it is necessary, otherwise, we can neither specify nor verify the *forward secrecy* property later on.

4.3.4 Modeling protocol execution of honest principals

Five observers `nw`, `ur`, `ui`, `uclk`, and `upqk` are introduced. The first one observes the network, which is modeled as an associative-commutative collection (AC-collection) of messages exchanged. The four remaining ones observe the sets of (1) random numbers, (2) session IDs, (3) ECDH secret keys, and (4) PQ KEM secret keys, respectively, have been used. They are all declared as follows:

```
op nw   : Sys -> Network
op ur   : Sys -> URand
op ui   : Sys -> USid
op uclk : Sys -> ClPriKeyExS
op upqk : Sys -> PqPriKeyS
```

where `URand`, `USid`, `ClPriKeyExS`, and `PqPriKeyS` are the sorts of sets of the above-mentioned data types (1), (2), (3), and (4), respectively.

In the case when client authentication is not requested, we define 17 transitions to model protocol execution of honest principals. For the sake of simplicity, we show here the definition of `shello`, which is one of the simplest transitions. For the complete specification with detailed interpretation, readers are referred to the webpage mentioned in Section 4.2 (and in Chapter 1 also).

```
op shello : Sys Rand CipherSuite Sid
    Prin Prin Prin Version Rand CipherSuites PqKemParams -> Bool
ceq nw(shello(S,R2,CS,I,A2,A,B,V,R,CSs,KEMs)) =
   (shM(B,B,A,V,R2,CS,I) , nw(S))
  if c-shello(S,R2,CS,I,A2,A,B,V,R,CSs,KEMs) .
ceq ur(shello(S,R2,CS,I,A2,A,B,V,R,CSs,KEMs)) = (R2 ur(S))
  if c-shello(S,R2,CS,I,A2,A,B,V,R,CSs,KEMs) .
eq uclk(shello(S,R2,CS,I,A2,A,B,V,R,CSs,KEMs)) = uclk(S) .
eq upqk(shello(S,R2,CS,I,A2,A,B,V,R,CSs,KEMs)) = upqk(S) .
ceq ui(shello(S,R2,CS,I,A2,A,B,V,R,CSs,KEMs)) = (I ui(S))
  if c-shello(S,R2,CS,I,A2,A,B,V,R,CSs,KEMs) .
eq ss(shello(S,R2,CS,I,A2,A,B,V,R,CSs,KEMs),A9,B9,I9) = ss(S,A9,B9,I9) .
eq clkLeaked(shello(S,R2,CS,I,A2,A,B,V,R,CSs,KEMs)) = clkLeaked(S) .
eq pqkLeaked(shello(S,R2,CS,I,A2,A,B,V,R,CSs,KEMs)) = pqkLeaked(S) .
eq hskLeaked(shello(S,R2,CS,I,A2,A,B,V,R,CSs,KEMs)) = hskLeaked(S) .
eq ltkLeaked(shello(S,R2,CS,I,A2,A,B,V,R,CSs,KEMs)) = ltkLeaked(S) .
eq time(shello(S,R2,CS,I,A2,A,B,V,R,CSs,KEMs)) = time(S) .
ceq shello(S,R2,CS,I,A2,A,B,V,R,CSs,KEMs) = S
  if not c-shello(S,R2,CS,I,A2,A,B,V,R,CSs,KEMs) .
eq c-shello(S,R2,CS,I,A2,A,B,V,R,CSs,KEMs) =
  (not(R2 \in ur(S) or I \in ui(S)) and
   chM(A2,A,B,V,R,CSs,KEMs) \in nw(S) and
   CS \in CSs) .
```

Parameters of the term `shello(...)`, such as `A`, `B`, and `R`, are variables of the corresponding sorts. `c-shello` is the effective condition of the transition. The equations say that if principal `B` has received a ClientHello message apparently sent from `A`, random `R2` and session ID `I` have not been used before, and cipher suite `CS` is in the cipher suite list sent in the ClientHello message, then `B` replies to `A` a ServerHello message using `R2`, `CS`, and `I`. Together with that, `R2` and `I` are put into the set of used random numbers and session IDs, respectively. If one of the conditions above is not satisfied, nothing changes. Note that the ClientHello message is actually sent by `A2`, possibly different from `A`.

Checking the specification

Before going to model the intruder's capabilities, we check that the CafeOBJ specification we have specified so far allows two principals successfully complete a handshake and obtain the symmetric handshake keys. This must be fulfilled, otherwise, anything we do after is entirely meaningless. We have confirmed that by showing a state (through an open-close fragment) satisfying the above-mentioned requirement.

4.3.5 Modeling the intruder

Our threat model assumes the compromises of (1) symmetric handshake keys, (2) ECDH secret keys, (3) PQ KEM secret keys, and (4) long-term private keys of honest principals. In the CafeOBJ formal specification, four observers `hskLeaked`, `clkLeaked`, `pqkLeaked`, and `ltkLeaked` are introduced to store the sets of (1), (2), (3), and (4), respectively:

```
op hskLeaked : Sys -> KeyS
op clkLeaked : Sys -> ClPriKeyExS
op pqkLeaked : Sys -> PqPriKeyS
op ltkLeaked : Sys -> PriKeyTimeS
```

Given a long-term private key sk and a time t, each entry of `PriKeyTimeS` is in the form of `pkNPair`(sk, t), where t is the time when sk is compromised. In the case when client authentication is not requested, there are 6 transitions modeling the compromises of different kinds of secret keys. We show here the definition of one among two transitions modeling the compromise of PQ KEM secret keys.

```
op leakPKE2 : Sys Prin Prin Prin ClPubKeyEx PqPubKey Cipher Nat -> Sys {constr}
ceq pqkLeaked(leakPKE2(S,A2,B,A,PK,PK',CI,N))
    = (pqkLeaked(S) priPqKey(PK'))
  if c-leakPKE2(S,A2,B,A,PK,PK',CI,N) .
...
ceq leakPKE2(S,A2,B,A,PK,PK',CI,N) = S
  if not c-leakPKE2(S,A2,B,A,PK,PK',CI,N) .
```

```
eq c-leakPKE2(S,A2,B,A,PK,PK',CI,N)
  = skexM(A2,B,A,PK,PK',CI,N) \in nw(S) .
```

where ... indicates that some more equations stating that the remaining observers are not changed by `leakPKE2` are omitted. The equations say that when there exists a ServerKeyExchange message in the network where the PQ KEM public key `PK'` is sent, then the associated secret key is compromised and added to the set of PQ KEM secret keys compromised in the successor state.

The intruder tries to learn several kinds of information from the network, among them including handshake keys, pre-master secrets, ECDH shared keys, and PQ KEM shared keys, which are specified in CafeOBJ by operators `chsk`, `cpms`, `cclk`, and `cpqk`, respectively:

```
op chsk : Sys -> ColHsK
op cpms : Sys -> ColPms
op cclk : Sys -> ColClKey
op cpqk : Sys -> ColPqKey
op existPqPriKexM : PqKey Network -> Bool .
-- a handshake key is available to intruder if it is compromised or intruder learned its
    pre-master secret
eq HSK \in chsk(S) =
  HSK \in hskLeaked(S) or getPms(getMs(HSK)) \in cpms(S) .
-- a pre-master secret is available to intruder if intruder learned both its ECDH and PQ
    KEM shared secrets
eq PMS \in cpms(S) =
  pmsClKey(PMS) \in cclk(S) and pmsPqKey(PMS) \in cpqk(S) .
-- a PQ KEM shared secret is available to intruder if intruder learned one of its two
    secret keys or existPqPriKexM holds.
eq KP \in cpqk(S) =
  $pqKey1(KP) \in pqkLeaked(S) or
  $pqKey2(KP) \in pqkLeaked(S) or
  existPqPriKexM(KP, nw(S)) .
eq existPqPriKexM(KP, void) = false .
eq existPqPriKexM(KP, (M , NW)) =
 (((skexM?(M) and
    (priPqKey(getPqKey(M)) = $pqKey1(KP) or
     priPqKey(getPqKey(M)) = $pqKey2(KP))) or
   (ckexM?(M) and
    (priPqKey(getPqCipher(M)) = $pqKey1(KP) or
     priPqKey(getPqCipher(M)) = $pqKey2(KP)))
  ) and crt(M) = intruder and time(M) = time(KP))
 or existPqPriKexM(KP, NW) .
```

where `HSK`, `PMS`, `KP`, `NW`, and `M` are variables of the sorts `Key`, `Pms`, `PqKey`, `Network`, and `Msg`, respectively. `skeyM?(M)` and `ckeyM?(M)` check whether message `M` is the ServerKeyExchange message and the ClientKeyMessage message, respectively. Recall that `getPms`, `getMs`, `pmsClKey`, and `pmsPqkey`

are projection operators. Given a number n and two PQ KEM secret keys k' and $k2'$, the projection operators `$pqKey1`, `$pqKey2`, and `time` on `pqKey`$(k'$ `&` $k2', n)$ return k', $k2'$, and n, respectively. The first equation says that a handshake key is available to the intruder if it is compromised or the intruder has learned its pre-master secret. The second equation says that a pre-master secret is available to the intruder if the intruder has learned both its ECDH and PQ KEM shared secrets. The third equation says that a PQ KEM shared secret `KP` is available to the intruder if either the intruder has learned one of its two secret keys or `existPqPriKexM(KP,nw(S))` holds. The predicate `existPqPriKexM(KP,nw(S))` checks whether there exists a ServerKeyExchange or a ClientKeyExchange message (called KeyExchange message for short) in the network such that one of the two secret keys of `KP` equals the secret key associated with the PQ KEM public key sent in that message and `intruder` is the actual creator of that message (only in that case, the intruder owns the secret key). We omit the equations defining `cclk`.

In the case when client authentication is not requested, in addition to the 6 transitions modeling the secret compromises, the specification has 15 transitions modeling the intruder's capabilities. Among them, let us consider a transition in which the intruder tries to fake a ServerKeyExchange message. For the others, readers are asked to check the webpage mentioned in Section 4.2. The following are some equations among the set of equations defining that transition:

```
op fkSkeyex : Sys Prin Prin ClPriKeyEx PqPriKey Rand Rand -> Sys {constr}
ceq nw(fkSkeyex(S,B,A,K,K',R,R2)) =
  (skexM(intruder,B,A,clPubKeyEx(K),pqPubKeyEn(K'),
    encH(priKey(B), hParams(R,R2,
      clPubKeyEx(K),pqPubKeyEn(K'))),time(S)) , nw(S))
 if c-fkSkeyex(S,B,K,K') .
ceq time(fkSkeyex(S,B,A,K,K',R,R2)) = s(time(S))
 if c-fkSkeyex(S,B,K,K') .
ceq fkSkeyex(S,B,A,K,K',R,R2) = S
 if not c-fkSkeyex(S,B,K,K') .
eq c-fkSkeyex(S,B,K,K') = priKey(B) \in' ltkLeaked(S)
  and not(K \in uclk(S) or K' \in upqk(S)) .
```

where `priKey(B)` denotes the long-term private key of `B`, `hParams` denotes the hash function, and `encH` encrypts the hash by some key. `\in'` is a predicate checking whether a given long-term private key exists in the set of compromised keys (checking the existence of the key in `ltkLeaked(S)`). The equations say that if the long-term private key of principal `B` is compromised and the two secret keys `K` & `K'` have not been used before, then the intruder can construct a ServerKeyExchange message from the two secret keys, sign them with the compromised key, and impersonate `B` to send the message to `A`. Together with that, the current time is embedded at the end of the message and the time of the system is incremented. Note that there are actually some more equations defining `fkSkeyex`, but they are omitted.

4.3.6 Client authentication is requested

In the case when client authentication is requested, we need to introduce three more constructors of `Msg`. They are `certReqM`, `ccertM`, and `certVerM` which denote CertificateRequest, client Certificate, and CertificateVerify messages, respectively. Three more transitions are also added, specifying a client/server sends messages that belong to those three kinds of messages. Consequently, some effective conditions of some existing transitions need to be updated. For example, before sending the Finished message, a server needs to ensure that Certificate and CertificateVerify messages have been received from a client in addition to the ClientKeyExchange, ChangeCipherSpec, and Finished messages. For the intruder, several transitions are also added to model the capability of faking messages that belong to those three kinds of messages.

In summary, the complete formal specification in case client authentication is requested consists of 2895 lines of CafeOBJ code. Among them, 2314 lines are dedicated to specifying the protocol execution and 581 lines are dedicated to defining invariants (and lemmas), which are used for the formal verification. On the other hand, the specification without client authentication includes 2276 lines of code in total, where 1931 lines are dedicated to specifying the protocol execution and 345 lines are used for specifying invariants and lemmas. Note that we limit neither the number of honest principals participating in the protocol nor the number of sessions that the protocol can execute in both models. Roughly speaking, the CafeOBJ specifications allow any principal (a variable of sort `Prin`) to initialize a session with any other principal regardless of how many times and without any restriction by executing the `chello` transition, which formalizes sending a ClientHello message.

4.4 Security verification

This section first mentions the security properties of the protocol claimed in the IETF Draft. We then separately present security verification of the protocol in each case without and with client authentication. In the latter case, tighter properties are proved and more invariants/lemmas are introduced.

4.4.1 Security properties

The IETF Draft [36] claims that the security of the PQ TLS protocol is as strong as the original TLS 1.2. The TLS 1.2 standard document [123] states the following two desired security properties:

- A shared secret is securely negotiated. The shared secret cannot be compromised by an external party for any authenticated connection, even by an active attacker placed in the middle of the connection.

- A shared secret is reliably negotiated. The communication to establish the shared secret cannot be silently modified by an attacker without being detected by the two parties.

Note that the TLS 1.2 standard document [123] states that authentication is not mandatory but is usually required for at least one peer. On the other hand, in the IETF Draft of the PQ TLS protocol [36], it requires that server authentication is mandatory. Furthermore, the IETF Draft also claims its forward secrecy property:

- The establishment of shared secrets achieves forward secrecy provided that all ephemeral keys are unique.

We formally specify and verify three properties including (1) *session key secrecy*, (2) *forward secrecy*, and (3) *authentication*. The first property makes sure that nobody can learn a session key established between a client and a server except those two principals. The second property guarantees that even if a long-term private key of a client or a server is compromised, session keys established before the compromise are still secure. The third property ensures that upon completion of a handshake, if client C has communicated apparently with server S, then the server is indeed S. The *authentication* property is also called the *correspondence* property by Woo and Lam [149] and by Lowe [97].

4.4.2 Without client authentication

Security verification in the case when client authentication is not requested is presented first.

Session key secrecy property

Handshake keys must be securely negotiated between an honest client and an honest server. Nobody except for the two honest principals can know the shared handshake keys. The secrecy of handshake keys established in the full handshake mode is specified by the following invariant:

```
op ssKeySe : Sys Prin Prin ClassicKey
         PqPriKey PqPriKey Seed Seed Hash Nat -> Bool
eq ssKeySe(S,A,B,KC,K',K2',SD,SD2,H,N) =
 (not(A = intruder or B = intruder or A = B) and
  not(K' \in pqkLeaked(S) or K2' \in pqkLeaked(S)) and
  not(prf-ckey(prf-ms(KC || pqKey(K' & K2',N),SD),SD2) \in hskLeaked(S)) and
  not(priKey(B) \in' ltkLeaked(S)) and
  cfM(A,A,B, encFin(
    prf-ckey(prf-ms(KC || pqKey(K' & K2',N),SD),SD2),
    prf-cfin(prf-ms(KC || pqKey(K' & K2',N),SD),H)))
   \in nw(S))
 implies
  not(prf-ckey(prf-ms(KC || pqKey(K' & K2',N),SD),SD2) \in chsk(S)) .
```

where cfM is a constructor of the sort Msg, representing Finished messages sent from the client side in the full handshake mode. prf-cfin computes the value for the verify_data field in those Finished messages from the master secret and the hash of the handshake messages (H). encFin denotes the encryption of the verify_data by the client symmetric handshake key. ssKeySe says that when honest client A has sent to honest server B a Finished message indicating that the key negotiation has been completed in which the established handshake key is not trivially compromised, its two PQ KEM secret keys and the long-term private key of B are not compromised, then the intruder cannot learn the handshake key.

ssKeySe is simply proved without induction by using the following lemma:

```
op pqKeySe : Sys Prin Prin ClassicKey
         PqPriKey PqPriKey Seed Seed Hash Nat -> Bool
eq pqKeySe(S,A,B,KC,K',K2',SD,SD2,H,N) =
 (not(A = intruder or B = intruder or A = B) and
  not(K' \in pqkLeaked(S) or K2' \in pqkLeaked(S)) and
  (not(priKey(B) \in' ltkLeaked(S)) or
   not(N > timeLeak(priKey(B), ltkLeaked(S)))) and
  cfM(A,A,B,encFin(
    prf-ckey(prf-ms(KC || pqKey(K' & K2',N),SD),SD2),
    prf-cfin(prf-ms(KC || pqKey(K' & K2',N),SD),H)))
   \in nw(S))
 implies
  not existPqPriKexM(pqKey(K' & K2',N), nw(S)) .
```

pqKeySe says that when honest client A has sent to honest server B a Finished message indicating that the key negotiation has been completed in which the two PQ KEM secret keys are not compromised and either the long-term private key of B is not compromised or it is compromised but the time when the compromise happens (denoted by timeLeak(priKey(B), ltkLeaked(S))) is after the handshake key establishment (denoted by N), then there does not exist a KeyExchange message created by the intruder in the network such that the secret key associated with the PQ KEM public key sent in that message equals one of the two PQ KEM secret keys. The proof of ssKeySe using pqKeySe is simply as follows:

```
open INV .
op s : -> Sys .   op n : -> Nat .   op h : -> Hash .
ops k' k2' : -> PqPriKey .          ops a b : -> Prin .
op kc : -> ClassicKey .             ops sd sd2 : -> Seed .
red pqKeySe(s,a,b,kc,k',k2',sd,sd2,h,n)
  implies ssKeySe(s,a,b,kc,k',k2',sd,sd2,h,n) .
close
```

CafeOBJ returns true for that **red** command, meaning that ssKeySe is proved. The proof is accomplished with respect to an unbounded number of sessions. Readers may argue that a and b are fresh constants, and thus the proof by the **red** command above is limited to only two

particular principals `a` and `b`. But this argument is a mistake. `a` and `b` are defined as fresh constants of the sort `Prin` without any constraint, denoting two arbitrary principals among all principals participating in the protocol (which is unbounded). The protocol execution does not limit to only `a` and `b`, but it also includes the involvement of other principals as well. Recall that the previous section has clarified that the CafeOBJ specifications allow an unbounded number of sessions and principals. From what has been clarified, we can affirm that `ssKeySe` is proved under an unbounded number of sessions and this is also true with the remaining invariant proofs in this verification case study.

We need to prove `pqKeySe` is also invariant, for which now we use simultaneous induction with some other lemmas. As mentioned before, we employ IPSG to infer proof scores of them, for which we prepare a simple script. IPSG successfully generates the proof score of `pqKeySe`, consisting of 369 open-close fragments in total after about 20 seconds on a MacBook Pro carrying 32 GB of memory with a processor i7 2.3 GHz (see Section 4.4.5). Note that this is the time taken for generating the executable proof score of `pqKeySe`, but not the time for executing the proof. Once the proof is generated, it is unnecessary to infer the proof again; just executing the generated proof with CafeOBJ is all we need to do for the verification.

Forward secrecy property

We consider the case that the long-term private key of an honest server is compromised. Even in that case, session keys established before the compromise are still secure. This is called the *forward secrecy* property, which is specified by the following invariant:

```
op forwardSe : Sys Prin Prin ClassicKey
          PqPriKey PqPriKey Seed Seed Hash Nat -> Bool
eq forwardSe(S,A,B,KC,K',K2',SD,SD2,H,N) =
 (not(A = intruder or B = intruder or A = B) and
  not(K' \in pqkLeaked(S) or K2' \in pqkLeaked(S)) and
  not(prf-ckey(prf-ms(KC || pqKey(K' & K2',N),SD),SD2) \in hskLeaked(S)) and
  priKey(B) \in' ltkLeaked(S) and
  not(N > timeLeak(priKey(B), ltkLeaked(S))) and
  cfM(A,A,B,encFin(
    prf-ckey(prf-ms(KC || pqKey(K' & K2',N),SD),SD2),
    prf-cfin(prf-ms(KC || pqKey(K' & K2',N),SD),H)))
   \in nw(S))
 implies
  not(prf-ckey(prf-ms(KC || pqKey(K' & K2',N),SD),SD2) \in chsk(S)) .
```

`forwardSe` says that if honest client `A` has sent to honest server `B` a Finished message indicating that the key negotiation has been completed in which the established handshake key is not trivially compromised, its two PQ KEM secret keys are not compromised, the long-term private key of `B` is compromised but the compromise happens after the handshake key establishment, then

the intruder cannot learn the handshake key. `forwardSe` guarantees that even if the long-term private key of an honest server is compromised, ciphertexts encrypted by using the handshake keys established between the server and some honest client cannot be decrypted by the intruder if the compromise happens after the establishment of the handshake keys. `forwardSe` is simply proved without induction by using `pqKeySe`.

Authentication property

The handshake process completes once the client and the server complete sending their Finished messages. After receiving a Finished message from a server in either a full handshake or an abbreviated handshake, from a client's point of view, the *authentication* property is stated as follows. If honest client `A` has been communicating apparently with server `B`, where `A` receives a valid Finished message seemingly sent from `B`, then the server that `A` has been communicating with is indeed `B`. This *authentication* property is verified through two invariants, one for the full handshake mode, and the other for the abbreviated handshake mode. The former is as follows:

```
op authent : Sys Prin Prin Prin ClassicKey PqPriKey
                  PqPriKey Seed Seed Hash Nat -> Bool
eq authent(S,A,B,B1,KC,K',K2',SD,SD2,H,N) =
 (not(A = intruder or B = intruder or A = B) and
  not(K' \in pqkLeaked(S) or K2' \in pqkLeaked(S)) and
  (not(priKey(B) \in' ltkLeaked(S)) or
   not(N > timeLeak(priKey(B), ltkLeaked(S)))) and
  cfM(A,A,B,encFin(
    prf-ckey(prf-ms(KC || pqKey(K' & K2',N),SD),SD2),
    prf-cfin(prf-ms(KC || pqKey(K' & K2',N),SD),H)))
   \in nw(S) and
  sfM(B1,B,A,encFin(
    prf-skey(prf-ms(KC || pqKey(K' & K2',N),SD),SD2),
    prf-sfin(prf-ms(KC || pqKey(K' & K2',N),SD),H)))
   \in nw(S))
 implies
  sfM(B,B,A,encFin(
    prf-skey(prf-ms(KC || pqKey(K' & K2',N),SD),SD2),
    prf-sfin(prf-ms(KC || pqKey(K' & K2',N),SD),H)))
   \in nw(S) .
```

where `sfM` is another constructor of the sort `Msg`, representing Finished messages sent from the server side in the full handshake mode. `prf-sfin` computes the value for the verify_data field in those Finished messages from the master secret and the hash of the handshake messages. Precisely, `authent` says that when a full handshake between two honest principals `A` and `B` has been completed, where `A` sent a Finished message to `B` and received back another valid Finished message apparently sent from `B` such that the two PQ KEM secret keys of the pre-master

secret are not compromised, and either the long-term private key of `B` is not compromised or the compromise happens after the handshake key establishment, then the Finished message A received really created by `B`. Note that because we allow the intruder to replay a Finished message in the network, it is impossible to use `B1 = B` in the conclusion of `authent`. In other words, it is possible that `B1` is `intruder`, making `B1` different from `B` (there are two different `sfM` messages, one is created by `B` and the other is created by `intruder`). `authent` is proved by using IPSG to automatically generate proof scores with the use of two lemmas, one of which is `pqKeySe`.

A similar invariant is defined to specify the *authentication* property in the abbreviated handshake mode, and its proof is also fully generated by IPSG with the use of `pqKeySe` and another new lemma. The invariant is defined as follows:

```
op authent2 : Sys Prin Prin Prin ClassicKey PqPriKey
        PqPriKey Seed Seed Seed Hash Hash Nat -> Bool
eq authent2(S,A,B,B1,KC,K',K2',SD,SD2,SD3,H,H2,N) =
 (not(A = intruder or B = intruder or A = B) and
  not(K' \in pqkLeaked(S) or K2' \in pqkLeaked(S)) and
  (not(priKey(B) \in' ltkLeaked(S)) or
   not(N > timeLeak(priKey(B), ltkLeaked(S)))) and
  cfM(A,A,B,encFin(
    prf-ckey(prf-ms(KC || pqKey(K' & K2',N),SD),SD2),
    prf-cfin(prf-ms(KC || pqKey(K' & K2',N),SD),H)))
   \in nw(S) and
  sf2M(B1,B,A,encFin(
    prf-skey(prf-ms(KC || pqKey(K' & K2',N),SD),SD3),
    prf-sfin2(prf-ms(KC || pqKey(K' & K2',N),SD),H2)))
   \in nw(S))
 implies
  sf2M(B,B,A,encFin(
    prf-skey(prf-ms(KC || pqKey(K' & K2',N),SD),SD3),
    prf-sfin2(prf-ms(KC || pqKey(K' & K2',N),SD),H2)))
   \in nw(S) .
```

where `sf2M` is another constructor of the sort `Msg`, representing Finished messages sent from the server side in the abbreviated handshake mode. `prf-sfin2` computes the value for the verify_data field in those Finished messages from the master secret and the hash of the handshake messages.

In summary, when client authentication is not requested, the verification consists of 31 invariants, among them, four invariants specify the three properties, whereas the remaining 27 ones serve as lemmas for the proofs. Six invariants are simply proved by using some others without induction, while the proofs of the remaining ones are completely generated by IPSG (all of them are available on the webpage mentioned in Section 4.2).

4.4.3 Invalid invariant candidates and counterexamples

As mentioned before, our threat model supposes the security of ECDH can be broken, i.e., the intruder can learn the ECDH shared secret from the two public keys exchanged between a client and a server. To illuminate that we have correctly modeled the intruder's capability in this assumption, let us use the following predicate:

```
op pd1 : Sys Prin Prin ClPriKeyEx ClPriKeyEx PqKey Seed Seed Hash -> Bool
eq pd1(S,A,B,K,K2,KP,SD,SD2,H) =
 (not(A = intruder or B = intruder or A = B) and
  not(K \in clkLeaked(S) or K2 \in clkLeaked(S)) and
  cfM(A,A,B,encFin(prf-ckey(prf-ms(K & K2 || KP, SD),
    SD2), prf-cfin(prf-ms(K & K2 || KP, SD), H)))
   \in nw(S))
 implies not((K & K2) \in cclk(S)) .
```

`pd1` says that when honest client `A` has sent to honest server `B` a Finished message indicating that the key negotiation has been completed in which the two ECDH secret keys of the shared secret are not compromised, then the intruder cannot learn ECDH shared secret. Because of the above-mentioned assumption, `pd1` should be violated (not invariant). The intruder can grasp the two ECDH public keys carried in the ServerKeyExchange and ClientKeyExchange messages sent in the same session with that Finished message, from that, it is possible for the intruder to derive the shared secret. Indeed, a counterexample is found for `pd1`. The counterexample and action sequence leading to it are described in detail on the webpage mentioned at the beginning of this chapter. Through this demonstration, it can be confirmed that at least we have correctly modeled the protocol execution and the intruder's capability of breaking ECDH.

Let us now turn to a more complicated case. Recall that `ssKeySe` confirms the *session key secrecy* property from a client's point of view. Following that, it is natural to think about the counterpart of `ssKeySe`, which expresses the property from a server's point of view. Let us consider the following predicate:

```
op pd2 : Sys Prin Prin ClassicKey PqPriKey PqPriKey Seed Seed Hash Nat -> Bool
eq pd2(S,A,B,KC,K',K2',SD,SD2,H,N) =
 (not(A = intruder or B = intruder or A = B) and
  not(K' \in pqkLeaked(S) or K2' \in pqkLeaked(S)) and
  not(prf-skey(prf-ms(KC || pqKey(K' & K2',N),SD),SD2) \in hskLeaked(S)) and
  not(priKey(A) \in' ltkLeaked(S)) and
  sfM(B,B,A,encFin(prf-skey(prf-ms(KC ||
    pqKey(K' & K2',N),SD),SD2),
    prf-sfin(prf-ms(KC || pqKey(K' & K2',N),SD),H)))
   \in nw(S))
 implies
  not(prf-skey(prf-ms(KC || pqKey(K' & K2',N),SD),SD2) \in chsk(S)) .
```

What pd2 specifies is the counterpart of ssKeySe from a server's point of view, so we do not explicitly clarify it. Actually, this predicate is not invariant, a counterexample violating it is found. The counterexample and action sequence leading to it are described in detail on the webpage mentioned at the beginning of this chapter. Basically, the reason is that the intruder can impersonate A to communicate with B. Consequently, the handshake key that B believes they have established with A is actually owned by the intruder. Together with the previous demonstration, the predicate and the counterexample of this case can be used for the purpose of checking the specification, through that, roughly speaking, we can confirm the correctness of the protocol execution specification as well as the intruder's capability specification.

4.4.4 With client authentication

Session key secrecy property: When client authentication is requested, ssKeySe remains to be invariant specifying the *session key secrecy* property from a client's point of view. Moreover, a new invariant, namely ssKeySeAu, is introduced to specify the property from a server's point of view:

```
op ssKeySeAu : Sys Prin Prin ClassicKey PqPriKey PqPriKey
                            Seed Seed Hash Nat -> Bool
eq ssKeySeAu(S,A,B,KC,K',K2',SD,SD2,H,N) =
 (not(A = intruder or B = intruder or A = B) and
  not(K' \in pqkLeaked(S) or K2' \in pqkLeaked(S)) and
  not(prf-skey(prf-ms(KC || pqKey(K' & K2',N),SD),SD2) \in hskLeaked(S)) and
  not(priKey(A) \in' ltkLeaked(S)) and
  sfM(B,B,A,encFin(
    prf-skey(prf-ms(KC || pqKey(K' & K2',N),SD),SD2),
    prf-sfin(prf-ms(KC || pqKey(K' & K2',N),SD),H)))
   \in nw(S))
 implies
  not(prf-skey(prf-ms(KC || pqKey(K' & K2',N),SD),SD2) \in chsk(S)) .
```

ssKeySeAu is simply proved without induction by using another lemma, which is the counterpart of pqKeySe. Like pqKeySe, the proof score of that lemma is also generated by IPSG.

Similar to the *session key secrecy* property, tighter versions of the *forward secrecy* property and the *authentication* property are formally verified in the case when client authentication is requested. Concretely, one more invariant, namely forwardSeAu, is defined with respect to the CafeOBJ formal specification with client authentication to specify the *forward secrecy* property, whereas two more invariants, namely authentAu and authentAu2, are defined specifying the *authentication* property corresponding to the full handshake and abbreviated handshake modes. In summary, in the case when client authentication is requested, we introduce four more invariants specifying the three properties and 15 additional lemmas to complete the formal verification.

Table 4.1: Time taken by IPSG to generate proof scores in case of non-client authentication

Invariant	Time (s)	No. open-close fragments	Invariant	Time (s)	No. open-close fragments
pqKeySe	17.2	329	inv13	9.4	190
authent	35.9	691	inv16	8.6	184
authent2	36.1	691	inv17	50.3	772
inv3	12	279	inv18	31	652
inv4	11.4	233	inv19	45.1	648
inv4'	10.5	233	inv20	27.3	553
inv4''	8.2	192	inv20'	25.8	541
inv5	7.9	187	inv21	8.3	184
inv6	8.6	173	inv24	47.3	772
inv9	16.1	329	inv25	24.1	569
inv10	29.4	323	inv26	52.1	565
inv11	7.3	171	inv27	15.6	336
inv12	9.7	188			

There are six other invariants that are proved without induction but simply by using others as lemmas.

Among those new 19 invariants, six invariants are simply proved by using some others without induction, while IPSG automatically generates the proof scores for the remaining 13 invariants.

4.4.5 IPSG experimental results

Table 4.1 reports the time taken by IPSG to generate the proof scores of the 25 invariants in case client authentication is not requested. The third column shows the number of open-close fragments in the generated proof score of each invariant. Recall that the six other invariants (i.e., `ssKeySe`, `forwardSe`, `inv7`, `inv8`, `inv15`, and `inv23`), in this case, are proved without induction but simply by using some other invariants as lemmas. The experiments have been conducted on a MacBook Pro carrying 32 GB of memory with a processor i7 2.3 GHz. Table 4.2 shows the time taken in case client authentication is requested. The table omits the times to produce the proof scores of the 25 old invariants in case client authentication is not requested (those in Table 4.1). Note that what is depicted in the two tables is the time taken for generating the proofs with IPSG, but not the time for executing the proofs (the verification is confirmed by running the generated proof scores with CafeOBJ).

4.5 Limitations

To model the intruder's capability of forging messages, we have defined some specific transitions. Concretely, for each kind of message, there are one or several transitions dedicated to specifying how the intruder impersonates a principal to send an instance of that message to another.

Table 4.2: Time taken by IPSG to generate proof scores in case of client authentication

Invariant	Time (s)	No. open-close fragments	Invariant	Time (s)	No. open-close fragments
pqKeySeAu	23.7	367	inv39	51.9	800
authentAu	48.1	733	inv41	52	800
authentAu2	62	733	inv43	55.5	691
inv33	20.7	327	inv44	35.8	591
inv36	14.6	239	inv44'	34.7	579
inv37	15.7	234	inv45	61.8	603
inv38	17.2	236			

There are six other invariants that are proved without induction but simply by using others as lemmas.

Even though the specification defines 15 transitions, allowing the intruder to forge every kind of message, covering most of the significant cases, it is hard to verify that the intruder is given the full capability of forging an arbitrary message synthesized from the information that has been learned. In other words, it can be said that the transitions specifying the intruder's capabilities of forging messages are dedicated to this PQ TLS protocol only. It is better to use some general transitions, for example, if the intruder knows two pieces of information A and B, then they can combine them to obtain the composition information in the form of $A \,||\, B$, from which they can impersonate a principal to send the faking message $A \,||\, B$ to some other. In this way, efforts can be saved when specifying different cryptographic protocols and it is easy to verify that the intruder is given the full capability of forging an arbitrary message synthesized from the information that they have learned. Besides, it is unnatural to embed time information into PQ KEM shared keys in order to verify the *forward secrecy* property. The above-mentioned limitations will be addressed in the case study presented in the next chapter.

Regarding the threat model in the quantum era used in this case study, essentially, the novelty of the intruder's capabilities is only the assumption of breaking classical key exchange schemes such as ECDH. The remaining considerations are known as non-novel things from a traditional cryptographic protocol analysis point of view. We expect some more non-trivial capabilities for a quantum attacker will be included in our future work.

When the input is an invalid invariant property, a counterexample violating the property cannot be automatically produced by IPSG as other well-known symbolic cryptographic analyzers, such as Tamarin [101] and Maude-NPA [64], can do. Manual efforts have been spent to find the counterexamples discussed in Section 4.4.3. This restriction is a limitation of interactive theorem proving in general compared to the model-checking approach. Despite that, we must emphasize that doing theorem proving did help human experts to find counterexamples of non-trivial security properties [21, 120, 113]. This is extra helpful in the case when the security protocol under verification is complicated so that model checking cannot deal with it due to the

state space explosion problem.

4.6 Summary

In this chapter, we have symbolically modeled the Hybrid Post-Quantum TLS Protocol and specified it in CafeOBJ. Our model covers both the full handshake and abbreviated handshake modes, and both cases when client authentication is requested and when it is not. We have formally verified that the protocol enjoys the three desired security properties including *session key secrecy*, *forward secrecy*, and *authentication* with respect to an unbounded number of protocol participants and session executions. The first property makes sure that the negotiation of session keys is secure (i.e., nobody can glean the shared key except for the client and the server who have established those keys). The second property ensures that the hybrid key exchange mechanism achieves forward secrecy (i.e., even if a long-term private key of a client or a server is compromised, sessions completed before the compromise remain secure). The third property guarantees that upon completion of the handshake, if a client C believes that he/she is communicating with a server S, then the server is indeed S. The three properties have been specified by equations in CafeOBJ, and then their formal proofs, i.e., proof scores, have been generated by using the tool IPSG. These proof scores are executable, and the formal verification is done by executing the proof scores with CafeOBJ.

Comprehending the protocol is the task that takes the most time to conduct the formal analysis. The protocol itself is complicated, with the combination of various handshake options, such as full or abbreviated handshake and with or without client authentication. The most difficult task would be to comprehend the post-quantum cryptographic primitives used in the protocol, such as CRYSTALS-Kyber KEM. That is a challenge in the verification/analysis of post-quantum cryptographic protocols compared to classical ones. After understanding such primitives, another challenge is to find a way to model them for formal verification, such as by abstracting KEMs as we did. To come up with a reasonable and strong threat model is also a creative task. Attackers must have quantum-based power, such as breaking classical key exchange algorithms, and must have powerful capabilities, such as fully controlling the network. To find out how to specify in CafeOBJ the attacker's capability of fully controlling the network is also a challenging task.

Through the case study, we once again demonstrate that IPSG is efficient and applicable to even complicated protocols like PQ TLS. The complicate of the case study can be partially seen from the size of the specifications, consisting of more than 2000 lines of CafeOBJ code, as well as the total number of invariants used, i.e., more than 30 invariants. Although in the previous chapter, a verification case study has been conducted with the TLS 1.2 protocol, there are several simplifications in that verification, such as client authentication has not been considered,

no consideration about the compromise of any secret keys, some kinds of messages have been excluded for ease of verification, and only the *session key secrecy* and *authentication* properties have been verified. That makes the formal specification and verification of TLS 1.2 much less complicated than these ones of PQ TLS presented in this chapter. Concretely, in this PQ TLS verification case study, we have covered both cases when client authentication is not requested and when it is requested, have considered the compromises of all secret keys, have not excluded any kind of messages, and have additionally verified the *forward secrecy* property.

Chapter 5

Formal analysis of Hybrid Post-Quantum SSH Transport Layer Protocol

This chapter presents a formal analysis case study with the Hybrid Post-Quantum SSH Transport Layer Protocol [84], a quantum-resistant version of the SSH Transport Layer Protocol [96]. The Hybrid Post-Quantum SSH Transport Layer Protocol is shortly called PQ SSH in this thesis. Similar to PQ TLS, the hybrid key exchange scheme used in PQ SSH enables two concurrent key exchanges, one is a classical key exchange algorithm and the other is a quantum-resistant KEM. We take into account four desired security properties including (1) *session key secrecy*, (2) *forward secrecy*, (3) *session identifier uniqueness*, and (4) *authentication* properties, where (1), (2), and (3) are formally verified. The formal verifications of (1), (2), and (3) share many commons with what has been used in the PQ TLS verification case study. However, there are some novel distinctions in this PQ SSH case study as follows:

- For the threat model, we specify the intruder's capabilities in a more general way, addressing the limitation mentioned in Section 4.5 of the previous chapter. The formal specification guarantees that the intruder is given the full capabilities of learning exchanged information, manipulating it, and forging messages. This part will be shown in Section 5.2.4.
- We find a counterexample of (4), meaning that the protocol does not enjoy the *authentication* property. We propose to slightly revise the protocol by adding the identifiers of the client and the server into the exchange hash. We revise the CafeOBJ formal specification accordingly so that we can formally verify that the improved protocol enjoys the *authentication* property as well as (1), (2), and (3). This will be presented in more detail in Sections 5.3.4 and 5.3.5.

We provide a webpage[1], from which readers can find the CafeOBJ formal specification of the PQ SSH protocol and the improved version, the counterexample of the *authentication* property,

[1] https://github.com/duongtd23/PQSSH

the proofs from which readers can execute them to verify the four properties, and other related materials used in this chapter.

5.1 Hybrid Post-Quantum SSH Transport Layer Protocol

The Secure Shell protocol (SSH) [95] is a cryptographic protocol that gives users a secure way to access a computer over an unsecured network. The most well-known application of SSH is to allow users, particularly system administrators, to securely remote login to a server and execute commands through the command line environment. An essential difference between SSH and TLS is that SSH operates at the Application Layer, the 7th layer of the OSI model, while TLS operates at the Transport Layer through the Application Layer (the 4th-7th layers) of the OSI model. The protocol consists of the following three sub-protocols:

- Transport Layer protocol [96]: executes key negotiation to establish symmetric keys, which are used by the Connection protocol to securely exchange information between two participants. This protocol provides server authentication to the client.
- User Authentication protocol [93]: provides client authentication to the server.
- Connection protocol [94]: provides channels to securely exchange information between two participants by using the negotiated keys to encrypt/decrypt data.

Due to the threat of quantum attacks, an IETF Working Group has proposed a post-quantum version of the SSH Transport Layer protocol (PQ SSH) [84]. The proposed protocol is being standardized and its latest version is 01 by June 2023. Similar to PQ TLS, the proposed protocol also bases its security on the post-quantum hybrid key exchange method, which uses a classical key exchange algorithm and a quantum-resistant Key Encapsulation Mechanism (KEM) in parallel. In the IETF Draft [84], the classical key exchange algorithm and the quantum-resistant KEM are fixed to ECDH and CRYSTALS-Kyber [35, 12], respectively. In this chapter, we present a security analysis of PQ SSH version 01 [84].

The messages exchanged in the PQ SSH protocol are depicted in Figure 5.1. Each server host B owns a public host key (LK_B) and a private host key (LSK_B), where the public host key is known by all clients. To initialize a new connection between client A and server B, a pair of VERSION_EX messages is sent by them, exchanging the protocol versions on each side. The message can be called the version exchange message. Then, they exchange a pair of KEX_ALGR messages, indicating their supported algorithms (cryptographic primitives) sorted in order of preference. The message can be called the key exchange algorithms message. After that A generates: (1) an ECDH ephemeral key pair, i.e., secret key and its associated public key ($\mathrm{ECDH}_{\mathrm{PK}_A}$), and (2) a KEM public key ($\mathrm{KEM}_{\mathrm{PK}_A}$), i.e., the output of the algorithm `KeyGen`.

version exchange	VERSION_EX	$A \to B:$ Version_A
	VERSION_EX	$B \to A:$ Version_B
key exchange algorithms	KEX_ALGR	$A \to B:$ Suites_A
	KEX_ALGR	$B \to A:$ Suites_B
key exchange initiation	KEX_HBR_INIT	$A \to B:$ $\text{ECDH}_{\text{PK}_A}, \text{KEM}_{\text{PK}_A}$
key exchange reply	KEX_HBR_REPLY	$B \to A:$ $\text{LK}_B, \text{ECDH}_{\text{PK}_B}, \text{KEM}_{\text{C}_B}, \text{SIGN}$

Figure 5.1: Messages exchanged in the PQ SSH protocol

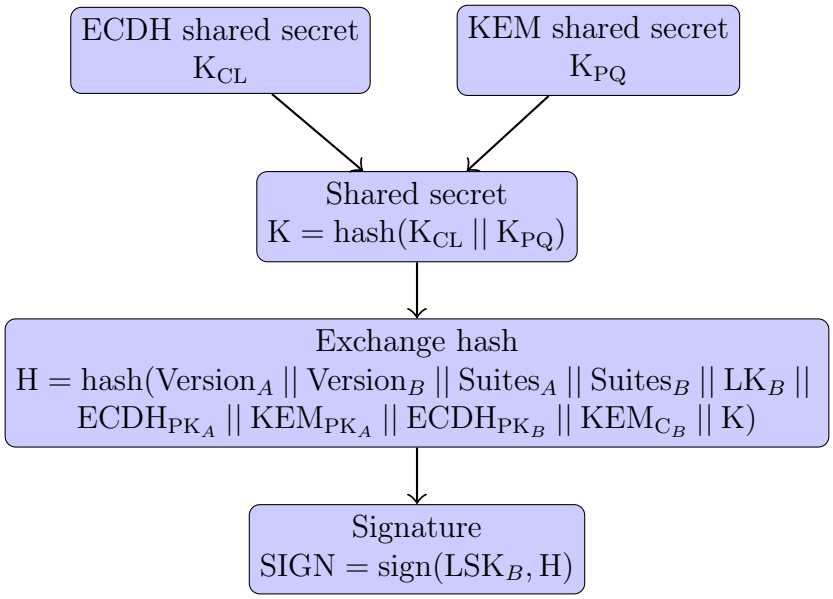


Figure 5.2: Exchange hash and signature calculation

A then sends the two public keys to B through a KEX_HBR_INIT message (key exchange initiation message). Upon receiving that KEX_HBR_INIT message, B also generates an ECDH ephemeral key pair and performs the algorithm `Encaps` to get a ciphertext (KEM_{C_B}). B then replies back to A with a KEX_HBR_REPLY message (key exchange reply message), consisting of the public host key of B (LK_B), the ECDH ephemeral public key, the ciphertext, and a signature of the "exchange hash." The precise computations of the exchange hash and the signature are depicted in Figure 5.2, where hash denotes the hash function. First, the shared secret K is computed by hashing the concatenation of the ECDH and KEM shared secrets. Then, the exchange hash H is computed by hashing the concatenation of the payloads of the two VERSION_EX messages and the two KEX_ALGR messages, the public host key of B, the ECDH and KEM ephemeral public keys of the client, the ECDH ephemeral public key and the KEM ciphertext of the server, and the shared secret K. Afterward, the signature SIGN is computed by signing H under the private host key of B (LSK_B).

5.2 Modeling the protocol

This section presents how to model the protocol in CafeOBJ. We also explain the threat model used in this case study, pointing out the difference with the one used in the PQ TLS case study.

5.2.1 Modeling ECDH and KEM

Elliptic Curve Diffie-Hellman

We introduce CafeOBJ sorts `EcSecretK`, `EcPublicK`, and `EcShareK` representing ECDH secret keys, public keys, and shared secrets, respectively, and some operators as follows:

```
[EcSecretK EcPublicK EcShareK]
-- the associated ECDH public key is derived from a secret key
op ecPublic : EcSecretK           -> EcPublicK {constr}
-- a shared key is computed from a public and a secret keys
op ecShare  : EcPublicK EcSecretK -> EcShareK
-- constructor of a shared key is a secret key pair
op _|_      : EcSecretK EcSecretK -> EcShareK  {constr comm}
```

The first operator represents the computation of the associated public key from a secret key, where the attribute `constr` states that the operator is a constructor of the sort `EcPublicK`. The second operator represents the computation of the shared secret from a public key and a secret key. Let `PK` and `K` be CafeOBJ variables of the sorts `EcPublicK` and `EcSecretK`, respectively, the semantic of `ecShare` is defined by the following equation:

```
eq ecShare(PK,K) = (ecSecret(PK) | K) .
```

where `ecSecret` returns the associated secret key of a given public key (it is the projection function of `ecPublic`). Given two ECDH secret keys k_1 and k_2, `ecPublic`(k_1) denotes the public key associated with k_1, and $(k_1$ `|` $k_2)$ denotes the shared secret obtained from that public key and the secret key k_2. The operator `_|_` is commutative, namely $(k_1$ `|` $k_2)$ and $(k_2$ `|` $k_1)$ are identical, thanks to the CafeOBJ attribute `comm`.

Key encapsulation mechanisms

As explained in the previous chapter, KEMs are modeled based on the abstract version defined in Definition 6. We introduce sorts `PqSecretK`, `PqPublicK`, `PqShareK`, and `PqCipher`, representing secret keys, public keys, shared secrets, and ciphertexts (or encapsulations), respectively. Let `K'`, `K2'`, and `K3'` are variables of `PqSecretK`, `PK'` and `C` are variables of `PqPublicK` and `PqCipher`, respectively. The algorithms `KeyGen`, `Encaps`, and `Decaps` are modeled by the following operators and equations:

```
op keygen  : PqSecretK           -> PqPublicK {constr}
```

```
-- Encaps algorithm: returns ciphertext
op encapsC : PqPublicK PqSecretK -> PqCipher  {constr}
-- Encaps algorithm: returns shared key
op encapsK : PqPublicK PqSecretK -> PqShareK
op decaps  : PqCipher  PqSecretK -> PqShareK
-- constructor of a shared key is a secret key pair
op _&_     : PqSecretK PqSecretK -> PqShareK  {constr}
eq encapsK(PK', K') = (pqSecret(PK') & K') .
ceq decaps(C, K')   = (K' & pqSecret(C))  if (pqPublic(C) = keygen(K')) .
ceq (decaps(C, K') = (K' & K2'))  = false if not(K2' = pqSecret(C)) .
ceq (decaps(C, K') = (K2' & K3')) = false if not(K' = K2') .
```

Note that `KeyGen` and `Encaps` are probabilistic algorithms. Thus, to specify them as deterministic procedures in CafeOBJ, an argument of the sort `PqSecretK` is added as the input argument. Note also that with `Encaps`, two separate operators `encapsC` and `encapsK` are defined, respectively returning the ciphertext and the shared secret. `pqPublic` and `pqSecret` are the projection functions of `encapsC`, returning its first and second arguments, respectively (`pqSecret` is also the projection function of `keygen`). Given an encapsulation `C` and a secret key `K'`, the second equation states that `Decaps(C,K')` properly outputs the shared secret only if `C` encapsulates some secret to the associated public key of `K'` (in other words, the public key of `C` is the associated public key of `K'`). The third equation states that `Decaps(C,K)` cannot be `(K' & K2')` if `C` does not encapsulate `K2'`.

5.2.2 Modeling cryptographic primitives and messages exchanged

We introduced sorts `Prin`, `PubKey`, `PriKey`, `Version`, and `Suites` representing principals, public host keys, private host keys, protocol versions, and lists of supported algorithms, respectively. We additionally introduce two generic sorts `Data` and `DataL`, which are the supersorts of all sorts mentioned before, such as `EcPublicK` and `PqPublicK`. `||` is used as the concatenation operator, where `assoc` indicates that the operator is associative:

```
[Data < DataL]
[EcPublicK EcShareK PqSecretK PqPublicK ... < Data]
-- concatenation operator
op _||_   : DataL DataL                -> DataL {assoc constr}
```

where `...` denotes that some other sorts are omitted.

To model the hash function, the Sign and Verify signature algorithms, we declare the following operators:

```
-- hash function
op h      : DataL                      -> Data {constr}
```

```
--          key    plaintext              signature
op sign   : DataL  DataL               -> Data {constr}
--          key    plaintext  signature
op verify : DataL  DataL      DataL     -> Bool
```

There are several equations defining those operators, for example:

```
eq verify(pubK(A), D, SIGN) = (SIGN = sign(priK(A),D)) .
```

which states that given a public host key of principal `A`, a message `D`, and a signature, Verify outputs true when the signature is obtained by signing `D` under the private host key of `A`.

We turn to model messages exchanged in the protocol. The VERSION_EX, KEX_ALGR, KEX_HBR_INIT, and KEX_HBR_REPLY messages depicted in Figure 5.1 are respectively represented by the following operators:

```
op verM    : Prin Prin Prin Version   -> Msg {constr}
op kexAlgM : Prin Prin Prin Suites    -> Msg {constr}
op hbrIniM : Prin Prin Prin DataL Nat -> Msg {constr}
op hbrRepM : Prin Prin Prin DataL Nat -> Msg {constr}
```

The first, second, and third arguments of each operator denote the real author, the seeming sender, and the recipient of a given message, respectively. Recall that the first argument is used for modeling and verification purposes only, but it can neither be seen by the receiver nor be controlled by the intruder. In contrast, the other arguments may be forged by the intruder, which will be described in Section 5.2.4. `Nat` is the sort of natural numbers. The argument of the sort `Nat` is embedded in the last of the two operators representing key exchange initiation and key exchange reply messages to store the time when the corresponding message is sent. That time information is necessary to specify and verify the *forward secrecy* property later on, for instance, to check whether the key concerned is established before the compromise of the server's private host key who established that key in a session with another client. The time is represented as a natural number. Initially, the time of the system is set to 0, and after each action such as a message being sent, it is incremented. With the first two operators, we could also embed an argument of the sort `Nat` in the last of each one, but saving time information of version exchange and key exchange algorithms messages is not strictly necessary to complete the verification.

5.2.3 Modeling the protocol execution

Sorts `Sys` and `Network` are defined, representing the state space and the network, where the network is modeled as an AC-collection of messages exchanged between principals. All initial states are represented by the constant `init`. Five observers `nw`, `usecret`, `time`, `leakscr`, and `knl` are defined, observing the network, the set of ECDH & KEM secret keys used by all principals, the

system time, the compromised secrets, and the knowledge of the intruder, respectively. `usecret` is used to guarantee the uniqueness of ephemeral secret keys. The compromised secrets can be ephemeral secret keys, private host keys, and shared secrets between two participants, which will be described in the next sections. The declarations of `init` and the five observers and the definition of initial states are as follows:

```
op init    :     -> Sys {constr}
op nw      : Sys -> Network
op usecret : Sys -> SecretKS
op time    : Sys -> Nat
op leakscr : Sys -> SecretKS
op knl     : Sys -> DataL
eq nw(init) = void .
eq usecret(init) = empty .
eq time(init) = 0 .
eq leakscr(init) = empty .
eq knl(init) = (priK(intru) || pubK(intru)) .
```

where `SecretKS` is the sort of sets of secret data types (e.g., private host keys, ECDH & KEM secret keys). From the five equations, it follows that in an initial state, the network is empty (denoted by `void`), the set of secrets used is also empty, time of the system is `0`, no secret is revealed, and the intruder knowledge is their own private and public host keys.

For each of the six messages depicted in Figure 5.1, we define a transition modeling how that message is sent. For instance, the transition `cHbrInit` below specifies how a client sends a key exchange initiation message.

```
op cHbrInit : Sys Prin Prin Prin EcSecretK PqSecretK
              Version Version Suites Suites -> Sys {constr}
ceq nw(cHbrInit(S,B2,A,B,K,K',V,V2,CSs,CSs2)) =
    (hbrIniM(A,A,B, ecPublic(K) || keygen(K'), time(S)) , nw(S))
  if c-cHbrInit(S,B2,A,B,K,K',V,V2,CSs,CSs2) .
ceq usecret(cHbrInit(S,B2,A,B,K,K',V,V2,CSs,CSs2)) =
    (K K' usecret(S))
  if c-cHbrInit(S,B2,A,B,K,K',V,V2,CSs,CSs2) .
ceq time(cHbrInit(S,B2,A,B,K,K',V,V2,CSs,CSs2)) =
    s(time(S))
  if c-cHbrInit(S,B2,A,B,K,K',V,V2,CSs,CSs2) .
eq leakscr(cHbrInit(S,B2,A,B,K,K',V,V2,CSs,CSs2)) = leakscr(S) .
ceq knl(cHbrInit(S,B2,A,B,K,K',V,V2,CSs,CSs2)) =
    (ecPublic(K) || keygen(K') || knl(S))
  if c-cHbrInit(S,B2,A,B,K,K',V,V2,CSs,CSs2) .
ceq cHbrInit(S,B2,A,B,K,K',V,V2,CSs,CSs2) = S
  if not c-cHbrInit(S,B2,A,B,K,K',V,V2,CSs,CSs2) .
eq c-cHbrInit(S,B2,A,B,K,K',V,V2,CSs,CSs2) =
  (kexInitM(A,A,B,CSs) \in nw(S) and
```

```
kexInitM(B2,B,A,CSs2) \in nw(S) and
not(K \in usecret(S) or K' \in usecret(S))) .
```

where `A`, `B`, `V`, etc., are CafeOBJ variables of the corresponding sorts. `\in` is the membership predicate. `c-cHbrInit` is the effective condition of the transition, which states that the transition cannot proceed unless two key exchange algorithms messages have been exchanged, and the two secret keys (ECDH and KEM) `K` and `K'` have not been used before. The first five (conditional) equations say that if the effective condition is satisfied, from the two secret keys `K` and `K'`, client `A` sends the two associated public keys to `B` under a key exchange initiation message (by putting that message into the network), the two secret keys are put into the set of secret keys used, the time is incremented, and the two public keys are added to the intruder knowledge (i.e., the intruder learned the two public keys). The sixth equation states that everything remains unchanged if the effective condition is not satisfied.

5.2.4 Threat model and modeling the intruder

The threat model used in this verification case study is an extended version of the Dolev-Yao intruder model [56]. As a Dolev-Yao intruder, the intruder can completely control the network, concretely:

(1) The intruder can intercept any message sent in the network and glean information carried in that message. This capability has been partially illustrated through the definition of the transition `cHbrInit` presented in Section 5.2.3. That is, whenever an honest principal sends two public keys to another one through a key exchange initiation message, the intruder will learn the two public keys.

(2) The intruder also knows all publicity information such as protocol versions, names of cryptographic primitives, and public host keys even without gleaning them from the network.

(3) The intruder can select an ephemeral secret key (either ECDH one or KEM one), and generate the corresponding public key or the shared secret provided that the secret key has not been used before (uniqueness).

(4) If a piece of information is available to the intruder, they can use any cryptographic primitive function taking the information as input and learning the output.

(5) The intruder can use the information available to them to build a message and impersonate some honest principal to send the message to another.

In addition to the Dolev-Yao capabilities above, our threat model also considers the following:

(6) The security of ECDH is broken. If two ECDH public keys are given to the intruder, the intruder can derive the corresponding shared secret, which is assumed by utilizing the power of large quantum computers.

(7) Secrets may be compromised and the intruder gleans them. All of ECDH & KEM ephemeral secret keys, private host keys, and shared secrets established between two principals are possibly revealed.

For each capability, multiple transitions are defined to specify it. One of the transitions specifying the capability (3) is as follows:

```
op gPqSecretK : Sys PqSecretK -> Sys {constr}
eq nw(gPqSecretK(S,K')) = nw(S) .
ceq usecret(gPqSecretK(S,K')) = (K' usecret(S))
  if c-gPqSecretK(S,K') .
eq time(gPqSecretK(S,K')) = time(S) .
eq leakscr(gPqSecretK(S,K')) = leakscr(S) .
ceq knl(gPqSecretK(S,K')) = (K' || keygen(K') || knl(S))
  if c-gPqSecretK(S,K') .
ceq gPqSecretK(S,K') = S
  if not c-gPqSecretK(S,K') .
eq c-gPqSecretK(S,K') = not(K' \in usecret(S)) .
```

It states that the intruder can randomly select a KEM secret key `K'`, add it and the public key generated by the algorithm `KeyGen` to their knowledge provided that `K'` has not been used before.

The following is a part of a transition, which partially specifies the capability (4):

```
op g1 : Sys DataL -> Sys {constr}
ceq knl(g1(S,DL)) = (h(DL) || knl(S))
  if c-g1(S,DL) .
eq c-g1(S,DL) = DL \in knl(S) .
```

It states that the intruder can learn the hash of a piece of information if that information is available to them. Multiple other transitions specify that the intruder can sign any available information by any available key, compute ECDH shared secrets from any available public/secret key pair, and compute KEM ciphertexts and shared secrets by `Encaps` and `Decaps` on the available inputs, among others.

With the capability (5), for instance, a part of the transition `fkHbrInit` shown below illustrates how the intruder can forge a key exchange initiation message:

```
op fkHbrInit : Sys Prin Prin EcPublicK PqPublicK -> Sys {constr}
ceq nw(fkHbrInit(S,A,B,PK,PK')) =
    (hbrIniM(intru,A,B, PK || PK', time(S)) , nw(S))
  if c-fkHbrInit(S,A,B,PK,PK') .
ceq time(fkHbrInit(S,A,B,PK,PK')) = s(time(S))
```

```
  if c-fkHbrInit(S,A,B,PK,PK') .
eq c-fkHbrInit(S,A,B,PK,PK') = (PK \in knl(S) and PK' \in knl(S)) .
```

The equations state that if the two ECDH and KEM public keys `PK` and `PK'` are in the intruder knowledge, the intruder can impersonate principal `A`, sending the two public keys through a key exchange initiation message to principal `B`. As mentioned before, the first parameter inside `hbrIniM` is `intru` but not `A`, which is impossible to be seen by the receiver. Note that the system time is incremented also. The formal specification also specifies how the intruder can forge other kinds of messages, such as a key exchange reply message.

With the capability (7), the following transition `lPqSecretK1` partially specifies the compromise of a KEM ephemeral secret key:

```
op lPqSecretK1 : Sys Prin Prin EcSecretK PqSecretK Nat -> Sys {constr}
ceq time(lPqSecretK1(S,A,B,K,K',N)) = s(time(S))
  if c-lPqSecretK1(S,A,B,K,K',N) .
ceq leakscr(lPqSecretK1(S,A,B,K,K',N)) = (K' leakscr(S))
  if c-lPqSecretK1(S,A,B,K,K',N) .
ceq knl(lPqSecretK1(S,A,B,K,K',N)) = (K' || knl(S))
  if c-lPqSecretK1(S,A,B,K,K',N) .
eq c-lPqSecretK1(S,A,B,K,K',N) =
  hbrIniM(A,A,B, ecPublic(K) || keygen(K'), N) \in nw(S) .
```

It states that if a key exchange initiation message is in the network, the KEM secret key associated with the public key sent in that message can be compromised. If that is the case, the secret key is added to the intruder knowledge as well as the set of compromised secrets, and the system time is incremented. Another transition specifies the compromise of KEM ephemeral secret keys through key exchange reply messages. The formal specification also specifies the compromises of KEM shared secrets and private host keys.

From what has been described, it can be seen that a more general way has been used to model the intruder's capabilities of learning information and forging messages rather than by using specific transitions dedicated to a protocol only. Besides, we no longer embed time information into shared keys, which is unnatural, in order to verify the *forward secrecy* property. In summary, the complete formal specification of the PQ SSH protocol consists of 993 lines of CafeOBJ code, where 723 lines are dedicated to specifying the protocol execution and 270 lines are dedicated to defining security properties and auxiliary lemmas, which are used for the formal verification. The formal specification limits neither the number of honest principals participating in the protocol nor the number of sessions that the protocol can execute. Generally speaking, it allows any principal (a variable of sort `Prin`) to initialize a session with any other principal regardless of how many times and without any restriction by executing the transition formalizing the sending of a VERSION_EX message.

5.3 Formal analysis

Security considerations of the SSH Transport Layer protocol are claimed in [96] as follows:

> This protocol provides a secure encrypted channel over an insecure network. It performs server host authentication, key exchange, encryption, and integrity protection. It also derives a unique session ID that may be used by higher-level protocols.

Such security requirements should be kept fulfilled in the PQ SSH protocol. This section presents the formal analysis of four properties including (1) *session key secrecy*, (2) *forward secrecy*, (3) *session identifier uniqueness*, and (4) *authentication*. (1) makes sure that nobody can learn a shared secret negotiated between a client and a server except those two principals (see Section 5.3.1). (2) guarantees that even if a private host key of a server is compromised, shared secrets established before the compromise remain secure (see Section 5.3.2). (3) ensures that the exchange hash, acting as the session identifier, is unique (see Section 5.3.3). (4) states that upon completion of a protocol execution, if client A has communicated apparently with server B, then the server is indeed B (see Section 5.3.4).

Checking the specification

We first check that the formal specification allows two principals successfully complete a protocol execution and obtain the shared secret. This must be fulfilled, otherwise, anything we do after is entirely meaningless. We have confirmed that by showing a reachable state satisfying that requirement through a sequence of transitions, which can be found on the webpage mentioned at the beginning of this chapter.

Checking intruder capability of learning ECDH shared secret

We also confirmed that the intruder is able to learn the ECDH shared secret established between two honest principals as what has been modeled for the intruder's capabilities. Similarly, we have verified that by pointing out a reachable state in which the intruder can learn such a secret.

5.3.1 Session key secrecy property

The negotiation of a shared secret between two principals must be secure against any third party. This is called the *session key secrecy* property, which is specified by the following predicate:

```
op keySe : Sys Prin Prin Prin Version Version Suites Suites
    EcSecretK PqSecretK EcPublicK PqCipher Data Nat Nat -> Bool
eq keySe(S,B2,A,B,V,V2,CSs,CSs2,K,K',PK2,C,SIGN,N,N2) =
 (not(A = intru or B = intru) and
```

```
  hbrIniM(A,A,B, ecPublic(K) || keygen(K'), N) \in nw(S) and
  hbrRepM(B2,B,A, pubK(B) || PK2 || C || SIGN, N2) \in nw(S) and
  verify(pubK(B), h(V || V2 || CSs || CSs2 ||
    pubK(B) || ecPublic(K) || keygen(K') || PK2 || C ||
    h(ecShare(PK2,K) || decaps(C,K'))),
   SIGN) and
  not(decaps(C,K') \in leakscr(S)) and
  not(K' \in leakscr(S) or pqSecret(C) \in leakscr(S)) and
  not(priK(B) \in' leakscr(S)))
 implies not(h(ecShare(PK2,K) || decaps(C,K')) \in knl(S)) .
```

The predicate states that when honest client `A` has sent to honest server `B` a key exchange initiation message and has received back a key exchange reply message apparently sent from `B` with a valid signature of the exchange hash, neither the KEM shared secret, the two corresponding ephemeral secret keys, nor the private host key of the server is revealed, then the intruder cannot learn the shared secret (i.e., the hash of the ECDH shared secret and the KEM shared secret).

All sub-constraints in the premise of the predicate are necessary, namely, if any of them is eliminated, the predicate will be no longer valid. Indeed, if the following constraint:

(1) the signature of the exchange hash in the key exchange reply message is valid

is eliminated, the reply message received may be actually forged by the intruder (`B2` is `intru`), who is trying to impersonate `B`. In this case, the signature will be failingly verified if `A` does a check, however, it is not performed actually because (1) is removed.

If the following constraint:

(2) the KEM shared secret is not revealed

is removed, the intruder can derive the shared secret because the intruder can break ECDH's security to learn the ECDH shared secret. It also explains why we only bind the non-reveal of the KEM shared secret in the premise of `keySe`.

If the following constraint:

(3) the two corresponding KEM ephemeral secret keys are not revealed

is removed, the intruder can easily derive the KEM shared secret, and then they can derive the shared secret as explained above.

If the following constraint:

(4) the private host key of the server is not revealed

is eliminated, the intruder can use the revealed key to sign the exchange hash to make a valid signature. Subsequently, the intruder can completely impersonate server `B` to do the key exchange with `A`. As a result, the shared secret is obviously available to the intruder. Readers can find on the above-mentioned webpage the counterexamples showing that the predicate will be no longer valid if any of (1), (2), (3), and (4) is eliminated. Appendix B gives a detailed clarification of the counterexample when the constraint (4) is eliminated, which can be regarded as the most

interesting one.

keySe is simply proved without induction by using the following two lemmas:

```
op inv0 : Sys EcSecretK PqSecretK EcPublicK PqCipher -> Bool
eq inv0(S,K,K',PK2,C) = h(ecShare(PK2,K) || decaps(C,K')) \in knl(S)
  implies decaps(C,K') \in knl(S) .

op secrecy : Sys Prin Prin Prin Version Version Suites Suites
    EcSecretK PqSecretK EcPublicK PqCipher Data Nat Nat -> Bool
eq secrecy(S,B2,A,B,V,V2,CSs,CSs2,K,K',PK2,C,SIGN,N,N2) =
 (not(A = intru or B = intru) and
  hbrIniM(A,A,B, ecPublic(K) || keygen(K'), N) \in nw(S) and
  hbrRepM(B2,B,A, pubK(B) || PK2 || C || SIGN, N2) \in nw(S) and
  verify(pubK(B), h(V || V2 || CSs || CSs2 ||
    pubK(B) || ecPublic(K) || keygen(K') || PK2 || C ||
    h(ecShare(PK2,K) || decaps(C,K'))),
  SIGN) and
  not(decaps(C,K') \in leakscr(S)) and
  not(K' \in leakscr(S) or pqSecret(C) \in leakscr(S)) and
  (not(priK(B) \in' leakscr(S)) or
    N2 < timeLeak(priK(B), leakscr(S))))
 implies not(decaps(C,K') \in knl(S)) .
```

inv0 states that if a shared secret is available to the intruder, then the KEM shared secret component must be available to the intruder. secrecy states that when honest client A has sent to honest server B a key exchange initiation message and has received back a key exchange reply message apparently sent from B with a valid signature of the exchange hash, neither the KEM shared secret nor the two corresponding ephemeral secret keys are revealed, and either the private host key of B is uncompromised or it is compromised but the time when the compromise happens (denoted by timeLeak(priK(B), leakscr(S))) is after the shared secret establishment (denoted by N2), then the intruder cannot learn the KEM shared secret.

The proof of keySe using the two lemmas is simply as follows:

```
red (inv0(s,k,k',pk2,c) and
     secrecy(s,b2,a,b,v,v2,css,css2,k,k',pk2,c,sign,n,n2))
  implies keySe(s,b2,a,b,v,v2,css,css2,k,k',pk2,c,sign,n,n2) .
```

where s, a, b, etc., are fresh constants of the corresponding sorts. As explained in Section 4.4.2, the proof is accomplished with respect to an unbounded number of protocol executions. To prove the two lemmas are invariants, we use the tool IPSG with some other lemmas to generate their proof scores. The complete proof scores are available on the webpage[2]. Detailed information about the time taken as well as the size of the generated proof scores are available in Section 5.3.6.

[2]https://github.com/duongtd23/PQSSH

5.3.2 Forward secrecy property

Forward secrecy property in general is defined as that the compromise of a long-term private key does not break the secrecy of a session key if the session is completed before the compromise. This property is specified by the following predicate:

```
op fwdSe : Sys Prin Prin Prin Version Version Suites Suites
    EcSecretK PqSecretK EcPublicK PqCipher Data Nat Nat -> Bool
eq fwdSe(S,B2,A,B,V,V2,CSs,CSs2,K,K',PK2,C,SIGN,N,N2) =
 (not(A = intru or B = intru) and
  hbrIniM(A,A,B, ecPublic(K) || keygen(K'), N) \in nw(S) and
  hbrRepM(B2,B,A, pubK(B) || PK2 || C || SIGN, N2) \in nw(S) and
  verify(pubK(B), h(V || V2 || CSs || CSs2 ||
    pubK(B) || ecPublic(K) || keygen(K') || PK2 || C ||
    h(ecShare(PK2,K) || decaps(C,K'))),
   SIGN) and
  not(decaps(C,K') \in leakscr(S)) and
  not(K' \in leakscr(S) or pqSecret(C) \in leakscr(S)) and
  priK(B) \in' leakscr(S) and
  N2 < timeLeak(priK(B), leakscr(S)))
 implies not(h(ecShare(PK2,K) || decaps(C,K')) \in knl(S)) .
```

The predicate states that if honest client `A` has sent to honest server `B` a key exchange initiation message and has received back a key exchange reply message apparently sent from `B` with a valid signature of the exchange hash, neither the KEM shared secret nor the two corresponding ephemeral secret keys are revealed, the private host key of the server is compromised but the compromise happens after the key exchange reply message is sent, then the intruder cannot learn the shared secret. `fwdSe` and `keySe` are almost identical, except for the only difference in the constraint about the compromise of the server private host key in the premise part. `fwdSe` guarantees that even if the server's private host key is compromised, shared secrets negotiated before the compromise remain secure. Similar to `keySe`, `fwdSe` is also simply proved without induction by using the two lemmas `inv0` and `secrecy`:

```
red (inv0(s,k,k',pk2,c) and
     secrecy(s,b2,a,b,v,v2,css,css2,k,k',pk2,c,sign,n,n2))
  implies fwdSe(s,b2,a,b,v,v2,css,css2,k,k',pk2,c,sign,n,n2) .
```

We cannot eliminate any constraint in the premise of `fwdSe` because of the same reasons explained previously with `keySe`. The three above-mentioned constraints (1), (2), and (3) are compulsory assumptions to guarantee the secrecy of a shared secret, that is, the signature of the exchange hash in the reply message received must be correctly verified and the KEM shared secret & the two KEM ephemeral secret keys must be not compromised. While with the server's private host key, we need to require that either the key is uncompromised or the compromise happens after the sending of the key exchange reply message.

5.3.3 Session identifier uniqueness property

During the key negotiation between two principals, the server authenticates himself/herself by signing the exchange hash with his/her private host key and sending the signature to the client. Besides that purpose, the exchange hash is also used as the session identifier for this connection. This session identifier must be unique in order to be used by some higher-level protocols as claimed in [96]. The following invariant specifies the uniqueness property of session identifiers:

```
op unique : Sys Prin Version Version Suites Suites
  EcSecretK PqSecretK EcSecretK PqSecretK
  Prin Version Version Suites Suites
  EcSecretK PqSecretK EcSecretK PqSecretK -> Bool
eq unique(S,B,V,V2,CSs,CSs2,K,K',K2,K2',
    B3,V3,V4,CSs3,CSs4,K3,K3',K4,K4') =
  h(V || V2 || CSs || CSs2 || pubK(B) || ecPublic(K) ||
    keygen(K') || ecPublic(K2) || encapsC(keygen(K'),K2') ||
    h((K | K2) || (K' & K2'))) =
  h(V3 || V4 || CSs3 || CSs4 || pubK(B3) || ecPublic(K3) ||
    keygen(K3') || ecPublic(K4) || encapsC(keygen(K3'),K4') ||
    h((K3 | K4) || (K3' & K4')))
  implies (K = K3 and K' = K3' and K2 = K4 and K2' = K4') .
```

The equation states that if two session identifiers are identical, the corresponding ephemeral secret keys that construct the two sessions must be equal. Provided that all ephemeral secret keys are one-time used, it can be deduced from `unique` that two session identifiers are always different. `unique` is proved by reduction only, that is CafeOBJ directly reduces the invariant to true:

```
red unique(s,b,v,v2,css,css2,k,k',k2,k2',b3,v3,v4,css3,css4,k3,k3',k4,k4') .
```

In summary, 17 additional lemmas are introduced to complete the verifications of the *session key secrecy*, *forward secrecy*, and *session identifier uniqueness* properties.

5.3.4 Authentication property

The IETF Draft [84] states that the protocol provides server authentication. This property is stated (from a client's point of view) as follows: if client A performs a key negotiation apparently with server B, then the server that A communicates with is really B. We attempt to specify this property in CafeOBJ by the following predicate, where `?M` is existentially quantified.

```
op auth : Sys Prin Prin Prin Version Version Suites Suites
  EcSecretK PqSecretK EcPublicK PqCipher Data Nat Nat Nat -> Bool
eq auth(S,B2,A,B,V,V2,CSs,CSs2,K,K',PK2,C,SIGN,N,N2,?M) =
 (not(A = intru or B = intru) and
  not(decaps(C,K') \in leakscr(S)) and
```

Step-1	A	$A \rightarrow B$:	$\mathrm{ECDH}_{\mathrm{PK}} \,\|\|\, \mathrm{KEM}_{\mathrm{PK}}$
Step-2	I	learns	$\mathrm{ECDH}_{\mathrm{PK}} \,\|\|\, \mathrm{KEM}_{\mathrm{PK}}$
Step-3	I	$A_2 \rightarrow B$:	$\mathrm{ECDH}_{\mathrm{PK}} \,\|\|\, \mathrm{KEM}_{\mathrm{PK}}$
Step-4	B	$B \rightarrow A_2$:	$\mathrm{LK}_B \,\|\|\, \mathrm{ECDH}_{\mathrm{PK}_2} \,\|\|\, \mathrm{KEM}_{\mathrm{C}} \,\|\|\, \mathrm{SIGN}$
Step-5	I	learns	$\mathrm{LK}_B \,\|\|\, \mathrm{ECDH}_{\mathrm{PK}_2} \,\|\|\, \mathrm{KEM}_{\mathrm{C}} \,\|\|\, \mathrm{SIGN}$
Step-6	I	$B \rightarrow A$:	$\mathrm{LK}_B \,\|\|\, \mathrm{ECDH}_{\mathrm{PK}_2} \,\|\|\, \mathrm{KEM}_{\mathrm{C}} \,\|\|\, \mathrm{SIGN}$

where I denotes the intruder

Figure 5.3: Counterexample of `auth`

```
 not(K' \in leakscr(S) or pqSecret(C) \in leakscr(S)) and
 not(priK(B) \in' leakscr(S)) and
 hbrIniM(A,A,B, ecPublic(K) || keygen(K'), N) \in nw(S) and
 hbrRepM(B2,B,A, pubK(B) || PK2 || C || SIGN, N2) \in nw(S) and
 verify(pubK(B), h(V || V2 || CSs || CSs2 ||
   pubK(B) || ecPublic(K) || keygen(K') || PK2 || C ||
   h(ecShare(PK2,K) || decaps(C,K'))),
  SIGN))
implies
 hbrRepM(B,B,A, pubK(B) || PK2 || C || SIGN, ?M) \in nw(S) .
```

The equation states that if honest client `A` has sent to honest server `B` a key exchange initiation message and has received back a key exchange reply message apparently sent from `B` with a valid signature of the exchange hash, neither the KEM shared secret, the two corresponding ephemeral secret keys, nor the private host key of the server is revealed, then `B` has indeed sent the key exchange reply message to `A` at some time denoted by `?M`. We repeat again that it is wrong to affirm `B2 = B` in the conclusion of `auth` because `B2` may be the intruder, who replayed the key exchange reply message originally sent by `B`. As a result, there exist two different messages in the network: an original one sent by `B` and the other one replayed by the intruder. Note that `?M`, which is existentially quantified, must be used because the time when the original message was sent by the honest server is unknown, in particular, it cannot be derived from the time when the replaying message was sent by the intruder (i.e., `N2`). The ?-symbol prefix is not mandatory from a syntactic point of view, but it helps to distinguish variables that are existentially quantified and universally quantified.

However, a counterexample of `auth` is found, namely, the property stated by `auth` does not hold. The counterexample can be found on the webpage mentioned before. Figure 5.3 briefly explains how the counterexample can happen. According to the figure, there are mainly six steps, where the first name at each step denotes the principal who performs the given action. In the first and second steps, A sends two ephemeral public keys to B under a key exchange initiation message, and then the intruder gleans them. Using the two public keys gleaned, in Step-3, the

intruder tries to impersonate another client A_2 to initialize a new session with B, sending the two public keys just gleaned to B. Upon receiving the faking message, B believes that it was truly sent from A_2, and then in Step-4, B replies back to A_2 with a key exchange reply message with the public host key, an ECDH ephemeral public key, a KEM ciphertext, and a signature over the exchange hash. In Step-5, the intruder once again intercepts the key exchange reply message, gleaning all pieces of information in that message. In the final step, by using the information just learned, the intruder tries to impersonate B to send a key exchange reply message to A. After this step, there exists in the network a valid key exchange reply message whose creator is the intruder, the seeming sender is B, and the receiver is A, i.e., the message in the form of `hbrRepM(intru,B,A,...)`. However, there does not exist a key exchange reply message with the same content really sent by B to A in the network. B sent such a message to A_2 instead. The CafeOBJ code showing the transition sequence leading to this counterexample is presented in detail in Appendix A.

The found counterexample can be regarded as a weakness of the protocol. This weakness does not affect the confidentiality of session keys shared by honest participants. Indeed, I cannot learn the shared secret derived from the keys generated by A and B depicted in Figure 5.3. The intruder could only learn the public information sent in the messages intercepted and forward/replay them to someone, but could not derive the associated secret information, such as the KEM secret key and shared key.

5.3.5 Revising the exchange hash

To address the weakness described above and to make the protocol enjoy the *authentication* property, we proposed to revise the protocol by including the identifiers of the client and the server in the exchange hash. Precisely, the exchange hash is computed as follows:

$$\begin{aligned}\mathrm{H} = \mathrm{hash}(&\mathrm{Version}_A \,||\, \mathrm{Version}_B \,||\, \mathrm{Suites}_A \,||\, \mathrm{Suites}_B \,||\, \mathrm{LK}_B \,||\\ &\mathrm{ECDH}_{\mathrm{PK}_A} \,||\, \mathrm{KEM}_{\mathrm{PK}_A} \,||\, \mathrm{ECDH}_{\mathrm{PK}_B} \,||\, \mathrm{KEM}_{\mathrm{C}_B} \,||\, \mathrm{K} \,||\, A \,||\, B)\end{aligned}$$

In this way, when the intruder tries to impersonate B to send the key exchange reply message to A (Step-6 in Figure 5.3), A will not accept that message because the signature SIGN will not be successfully verified. Upon reception of that message, A expects that the identifier of A is included in the signature, but actually, SIGN is signed over A_2 rather than A. Therefore, the counterexample will be prevented.

That is the informal argument, to prove that revising the protocol in that way does indeed make it enjoy the *authentication* property, formal verification must be conducted. We revise the CafeOBJ formal specification accordingly and verify the *authentication* property again. With the improved version, we successfully prove `auth` with the employment of IPSG. Besides, the

three other properties remain secure with respect to the improved protocol. Note that we need to slightly revise the four predicates specifying the four properties to make the client and the server identifiers included in the exchange hash. Again, the proofs with respect to the improved protocol are available on the webpage[3].

IPSG has been used to produce the proof of `auth`. To this end, IPSG has been extended so that it can handle existentially quantified variables based on the *Skolemization* process, a way of removing existential quantifiers from a formula. This is done by introducing a *Skolem constant* or a *Skolem function* to replace a variable that is existentially quantified. With variables bound by existential quantifiers which are not inside the scope of universal quantifiers, the variables can simply be replaced by Skolem constants. For example, $\exists m\, P(m)$ can be changed to $P(c)$, where c is a suitable constant. With an existentially quantified variable inside universal quantifiers, the variable can be replaced by a Skolem function on the variables that are universally quantified. For example, $\forall a \exists m\, Q(a, m)$ can be changed to $\forall a\, Q(a, f(a))$, where f is a new function.

The 17 auxiliary lemmas can be reused to formally verify again the *session key secrecy*, *forward secrecy*, and *session identifier uniqueness* properties with respect to the improved protocol. To this end, we need to slightly revise 5 among those lemmas by simply adding the identifiers of the client and the server concerned into the exchange hash. No new lemma is needed. The remaining job is just to ask IPSG to produce the proofs again for the three properties and the lemmas. The experimental results, which are available in the upcoming section, indicate that IPSG took 1 second up to around 8 seconds to produce each invariant proof score, which is reasonably small. Therefore, the verification process helps us to save a lot of time and effort. Once a property has been successfully proved, when the protocol and/or the property are slightly changed, the verification by proof scores allows us to reuse most of the auxiliary lemmas, while the others are only needed to be slightly revised. Regenerating the proofs is trivial because it is automated by the tool taking only a bit of time. This is an advantage of the verification approach compared to model checking-based and its variant approaches. When conducting model checking, each time the protocol or the properties under verification are changed even a little bit, verification should be redone from the beginning, meaning that it is time-consuming because the model checker takes time to terminate for each verification experiment.

5.3.6 IPSG experimental results

In summary, to complete the formal verification of the four properties with respect to the improved protocol, 20 lemmas have been conjectured and used. Table 5.1 reports the time taken by IPSG to generate the proofs of those 20 lemmas and `auth`, which specifies the *authentication* properties. Recall that `keySe` and `fwdSe` are simply proved without induction by using the lemma

[3]https://github.com/duongtd23/PQSSH

Table 5.1: Time taken by IPSG to generate proofs (with respect to the improved protocol)

Invariant	Time (s)	No. open-close fragments	Invariant	Time (s)	No. open-close fragments
auth	7.1	132	inv9	1.3	74
secrecy	9.3	192	inv10	1.8	112
inv0	1.0	69	inv11	1.5	81
inv1	1.8	95	inv12	1.2	66
inv2	1.2	72	inv13	2.1	98
inv3	1.5	84	inv14	1.7	91
inv4	1.6	86	inv15	1.3	68
inv5	6.6	120	inv16	2.9	107
inv6	8.4	359	inv17	2.0	99
inv7	1.0	70	inv18	1.6	98
inv8	1.1	66			

secrecy, while unique is proved by reduction only. In the table, the last column shows the number of open-close fragments in the generated proof of each invariant. The experiments have been conducted on a MacBook Pro carrying 32 GB of memory with a processor i7 2.3 GHz.

Three lemmas inv16, inv17, and inv18 are used only for the auth's proof, meaning that they are not valid with the original protocol. The experimental results of the 17 remaining lemmas (i.e., secrecy, inv0, ..., inv15) with the original protocol are almost similar to those in Table 5.1 in terms of both the time taken and the number of open-close fragments generated.

5.3.7 Incorrect KEX_HBR_REPLY message format in the IETF Draft

The analysis reported above has been tackled with version 01 of the proposed protocol, which is the latest version by June 2023. Initially, however, we conducted the analysis with version 00[4] instead, which was the latest version by the time we started. There is an essential difference between the two versions, that is the IETF Draft version 00 actually defines the format of the key exchange reply message as a string S_REPLY, where:

> S_REPLY is the concatenation of S_PK1 and S_CT2. Typically, S_PK1 represents the ephemeral (EC)DH server public key. S_CT2 represents the ciphertext c output of the corresponding KEM's algorithm Encaps generated by the server which encapsulates a secret to the client public key.

That means the server's public host key and the signature over the exchange hash are not included. The missing of these two parts may result in server authentication, which is stated as a security property in the draft, is not guaranteed. Indeed, we have attempted to make the

[4]https://datatracker.ietf.org/doc/draft-kampanakis-curdle-ssh-pq-ke/00/

CafeOBJ formal specification of the protocol, in which the key exchange reply message does not contain the server public host key and the signature, and then we could point out counterexamples of both `keySe` and `auth`, meaning that both the *session key secrecy* and *authentication* properties were not valid. The reason essentially is that the intruder can completely impersonate server `B` to communicate with client `A`. The shared key that `A` believes is established with `B` is available to the intruder because the opposite peer is actually the intruder. As a result, this lack of server authentication also leads to a man-in-the-middle attack, i.e, in the connection between `A` and `B`, the intruder in the middle of the connection does: (1) impersonates `B`, acting as a server role to communicate with `A`, and (2) impersonates `A`, acting as a client role to communicate with `B`.

This difference can be regarded as a writing mistake in the IETF Draft version 00 because it has been corrected in the updated version 01. Moreover, both IETF Internet Standards of the SSH Transport Layer protocol based on DH [96] and ECDH [140] consistently define the format of such a message (SSH_MSG_KEXDH_REPLY in [96] and SSH_MSG_KEX_ECDH_REPLY in [140]) include the server public host key and the signature over the exchange hash. Even though the mistake has been corrected in the latest version, we have indeed found it in the formal analysis initially conducted.

5.4 Limitations

In this case study, we have addressed the limitations of modeling the intruder's capability by some specific transitions as well as embedding the time information in the shared secrets used in the PQ TLS case study. However, there are other two limitation points remaining unsolved. First, regarding the threat model we used, the only novel point of the intruder's capabilities is the presumption that they can break classical key exchange algorithms. Under the assumption of the presence of quantum computers, we expect the intruder will have some other novel quantum-based capabilities in our future work. Second, IPSG is not able to automatically produce a counterexample against an invalid invariant property. We repeat again that: (1) this is generally a drawback of interactive theorem proving compared to the model checking approach, and (2) doing theorem proving could help human experts to find counterexamples of non-trivial security properties, where model checking may be unable to do so due to the state explosion problem.

5.5 Summary

We have presented in this chapter the formal analysis of the PQ SSH protocol, a quantum-resistant version of the SSH Transport Layer protocol. Four properties have been taken into account including (1) *session key secrecy*, (2) *forward secrecy*, (3) *session identifier uniqueness*,

and (4) *authentication*. The analysis has formally verified that the protocol enjoys (1), (2), and (3), while it does not enjoy (4). The protocol was then proposed to be slightly improved by adding the identifiers of the client & server into the exchange hash. The formal verification has confirmed that the improved protocol enjoys (4). The properties have been proved with respect to an unbounded number of protocol participants and session executions, by using the tool IPSG to produce the proof scores in CafeOBJ. Even though the verification process is not completely automated, the use of IPSG allows us to only focus on only one task, namely to conjecture lemmas. The reuse of most auxiliary lemmas has helped us to save a lot of time and effort when conducting verifications again for the three properties (1), (2), and (3) with respect to the improved protocol. The formal verification shares many commons with what has been used in the PQ TLS verification case study presented in the previous chapter. However, there is also a novel distinction in this case study, that is, we no longer modeled the intruder's capability of learning information from sent messages and forging messages by some specific transitions as in the PQ TLS case study. Instead, we have chosen a more general way to model the intruder's capabilities, which results in at least two superiorities: (1) efforts can be saved when specifying other cryptographic protocols because those transitions may be reused, and (2) making sure that the intruder is given the full capability of learning information from messages exchanged in the network and forging an arbitrary message synthesized from the information that has been learned.

Similar to the PQ TLS verification case study, comprehending the protocol is the task that takes the most time before conducting the formal analysis. The most difficult task would be to comprehend the post-quantum cryptographic primitives used in the protocol, such as CRYSTALS-Kyber KEM. The challenges in the analysis are also to come up with a reasonable and strong threat model, comprehend the post-quantum cryptographic primitives, and abstract them to use in the formal specification. Another challenge in the analysis is to find out the general way to specify in CafeOBJ the attacker's capability of fully controlling the network, which was not carried out in the last PQ TLS verification case study.

Symbolic and computational approaches to cryptographic protocol verification are complementary to each other. Historically, both of them have been widely used for protocol security analysis, resulting in the modern cryptographic protocols used today. To prepare for the upcoming quantum computing era, their combination once again is necessary for security analysis research to construct secure post-quantum cryptosystems. So far, however, to the best of our knowledge, the number of case studies on symbolic verification of post-quantum cryptographic protocols, like our case studies presented in this chapter and the previous chapter, is very limited.

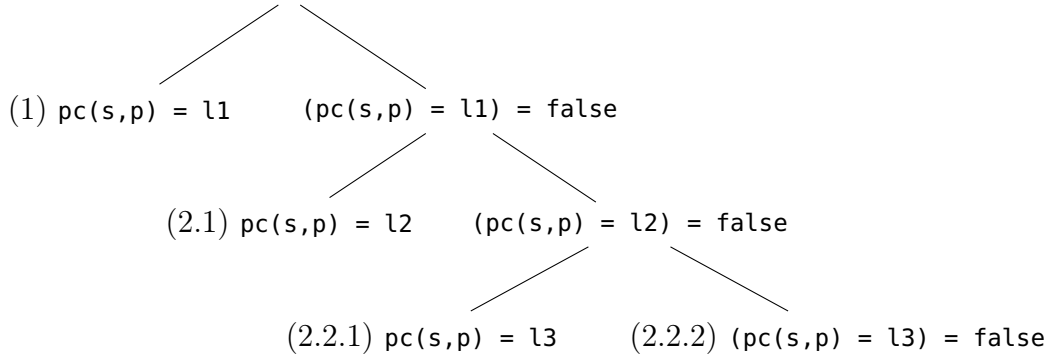
Chapter 6

Related work

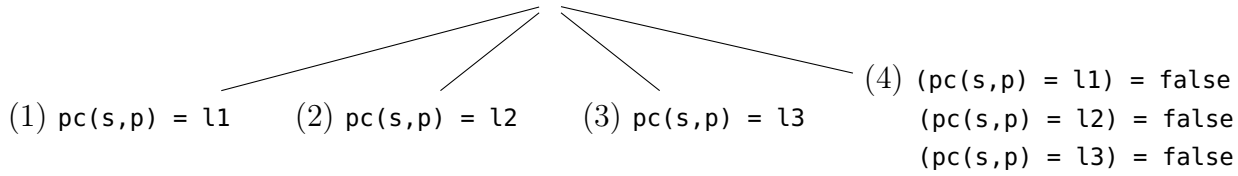
In this chapter, we discuss some research closely related to our work presented in this thesis. We mention some existing tools supporting formal verification with CafeOBJ as well as several most advanced interactive theorem provers. We mention some case studies on security verification of post-quantum cryptographic protocols as well as several state-of-the-art tools supporting cryptographic protocol analysis.

6.1 CafeOBJ formal verification

Providing a formal specification written in CafeOBJ or Maude, CiMPG+F [125, 124], CITP [73], and Creme [106] are three existing tools that can generate different kinds of formal proofs for formal verification of invariant properties to some extent. CiMPG+F (CafeInMaude Proof Generator & Fixer-upper) [125, 124], which is an extension of CiMPG, can infer proof scripts of invariant properties from CafeOBJ specifications. Given an invariant property, to generate its formal proof, similar to IPSG, CiMPG+F also requires human users to provide all necessary lemmas and the input argument where induction is used. The formal verification of the property is done by executing the generated proof script with the proof assistant CiMPA. CiMPG+F conducts case splitting based on sort constructors, while the case splitting used by IPSG can be regarded as based on constructors of Boolean, which are true and false. This difference results in an advantage of CiMPG+F over IPSG, that is, with some case studies, CiMPG+F may use fewer equations than IPSG to characterize a sub-case in the proof generated. To make it clearer, let us return to the MCS protocol introduced in Section 3.4 and consider an example in the following. Suppose that case splittings are used to categorize the location of process `p` in state `s` into `l1`, `l2`, `l3`, or somewhere else. Four sub-cases are generated by each tool as depicted in Figure 6.1, where sub-cases (1), (2.1), (2.2.1), and (2.2.2) of IPSG respectively correspond to sub-cases (1), (2), (3), and (4) of CiMPG+F. IPSG uses the following three equations to characterize the sub-case (2.2.1):



(a) Case splitting by IPSG



(b) Case splitting by CiMPG+F

Figure 6.1: An example of case splitting by IPSG and by CiMPG+F

```
eq (pc(s,p) = l1) = false .
eq (pc(s,p) = l2) = false .
eq pc(s,p) = l3 .
```

While CiMPG+F uses only one equation to characterize the sub-case (3):

```
eq pc(s,p) = l3 .
```

CiMPG+F is able to do so because `l1`, `l2`, and `l3` are constructors of the sort `Label`, i.e., the sort of process locations. Initially, CiMPG+F was dedicated to fixing incomplete proof scores during the proof script generation of CiMPG. That is, if CiMPG cannot generate proof scripts properly because of incomplete proof scores input, CiMPG+F will be invoked to infer the proof for the missing part. To do so, CiMPG+F uses a configurable bounded depth. For each sub-case in which CiMPG+F finds that the corresponding proof score part is missing, CiMPG+F tries to check whether the sub-case can be reduced to true by using induction hypotheses as premises or a case splitting can be used. When a case splitting is used, the procedure is recursively re-invoked provided that the total number of case splitting used so far is smaller than the bounded depth. If CiMPG+F could solve the sub-case up to the configured depth, it returns the part of the proof script for the missing proof score; otherwise, a `:postpone` command is returned, meaning that the sub-case is left to human users for resolving. IPSG produces proof scores, while CiMPG produces proof scripts from proof scores. As we already mentioned previously, a proof score is typically easier for human users to understand and/or analyze than the corresponding proof

script because, for example, we can see clearly the set of equations characterizing a sub-case in the proof score, while it is not straightforward to do the same with the proof script. Besides, CiMPG may take a quite long time to produce proof scripts when complicated specifications and properties are taken into account. For instance, as reported in Chapter 3, with the TLS 1.2 protocol, CiMPG took nearly 10 hours to complete the job.

D. Gâinâ et al. [73] have developed CITP (Constructor-based Inductive Theorem Prover), which is also implemented in Maude using its meta-level functionalities. CITP can automatically verify invariant properties if all needed lemmas are provided. Similar to IPSG and CiMPG+F, CITP is essentially based on inductive proof, however, it takes as inputs formal specifications written in Maude rather than in CafeOBJ. The difference between the two specification languages leads to some additional jobs that would be required for human users. For instance, when writing Maude specifications, human users are supposed to define the counterpart of the CafeOBJ predicate `_=_` (which is typically defined as `_=?_`). Although the predicate `_==_` is pre-defined in Maude, it cannot be used. With IPSG, it takes specifications written in CafeOBJ, and so we can straightforwardly use the built-in predicate `_=_`. To handle case splitting, CITP requires human users to annotate metadata attributes to equations that will be used for case splitting in the Maude specification. Afterward, the users are supposed to write a list of *proof tactics*, where each tactic can be, for example, applying theorem of constants, case splittings, implication, and reduction, to do theorem proof. With IPSG, case splitting is conducted automatically without annotating any metadata attributes. For the verification, we only need to declare invariants & lemmas, and indicate the argument where induction is used but need not do anything else. A case study using the Alternating Bit Protocol [18] was presented in the paper to demonstrate the practicability of CITP.

M. Nakano et al. [106] have proposed an automated invariant verification method as well as a supporting tool, namely Creme. OTSs are also used as state machines, but formal specifications need to be written in Maude. They have shown that Creme could automatically prove the NSLPK authentication protocol [98] satisfies the secrecy property without human creativity in providing auxiliary lemmas. It is not clear whether Creme is applicable to other case studies, especially complicated ones, such as the PQ TLS and PQ SSH protocols presented in this thesis. Generally speaking, systematically conjecturing correct lemmas for any formal verification problem is an issue that has no solution. IPSG although leaves the lemma conjecture task to human users, it has been demonstrated to be applicable to various systems/protocols. Because Creme takes as input Maude specification similar to CITP, users are also supposed to do some additional jobs when specifying the protocol under verification, such as to define the counterpart of the CafeOBJ predicate `_=_`.

The induction approach has been studied for security verification of cryptographic protocols by Lawrence C. Paulson [120]. In his inductive approach, the protocol under verification is

formalized as a set of *traces*. Each *trace* is a list of *events*, such as, Alice sending a message M to Bob. Desired security properties are then specified and proved by induction on traces. The verification is assisted by the interactive theorem prover Isabelle/HOL [108]. The verification process requires an amount of manual effort to guide the proof, i.e., to specify a series of *tactics* (procedures) to apply, such as to tell Isabelle/HOL to apply rules & theorems, to use lemmas, and to do case splittings. Paulson's verification approach and our approach both rely on induction, but our tool IPSG does case splitting automatically. Automating proof tactics in order to attain the tactic series and the proof of the theorem automatically has been studied well, resulting in many automated tactics being introduced and integrated into Isabelle/HOL over time. In summary, the only essential difference is that just different theorem provers and different tools are used for the two verification approaches.

In addition to the *induction-based* verification method, there exists also the *simulation-based* method for verifying state machines enjoy invariant properties. Simulation has originally been proposed and used by N. A. Lynch [99] for formal verification of distributed algorithms. The *simulation-based* verification involves two state machines: an abstract state machine and a concrete state machine. Simulation proves that an abstract state machine simulates a concrete state machine or a concrete state machine is a proper implementation of an abstract state machine. We may need to use mathematical induction to prove that an abstract state machine enjoys some invariant properties so that we can conclude that a concrete state machine enjoys the invariant properties as well from the simulation proof. We may also need to prove that the simulation preserves the invariant properties. K. Ogata and K. Futatsugi [117] have conducted a case study in which they formally proved that the Alternating Bit Protocol (ABP) [18] enjoys an invariant property by both the induction-based technique (by writing CafeOBJ proof scores) and the simulation-based technique. In the simulation-based proof, two more abstract protocols, namely BCP and SCP, were used. CafeOBJ and OTSs were used to formalize and specify all protocols as state machines. They defined a simulation from an OTS $\mathcal{S}$ to an OTS $\mathcal{S}_A$, which is more abstract than $\mathcal{S}$, as a *relation* r between the reachable states of $\mathcal{S}$ and those of $\mathcal{S}_A$ that satisfies some conditions. They then proved a theorem stating that a state predicate p is invariant wrt $\mathcal{S}$ if a state predicate p_A is invariant wrt $\mathcal{S}_A$ and p is deduced from p_A assuming the simulation relation r. The OTS formalizing ABP acts as $\mathcal{S}$, while the OTSs formalizing the abstract protocols (BCP and SCP) act as $\mathcal{S}_A$, from that the desired property of ABP was proved. They concluded that there is no significant difference between the two formal proofs (induction-based and *simulation-based*) in terms of the proof sizes.

Extending the simulation relation between two OTSs, D. D. Tran et al. [144] have defined a variant of it, a so-called *observably equivalent simulation*. They have reported two case studies in which the MCS and Anderson mutual exclusion protocols [102, 9] are formally verified that enjoying the mutual exclusion property with the simulation-based technique. They made two

modified versions of the MCS protocol called MCS2 and MCS3 such that MCS2 observably equivalently simulates MCS and MCS3 observably equivalently simulates MCS, and proved that the observably equivalent simulations preserve the mutual exclusion property. The two formal proofs of MCS obtained by the induction-based technique and the simulation-based technique are comparable in terms of the proof sizes. However, they completed the proof by the simulation-based technique earlier than the one by the induction-based technique even though they started the latter formal verification earlier. With the Anderson case study, they attempted but could not complete the proof that the protocol enjoys the mutual exclusion property by the induction-based technique, which is mostly because an unbounded series of lemmas are required. They then made an abstract version of the protocol by replacing the finite array with an infinite array, which is called A-Anderson, and proved that A-Anderson simulates Anderson and the mutual exclusion property is preserved by the simulation relation used. Subsequently, Anderson was proved to enjoy the mutual exclusion property because A-Anderson was already proved to enjoy the property by the proof score method (or induction-based method) [145]. Through the two case studies, they concluded that the simulation-based technique is helpful for them to complete the verifications of the two protocols.

Refinement, which is the reverse direction of simulation, has been extensively used in one main stream of formal methods, such as Vienna Development Method (VDM) (or VDM++) [83], B method [3] and Event-B [2], Z method [129], and Abstract State Machine (ASM) [34]. Starting with a very abstract formal specification (or requirements specification), a bit more concrete formal specification is made such that the latter refines the former or the latter is a proper implementation of the former, which is repeated until an executable program or a detailed specification that can be straightforwardly written in a programming language is obtained. This way to develop software systems is known as the *correct-by-construction* method. With this method, we can verify the final specification/implementation by verifying each individual refinement step.

There exist many proof assistants (interactive theorem provers) in the field of theorem proving, such as Isabelle/HOL [108], ACL2 [85], and Coq [24]. Through human-computer collaboration, those provers facilitate the development of formal proofs for some specific mathematical theorems. Not limited to only mathematical theorems like those theorem provers, CafeOBJ is a high-level specification language and also a verification tool that can be used for a wide variety of systems. CafeOBJ is equipped with many useful features and a powerful specification syntax for writing formal specifications and required properties of even complex systems. For example, module expressions, modules instantiated parameters using views, operators & equations for specifying system transitions, and the flexible mix-fix syntax are some superiorities of CafeOBJ. Particularly, talking about cryptographic protocols, these features allow the users to specify algebraic properties of cryptographic functions, such as Diffie-Hellman exponentiation and cancellation between symmetric public encryption & decryption.

It is worth mentioning some lemma conjecture techniques because lemma conjecture can be regarded as one of the most challenging tasks in theorem proving. When we want to prove that a state predicate p is an invariant wrt a state machine $\mathcal{S}$ by induction, there often exists a non-trivial case such that $p(\upsilon)$ holds but $p(\upsilon')$ does not, where $\upsilon \rightarrow \upsilon'$ is a transition instance. To deal with such a case, we need to find p_{str} that is stronger than p such that $p_{\text{str}}(\upsilon)$ does not hold (which implies the source υ is not reachable wrt $\mathcal{S}$) and to prove that p_{str} is an invariant wrt $\mathcal{S}$. p_{str} is typically in the form $p \wedge p'$ and p' is typically in the form $q_1 \wedge \ldots \wedge q_n$. LS tries to make each individual lemma $q_1, \ldots, q_n$ as strong as possible. Whereas LW tries to make some q_i a bit weaker such that it is still enough to make p_{str} inductive and it is possible to prove q_i. However, J. M. Rushby [131] has demonstrated that the use of disjunctive invariants $q_1 \vee \ldots \vee q_n$ makes invariant verification easier for synchronous concurrent (and/or distributed) systems. Precisely, with his technique, p_{str} is in the form $p \wedge (q_1 \vee \ldots \vee q_n)$. LW, together with LS, can be regarded as a generalized version of his technique. Instead of $p \wedge q_1 \wedge \ldots \wedge q_i \wedge \ldots \wedge q_n$, we prove that $p \wedge q_1 \wedge \ldots \wedge q'_i \wedge \ldots \wedge q_{n'}$ is an inductive invariant wrt a system, where q'_i is weaker than q_i (and n' is much less than n in our case study). q'_i may be in the form $q'_{i1} \vee \ldots \vee q'_{im}$. We have demonstrated that LW may be effective for asynchronous concurrent (and/or distributed) systems as well.

6.2 Post-quantum cryptographic protocol analysis

There are essentially two main approaches to security analysis of cryptographic protocols including symbolic analysis and computational analysis. In the survey [31], plenty of case studies and tools for analyzing cryptographic protocols in both the symbolic and computational approaches have been discussed. In the symbolic approach, talking about the class of post-quantum cryptographic protocols, to the best of our knowledge, [81] and [79] are the only two case studies on the security analysis of these protocols. The former has presented a formal analysis of the Ephemeral Diffie Hellman Over COSE (EDHOC) protocol [133], a variant of the Diffie Hellman (DH) protocol designed by the IETF's Lightweight Authenticated Key Exchange Working Group to be used on IoT devices. EDHOC offers multiple options for authentication methods and key exchange mechanisms. Among them, a so-called KEM-based version supports post-quantum security, which is achieved by replacing DH with a quantum-resistant KEM. The KEM-based version of the protocol is also covered in their analysis. An interesting point in this work is that they used SAPIC$^+$ [37] protocol verification platform so that their formal specification written in pi-calculus can be exported to some other security analyzer tools including ProVerif [33] and Tamarin [19].

WireGuard [57] is a VPN protocol focusing on simplicity, fast speed, and high performance. Facing the quantum attack threat, A. Hülsing et al. [79] have proposed its quantum-resistant

version, namely post-quantum WireGuard (PQ-WireGuard), and presented its symbolic verification of the desired security properties inherited from the WireGuard protocol. The symbolic proof used Tamarin prover [19], a well-known formal method tool for symbolic verification of cryptographic protocols. They first symbolically modeled the primitives, messages, etc. used in the protocol as function symbols and terms, and then specified the desired security properties. To do so, the security properties have been formalized as Tamarin lemmas, and some auxiliary lemmas have been introduced as well. Moreover, the authors also presented security verification in the computational model. In comparison, the symbolic proof exposes the superiority to the computational proof in two points: first, it covers more security properties, and second, it is computer-verified. On the other hand, the computational proof gives higher security assurances because it took probability and complexity into account, and fewer idealizing assumptions were made.

Formal methods with their assistant tools have been successfully used for cryptographic protocol verification. Scyther [47] is a cryptographic protocol analysis based on multiset rewriting [103]. Scyther can provide a security verification with respect to an unbounded number of sessions, but it supports only a fixed set of cryptographic primitives (symmetric & asymmetric encryptions and digital signatures) and does not allow for user-specified equational theories. Its successor, namely Tamarin [101, 19], does support equational theories. Tamarin operates based on multiset rewriting and its verification algorithm is based on constraint solving. A specification written in Tamarin is essentially a state machine where each state is an AC-collection of *facts*. Transitions between states are defined by *rules*, which specify the protocol execution, the behavior of honest parties as well as the capabilities of the Dolev-Yao intruder [56]. A security property is specified as a trace property, then Tamarin checks the satisfiability and/or the validity of the formula formalizing the property under verification. If it is the validity checking, Tamarin first converts the formula to its negated form in order to perform a satisfiability checking instead. After that, Tamarin performs an exhaustive, symbolic search based on constraint solving until either a satisfying trace is found or no more rewrite rules can be applied. However, the search is not guaranteed to be terminated for every analysis attempt, and when it is the case, the tool allows manual interaction from human users to operate it with the supplementation of some extra lemmas. There exist many case studies on security analysis of cryptographic protocols with Tamarin, such as the Authentication and Key Agreement (AKA) protocol for 5G Authentication [20] and the IEEE 802.11 WPA2 protocol [46]. Roughly speaking, *facts* and *rules* in Tamarin correspond to *observers* and *transitions*, respectively, in OTSs of the proof score verification approach. Tamarin and our verification method both require manual efforts on conjecturing lemmas. On the other hand, Tamarin's verification algorithm and the simultaneous induction proof method are technically different, and the applications of Tamarin are limited to cryptographic protocols only, while IPSG is applicable to various systems/protocols

as demonstrated in Chapter 3.

ProVerif [28, 33] is an automated tool for symbolically reasoning cryptographic protocols with the presence of a Dolev-Yao intruder [56]. A variant of the pi-calculus [30] is used to model a cryptographic protocol, and then ProVerif translates it to a set of Horn clauses. This Horn clause representation makes some abstractions, which is the cost for the support of an unbounded number of sessions. Given a security property that we want to prove, the tool reduces the problem of finding an attack against the property to the derivability of a fact on the Horn clauses representing the protocol execution. If the fact is not derivable from the clauses, the protocol is verified to enjoy the property. Otherwise, there may be an attack violating the property under analysis, but it may also be a "false attack", that is the found derivation actually does not correspond to a real attack. Using ProfVerif, a number of cryptographic protocols have been analyzed, such as LINE [134], Signal [86], and the ARINC823 avionic protocols [32]. On the one hand, similar to Tamarin, the applications of ProVerif are limited to cryptographic protocols, while IPSG supports a more wide range of systems/protocols. On the other hand, ProVerif can produce a counterexample violating the security property under analysis, which IPSG cannot. However, the counterexample found by ProVerif does not always represent a correct attack, which is a weakness of ProVerif. To show why ProVerif can output a false attack, let us consider a simple protocol example with the following two exchanged messages:

(i) $A \rightarrow B \ : \ x \leftarrow \text{senc}(k, \text{senc}(k, s))$

(ii) $B \rightarrow A \ : \ \text{sdec}(k, x)$

where senc and sdec denote the symmetric encryption and decryption, respectively. k is a shared key supposed to be known by only A and B. A choose a secret s, encrypts it with the key k twice, obtains a ciphertext x, and sends x to B. On reception of x, B decrypts it with the key k and sends the result back to A. We would like to verify the secrecy of s. The protocol execution is specified in ProVerif as the following process:

```
new k: key; out(c, senc(k,senc(k,s)));
in(c, x: bitstring); out(c, sdec(k,x))
```

The first line indicates A outputs to public channel `c` the ciphertext x. The second line indicates B receives x from channel `c` and performs the decryption. When translating to Horn clause representation, the process above has the same representation as the following process:

```
new k: key; out(c, senc(k,senc(k,s)));
!in(c, x: bitstring); out(c, sdec(k,x))
```

where the replication `!in()` is the infinite composition `in() | in() | ...`. As a result, ProVerif thinks that the result of the first decryption, i.e., $\text{senc}(k, s)$, can be sent again to the input, and then the obtained result, i.e., secret s, is outputted to the public channel c. Afterward, the intruder can easily grasp it. However, this is impossible in reality because the input can be executed only once. The invalid attack was found because ProVerif made an abstraction when

translating processes to Horn clause representations.

Maude-NPA [65] is another formal method tool for cryptographic protocol analysis based on narrowing & rewriting logic [68]. The tool is implemented in Maude [43, 62]. The Dolev-Yao intruder model [56] and the strand model [143] are used to model the intruder's capabilities. In this manner, the intruder is given the capability of fully controlling the network, for example, intercepting & modifying messages and impersonating some protocol participants to send some messages to other participants. For the analysis, Maude-NPA uses a backward narrowing reachability analysis modulo an equational theory. Narrowing [43] is a generalization of term rewriting that allows logical variables in subject terms and replaces pattern matching by unification, which provides Maude-NPA with symbolic execution capabilities. The backward narrowing reachability analysis starts from a final insecure state pattern specified by human users that represents insecure states, a so-called attack pattern, to check whether it is reachable from an initial state, which has no further backward steps. If that is the case, the protocol is insecure with the attack; otherwise, it is secure against the attack. Roughly speaking, the attack pattern and the initial state (if found) in Maude-NPA correspond to the negated formula formalizing the validity property and the satisfying trace (if found), respectively, in Tamarin. The exhaustive search has pros as it is fully automated, but it poses cons because the search would take a long time to terminate when the state space is large. Several optimization techniques to reduce the search state space have been developed [66, 65, 67], such as to generate formal grammars representing terms (states information) unreachable from initial states, but the problem remains a limitation of Maude-NPA. In contrast, running time is not a problem with our verification approach presented in this thesis since proof score execution normally takes only a short time. However, the proof is not fully automated because manual efforts are spent to construct some additional lemmas.

Security properties of cryptographic protocols can be classified into two categories: *trace properties* and *equivalence properties*. The former are properties that are defined on each execution trace of the protocol, where *secrecy* and *authentication* are the two most basic ones. The latter are properties stating that the attacker cannot distinguish two processes, so they can also be called *indistinguishability properties*. For example, considering the Helios voting protocol [4] with two honest participants A and B, and two voting options $\#1$ and $\#2$, we would like to verify the vote privacy property, which states that the attacker cannot distinguish two instances of the protocol execution: (1) A votes for option $\#1$ while B votes for option $\#2$, and (2) A votes for option $\#2$ while B votes for option $\#1$. In this thesis, such properties and protocols have not been considered, we leave it as a piece of our future work. DEEPSEC [38] is a verification tool dedicated to verifying *equivalence properties* of cryptographic protocols. The tool decides trace equivalence for cryptographic protocols that are specified in a dialect of the applied pi calculus [1]. Given two processes P_1 and P_2 representing two protocol execution instances, DEEPSEC

constructs a so-called *partition tree* to guide decision of equivalence of the two processes. Each node in this tree consists of a set of symbolic processes and constraints. From the root node, which contains P_1 and P_2 and empty constraints, the tree is constructed from top to down based on some rules. If there exists a node in the tree that does not contain both a process originating from P_1 and a process originating from P_2, then the two processes are not equivalent, or the property under verification is not satisfied. Otherwise, the property is proved. A drawback of DEEPSEC is that it supports only proof for a bounded number of protocol executions.

The proof scores approach has been used to formally verify the TLS 1.0 handshake protocol [6] by K. Ogata and K. Futatsugi [110]. L. C. Paulson [119] has also analyzed the TLS 1.0 handshake protocol with his inductive method [120] and the proof assistant Isabelle/HOL [108]. Although verification of designs and specifications of cryptographic protocols has significantly contributed to their reliability, some researchers claimed that formal verification by using some formal specification languages to specify cryptographic protocols often lacks some aspect of details. They then proposed to verify detailed protocol implementations and deployments [26, 27, 50]. Their verification method consists of selecting a part of the implementation and writing additional verification harness code that specifies the attacker model, the cryptographic assumptions, and the target security properties, and then compiling their combination to ProVerif [28]. Because the verification toolchain is automated, one can easily re-verify the code base as it evolves, like regression testing. On the one hand, this verification technique has the benefit of not having to worry about some potentially erroneous details of the protocol implementation code being missed. On the other hand, the verification may be very costly. First, the verifier may take a very long time to terminate or even may not terminate in some circumstances, especially with a large implementation. Second, a large amount of memory may be consumed.

Chapter 7

Conclusion

In this chapter, we summarise the work have been done and discuss several lines of our future work.

7.1 Concluding remarks

This thesis has presented a formal verification approach with the employment of the CafeOBJ algebraic specification language, applied to verify the requirement properties of systems including concurrent and distributed systems. Although the approach has been demonstrated to be applicable to various systems/protocols, this thesis has mainly focused on reporting two verification case studies with two post-quantum cryptographic protocols, namely PQ TLS and PQ SSH.

We have proposed an approach to automation of the proof score writing process for formal verifications of invariant properties and implemented the supporting tool IPSG. This has been motivated by the fact that writing proof scores is time- and effort-consuming, especially with complicated systems or specifications, and they are subject to human errors because they are user-defined, while CafeOBJ does not check their correctness. To demonstrate the efficiency and the practicability of the tool, experiments with various systems/protocols have been conducted, ranging from a classical key distribution protocol to authentication protocols, from a real-time system to mutual exclusion protocols, and from a distributed protocol to real cryptographic protocols currently in use. Given a CafeOBJ formal specification and an invariant property list, the tool can produce the proof scores of those properties provided that all necessary lemmas are given as well. Lemma conjecture is indeed a creativity task, which IPSG leaves to the users conducting formal verification. In this thesis, we have also proposed the Lemma Weakening technique for conjecturing lemmas. A non-trivial invariant typically cannot be proved standalone, instead, we often prove a stronger version of it, which is in the form of a conjunction of that invariant and some auxiliary lemmas. Lemma Strengthening is typically used to make each of

such lemmas generic enough, otherwise, its proof may be tough or even impossible. We found, however, that Lemma Weakening, which replaces a lemma with a weaker version of it, is an effective way to make the verification attempt of the MCS protocol converge.

PQ TLS, a quantum-resistant version of the TLS 1.2 protocol, has been proposed by the Amazon Web Services team as a precaution against the quantum attack threat. We have constructed a comprehensive symbolic model of the proposed protocol, specified it in CafeOBJ, and formally proved the claimed security properties with the employment of the tool IPSG. The CafeOBJ specifications cover both the full handshake and abbreviated handshake modes, and both cases when client authentication is requested and when it is not. The three desired security properties have been proved, including *session key secrecy*, *forward secrecy*, and *authentication*. The formal verification has been achieved under a threat model with the presence of an active attacker who can control the network. The attacker could break the classical key exchange algorithms, i.e., ECDH, by utilizing the power of large quantum computers. Moreover, the threat model has also considered the compromises of (1) symmetric handshake keys, (2) ECDHE secret keys, (3) PQ KEM secret keys, and (4) long-term private keys of honest principals.

PQ SSH has been proposed as a quantum-resistant version of the SSH Transport Layer protocol, where Amazon Web Services is also one of its authors. This thesis has formally verified that PQ SSH enjoys the three desired security properties, including *session key secrecy*, *forward secrecy*, and *session identifier uniqueness*. Similar to the PQ TLS verification case study, we first formally specified the protocol in CafeOBJ, and then used IPSG to generate the proofs of the three properties. However, in this case study, for the threat model, we have used more general transitions to model the intruder's capability of learning information and forging messages, making sure that the intruder is given the full capability of forging an arbitrary message synthesized from the information that has been learned. With another property, namely the *authentication* property, we have found a counterexample showing that the protocol does not enjoy the property, although what we found does not affect the confidentiality of session keys shared by honest participants. We then proposed to slightly revise the protocol by adding the identifiers of the client and the server into the exchange hash. After revising the CafeOBJ formal specification accordingly, we could verified that the improved protocol enjoy the *authentication* property as well as the three other properties.

Through the two case studies with PQ TLS and PQ SSH, we have once again demonstrated that IPSG is efficient and applicable to even complicated protocols. Even though the verification process is not completely automated, the use of IPSG allows us to only focus on only one task, namely to conjecture lemmas. Once a property has been successfully proved, the automation of the tool could help us to save a lot of time and effort when conducting verifications again after the protocol and/or the property are slightly changed. This is an advantage of the verification approach compared to model checking-based and its variant approaches. When conducting

model checking, each time the protocol or the properties under verification are changed even a little bit, verification should be redone from the beginning, meaning that it is time-consuming because the model checker takes time to terminate for each verification experiment. With the rapid advance in quantum computer construction over the years, large-scale quantum computers seem closely available in the near future. Cryptographers have proposed more and more post-quantum cryptographic protocols. Thus, it is indeed very useful and meaningful to apply formal method to post-quantum cryptographic protocol security verification like our work presented in this thesis. Historically, it took several decades with the involvement of many verification/-analysis approaches to construct the modern cryptographic protocols used today. Therefore, to prepare for the upcoming quantum computing era, it is crucial to start research on building post-quantum cryptographic protocols and verifying/analyzing their security now.

7.2 Future work

In addition to PQ TLS and PQ SSH, there exist some more proposals by some IETF working groups to standardize new post-quantum cryptographic protocols, among them including (1) a quantum-resistant version of the Internet Key Exchange Protocol Version 2 (IKEv2) [71], (2) an alternative quantum-resistant version of IKEv2 [136], and (3) post-quantum OpenPGP [88]. One piece of our future work is to conduct formal verification/analysis of these protocols. The IKE protocol is used to set up a security association (SA) in the Internet Protocol Security (IPsec) protocol [72]. The IPsec protocol provides secure encrypted communication between two computers over an Internet Protocol network, where the most typical application is to use to set up virtual private networks (VPNs). IPsec operates at the Network Layer, the 3rd layer of the OSI model, which is its essential difference from SSH and TLS. Dealing with the quantum attack threat, the IETF Draft [71] defined a post-quantum version of IKEv2 for protecting today's VPN traffic against future quantum computers. The IETF Draft [136] proposed an alternative post-quantum version of the protocol, addressing the problem that [71] did not provide protection for IKE SA against quantum attackers from the very beginning when an initial IKE SA was created. It was believed no sensitive information is transferred over IKE SA and so it suffices to protect only IPsec traffic. However, it is claimed in [136] that the lack of protection for an initial IKE SA might be unacceptable in some situations. Post-quantum OpenPGP [88] defined a public-key algorithm extension for the OpenPGP protocol [69], one of the standard protocols for encrypting and decrypting data. OpenPGP is widely used for email encryption, whereas, VPNs are widely used nowadays to protect the user's personal data and communications sent over public networks. Therefore, it is worth conducting security analyses of the post-quantum proposals of these protocols.

It will be better if the running performance of IPSG could be improved so that the time

taken to generate proofs of complicated case studies could be reduced. One possible way to do so is by parallelizing the tool. Recall that for each induction case, IPSG tries to reduce the term representing the target goal of that induction case. If the obtained result is a composite term different from `true` and `false`, IPSG will select a sub-term of it, say t', and split the case into two sub-cases: one when t' holds and another when it does not. For each of the two sub-cases, the same procedure is applied. Therefore, the proof of each induction case can be produced independently. It means that one possible way to parallelize the tool is to parallelize the proof generation of induction cases. Maude 3.2 is equipped with functionalities supporting concurrent computation so that we can implement that idea without difficulty. A coordinator-worker model will be used with one coordinator and multiple workers. Initially, each worker is in the idle state, and a task queue is defined where each task (job) is in charge of generating the proof for an independent induction case. An idle worker takes a task from the task queue, updating its state to different from idle to indicate it is unable to receive a new task. Upon finishing the task, it sends the result to the coordinator, resets its state to idle, and ready to receive another task from the task queue. In this way, workers can handle tasks in parallel on each core processor of a single computer so that the running performance of the tool can be improved.

In the context of cryptographic protocol verification, security properties can be classified into two categories as mentioned before: *trace properties* and *equivalence properties*. The properties considered in the PQ TLS and PQ SSH protocols, such as the *secrecy* and *authentication* properties, all are *trace properties*, which can be defined on each execution trace of the protocol under verification. On the other hand, an equivalence property (or indistinguishability property) states that the attacker cannot distinguish two processes representing two protocol execution instances. Privacy property of voting protocols, such as [4, 49, 147], is the most common property that needs to be expressed in a form of an equivalence property, but cannot be in a form of a trace property. Recall that the privacy property of the Helios voting protocol [4] with two honest participants A and B, and two voting options #1 and #2 states that the attacker cannot distinguish two instances of the protocol execution: (1) A votes for option #1 while B votes for option #2, and (2) A votes for option #2 while B votes for option #1. This kind of protocol and property has not been taken into account in this thesis. Therefore, as one line of our future work, we are also interested in conducting some formal verification case studies with equivalence properties.

References

[1] Martín Abadi, Bruno Blanchet, and Cédric Fournet. "The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication". In: *J. ACM* 65.1 (2018), 1:1–1:41. DOI: 10.1145/3127586.

[2] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010. ISBN: 978-0-521-89556-9. DOI: 10.1017/CBO9781139195881.

[3] Jean-Raymond Abrial. *The B-book - Assigning programs to meanings*. Cambridge University Press, 1996. ISBN: 978-0-521-02175-3. DOI: 10.1017/CBO9780511624162.

[4] Ben Adida. "Helios: Web-based Open-Audit Voting". In: *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*. Ed. by Paul C. van Oorschot. USENIX Association, 2008, pp. 335–348. URL: http://www.usenix.org/events/sec08/tech/full_papers/adida/adida.pdf.

[5] Erdem Alkim, Joppe W. Bos, Léo Ducas, Patrick Longa, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Chris Peikert, and Ananth Raghunathan. *FrodoKEM: Learning With Errors Key Encapsulation*. 2021. URL: https://frodokem.org/files/FrodoKEM-specification-20210604.pdf.

[6] Christopher Allen and Tim Dierks. *The TLS Protocol Version 1.0*. RFC 2246. Jan. 1999. DOI: 10.17487/RFC2246.

[7] Bowen Alpern and Fred B. Schneider. "Defining Liveness". In: *Inf. Process. Lett.* 21.4 (1985), pp. 181–185. DOI: 10.1016/0020-0190(85)90056-0.

[8] Ross J. Anderson. *Security engineering - A guide to building dependable distributed systems*. Wiley, 2001. ISBN: 978-0-471-38922-4.

[9] Thomas E. Anderson. "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors". In: *IEEE Trans. Parallel Distrib. Syst.* 1.1 (1990), pp. 6–16. DOI: 10.1109/71.80120.

[10] Nicolas Aragon, Paulo Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Shay Gueron, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, and Gilles Zémor. "BIKE: Bit Flipping Key Encapsulation - Round 3 Submission". In: 2019. URL: https://bikesuite.org/files/v4.2/BIKE_Spec.2021.09.29.1.pdf.

[11] Egidio Astesiano, Hans-Jörg Kreowski, and Bernd Krieg-Brückner, eds. *Algebraic Foundations of Systems Specification*. IFIP State-of-the-Art Reports. Springer, 1999. ISBN: 978-3-642-64151-0. DOI: 10.1007/978-3-642-59851-7.

[12] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. *CRYSTALS-Kyber: Algorithm Specifications And Supporting Documentation (version 3.02)*. 2021. URL: https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf.

[13] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. ISBN: 978-0-262-02649-9.

[14] Elaine Barker, Lily Chen, Allen Roginsky, Apostol Vassilev, and Richard Davis. *Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography*. SP 800-56A Rev. 3. 2018. URL: https://csrc.nist.gov/publications/detail/sp/800-56a/rev-3/final.

[15] Richard Barnes, Martin Thomson, Alfredo Pironti, and Adam Langley. *Secure Sockets Layer (SSL) Version 3.0*. RFC 7568. June 2015. DOI: 10.17487/RFC7568.

[16] Gilles Barthe, Franccois Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. "EasyCrypt: A Tutorial". In: *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*. Ed. by Alessandro Aldini, Javier López, and Fabio Martinelli. Vol. 8604. Lecture Notes in Computer Science. Springer, 2013, pp. 146–166. DOI: 10.1007/978-3-319-10082-1_6.

[17] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. "Computer-Aided Security Proofs for the Working Cryptographer". In: *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*. Ed. by Phillip Rogaway. Vol. 6841. Lecture Notes in Computer Science. Springer, 2011, pp. 71–90. DOI: 10.1007/978-3-642-22792-9_5.

[18] Keith A. Bartlett, Roger A. Scantlebury, and Peter T. Wilkinson. "A note on reliable full-duplex transmission over half-duplex links". In: *Commun. ACM* 12.5 (1969), pp. 260–261. DOI: 10.1145/362946.362970.

[19] David A. Basin, Cas Cremers, Jannik Dreier, and Ralf Sasse. "Symbolically analyzing security protocols using TAMARIN". In: *ACM SIGLOG News* 4.4 (2017), pp. 19–30. DOI: 10.1145/3157831.3157835.

[20] David A. Basin, Jannik Dreier, Lucca Hirschi, Sasa Radomirovic, Ralf Sasse, and Vincent Stettler. "A Formal Analysis of 5G Authentication". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang. ACM, 2018, pp. 1383–1396. DOI: 10.1145/3243734.3243846.

[21] Giampaolo Bella, Fabio Massacci, and Lawrence C. Paulson. "Verifying the SET Purchase Protocols". In: *J. Autom. Reason.* 36.1-2 (2006), pp. 5–37. DOI: 10.1007/s10817-005-9018-6.

[22] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, Chitchanok Chuengsatiansup, Tanja Lange, Adrian Marotzke, Bo-Yuan Peng, Nicola Tuveri, Christine van Vredendaal, and Bo-Yin Yang. *NTRU Prime: NIST round 3 submission*. 2020. URL: https://ntruprime.cr.yp.to/nist/ntruprime-20201007.pdf.

[23] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. ISBN: 978-3-642-05880-6. DOI: 10.1007/978-3-662-07964-5.

[24] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. ISBN: 978-3-642-05880-6. DOI: 10.1007/978-3-662-07964-5.

[25] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. "Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate". In: *2017 IEEE Symposium on Security and Privacy*. 2017, pp. 483–502. DOI: 10.1109/SP.2017.26.

[26] Karthikeyan Bhargavan, Cédric Fournet, Ricardo Corin, and Eugen Zalinescu. "Cryptographically verified implementations for TLS". In: *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*. Ed. by Peng Ning, Paul F. Syverson, and Somesh Jha. ACM, 2008, pp. 459–468. DOI: 10.1145/1455770.1455828.

[27] Karthikeyan Bhargavan, Cédric Fournet, Ricardo Corin, and Eugen Zalinescu. "Verified Cryptographic Implementations for TLS". In: *ACM Trans. Inf. Syst. Secur.* 15.1 (2012), 3:1–3:32. DOI: 10.1145/2133375.2133378.

[28] Bruno Blanchet. "Automatic Verification of Security Protocols in the Symbolic Model: The Verifier ProVerif". In: *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*. Vol. 8604. Springer, 2013, pp. 54–87. DOI: 10.1007/978-3-319-10082-1_3.

[29] Bruno Blanchet. "Mechanizing Game-Based Proofs of Security Protocols". In: *Software Safety and Security - Tools for Analysis and Verification*. Ed. by Tobias Nipkow, Orna Grumberg, and Benedikt Hauptmann. Vol. 33. NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, 2012, pp. 1–25. DOI: 10.3233/978-1-61499-028-4-1.

[30] Bruno Blanchet. "Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif". In: *Found. Trends Priv. Secur.* 1.1-2 (2016), pp. 1–135. DOI: 10.1561/3300000004.

[31] Bruno Blanchet. "Security Protocol Verification: Symbolic and Computational Models". In: *Principles of Security and Trust - First International Conference, POST 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012, Proceedings*. Ed. by Pierpaolo Degano and Joshua D. Guttman. Vol. 7215. Lecture Notes in Computer Science. Springer, 2012, pp. 3–29. DOI: 10.1007/978-3-642-28641-4_2.

[32] Bruno Blanchet. "Symbolic and Computational Mechanized Verification of the ARINC823 Avionic Protocols". In: *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*. IEEE Computer Society, 2017, pp. 68–82. DOI: 10.1109/CSF.2017.7.

[33] Bruno Blanchet, Vincent Cheval, and Véronique Cortier. "ProVerif with Lemmas, Induction, Fast Subsumption, and Much More". In: *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2022, pp. 69–86. DOI: 10.1109/SP46214.2022.9833653.

[34] Egon Börger. "The ASM Refinement Method". In: *Formal Aspects Comput.* 15.2-3 (2003), pp. 237–257. DOI: 10.1007/s00165-003-0012-7.

[35] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. "CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM". In: *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*. IEEE, 2018, pp. 353–367. DOI: 10.1109/EuroSP.2018.00032.

[36] Matt Campagna and Eric Crockett. *Hybrid Post-Quantum Key Encapsulation Methods (PQ KEM) for Transport Layer Security 1.2 (TLS)*. Internet-Draft. Work in Progress. Internet Engineering Task Force, Sept. 2021. 17 pp. URL: https://datatracker.ietf.org/doc/html/draft-campagna-tls-bike-sike-hybrid-07.

[37] Vincent Cheval, Charlie Jacomme, Steve Kremer, and Robert Künnemann. "SAPIC+: protocol verifiers of the world, unite!" In: *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. Ed. by Kevin R. B. Butler and Kurt Thomas. USENIX Association, 2022, pp. 3935–3952. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/cheval.

[38] Vincent Cheval, Steve Kremer, and Itsaka Rakotonirina. "The DEEPSEC Prover". In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*. Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10982. Lecture Notes in Computer Science. Springer, 2018, pp. 28–36. DOI: 10.1007/978-3-319-96142-2_4.

[39] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-guided abstraction refinement for symbolic model checking". In: *J. ACM* 50.5 (2003), pp. 752–794. DOI: 10.1145/876638.876643.

[40] Edmund M. Clarke, Orna Grumberg, and David E. Long. "Model Checking and Abstraction". In: *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*. Ed. by Ravi Sethi. ACM Press, 1992, pp. 342–354. DOI: 10.1145/143165.143235.

[41] Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron A. Peled. "State Space Reduction Using Partial Order Techniques". In: *Int. J. Softw. Tools Technol. Transf.* 2.3 (1999), pp. 279–287. DOI: 10.1007/s100090050035.

[42] Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. "Model Checking and the State Explosion Problem". In: *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*. Ed. by Bertrand Meyer and Martin Nordio. Vol. 7682. Lecture Notes in Computer Science. Springer, 2011, pp. 1–30. DOI: 10.1007/978-3-642-35746-6_1.

[43] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, eds. *All About Maude*. Vol. 4350. Lecture Notes in Computer Science. Springer, 2007. DOI: 10.1007/978-3-540-71999-1.

[44] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. "A Comprehensive Symbolic Analysis of TLS 1.3". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 1773–1788. DOI: 10.1145/3133956.3134063.

[45] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. "Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication". In: *2016 IEEE Symposium on Security and Privacy*. 2016, pp. 470–485. DOI: 10.1109/SP.2016.35.

[46] Cas Cremers, Benjamin Kiesl, and Niklas Medinger. "A Formal Analysis of IEEE 802.11's WPA2: Countering the Kracks Caused by Cracking the Counters". In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, 2020, pp. 1–17. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/cremers.

[47] Cas J. F. Cremers. "The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols". In: *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*. Ed. by Aarti Gupta and Sharad Malik. Vol. 5123. Lecture Notes in Computer Science. Springer, 2008, pp. 414–418. DOI: 10.1007/978-3-540-70545-1_38.

[48] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. "Saber: Module-LWR Based Key Exchange, CPA-Secure Encryption and CCA-Secure KEM". In: *Progress in Cryptology - AFRICACRYPT 2018 - 10th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 7-9, 2018, Proceedings*. Ed. by Antoine Joux, Abderrahmane Nitaj, and Tajjeeddine Rachidi. Vol. 10831. Lecture Notes in Computer Science. Springer, 2018, pp. 282–305. DOI: 10.1007/978-3-319-89339-6_16.

[49] Stéphanie Delaune, Steve Kremer, and Mark Ryan. "Verifying privacy-type properties of electronic voting protocols". In: *J. Comput. Secur.* 17.4 (2009), pp. 435–487. DOI: 10.3233/JCS-2009-0340.

[50] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella Béguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. "Implementing and Proving the TLS 1.3 Record Layer". In: *2017 IEEE Symposium on Security and Privacy*. 2017, pp. 463–482. DOI: 10.1109/SP.2017.58.

[51] Dorothy E. Denning and Giovanni Maria Sacco. "Timestamps in Key Distribution Protocols". In: *Commun. ACM* 24.8 (1981), pp. 533–536. DOI: 10.1145/358722.358740.

[52] Razvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report - The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. Vol. 6. AMAST Series in Computing. World Scientific, 1998. ISBN: 978-981-02-3513-0. DOI: 10.1142/3831.

[53] Whitfield Diffie and Martin E. Hellman. "New directions in cryptography". In: *IEEE Trans. Inf. Theory* 22.6 (1976), pp. 644–654. DOI: 10.1109/TIT.1976.1055638.

[54] Edsger W. Dijkstra. "Notes on structured programming". In: Apr. 1970. URL: http://www.informatik.uni-bremen.de/agbkb/lehre/programmiersprachen/artikel/EWD-notes-structured.pdf.

[55] Jintai Ding and Dieter Schmidt. "Rainbow, a New Multivariable Polynomial Signature Scheme". In: *Applied Cryptography and Network Security, Third International Conference, Proceedings*. Vol. 3531. Lecture Notes in Computer Science. 2005, pp. 164–175. DOI: 10.1007/11496137_12.

[56] Danny Dolev and Andrew Chi-Chih Yao. "On the security of public key protocols". In: *IEEE Trans. Inf. Theory* 29.2 (1983), pp. 198–207. DOI: 10.1109/TIT.1983.1056650.

[57] Jason A. Donenfeld. "WireGuard: Next Generation Kernel Network Tunnel". In: *24th Annual Network and Distributed System Security Symposium, NDSS 2017*. https://www.wireguard.com/papers/wireguard.pdf. 2017.

[58] Ling Dong and Kefei Chen. "Cryptographic protocol: security analysis based on trusted freshness". In: 2012.

[59] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. "A Cryptographic Analysis of the TLS 1.3 Handshake Protocol". In: *J. Cryptol.* 34.4 (2021), p. 37. DOI: 10.1007/s00145-021-09384-1.

[60] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. "A Cryptographic Analysis of the TLS 1.3 Handshake Protocol Candidates". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 2015, pp. 1197–1210. DOI: 10.1145/2810103.2813653.

[61] Léo Ducas, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. "CRYSTALS-Dilithium: Digital Signatures from Module Lattices". In: *IACR Cryptol. ePrint Arch.* (2017), p. 633. URL: http://eprint.iacr.org/2017/633.

[62] Francisco Durán, Steven Eker, Santiago Escobar, Narciso Martí-Oliet, José Meseguer, Rubén Rubio, and Carolyn L. Talcott. "Programming and symbolic computation in Maude". In: *J. Log. Algebraic Methods Program.* 110 (2020). DOI: 10.1016/j.jlamp.2019.100497.

[63] U.S. Department of Energy and Canada Department of Natural Resources. *Final Report on the August 14, 2003 Blackout in the United States and Canada: Causes and Recommendations*. Tech. rep. U.S.-Canada Power System Outage Task Force, 2004. URL: https://www.energy.gov/sites/prod/files/oeprod/DocumentsandMedia/BlackoutFinal-Web.pdf.

[64] Santiago Escobar, Catherine A. Meadows, and José Meseguer. "A rewriting-based inference system for the NRL Protocol Analyzer and its meta-logical properties". In: *Theor. Comput. Sci.* 367.1-2 (2006), pp. 162–202. DOI: 10.1016/j.tcs.2006.08.035.

[65] Santiago Escobar, Catherine A. Meadows, and José Meseguer. "Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties". In: *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*. Ed. by Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri. Vol. 5705. Lecture Notes in Computer Science. Springer, 2007, pp. 1–50. DOI: 10.1007/978-3-642-03829-7_1.

[66] Santiago Escobar, Catherine A. Meadows, and José Meseguer. "State Space Reduction in the Maude-NRL Protocol Analyzer". In: *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*. Vol. 5283. Lecture Notes in Computer Science. Springer, 2008, pp. 548–562. DOI: 10.1007/978-3-540-88313-5_35.

[67] Santiago Escobar, Catherine A. Meadows, José Meseguer, and Sonia Santiago. "State space reduction in the Maude-NRL Protocol Analyzer". In: *Inf. Comput.* 238 (2014), pp. 157–186. DOI: 10.1016/j.ic.2014.07.007.

[68] Santiago Escobar, José Meseguer, and Prasanna Thati. "Narrowing and Rewriting Logic: from Foundations to Applications". In: *Proceedings of the 15th Workshop on Functional and (Constraint) Logic Programming, WFLP 2006, Madrid, Spain, November 16-17, 2006*. Ed. by Francisco Javier López-Fraguas. Vol. 177. Electronic Notes in Theoretical Computer Science. Elsevier, 2006, pp. 5–33. DOI: 10.1016/j.entcs.2007.01.004.

[69] Hal Finney, Lutz Donnerhacke, Jon Callas, Rodney L. Thayer, and David Shaw. *OpenPGP Message Format*. RFC 4880. Nov. 2007. DOI: 10.17487/RFC4880.

[70] Marc Fischlin, Felix Günther, Benedikt Schmidt, and Bogdan Warinschi. "Key Confirmation in Key Exchange: A Formal Treatment and Implications for TLS 1.3". In: *2016 IEEE Symposium on Security and Privacy*. 2016, pp. 452–469. DOI: 10.1109/SP.2016.34.

[71] Scott Fluhrer, Panos Kampanakis, David McGrew, and Valery Smyslov. *Mixing Preshared Keys in the Internet Key Exchange Protocol Version 2 (IKEv2) for Post-quantum Security*. RFC 8784. June 2020. DOI: 10.17487/RFC8784.

[72] Sheila Frankel and Suresh Krishnan. *IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap*. RFC 6071. Feb. 2011. DOI: 10.17487/RFC6071.

[73] Daniel Gâinâ, Ionut Tutu, and Adrián Riesco. "Specification and Verification of Invariant Properties of Transition Systems". In: *APSEC 2018*. 2018, pp. 99–108. DOI: 10.1109/APSEC.2018.00024.

[74] Taher El Gamal. "A public key cryptosystem and a signature scheme based on discrete logarithms". In: *IEEE Trans. Inf. Theory* 31.4 (1985), pp. 469–472. DOI: 10.1109/TIT.1985.1057074.

[75] Shafi Goldwasser and Silvio Micali. "Probabilistic Encryption". In: *J. Comput. Syst. Sci.* 28.2 (1984), pp. 270–299. DOI: 10.1016/0022-0000(84)90070-9.

[76] Lov K. Grover. "A Fast Quantum Mechanical Algorithm for Database Search". In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*. Ed. by Gary L. Miller. ACM, 1996, pp. 212–219. DOI: 10.1145/237814.237866.

[77] John Harrison, Josef Urban, and Freek Wiedijk. "History of Interactive Theorem Proving". In: *Computational Logic*. Ed. by Jörg H. Siekmann. Vol. 9. Handbook of the History of Logic. Elsevier, 2014, pp. 135–214. DOI: 10.1016/B978-0-444-51624-4.50004-6.

[78] Lars Helge Haß and Thomas Noll. "Equational Abstractions for Reducing the State Space of Rewrite Theories". In: *Proceedings of the Seventh International Workshop on Rewriting Logic and its Applications, WRLA 2008, Budapest, Hungary, March 29-30, 2008*. Ed. by Grigore Rosu. Vol. 238. Electronic Notes in Theoretical Computer Science 3. Elsevier, 2008, pp. 139–154. DOI: 10.1016/j.entcs.2009.05.017.

[79] Andreas Hülsing, Kai-Chun Ning, Peter Schwabe, Florian Weber, and Philip R. Zimmermann. "Post-quantum WireGuard". In: *2021 IEEE Symposium on Security and Privacy*. 2021, pp. 304–321. DOI: 10.1109/SP40001.2021.00030.

[80] Andreas Hülsing, Joost Rijneveld, John M. Schanck, and Peter Schwabe. "High-Speed Key Encapsulation from NTRU". In: *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 232–252. DOI: 10.1007/978-3-319-66787-4_12.

[81] Charlie Jacomme, Elise Klein, Steve Kremer, and Maiwenn Racouchot. "A comprehensive, formal and automated analysis of the EDHOC protocol". In: *32nd USENIX Security Symposium, USENIX Security 2023, to appear*. 2023.

[82] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. "On the Security of TLS-DHE in the Standard Model". In: *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference*. Vol. 7417. Lecture Notes in Computer Science. Springer, 2012, pp. 273–293. DOI: 10.1007/978-3-642-32009-5_17.

[83] Clifford B. Jones. *Systematic software development using VDM (2. ed.)* Prentice Hall International Series in Computer Science. Prentice Hall, 1991. ISBN: 978-0-13-880733-7.

[84] Panos Kampanakis, Douglas Stebila, and Torben Hansen. *Post-quantum Hybrid Key Exchange in SSH*. Internet-Draft draft-kampanakis-curdle-ssh-pq-ke-01. Work in Progress. Internet Engineering Task Force, Apr. 2023. 15 pp. URL: https://datatracker.ietf.org/doc/draft-kampanakis-curdle-ssh-pq-ke/01/.

[85] Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach*. USA: Kluwer Academic Publishers, 2000. ISBN: 0792377443. DOI: 10.1007/978-1-4615-4449-4.

[86] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. "Automated Verification for Secure Messaging Protocols and Their Implementations: A Symbolic and Computational Approach". In: *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*. IEEE, 2017, pp. 435–450. DOI: 10.1109/EuroSP.2017.38.

[87] Weiqiang Kong, Kazuhiro Ogata, and Kokichi Futatsugi. "Algebraic Approaches to Formal Analysis of the Mondex Electronic Purse System". In: *IFM 2007*. Vol. 4591. 2007, pp. 393–412. DOI: 10.1007/978-3-540-73210-5_21.

[88] Stavros Kousidis, Falko Strenzke, and Aron Wussler. *Post-Quantum Cryptography in OpenPGP*. Internet-Draft draft-wussler-openpgp-pqc-01. Work in Progress. Internet Engineering Task Force, Mar. 2023. 39 pp. URL: https://datatracker.ietf.org/doc/draft-wussler-openpgp-pqc/01/.

[89] Laura Kovács and Andrei Voronkov. "First-Order Theorem Proving and Vampire". In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 1–35. DOI: 10.1007/978-3-642-39799-8_1.

[90] Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. "On the Security of the TLS Protocol: A Systematic Analysis". In: *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference*. Vol. 8042. Lecture Notes in Computer Science. Springer, 2013, pp. 429–448. DOI: 10.1007/978-3-642-40041-4_24.

[91] Leslie Lamport. "Proving the Correctness of Multiprocess Programs". In: *IEEE Trans. Software Eng.* 3.2 (1977), pp. 125–143. DOI: 10.1109/TSE.1977.229904.

[92] Jacques-Louis Lions. *Ariane 5 Flight 501 Failure*. Technical Report. European Space Agency, 1996. URL: https://www-users.cse.umn.edu/~arnold/disasters/ariane5rep.html.

[93] Chris M. Lonvick and Tatu Ylonen. *The Secure Shell (SSH) Authentication Protocol*. RFC 4252. Jan. 2006. DOI: 10.17487/RFC4252.

[94] Chris M. Lonvick and Tatu Ylonen. *The Secure Shell (SSH) Connection Protocol*. RFC 4254. Jan. 2006. DOI: 10.17487/RFC4254.

[95] Chris M. Lonvick and Tatu Ylonen. *The Secure Shell (SSH) Protocol Architecture*. RFC 4251. Jan. 2006. DOI: 10.17487/RFC4251.

[96] Chris M. Lonvick and Tatu Ylonen. *The Secure Shell (SSH) Transport Layer Protocol*. RFC 4253. Jan. 2006. DOI: 10.17487/RFC4253.

[97] Gavin Lowe. "A Hierarchy of Authentication Specification". In: *10th Computer Security Foundations Workshop (CSFW '97), June 10-12, 1997, Rockport, Massachusetts, USA*. IEEE Computer Society, 1997, pp. 31–44. DOI: 10.1109/CSFW.1997.596782.

[98] Gavin Lowe. "An Attack on the Needham-Schroeder Public-Key Authentication Protocol". In: *Inf. Process. Lett.* 56.3 (1995), pp. 131–133. DOI: 10.1016/0020-0190(95)00144-2.

[99] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996. ISBN: 1-55860-348-4.

[100] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems - safety*. Springer, 1995. ISBN: 978-0-387-94459-3. DOI: 10.1007/978-1-4612-4222-2.

[101] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. "The TAMARIN Prover for the Symbolic Analysis of Security Protocols". In: *Computer Aided Verification - 25th International Conference, CAV 2013*. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 696–701. DOI: 10.1007/978-3-642-39799-8_48.

[102] John M. Mellor-Crummey and Michael L. Scott. "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors". In: *ACM Trans. Comput. Syst.* 9.1 (Feb. 1991), pp. 21–65. DOI: 10.1145/103727.103729.

[103] John C. Mitchell. "Multiset Rewriting and Security Protocol Analysis". In: *Rewriting Techniques and Applications, 13th International Conference, RTA 2002, Copenhagen, Denmark, July 22-24, 2002, Proceedings*. Ed. by Sophie Tison. Vol. 2378. Lecture Notes in Computer Science. Springer, 2002, pp. 19–22. DOI: 10.1007/3-540-45610-4_2.

[104] Bodo Moller, Thai Duong, and Krzysztof Kotowicz. *This POODLE Bites: Exploiting The SSL 3.0 Fallback*. Tech. rep. Google Security Team, 2014. URL: https://www.openssl.org/~bodo/ssl-poodle.pdf.

[105] Ataru T. Nakagawa, Toshimi Sawada, Kokichi Futatsugi, and Norbert Preining. *CafeOBJ User's Manual*. 2016. URL: https://cafeobj.org/files/manual.pdf.

[106] Masahiro Nakano, Kazuhiro Ogata, Masaki Nakamura, and Kokichi Futatsugi. "CrÈme: an Automatic Invariant Prover of Behavioral Specifications". In: *Int. J. Softw. Eng. Knowl. Eng.* 17.6 (2007), pp. 783–804. DOI: 10.1142/S0218194007003458.

[107] Roger M. Needham and Michael D. Schroeder. "Using Encryption for Authentication in Large Networks of Computers". In: *Commun. ACM* 21.12 (1978), pp. 993–999. DOI: 10.1145/359657.359659.

[108] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer, 2002. ISBN: 3-540-43376-7. DOI: 10.1007/3-540-45949-9.

[109] Kazuhiro Ogata and Kokichi Futatsugi. "Compositionally Writing Proof Scores of Invariants in the OTS/CafeOBJ Method". In: *J. Univers. Comput. Sci.* 19.6 (2013), pp. 771–804. DOI: 10.3217/jucs-019-06-0771.

[110] Kazuhiro Ogata and Kokichi Futatsugi. "Equational Approach to Formal Analysis of TLS". In: *25th International Conference on Distributed Computing Systems (ICDCS) 2005, 6-10 June 2005, Columbus, OH, USA*. IEEE Computer Society, 2005, pp. 795–804. DOI: 10.1109/ICDCS.2005.32.

[111] Kazuhiro Ogata and Kokichi Futatsugi. "Flaw and modification of the *i*KP electronic payment protocols". In: *Inf. Process. Lett.* 86.2 (2003), pp. 57–62. DOI: 10.1016/S0020-0190(02)00480-5.

[112] Kazuhiro Ogata and Kokichi Futatsugi. "Formal Analysis of the *i*KP Electronic Payment Protocols". In: *Software Security – Theories and Systems, Mext-NSF-JSPS International Symposium, ISSS 2002*. Ed. by Mitsuhiro Okada, Benjamin C. Pierce, Andre Scedrov, Hideyuki Tokuda, and Akinori Yonezawa. Vol. 2609. Lecture Notes in Computer Science. Springer, 2002, pp. 441–460. DOI: 10.1007/3-540-36532-X_25.

[113] Kazuhiro Ogata and Kokichi Futatsugi. "Formal Analysis of the *i*KP Electronic Payment Protocols". In: *Software Security – Theories and Systems, Mext-NSF-JSPS International Symposium, ISSS 2002, Tokyo, Japan, November 8-10, 2002, Revised Papers*. Ed. by Mitsuhiro Okada, Benjamin C. Pierce, Andre Scedrov, Hideyuki Tokuda, and Akinori Yonezawa. Vol. 2609. Lecture Notes in Computer Science. Springer, 2002, pp. 441–460. DOI: 10.1007/3-540-36532-X_25.

[114] Kazuhiro Ogata and Kokichi Futatsugi. "Modeling and verification of real-time systems based on equations". In: *Sci. Comput. Program.* 66.2 (2007), pp. 162–180. DOI: 10.1016/j.scico.2006.10.011.

[115] Kazuhiro Ogata and Kokichi Futatsugi. "Proof Score Approach to Analysis of Electronic Commerce Protocols". In: *Int. J. Softw. Eng. Knowl. Eng.* 20.2 (2010), pp. 253–287. DOI: 10.1142/S0218194010004712.

[116] Kazuhiro Ogata and Kokichi Futatsugi. "Proof Scores in the OTS/CafeOBJ Method". In: *Formal Methods for Open Object-Based Distributed Systems, 6th IFIP WG 6.1 International Conference, FMOODS 2003, Paris, France, November 19.21, 2003, Proceedings*. Vol. 2884. Lecture Notes in Computer Science. Springer, 2003, pp. 170–184. DOI: 10.1007/978-3-540-39958-2_12.

[117] Kazuhiro Ogata and Kokichi Futatsugi. "Simulation-based Verification for Invariant Properties in the OTS/CafeOBJ Method". In: *Proceedings of the BCS-FACS Refinement Workshop, REFINE@IFM 2007, Oxford, UK, July 2007*. Ed. by Eerke A. Boiten, John Derrick, and Graeme Smith. Vol. 201. Electronic Notes in Theoretical Computer Science. Elsevier, 2008, pp. 127–154. DOI: 10.1016/j.entcs.2008.02.018.

[118] Kazuhiro Ogata and Kokichi Futatsugi. "Some Tips on Writing Proof Scores in the OTS/CafeOBJ Method". In: *Algebra, Meaning, and Computation*. Ed. by Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer. Vol. 4060. Lecture Notes in Computer Science. Springer, 2006, pp. 596–615. DOI: 10.1007/11780274_31.

[119] Lawrence C. Paulson. "Inductive Analysis of the Internet Protocol TLS". In: *ACM Trans. Inf. Syst. Secur.* 2.3 (1999), pp. 332–351. DOI: 10.1145/322510.322530.

[120] Lawrence C. Paulson. "The Inductive Approach to Verifying Cryptographic Protocols". In: *J. Comput. Secur.* 6.1-2 (1998), pp. 85–128. DOI: 10.3233/jcs-1998-61-205.

[121] Tim Polk and Sean Turner. *Secure Sockets Layer (SSL) Version 2.0*. RFC 6176. Mar. 2011. DOI: 10.17487/RFC6176.

[122] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446.

[123] Eric Rescorla and Tim Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. Aug. 2008. DOI: 10.17487/RFC5246.

[124] Adrián Riesco and Kazuhiro Ogata. "An integrated tool set for verifying CafeOBJ specifications". In: *J. Syst. Softw.* 189 (2022), p. 111302. DOI: 10.1016/j.jss.2022.111302.

[125] Adrián Riesco and Kazuhiro Ogata. "CiMPG+F: A Proof Generator and Fixer-Upper for CafeOBJ Specifications". In: *Theoretical Aspects of Computing - ICTAC 2020 - 17th International Colloquium, Macau, China, November 30 - December 4, 2020, Proceedings*. Ed. by Violet Ka I Pun, Volker Stolz, and Adenilso Simão. Vol. 12545. Lecture Notes in Computer Science. Springer, 2020, pp. 64–82. DOI: 10.1007/978-3-030-64276-1_4.

[126] Adrián Riesco and Kazuhiro Ogata. "Prove it! Inferring Formal Proof Scripts from CafeOBJ Proof Scores". In: *ACM Trans. Softw. Eng. Methodol.* 27.2 (2018), 6:1–6:32. DOI: 10.1145/3208951.

[127] Adrián Riesco, Kazuhiro Ogata, and Kokichi Futatsugi. "A Maude environment for CafeOBJ". In: *Formal Asp. Comput.* 29.2 (2017), pp. 309–334. DOI: 10.1007/s00165-016-0398-7.

[128] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In: *Commun. ACM* 21.2 (1978), pp. 120–126. DOI: 10.1145/359340.359342.

[129] Julian Rose. "Z - An introduction to formal methods (2nd edn) : Antoni Diller John Wiley, Chichester (1994)". In: *Inf. Softw. Technol.* 37.9 (1995), pp. 521–523. DOI: 10.1016/0950-5849(95)90016-0.

[130] John M. Rushby. "Tutorial: Automated Formal Methods with PVS, SAL, and Yices". In: *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), 11-15 September 2006, Pune, India*. IEEE Computer Society, 2006, p. 262. DOI: 10.1109/SEFM.2006.37.

[131] John M. Rushby. "Verification Diagrams Revisited: Disjunctive Invariants for Easy Verification". In: *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*. Ed. by E. Allen Emerson and A. Prasad Sistla. Vol. 1855. Lecture Notes in Computer Science. Springer, 2000, pp. 508–520. DOI: 10.1007/10722167_38.

[132] Stephan Schulz. "System Description: E 1.8". In: *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*. Ed. by Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov. Vol. 8312. Lecture Notes in Computer Science. Springer, 2013, pp. 735–743. DOI: 10.1007/978-3-642-45221-5_49.

[133] Göran Selander, John Preuß Mattsson, and Francesca Palombini. *Ephemeral Diffie-Hellman Over COSE (EDHOC)*. Internet-Draft draft-ietf-lake-edhoc-17. Work in Progress. Internet Engineering Task Force, Oct. 2022. 95 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-lake-edhoc/17/.

[134] Cheng Shi and Kazuki Yoneyama. "Verification of LINE Encryption Version 1.0 Using ProVerif". In: *IEICE TRANSACTIONS on Information and Systems*. Vol. E102-D. 2019, pp. 1439–1448. DOI: 10.1587/transinf.2018FOP0001.

[135] Peter W. Shor. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring". In: *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*. IEEE Computer Society, 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700.

[136] Valery Smyslov. *Alternative Approach for Mixing Preshared Keys in IKEv2 for Post-quantum Security*. Internet-Draft draft-smyslov-ipsecme-ikev2-qr-alt-07. Work in Progress. Internet Engineering Task Force, Apr. 2023. 9 pp. URL: https://datatracker.ietf.org/doc/draft-smyslov-ipsecme-ikev2-qr-alt/07/.

[137] SSL-Labs. *SSL/TLS protocol support statistic*. SSL-Labs. Dec. 2022. URL: https://www.ssllabs.com/ssl-pulse/.

[138] National Institute of Standards and Technology. *Post-Quantum Cryptography*. NIST. 2017. URL: https://csrc.nist.gov/projects/post-quantum-cryptography.

[139] National Institute of Standards and Technology. *Post-Quantum Cryptography: Round 3 Submissions*. NIST. 2020. URL: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[140] Douglas Stebila and Jonathan Green. *Elliptic Curve Algorithm Integration in the Secure Shell Transport Layer*. RFC 5656. Dec. 2009. DOI: 10.17487/RFC5656.

[141] Ichiro Suzuki and Tadao Kasami. "A Distributed Mutual Exclusion Algorithm". In: *ACM Trans. Comput. Syst.* 3.4 (1985), pp. 344–349. DOI: 10.1145/6110.214406.

[142] Carst Tankink and Pim Vullers. *Verification of the TLS Handshake protocol*. Technical Report. https://www.academia.edu/785834/Verification_of_the_TLS_Handshake_protocol. May 2008.

[143] F. Javier Thayer, Jonathan C. Herzog, and Joshua D. Guttman. "Strand Spaces: Why is a Security Protocol Correct?" In: *Security and Privacy - 1998 IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 3-6, 1998, Proceedings*. IEEE Computer Society, 1998, pp. 160–171. DOI: 10.1109/SECPRI.1998.674832.

[144] Duong Dinh Tran, Dang Duy Bui, and Kazuhiro Ogata. "Simulation-Based Invariant Verification Technique for the OTS/CafeOBJ Method". In: *IEEE Access* 9 (2021), pp. 93847–93870. DOI: 10.1109/ACCESS.2021.3093211.

[145] Duong Dinh Tran and Kazuhiro Ogata. "Formal verification of an abstract version of Anderson protocol with CafeOBJ, CiMPA and CiMPG". In: *SEKE 2020*. 2020, pp. 287–292. DOI: 10.18293/SEKE2020-064.

[146] Nikolaos Triantafyllou, Iakovos Ouranos, Petros S. Stefaneas, and Panayiotis Frangos. "Formal Specification and Verification of the OMA License Choice Algorithm in the OTS/CafeOBJ Method". In: *WINSYS 2010*. 2010, pp. 173–180.

[147] Jingzhong Wang, Yue Zhang, and Haibin Li. "Electronic voting protocol based on ring signature and secure multi-party computing". In: *International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, CyberC 2020, Chongqing, China, October 29-30, 2020*. IEEE, 2020, pp. 50–55. DOI: 10.1109/CyberC49757.2020.00018.

[148] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. "SPASS Version 3.5". In: *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*. Ed. by Renate A. Schmidt. Vol. 5663. Lecture Notes in Computer Science. Springer, 2009, pp. 140–145. DOI: 10.1007/978-3-642-02959-2_10.

[149] Thomas Y. C. Woo and Simon S. Lam. "A semantic model for authentication protocols". In: *1993 IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, USA, May 24-26, 1993*. IEEE Computer Society, 1993, pp. 178–194. DOI: 10.1109/RISP.1993.287633.

[150] Andrew Chi-Chih Yao. "Theory and Applications of Trapdoor Functions (Extended Abstract)". In: *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*. IEEE Computer Society, 1982, pp. 80–91. DOI: 10.1109/SFCS.1982.45.

Publications

Journal Articles

[J1] Duong Dinh Tran and Kazuhiro Ogata. "Formal verification of TLS 1.2 by automatically generating proof scores". In: *Computers & Security* 123 (2022), p. 102909. DOI: 10.1016/j.cose.2022.102909.

[J2] Duong Dinh Tran, Dang Duy Bui, and Kazuhiro Ogata. "Simulation-Based Invariant Verification Technique for the OTS/CafeOBJ Method". In: *IEEE Access* 9 (2021), pp. 93847–93870. DOI: 10.1109/ACCESS.2021.3093211.

[J3] Duong Dinh Tran, Thet Wai Mon, and Kazuhiro Ogata. "Transport Layer Security 1.0 handshake protocol formal verification case study: How to use a proof script generator for existing large proof scores". In: *PeerJ Computer Science* 9 (2023), e1284. DOI: 10.7717/peerj-cs.1284.

Conference papers

[C1] Duong Dinh Tran and Kazuhiro Ogata. "IPSG: Invariant Proof Score Generator". In: *46th IEEE Annual Computers, Software, and Applications Conferenc, COMPSAC 2022, Los Alamitos, CA, USA, June 27 - July 1, 2022*. IEEE, 2022, pp. 1050–1055. DOI: 10.1109/COMPSAC54236.2022.00164.

[C2] Duong Dinh Tran, Kazuhiro Ogata, Santiago Escobar, Sedat Akleylek, and Ayoub Otmani. "Formal specification and model checking of Saber lattice-based key encapsulation mechanism in Maude". In: *The 34th International Conference on Software Engineering and Knowledge Engineering, SEKE 2022, USA, July 1 - 10, 2022*. 2022, pp. 382–387. DOI: 10.18293/SEKE2022-097.

[C3] Duong Dinh Tran, Kentaro Waki, and Kazuhiro Ogata. "Formal specification and model checking of a recoverable wait-free version of MCS". In: *The 33rd International Conference on Software Engineering and Knowledge Engineering, SEKE 2021, USA, July 1 - 10, 2021*. 2021, pp. 138–143. DOI: 10.18293/SEKE2021-065.

[C4] Duong Dinh Tran, Dang Duy Bui, Parth Gupta, and Kazuhiro Ogata. "Lemma Weakening for State Machine Invariant Proofs". In: *27th Asia-Pacific Software Engineering Conference, APSEC 2020, Singapore, December 1-4, 2020*. IEEE, 2020, pp. 21–30. DOI: 10.1109/APSEC51365.2020.00010.

[C5] Duong Dinh Tran and Kazuhiro Ogata. "Formal verification of an abstract version of Anderson protocol with CafeOBJ, CiMPA and CiMPG". In: *The 32nd International Conference on Software Engineering and Knowledge Engineering, SEKE 2020, USA, July 9 - 19, 2020*. 2020, pp. 287–292. DOI: 10.18293/SEKE2020-064.

[C6] Minxuan Liu, Dang Duy Bui, Duong Dinh Tran, and Kazuhiro Ogata. "Formal Specification and Model Checking of an Autonomous Vehicle Merging Protocol". In: *21st IEEE International Conference on Software Quality, Reliability and Security, QRS 2021 - Companion, Hainan, China, December 6-10, 2021*. IEEE, 2021, pp. 333–342. DOI: 10.1109/QRS-C55045.2021.00057.

[C7] Dang Duy Bui, Duong Dinh Tran, Kazuhiro Ogata, and Adrián Riesco. "Integration of SMGA and Maude to Facilitate Characteristic Conjecture". In: *The 28th International DMS Conference on Visualization and Visual Languages, USA, June 29-30, 2022*. Ed. by Shi-Kuo Chang. 2022, pp. 45–54. DOI: 10.18293/DMSVIVA22-006.

Appendix

A PQ SSH: `auth` counterexample

In the PQ SSH case study, recall that `auth` formalizes the *authentication* property. A counterexample of `auth` has been found, where the transition sequence leading to the counterexample is as follows:

```
open INV .
  ops s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 : -> Sys .
  ops a b a2 : -> Prin .                op c : -> PqCipher .
  ops css css2 : -> Suites .            ops v v2 : -> Version .
  ops k k2 : -> EcSecretK .             ops pk' : -> PqPublicK .
  ops k' k2' : -> PqSecretK .           op sign : -> Data .
  ops pk pk2 : -> EcPublicK .
-- some conditions
  eq (a = intru) = false .              eq (a2 = intru) = false .
  eq (b = intru) = false .              eq (a = a2) = false .
  eq (k = k2) = false .                 eq (k' = k2') = false .
-- macros
  eq pk = ecPublic(k) .                 eq pk2 = ecPublic(k2) .
  eq pk' = keygen(k') .                 eq c = encapsC(keygen(k'), k2') .
  eq sign = sign(priK(b), h(v || v2 || css || css2 ||
    pubK(b) || ecPublic(k) || keygen(k') || ecPublic(k2) ||
    encapsC(keygen(k'), k2') || h((k | k2) || (k' & k2')))) .
-- a sends {pk || pk'} to b
  eq s1 = cVer(init,a,b,v) .
  eq s2 = sVer(s1,a,a,b,v2,v) .
  eq s3 = cKexInit(s2,b,a,b,css,v,v2) .
  eq s4 = sKexInit(s3,a,a,b,css2,v,v2,css) .
  eq s5 = cHbrInit(s4,b,a,b,k,k',v,v2,css,css2) .
-- intru gleans the message, impersonates a2, and sends {pk || pk'} to b
  eq s6 = fkHbrInit(s5,a2,b,pk,pk') .
  eq s7 = fkVer(s6,a2,b,v) .
  eq s8 = sVer(s7,intru,a2,b,v2,v) .
  eq s9 = fkKexInit(s8,a2,b,css) .
  eq s10 = sKexInit(s9,intru,a2,b,css2,v,v2,css) .
  eq s11 = sHbrReply(s10,intru,a2,b,k2,k2',v,v2,css,css2,pk,pk',1) .
-- intru gleans the message, impersonates b, and sends {pk2 || c || v-sign} to a
  eq s12 = gBasic(s11,v,css,pubK(b)) .
  eq s13 = fkHbrReply(s12,b,a,pubK(b),pk2,c,sign) .
```

```
-- time(s13) is 4, which can be check by command   "red time(s13) ."
-- to check there does not exist such a hbrRepM message (in the conclusion part of auth)
    in the network,
-- we show that for each ?M = {0,1,2,3}, auth is always false
  red auth(s13,intru,a,b,v,v2,css,css2,k,k',pk2,c,sign,0,3,2) .      -- result: false
  red auth(s13,intru,a,b,v,v2,css,css2,k,k',pk2,c,sign,0,3,1) .      -- result: false
  red auth(s13,intru,a,b,v,v2,css,css2,k,k',pk2,c,sign,0,3,0) .      -- result: false
  red auth(s13,intru,a,b,v,v2,css,css2,k,k',pk2,c,sign,0,3,3) .      -- result: false
-- message hbrRepM(b,b,a2,...,2) is in the network,
-- but message hbrRepM(b,b,a,...,2) is not
  red hbrRepM(b,b,a2, pubK(b) || pk2 || c || sign, 2) \in nw(s13) .  -- result: true
  red hbrRepM(b,b,a,  pubK(b) || pk2 || c || sign, 2) \in nw(s13) .  -- result: false
close
```

Thirteen fresh constants `s1`, . . . , `s13` of the sort `Sys` denote thirteen states in which there is a sequence of transition instances: `init` → `s1`, `s1` → `s2`, . . . , `s12` → `s13`. The five macros `pk`, `pk2` `pk'`, `c`, and `sign` are used because there are several occurrences of the corresponding terms in the open-close fragment. Note, however, that their employments are not mandatory. Running this open-close fragment, `false` will be returned for the first four **`red`** commands, `true` will be returned for the fifth one, and `false` will be returned for the last one. From the initial state, client `a` starts sending a version exchange message to server `b`, and the system state changes to `s1`. `b` then replies back to `a` with another version exchange message. Afterward, `a` and `b` exchange a pair of key exchange algorithms messages, and the system state changes to `s4`. `a` then sends two ephemeral public keys `pk` and `pk'` to `b` under a key exchange initiation message, and the system state changes to `s5`. Meanwhile, the intruder also gleans the two public keys. Then, using the two public keys gleaned, the intruder tries to impersonate another client `a2` to send them to `b` (the system state changes to `s6`). The intruder also impersonates `a2` to consecutively exchange with `b` a pair of version exchange messages (the system state changes to `s8`) and a pair of key exchange algorithms messages (the system state changes to `s10`). In this state `s10`, the messages exchanged so far trigger `b` to send to `a2` an ECDH public key `pk2`, a KEM ciphertext `c`, and a valid signature `sign` under a key exchange reply message (the system state changes to `s11`). The intruder gleans all of them. In the final step, the intruder impersonates `b` to send to `a` a key exchange reply message, whose content is the information just learned, i.e., `pk2`, `c`, and `sign`. In this state (`s13`), there exists in the network a valid key exchange reply message apparently sent by `b` to `a`, but the actual creator is the intruder, not `b`. `b` sent such a message to `a2`, not to `a`. The system time in this state is 4, which can be checked by the command **`red`** `time(s13)`. The first four **`red`** commands, whose results all are `false`, show that there does not exist `?M` such that `auth` is true. The fifth **`red`** command confirms that there exists in the network a valid key exchange reply message sent by `b` to `a2` (but not to `a`), while the last command confirms that `b` did not send such a message to `a`.

B PQ SSH: keySe counterexample if server private host key is revealed

In the PQ SSH case study, recall that keySe formalizes the *session key secrecy* property. If we eliminate the following constraint from the premise of keySe:

(4) the private host key of the server is not revealed

the following predicate is obtained:

```
op keySe-f4 : Sys Prin Prin Prin Version Version Suites Suites
  EcSecretK PqSecretK EcPublicK PqCipher Data Nat Nat -> Bool
eq keySe-f4(S,B2,A,B,V,V2,CSs,CSs2,K,K',PK2,C,SIGN,N,N2) =
  (not(A = intru or B = intru) and
    hbrIniM(A,A,B, ecPublic(K) || keygen(K'), N) \in nw(S) and
    hbrRepM(B2,B,A, pubK(B) || PK2 || C || SIGN, N2) \in nw(S) and
    verify(pubK(B), h(V || V2 || CSs || CSs2 ||
        pubK(B) || ecPublic(K) || keygen(K') || PK2 || C ||
        h(ecShare(PK2,K) || decaps(C,K'))),
      SIGN) and
    not(decaps(C,K') \in leakscr(S)) and
    not(K' \in leakscr(S) or pqSecret(C) \in leakscr(S)))
  implies not(h(ecShare(PK2,K) || decaps(C,K')) \in knl(S)) .
```

Suppose that this predicate is defined in the module PRED. A counterexample of the predicate is found, where the transition sequence leading to the counterexample is as follows:

```
open PRED .
  ops s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 s14 s15 : -> Sys .
  ops a b b2 : -> Prin .              ops c : -> PqCipher .
  ops css css2 : -> Suites .          ops v v2 : -> Version .
  ops k k2 : -> EcSecretK .           op  dl : -> DataL .
  ops k' k2' : -> PqSecretK .         ops d sign : -> Data .
-- some conditions
  eq (a = intru) = false .            eq (k = k2) = false .
  eq (b = intru) = false .            eq (k' = k2') = false .
-- macros
  eq c = encapsC(keygen(k'), k2') .
  eq dl = v || v2 || css || css2 || pubK(b) ||
    ecPublic(k) || keygen(k') || ecPublic(k2) || c ||
    h((k | k2) || (k' & k2')) .
-- the private host key of server b is revealed to the intruder
  eq s1 = lLtK(init,b) .
-- a sends the public keys associated with k and k' to b
  eq s2 = cVer(s1,a,b,v) .
  eq s3 = sVer(s2,a,a,b,v2,v) .
  eq s4 = cKexInit(s3,b,a,b,css,v,v2) .
```

```
  eq s5 = sKexInit(s4,a,a,b,css2,v,v2,css) .
  eq s6 = cHbrInit(s5,b,a,b,k,k',v,v2,css,css2) .
-- intru gleans the message, selects two ephemeral keys k2 and k2,
-- computes the shared secrets, and
-- impersonates b to send a faking reply message to a,
-- in which the revealed private key of b is used to sign the exchange hash
  eq s7 = gBasic(s6,v,css,pubK(b)) .
  eq s8 = gEcSecretK(s7,k2) .
  eq s9 = gPqSecretK(s8,k2') .
  eq s10 = gEcShare(s9,k2,ecPublic(k)) .
  eq s11 = gPqShare(s10,k2',keygen(k')) .
  eq s12 = g1(s11,(k | k2) || (k' & k2')) .
  eq s13 = g1(s12,dl) .
  eq s14 = gSign(s13,pkNPair(priK(b), 0), h(dl)) .
  eq s15 = fkHbrReply(s14, b, a,
    pubK(b), ecPublic(k2), c, sign(priK(b), h(dl))) .
  red keySe-f4(s15,intru,a,b,v,v2,css,css2,k,k',
    ecPublic(k2), c, sign(priK(b), h(dl)), 1, 2) .      -- result: false
close
```

Running this open-close fragment, `false` will be returned, meaning that the predicate does not hold in state `s15`. From the initial state, the private host key of server `b` is revealed to the intruder, and the system state changes to `s1`. `a` and `b` consecutively exchange a pair of version exchange messages and another pair of key exchange algorithms messages. Then, `a` sends two public keys to `b` through a key exchange initiation message, and the system state changes to `s6`. The intruder intercepts that message, learning the two public keys. The intruder selects two secret keys `k2` and `k2'`, computes from them and the gleaned public keys the corresponding shared secrets, i.e., `(k | k2)` and `(k' & k2')`, and the system state changes to `s12`. Afterward, the intruder computes the hash of the exchange hash, uses the revealed private host key of `b` to sign the hash, and the system state changes to `s14`. Finally, the intruder impersonates `b` to send back to `a` a key exchange reply message with the signature just computed. In this last state (`s15`), `a` receives a key exchange reply message apparently sent from `b` with the valid signature, but that message actually sent by the intruder, and obviously, the share secret is available to the intruder.