Modelling and Verification of Security Requirements and Stealthiness in Security Protocols

By

Rajiv Ranjan Singh

A thesis submitted to
the University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

Centre for Cyber Security and Privacy
School of Computer Science
College of Engineering and Physical Sciences
University of Birmingham
March 2022

ABSTRACT

Traditionally, formal methods are used to verify security guarantees of a system by proving that the system meets its desired specifications. These guarantees are achieved by verifying the system's security properties, in a formal setting, against its formal specifications. This includes, for example, proving the security properties of confidentiality and authentication, in an adversarial setting, by constructing a complete formal model of the protocol. Any counterexample to this proof implies an attack on the security property. All such proofs are usually based on an ordered set of actions, generated by the protocol execution, called a *trace*. Both the proofs and their counterexamples can be investigated further by analysing the behaviour of these protocol traces. The attack trace might either follow the standard behaviour as per protocol semantics or show deviation from it. In the latter case, however, it should be easy for an analyst to spot any attack based on its comparison from standard traces.

This thesis makes two key contributions: a novel methodology for verifying the security requirements of security protocols by only modelling the attacks against a protocol specification, and, secondly, a formal definition of *'stealthiness'* in a protocol trace which is used to classify attacks on security protocols as either *'stealthy'* or *'non-stealthy'*.

Our first novel proposal tests security properties and then verifies the security requirements of a protocol by modelling only a subset of interactions that constitute the attacks. Using this both time and effort saving methodology, without modelling the complete protocol specifications, we demonstrate the efficacy of our technique using real attacks on one of the world's most used protocols—WPA2. We show that the process of modelling the complete protocol specifications, for verifying security properties, can be simplified by modelling only a subset of protocol specifications needed to model a given attack. We establish the merit of our novel simplified approach by identifying the inadequacy of security properties apart from augmenting and verifying the new security properties, by modelling only the attacks versus the current practice of modelling the complete protocol which is a time and effort intensive process. We find that the current security requirements for WPA2, as stated in its specification, are insufficient to ensure security. We then propose a set of security properties to be augmented to the specification to stop these attacks. Further, our method also allows us to verify if the proposed additional security requirements, if enforced correctly, would be enough to stop attacks.

Second, we seek to verify the 'stealthiness' of protocol attacks by introducing a novel formal definition of a 'stealthy' trace. *'Stealthy'* actions by a participating entity or an adversary in a protocol interaction are about camouflaging fraudulent actions as genuine ones by fine-tuning their actions to make it look like honest ones. In our model, protocols are annotated to indicate what each party will log about each communication. Given a particular logging strategy, our framework determines whether it is possible to find an attack that produces log entries indistinguishable from normal runs of the protocol, or if any attack can be detected from the log entries alone. We present an intuitive definition of when an attack is 'stealthy', which cannot be automatically checked directly, with regard to some logging strategy. Next, we introduce session IDs to identify unique sessions. We show that our initial intuitive definition is equivalent to a second definition using these session IDs, which can also be tested automatically in Tamarin. We analyse various attacks on known vulnerable protocols to see, for a range of logging strategies, which can be made into stealth attacks and which cannot. This approach compares the stealthiness of various known attacks against a range of logging strategies.

DEDICATION

Dedicated to *Barka Babu* (my eldest uncle)

ACKNOWLEDGMENTS

My Ph.D. journey has been replete with sweet and sour experiences all along. I got to learn a lot of things; some about the subject and research, some of them about life, and made many friends. I could not have wished for a better person than my supervisor, Dr Tom Chothia, to guide me through the whole process. Tom, my supervisor in both M.Sc. and PhD, always led me in the right direction whenever I went astray, made sure I stayed focused at the right places, and criticised constructively when needed.

I owe my gratitude to my thesis group members Professor Dave Parker and my co-supervisor Dr Ian Batten. My thesis group made sure that I was on track and helped by constantly asking uncomfortable questions. Thanks, Ian, for sharing your experiences and all the pep talks. I am also thankful to Professor Mark Ryan and Dr. José Moreira for their support and fruitful discussions. Special thanks to our co-author, Dominic Duggan, for his contribution to our paper, and useful comments apart from hosting me at Stevens Institute of Technology, NJ during my visit. I would also like to thank Ralf Sasse from ETH Zurich, and Professor Cas Cremers from CISPA, for their help on Tamarin through emails and personal discussion.

There is a long list of people I owe my gratitude to; for they believed in me and helped me to stay motivated. Special thanks to School of Computer Science, UoB for its studentship offer. I was lucky to meet Richard Thomas, Kris Hicks, Abdullah, Ming, and all my friends from CS 117 at SoCS. Chris McMahon Stone deserves a special mention for endless talks and good times together. A big thanks to Dr. Sujoy Sinha Roy, Dr. Rajesh Chitnis, and Brian Mitchell for their help with the proofreading and suggestions. I was indeed lucky to be around a joyful bunch of people, Dr.Satish Kumar, Dr. Aditya Acharya, Dr. Harish Madabushi, Er. Kumar Rahul, and Dr. Jyotsna Talreja Wassan, who always kept me motivated.

I would like to thank the Chairperson, GB, and Principals Dr Pravin Kumar and Dr. Ramesh Kumar of Shyam Lal College (Evening), (D.U.), for granting study leave and support. Special thanks to my colleagues from Computer Science, especially to (Late) Ms. Bharti Kumar for supporting me in more than one way, who unfortunately, succumbed to Covid-19 just months before thesis submission. Ma'm, you will be dearly missed.

From my home institution, University of Delhi, Professor Naveen Kumar has been a pillar of support and always motivated me to go for further studies and attain the global experience. Dr. Manoj Agarwal has been a dear friend and support throughout this journey. Special mention to my friend Dr. Vishal Chauhan and family for being the constant companion and if I may add, motivator, in Birmingham for a good part of this journey. Dr. Chandrachur Singh and Dr. Hena Singh, a big thank you for your welcome and hospitality during the short time we were together at Birmingham.

My family has been my strength, I cannot thank them enough for everything they have done. My parents, who supported me beyond their means, did everything they could, and gave me all the freedom to make my choices in life. My brothers, Sanjiv, Ravi, Priya Ranjan, and sister, Priyanka, have been pillar of support and constants.

My wife, Seema, 'the wind beneath my wings', kept me motivated and been a true companion. If there is one person who deserves most credit for this thesis, it is her. Thanks to my boys, Sarthak and Hardik, for being so supportive throughout, for being understanding at the age that you are. While we may have lost some expected fun due to my schedule, let us promise each other to more than make up for them going forward. I promise to keep my end of the bargain.

I would not be honest if I do not acknowledge the role of friends from Birmingham, who made my life much more entertaining and comfortable than I expected. Thanks to

Dr Shishank and family, and Mr Manohar and family, for great family times together and always being there when needed.

My cricket team, Bharat Parivar Cricket Club (BPCC), made sure that I got a beautiful bunch of people to interact, and an extended sporting family in UK. I along with my boys, Sarthak and Hardik, are proud to be part of this group. Special thanks to each and every team-mate of mine for all the memories and beautiful time together.

My life is incomplete without the presence and support of all my college time friends, especially Shiv, Kapil, Raju, Sonu, Ashish, Manoj, Shrikant, Gaurav, Vikas, Bhuwan and the list is endless. Guys, thank you for all your support, critiques, and banters that helped me sail through and maintain my sanity. Special thanks to Shiv for regular interaction and support during my research.

I am unable to fathom of any other generation that had to face such conditions as perpetuated by COVID-19, at least in the living memory. The final stages of research, and especially writing up period, was challenging due to isolation and perpetual lock-downs. With the arrival of vaccines, we have started seeing the light at the end of the tunnel. I wish the world gets back to normal at the earliest, and we do not ever have to face anything like this ever again. I would like to thank all the front line workers along with the scientific community who, despite challenging situations, made sure of keeping us safe.

Contents

Page

Part I

Introduction and Background

Chapter One

Introduction

> *"Security protocols are three line programs that people still manage to get wrong"*
>
> NEEDHAM SCHROEDER

1.1 Overview

Security protocols can be regarded as an exchange of groups of messages in a specified sequence among two or more parties, e. g., Alice and Bob, using cryptography to provide various security guarantees including confidentiality, integrity, authentication [68]. These protocols are designed to provide trust, using security guarantees, to the end users while communicating online on an adversary controlled unsecured network. Over time and with advancements in technology, attempts have been made continuously to replace vulnerable security protocols with more robust versions. However, attackers have also devised newer techniques to attack the security protocols, exploit the newer vulnerabilities, and expose the security protocols to new challenges that require new notions of security guarantees. Such situations present multiple obstacles for protocol designers to design robust and more secure protocols, historically an arduous task. In 1996, after 17 years of use, the famous Needham-Schroeder public key protocol was found to be broken and fixed [92, 93]. Recently, in 2017, the latest draft of Transport Layer Security (TLS) [114], TLS1.3 was found to be vulnerable [135].

The security protocols base their security guarantees on their design and use of cryptographic primitives in an adversarial setting. Protocol designers employ multiple approaches for modelling and verification of security protocols for their security properties [30]. They perform various tests, such as conformance of implementation, to make sure that the protocol implementations adhere to specifications making it less prone to be attacked [125, 130]. Any attack on a security protocol implies that at least one of its security guarantees has been compromised. For many years since 1993 [97], formal analysis and verification of protocols has been used to encode the protocols and their security properties to verify if the security guarantees claimed by the protocol are satisfied or not. These claims about the security guarantees are usually placed in the formal model as an assertion. The idea is that all such assertions, i. e., lemmas, must be verified in the mathematical model even in the presence of an all-powerful network controlling attacker. In case they are not, i. e., there exists a counterexample, we conclude that an attack exists violating a security guarantee.

In the formal verification universe, it is standard practice to model the complete specification of a protocol and its security properties in order to verify their security properties. However, in this thesis, we develop and present a novel methodology of modelling only the attacks on the protocols to check if the given set of security properties is sufficient to capture the attacks. To the best of our knowledge, we are the first to establish that there is merit in modelling only the attacks and not the whole protocol. We also show that the results from such modelling can be highly effective to improve the protocol by verifying security requirements.

In late 2018, the University of Toronto and Citizen Lab published a report on suspected infections caused by Pegasus, a spyware developed by Israel-based NSO Group, in almost 45 countries around the world [98, 99]. In most of these cases, Zero-day exploits [9] were installed to take over the control on the victim's cell phone. However, in July 2021, the news of spyware *Pegasus* having been used against journalists, civil society members, and politicians was all over the media spectrum. What left the citizens and researchers astonished was the mention and use of zero-click exploits [107] that did not require active participation of users, and still are successful in taking over complete control of the devices, mainly mobile phones. These hacking attacks work by gaining admin privileges and hence making it difficult to be detected, leaving no trace.

Cyber attackers wish to carry out their attacks, *and* get away with it. Therefore, the attacks which cannot be detected are both much more powerful and useful than attacks that easily show up in system logs. This is particularly true for new zero-day attacks,for which using the attack may alert system owners to the existence of the newly discovered attack, thus leading to system to being patched [24, 145].

If protocol steps are followed exactly as per the protocol specification, its execution will produce a non-attack trace. *Stealth attacks* are where the attacker looks like following the steps in the expected order specified by the standard protocol behaviour, and are thus difficult to detect. These attacks have been studied in diverse contexts such as attacks on Operating Systems using Malware and Intrusion-detection Systems (IDS), Supervisory Control and Data Acquisition (SCADA) systems and Critical Infrastructures (CIs). To analyse the stealth attacks, many studies have focused on monitoring the difference between a benign and malicious behaviour of the systems [38, 112, 116]. While Carzola et al. [38] have compiled a list of stealth attacks against Critical infrastructures (CIs) and classifying the attacks based on objectives of an adversary, Rudd et al. [116] have analysed and shown flaws with the assumptions used by machine learning algorithms in identifying the stealth malware intrusive behaviour. Some studies have also focused on developing a platform to detect memory-based attacks using attack signatures, such as HexPADS [112], that are likely to be missed not only by memory defence mechanisms such as Data Execution Prevention (DEP), Address Space Layout Randomization (ASLR), but also by systems such as IDS.

There are a variety of possible attacks that exploit vulnerabilities of security protocols. Some of these attacks could be stopped by modifying the protocols making them more stringent. The rest of the successful attacks can be detected during post-attack analysis of the system logs that shows an altered behaviour such as incomplete run, replay or not adhering to the usual sequence of steps. There would, however, possibly be some attacks which exploit the weakness of the protocols yet remain undetected through their stealthy actions. We classify such attacks as *Stealth Attacks* that do not look like altering the normal or standard run of the protocols producing a trace which is indistinguishable from the standard run. It may also be possible that these stealth attacks might not be detected even during post-attack analysis of the protocol run logs.

Analysis of security protocols is usually performed either to uncover the flaws in protocols or to prove the correctness of security properties [68]. This thesis adopts a novel approach of studying known attacks, and analysing them to strengthen the protocol specifications, rather than trying to uncover new ones. We use several known attacks on security protocols as case studies to investigate their conformity against our framework and label them as either stealth attacks or non-stealth attacks based on the attacker following the normal protocol behaviour or deviating from it respectively.

The study of stealth attacks has seen active research since 2003 with Jakobsson et al. [81] describing the stealth attacks as one minimising the cost to, and visibility of, the attacker. However, to the best of our knowledge, there has been no published attempt to formalise the notion of stealth attacks on security protocols, making our contribution novel. Selection of log parameters plays an important role in detecting a stealth attack. To the best of our knowledge, we are the first to investigate the stealth attacks in the context of security protocols and analyse the stealthiness of protocol traces containing logs.

1.2 Research Objective and Questions

Expressing the security protocols on paper is not a difficult task. However, manual verification of their claimed security guarantees can be a challenging task due to inherent high degree of concurrency involved [15]. This challenge has motivated researchers to develop formal methods, models, and automatic verification tools to model the protocols and verify their security properties [138].

Usually, the formal analysis of a security protocols performed in anticipation of finding out a new bug or to establish the correctness of their security properties [68]. Most of the studies and analysis using formal methods revolve around modelling of complete specification of protocols and encoding their security properties to see if there exists a violation of any security property, implying that an attack exists. This process involves modelling of the complete protocol specifications, and is a time-consuming process. In contrast to this, we set out to answer whether there is any merit in modelling just the attacks over modelling the complete protocol specifications. If so, is it possible to test all the security properties mandated by the protocol standard in the attack environment? If yes, what can we learn from them and if not, why not?

The work presented in this thesis can be broadly divided in two sections. The first section, i. e., *Modelling of Attacks on Security Protocols*, consists of two units. The first unit presents a methodology of modelling only attacks on security protocols using attacks such as KRACK [137] and Downgrade attacks [131] on the 802.11 *4-way handshake*. The second unit focuses on analysing the security properties of 802.11 *4-way handshake* under attack scenario. The results of this analysis is then extended to present a novel methodology demonstrating the improved protocol specification by augmenting and verifying new security properties.

Stealthy attacks attempt to camouflage the network activities of an attacker with message traffic that appears to be standard, as defined by the protocols for standard messaging not leading to an attack scenario. Such an attack is potentially indistinguishable from a standard run of a protocol. Much previous research has investigated attempting to discover from system logs that a stealthy attack is in progress.

This thesis examines this issue from another perspective: Given the protocols that message traffic should be following, do these protocols admit stealthy attacks by allowing an attacker to "hide" an attack in the underlying message traffic? Our approach is to use well-known techniques for specifying security protocols, the framework of multiset-rewriting, to provide the specification of the protocols of interest. The underlying idea is to augment the protocol specifications with the logging specifications that will be used in detection of attacks. This process is followed by the application of formal protocol analysis under the stealthiness restrictions.

Based on the above perspective, the second section of this thesis defines *Stealthiness in a Trace Model.* The first unit defines a formal model of stealthy run, starting by tagging a protocol model with what would be logged on each step. The protocol execution is modelled using the labelled transition rules, and the traces generated therefore will contain our logs. Subsequently, formal definitions of *standard looking run*, *attack run*, and stealth attack are proposed using the traces generated by the executions. The next unit of this section implements formal definition of stealth attack, developed earlier, using TAMARIN [103]. In this model, we add a session ID to each protocol run instance, to identify a specific run instance from interleaving of multiple runs, while restricting each run to follow the stealthiness properties, the attacker is challenged to have an attack (stealth attack) on the protocol. A summary of testing the stealthiness of various known attacks on security protocols for their stealthiness is presented along with a discussion. Finally, we use the formal proof management system Coq [17, 110] to model TAMARIN semantics along with many lemmas and propositions regarding the system behaviour. Using these tools, we provide a manual proof that checking this restriction-based definition of a standard looking run, with a protocol with session identifiers using TAMARIN, is equivalent to our intuitive definition of a standard looking run of a protocol without session IDs.

This thesis makes use of term algebra and labelled multiset rewriting in order to abstract from implementation details and provide a high level description of the interaction, communication, and synchronization among the protocol participants including the adversary. We use Tamarin to specify protocols and semantics extended by us to model the single protocol run, interaction among various protocol runs, verification of security properties using trace properties etc. These semantics allow modelling of cryptographic primitives, and also provide an equational theory to model powers of an attacker, and active substitutions to model an attacker's actual knowledge.

Our modelling is quite abstract yet powerful, considering only the values placed in the logs, typically excluding features such as packet timing and size, usually used to detect an anomaly. We also do not consider likelihood of a particular action. However, if our method marks an attack as stealthy, then we can be sure that there is some innocent mix of runs of the protocol that would produce exactly the same logged actions that the attack produces. Alternatively, if our framework shows absence of stealth attack, then the attack will also be detectable from the logged values for a correct implementation of the protocol. We summarise our research objective in the form of the following research questions:

Formal methods have been successful in modelling various protocols, uncovering vulnerabilities, and help fix many attacks. This thesis, however, tries to answer the question:

RQ1: *Can the modelling of attacks on protocols be used to test the security requirements to improve the protocol specification?*

There have been many studies related to stealth attacks in various contexts and various detection mechanisms. However, to the best of our knowledge, there does not exist any formal definition of stealth attack presenting a research gap and opportunity. Based on such scenarios, we are interested in answering the question:

RQ2: *Is it possible to* ***formally define*** *attacks on protocols in two broad categories: first, where the attacker is able to hide the attack in the message traffic by fine-tuning the attack steps following the protocol sequence to camouflage the attacks, i. e., stealth attack, and second, where the protocol sequences are violated making it easy to capture, i. e., non-stealth attack?*

Given a mechanism to compare various logging strategies, it would be easier to analyse various attacks and see for what set of logging parameters, the attacks remain undetected or vice-versa. With this objective, we would like to answer the question:

RQ3: *Given a protocol and attacks thereon, can we apply formal methods to come up with a minimal set of logging parameters, required to be logged, in order to capture the stealth attacks thereby converting stealthy attacks to non-stealthy attacks?*

1.3 Thesis Overview & Structure

The structure of this thesis, also shown is Figure 1.1, is as follows:

Chapter 2 provides the preliminaries and background information used in our work. It talks about security properties, and attacks on them. A literature review of various tools used in the formal verification universe, including Tamarin use cases, is followed by a detailed survey of stealth attacks used in various contexts. This chapter sets the foundation for verifying the security requirements and stealthiness of protocol traces in the subsequent chapters.

Chapter 3 presents a detailed introduction to the automatic protocol verification tool Tamarin and its front-end tool SAPiC. Using KRACK [137] and Downgrade [131] attacks as case studies, the process of developing a novel formal model of these attacks using Tamarin and its front-end tool SAPiC, is demonstrated here.

Chapter 4 presents a novel methodology of improving the protocols using modelling of attacks. On modelling KRACK [137] and Downgrade [131] attacks, it was discovered that the security properties present in the 802.11 standard document are insufficient to capture these attacks. Subsequently, using our methodology, we propose additional security properties to be added to the standard and successfully demonstrate that the additional properties are not only able to capture these attacks but also verify that the attacks do not exist in the improved protocol augmented with our additional security properties.

Work presented in chapters 3 and 4 align with our research question ***RQ1*** *from Section 1.2.*

Chapter 5 presents the process of developing a **'Formal Stealth Model'** explaining how to use formal methods to define stealthiness in a protocol trace. It explains the steps and process of extending the semantics of a *'Labelled Multiset Rewriting (MSR)'* system used in Tamarin. The model supports the addition of a custom log and allows the user to choose the parameters to be logged. We then define a standard trace based on actions, containing the custom logs, generated during the standard execution of the protocol, i. e., a standard run, where the attacker simply passes outputs to inputs. A stealth attack is considered as a successful attack yet producing a standard looking trace.

Chapter 6 explains the process of implementing the definitions proposed by the **'Formal Stealth Model'**, presented in the previous chapter, using the protocol verification tool Tamarin. We start by arguing that it is difficult to implement the formal definitions of stealthiness directly in any protocol tool, as there is always a possibility of exponential blow-up due to the concurrent execution nature of the protocol steps. Since it is impossible to distinguish between the actions of various instances, we introduce session identifiers to be included as part of actions to uniquely identify each log entry and run of the protocol they correspond to. Based on the new definitions, we propose two restrictions, namely Uniqueness and Correspondence, to be added to the Tamarin model of the protocol in order to generate only stealthy traces. At the end, we provide a proof sketch of our central theoretical result, i. e., *"An attack is present in a trace with session IDs, under certain restrictions, if and only if a stealth attack is present in a trace without session IDs".*

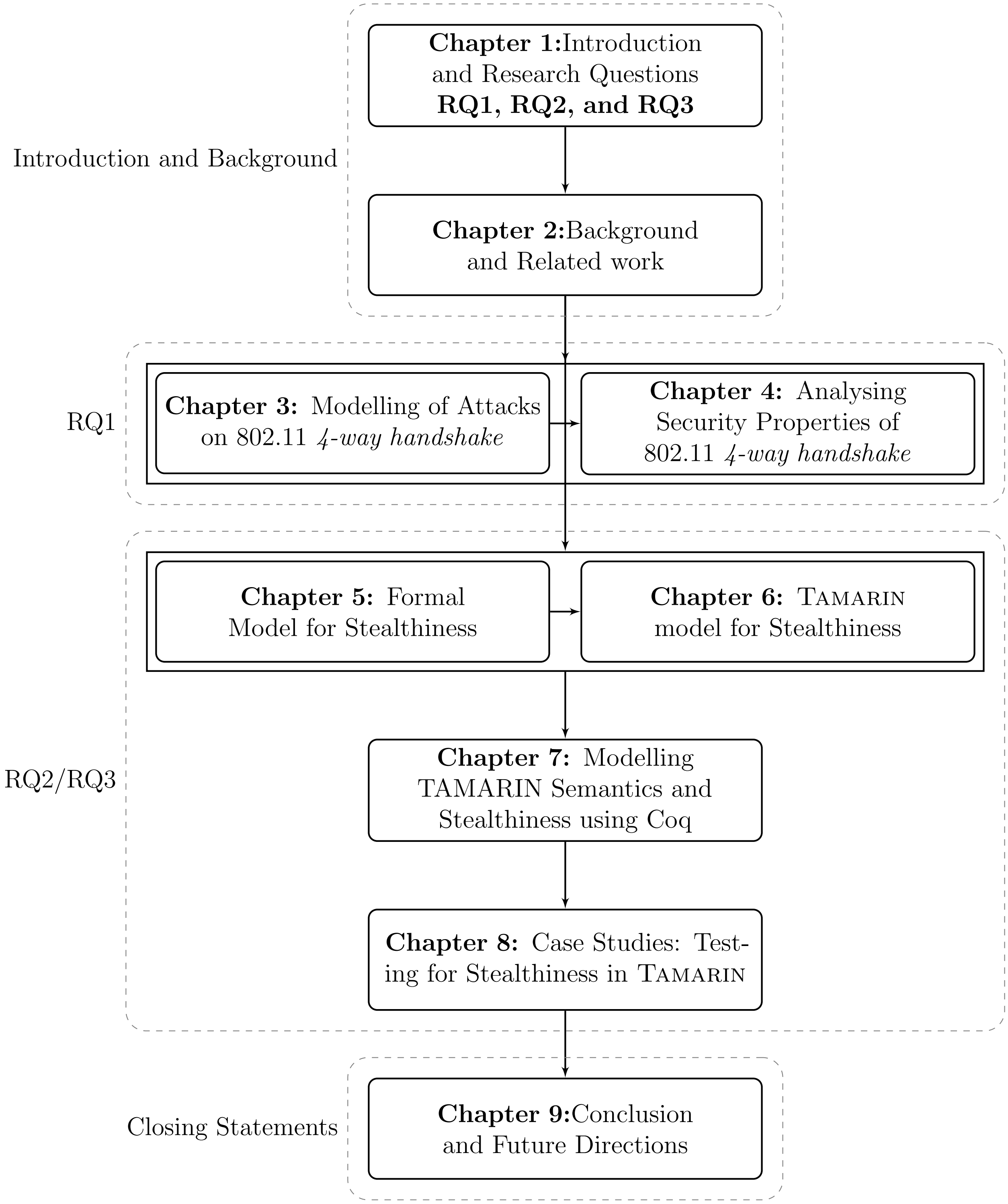

Introduction and Background
Chapter 1: Introduction and Research Questions **RQ1, RQ2, and RQ3**
Chapter 2: Background and Related work
RQ1
Chapter 3: Modelling of Attacks on 802.11 *4-way handshake*
Chapter 4: Analysing Security Properties of 802.11 *4-way handshake*
RQ2/RQ3
Chapter 5: Formal Model for Stealthiness
Chapter 6: TAMARIN model for Stealthiness
Chapter 7: Modelling TAMARIN Semantics and Stealthiness using Coq
Chapter 8: Case Studies: Testing for Stealthiness in TAMARIN
Closing Statements
Chapter 9: Conclusion and Future Directions

Figure 1.1: Thesis Structure and Outline

Chapter 7 presents use of interactive theorem prover Coq in modelling the semantics of TAMARIN. We start by presenting modelling of standard TAMARIN semantics that can also be reused by anyone interested in modelling another TAMARIN case study using Coq.

Subsequently, we also present modelling of TAMARIN extensions developed by us. Finally, we augment Coq model with useful lemmas and theorems to enforce correctness conditions in the model.

*Work presented in chapters 5, 6 and 7 align with our research question **RQ2** from Section 1.2.*

Chapter 8 discusses the algorithm used by of the automated python based tool *'StealthCheck'* developed by us. *StealthCheck* automatically adds session identifiers and stealthiness restrictions to a TAMARIN model of a protocol. We use *StealthCheck* to test the stealthiness of various attacks from both the publicly available TAMARIN models, and of KRACK [137] attack model developed by us. Our results show, for various common logging strategies, which attacks can be made into stealth attacks and which cannot.

*Work presented in chapter 8 align with our research question **RQ3** from Section 1.2.*

Chapter 9 presents a discussion on our research presented in the thesis, followed by possible future directions.

1.4 Publications

During work of this thesis, the following work was published. The ideas and results presented in this thesis are also derived from the following publications.

- Rajiv Ranjan Singh, José Moreira, Tom Chothia, and Mark D Ryan, *"Modelling of 802.11 4-Way Handshake Attacks and Analysis of Security Properties"* In the Proceedings of the 16th International Workshop on Security and Trust Management Co-located with ESORICS 2020, 17–18 September 2020. Guildford, UK, pages 3-21. Springer, 2020.
- Rajiv Ranjan Singh, Tom Chothia, and Dominic Duggan, *"Defining Stealthiness in a Trace Model"* In preparation for submission.

Chapter Two

Background & Related Work

If I have seen further, it is by standing on the shoulders of Giants.

Sir Isaac Newton

2.1 Overview

This chapter introduces background concepts such as security protocols, their security properties, and attacks on them. We also cover topics such as background for understanding the formal protocol verification universe, preliminary concepts related to term rewriting and labelled multiset rewriting to develop our model. Further, this chapter presents a short summary of various studies conducted related to formal modelling of protocols and their security properties. We also provide a summary of *stealth attacks* discovered, in various contexts, so far. We show that, based on our analysis, stealth attacks do not find mention either in context of security protocols or being analysed in a trace model.

2.2 Introduction

A security protocol can be regarded as an ordered set of messages between two or more agents to facilitate a secure two-way message exchange. It uses cryptography to either authenticate the communicating parties or provide key distribution for new sessions [68]. The area of security protocol verification has seen a lot of research since 1990s [28]. Most of the recent attacks on the security protocols have been successful either because of cryptographic weaknesses (BEAST, CRIME [120]), protocol logic flaws (HeartBleed, 3Shake [66]), implementation bugs (LOGJAM [7], SLOTH [22]) or a combination of them. Accordingly, in line with the identified factors responsible for the attacks, these attacks can be identified and prevented in two ways. First, the cryptographic scheme used in the protocol can be analysed and proven secure. Second, we can analyse the full specification of the protocol and verifying the implementations. All such analysis can be performed either manually or by using automated analysis tools. Some of these techniques posses a few advantages over the other, although there always remains some gaps between theoretical assumptions of the schemes and their implementations [20].

The security community has long been debating over preferring one class of tools over the other. However, it has been accepted that the formal analysis of security protocols is certainly the most economical process, at least in terms of the time and effort, compared to the actual implementation followed by testing. It has become a popular research area during the last three decades. Formal security analysis has been put to use in successfully analysing many protocols [19, 44, 51, 65] and uncovering flaws.

The field of formal verification has been strengthened with the development and support of many advanced tools such as ProVerif [29], Avispa [10, 138], Scyther [58], and Tamarin Prover[103]. All these tools can be used to verify whether a security protocol successfully preserves the security goals or not.

In the next sections, we will provide an overview of security properties, and some attacks on them. This is followed by an introduction to formal protocol verification tools, their use cases, and a literature review of stealth attacks. Finally, we introduce term rewriting and *labeled multiset rewriting rules*which we would be using in our formal analysis, to be used in subsequent chapters.

2.3 Security Properties

A security protocol is designed with focus on providing some form of assurance against unauthorised access and usage, to both the data being processed and the parties processing them, in an environment controlled by an adversary. A security guarantee is an assurance provided by a security protocol which, if used according to the designer's specifications, provides provable specified security goals. However, not all security protocols are designed to protect every security goal. As discussed earlier, we have presented a novel methodology of not modelling the whole protocol but only the part of it enough to model an attack, it is imperative that we discuss some common security guarantees that the protocols usually claim. The list is not an exhaustive one, as the idea is to make the reader aware of the methodology adopted in encoding these security properties.

Secrecy: The concept of secrecy, the most basic and important security notion, requires that the attacker is unable to access the protected sensitive information, even if the communication uses an unsecured network. The formal models of cryptography assumes perfect cryptographic primitives and consider terms built using these primitives. *Secrecy* is sometimes classified in two categories; namely *standard secrecy* and *strong secrecy.* While the *standard secrecy* requires that the attacker must not be successful into obtaining the value of secret, the notion of *strong secrecy* mandates that the attacker must not be able to differentiate between the events that lead to change in the secret values [27].

Authentication: Authentication is one of the most studied property by researchers in security protocol analysis. Though the notion of 'secrecy' is quite clear in every context, 'authentication' as a property is still evolving [57]. Authentication is defined as a party being sure of the identity of the other. Lowe [95] has proposed a hierarchy of authentication starting from aliveness to agreement. While the weakest form of authentication, i. e., aliveness may not guarantee both the agents about guaranteed participation of the other in the protocol run, the agreement or injective agreement seeks to enforce one-to-one relationship among the agents over the variable values used upon successful completion of the protocol. Between aliveness and agreement, two more forms namely weak agreement and non-injective agreement have also been placed in the hierarchy.

Cremers et al [56] have extended Lowe's hierarchy of authentication with the notion of synchronisation. In their other work [57], they have argued that the synchronisation property implies agreement and is strictly stronger than agreement. The need for stronger property is justified given that certain classes of attacks such as replay attacks may be successful even on a security protocol satisfying agreement.

2.4 Attacks on Security Protocols

There is rich availability of literature related to the study of protocols, attacks, vulnerabilities, and fixes proposed on them. We list a few attack examples, and their categories, in this section. For more information on security protocols and attacks on them, we refer the readers to study the authentication and key establishment protocols [34] in detail.

2.4.1 Replay/Pre-play Attack

In this attack, the intruder replays a message from previous run or other protocol in a different context order to trick an honest agent in believing the completion of successful run of the protocol [96]. These attacks can be prevented by making use of session ID attached with every message to make them unique. Implementing synchronisation using one time passwords (OTPs), timestamps etc. can also be helpful. Lowe's man-in-the-middle attack on Needham-Schroeder Public key protocol [46, 93], presented in Sec. 2.4.3, is a famous example of replay attack.

2.4.2 Type Flaw Attack

The type flaw attack on a security protocol works by interpreting a term, originally intended to be of one type, as being of another type [140]. The Otway-Rees protocol [108] is expected to provide symmetric assurance of key freshness using a trusted server, and is prone to type-flaw attack explained in [42]. In this attack, the responder is fooled into interpreting combination of message and participants name as session key [140].

1: $A \rightarrow B{:}M, A, B, \{N_A, M, A, B\}_{K_{AS}}$
2: $B \rightarrow S{:}M, A, B, \{N_A, M, A, B\}_{K_{AS}}, \{N_B, M, A, B\}_{K_{BS}}$
3: $S \rightarrow B{:}M, \{N_A, K_{AB}\}_{K_{AS}}, \{N_B, K_{AB}\}_{K_{BS}}$
4: $B \rightarrow A : M, \{N_A, K_{AB}\}_{K_{AS}}$

Figure 2.1: Otway-Rees protocol [108]

Fig. 2.2 describes a type-flaw attack on Otway-Rees protocol [108]. This attack considers an intruder I who has obtained the encrypted message $\{M, C, B\}_{K_{BS}}$by engaging in a previous run of the protocol with B. The attack is successful as it successfully tricks A

1: $:A \rightarrow I_B{:}M, A, B, \{N_A, M, A, B\}_{K_{AS}}$
2: $:B \rightarrow S{:}M, A, B, \{N_A, M, A, B\}_{K_{AS}}, \{N_B, M, A, B\}_{K_{BS}}$
3: $:S \rightarrow B{:}M, \{N_A, K_{AB}\}_{K_{AS}}, \{N_B, K_{AB}\}_{K_{BS}}$
4: $I_B \rightarrow A{:}M, \{N_A, M, A, B\}_{K_{AS}}$

Figure 2.2: Otway-Rees Protocol and type-flaw attack [34]

into believing that it is talking to B. As a result, A ends up accepting the triple (M, A, B) as a fresh key K_{AB} in the last message 4.

2.4.3 Man-in-the-Middle Attack

In Man-in-the-middle (MITM) attack, an adversary sitting between the two communicating parties, is able to modify as well as control the sequence of the messages without any of the parties detecting its presence. It is one of the oldest known forms of attack and has been known to be successful against various entity authentication protocols including HTTPS [37] and very recently on TLS 1.3 specifications [53].

1:$A \rightarrow B$: $\{N_A, A\}_{pubK_B}$
2:$B \rightarrow A$: $\{N_A, N_B\}_{pubK_A}$
3:$A \rightarrow B$: $\{N_B\}_{pubK_B}$

Figure 2.3: Needham-Schroeder Public Key Protocol [93]

1: $A \rightarrow C$: $\{N_A, A\}_{pubK_C}$
1' :$C_A \rightarrow B$: $\{N_A, A\}_{pubK_B}$
2': $B \rightarrow C_A$: $\{N_A, N_B\}_{pubK_A}$
2: $C \rightarrow A$: $\{N_A, N_B\}_{pubK_A}$
3: $A \rightarrow C$: $\{N_B\}_{pubK_C}$
3': $C_A \rightarrow B$: $\{N_B\}_{pubK_B}$

Figure 2.4: NSPK Protocol and Lowe's attack [34, 92]

In an MITM attack, the attacker uses some form of replay trick to authenticate itself with the victim. The classic Needham-Schroeder Public Key Protocol [92, 93] is vulnerable to this attack as shown in Fig. 2.4.

2.4.4 Reflection Attack

A reflection attack is a special form of replay attack where the attacker, rather than replying to the query, sends it back to the originator itself and waits for the originator to send back the response. This response can then be sent back to the originator to complete the protocol run thereby breaking the challenge-response authentication [34, 117].

A reflection attack on NSPK [93] is shown in Fig. 2.5. Here, A and B wish to authenticate each other by sending nonces across encrypted with a pre-shared key. Attacker C, masquerading as A, can easily suppress the real message 3, and instead use message 3' to complete the protocol run with A leading to an attack. The attacker can start a parallel run and gets authenticated using reflection attack as described below:

1 :$A \rightarrow B : \{N_A\}_K$

2 :$B \rightarrow A : \{N_B\}_K, N_A$

1':$C_B \rightarrow A : \{N_B\}_K$

2':$B \rightarrow C_A : \{N_X\}_K, N_B$

3':$C_A \rightarrow B : N_B$

Figure 2.5: Reflection attack on NSPK Protocol [34]

2.4.5 Attacks on Authentication

Any attack on authentication attempts to exploit the weakness in an authentication process used to verify the identity of a protocol participant. Most of the above-mentioned type of attacks can be launched on security protocol to break the authentication in a protocol run. The Station-To-Station Protocol, or STS protocol [26], seeks to provide mutual key authentication and confirmation along with forward secrecy [63]. The protocol version suffers from unknown key share attacks against both the Initiator and Responder, with the adversary able to register their public keys as its own and successfully intercept the message intended for honest parties, violating the perfect forward secrecy.

1: $A \rightarrow B : A, B, g^x$
2: $B \rightarrow A : B, A, g^y, sig_B(g^y, g^x), mac_K(sig_B(g^y, g^x))$
3: $A \rightarrow B : A, B, sig_A(g^x, g^y), mac_K(sig_A(g^x, g^y))$
4: $A \rightarrow B : A, B, g^x$
5: $B \rightarrow A : B, A, g^y, \{Sig_B(g^y, g^x)\}_{K_{AB}}$
6: $A \rightarrow B : A, B, \{Sig_A(g^x, g^y)\}_{K_{AB}}$

Figure 2.6: Modified STS Protocol [34]

1 :$A \rightarrow C_B : A, B, g^x$
1':$C \rightarrow B : C, B, g^x$
2':$B \rightarrow C : B, C, g^y, \{Sig_B(g^y, g^x)\}_{K_{AB}}$
2:$C_B \rightarrow A : B, A, g^y, \{Sig_B(g^y, g^x)\}_{K_{AB}}$
3:$A \rightarrow C_B : A, B, \{Sig_A(g^x, g^y)\}_{K_{AB}}$

Figure 2.7: Attack on Modified STS Protocol [34]

The attack example shown in Fig. 2.7 is a combination of replay attack (message 2' used in 2) along with modified message. C has successfully convinced A that he is talking to B by starting a parallel run of the protocol and mixing messages from these runs.

Defence mechanisms against these attacks exist in the form of newer versions of protocols and guidelines. In their landmark paper, Abadi and Needham [5] have outlined some principles for implementations of cryptographic protocols that would make the attacks on protocol more difficult, if not impossible. Adoption of these principles can help protocol designers in strengthening their protocols against some known attacks.

2.5 Formal Protocol Verification

Developing a security protocol as well as the methods to analyse them is a rigorous mathematical exercise. To prove the reasoning about their correctness is also a complex task due to the inherent complexity of the protocols [12]. The discovery of a flaw affecting the Needham-Schroeder public-key protocol using formal methods was an important milestone in the development and acceptance of formal methods.

Formal verification of protocols is fast becoming one of the popular research topics in recent years [60, 86, 93]. Various approaches, e. g., logic-based, process calculus base and symbolic approaches have been proposed to model the protocols and verify their security properties [100]. A logic based approach is based on representing protocols in terms of logic formulae combined with inference rules, whereas a process calculus based approach models the protocol participants as processed executing in parallel that interact via exchanging messages.

The model checking approach models the protocol by defining a set of states and transitions along with an intruder. The knowledge of an intruder along with the information known to each principal is taken into account while searching the state space to check the reachability of some particular state. Such models can also verify or falsify the possibility of generating specific traces [100].

2.5.1 Security Protocol Verification Models

Presently, two models of verification are popular in the security protocol verification landscape : computational and symbolic models [28]. While the computational methods concerns about the verification of the cryptographic functions used in security protocols, symbolic methods model security protocols as sequences of interactions among the protocol participants while assuming perfect cryptography.

Computational Model: The computational model is built on following the ideas forwarded by Goldwasser and Micali [72] involving complexity and probability theories. The computational (or cryptographic) model represents the messages using bit-strings and cryptographic primitives using functions that map bit-strings to bit-strings, with the adversary modelled as a probabilistic Turing machine. The security of a protocol is measured by the probability of a security property holding true or otherwise [28].

Symbolic Model: The symbolic model is built around abstract, symbolic modelling and has its origin in the pioneering work by Dolev and Yao [64]. The messages here are modelled using terms in a term algebra and various operations on messages by message deduction rules. A term such as $enc(Msg, K)$ represents the encryption of the message, i. e., term Msg with the key, i. e., term K. The terms known in the system can be paired, encrypted, and decrypted to form newer terms. However, the decryption can only be performed when the key is known assuming perfect cryptography [28]. First such formalism was proposed by Dolev-Yao that came to be known by a powerful Dolev-Yao adversary [64] controlling the network. It can read, intercept and send all the messages on the network and is limited in its capability only by cryptographic constraints.

Some researchers, such as Creese et al. [115], have argued that Dolev-Yao model presents a worst-case scenario which is very difficult if not impossible in real world assumptions. We agree with their assertion that it is always better for a protocol analysis to be failing in a strict environment compared to satisfying the security goals in a restricted threat model [115].

Both the approaches have witnessed huge progress especially in the last decade as researchers started looking at formal verification of protocols before their implementation [53]. The symbolic approach is an abstract model and allows for better proof automation, but its results are not easy to relate to real world security goals. In contrast, the computational approach provides more realistic security assurance while sacrificing the ease of proof automation [12]. In a significant seminal paper [6], Abadi and Rogaway tried to bridge the gap by proposing a computational justification for treating encryption formally. Subsequently, there have been multiple attempts to establish a relation between the two approaches. Some of these approaches focused on proving computational soundness for the symbolic model, whereas others applied reasoning techniques from the symbolic model to the computational model [12].

Applications of formal research outcome have demonstrated the usefulness of formal verification tools in detecting and correcting the protocol flaws even after their standardisation [93]. However, all generic protocol verification tools may not necessarily support the verification of the security properties of a cryptographic protocol.

2.5.2 Formal Protocol Verification Tools

The research community, working on formal verification, has developed various security protocol verification tools over the years. Most popular of them include, but not limited to, Tamarin[103] (Meier, Cremers, Basin, 2013-), Scyther [58] (Cremers etc 2006-), AVISPA [10, 138] (Large EU project and team, 2005-), Pro-Verif [29] (B. Blanchet, and others, 2001-), Athena [129] (D. Song, 1999-), Casper/FDR [94] (G. Lowe, 1997-), NRL [101] (C. Meadows, 1994-), etc. to name a few.

The approach used by these tools are similar in the sense that the security properties of protocols are specified either as trace-based or equivalence properties. Trace-based reachability property seeks to establish the presence of a given security property, such as secrecy and authentication, in all possible generated traces. The security properties such as anonymity, unlinkability, and vote secrecy are collectively known as privacy properties. The equivalence properties can express privacy properties by establishing the inability of an adversary in differentiating among two scenarios [41]. As stated earlier, this thesis makes use of symbolic model for formal verification using mathematical framework defined in Tamarin. Other verification tools, such as ProVerif [29] support unbounded verification, but are not guaranteed to terminate. Although Tamarin is also not guaranteed to terminate, it does have an interactive mode that provides for manual guided proof mechanism. An introduction to fundamentals of Tamarin is presented in Sec. 3.5.

2.5.3 Protocols and Trace Properties

A run of a security protocol involves execution of roles by participant(s) or agent(s) performing some actions. A trace can be generated by compiling these sequence of actions performed by these agents. A security protocol can then be inductively defined as a set of traces, with each trace resulting from multiple interleaved runs. An agent is capable of executing multiple interleaved runs, with each run as an instantiation of a role, in the protocol definition [56]. In the presence of an attacker, trace of a process may be defined as any possible interleaving of transitions with an attacker able to intercept, modify, and forward the messages. The trace can be analysed for verifying conformance to various security properties by the protocol. The manual proof of these security properties are very long and time-consuming [111]. However, many automatic tools, such as Pro-Verif, Scyther, TAMARIN etc., present in Section 2.5.2 provide a push-button approach for such analysis by modelling the security properties as trace properties [48, 123].

Security properties such as secrecy, authentication, observational equivalence etc. have been defined on traces. The reachability property in a trace is used to check if some security property holds for any execution trace or not, and is used for properties such as secrecy and authentication. The equivalence property expresses the inability of an adversary to distinguish between the traces generated by two different scenarios [41], and is commonly used to express privacy-related properties such as anonymity and unlinkability etc. [61, 77]. However, some properties, such as information flow properties, can not be verified through trace properties [35].

2.6 Formal Verification of Security Properties

The formal methods have been studied and applied extensively in recent decades to analyse the specifications of the system and proving their security properties. A complete formal model of a system encompasses an abstract view of the system, its desired security properties along with the powers of adversary. The formal protocol verification process must start with a complete specification of the protocol and make explicit assumption about the capability of the adversary. It must be proved that the specified protocol achieved the target security guarantee given the adversary assumptions hold [122]. These assumptions help the protocol designers to claim if the protocol behaves in an intended fashion in an adversary controlled environment.

Formal verification has been helpful in finding many bugs and prove security guarantees during both the pre- and post-deployment phases of various protocols. Automated analysis of the security protocol has been successful in uncovering many flaws in the security protocols starting from the famous Lowe's attack [93] on the Needham-Schroeder public key protocol after 17 years of its publication. This beginning was followed by several famous cases such as flaws discovered in Google Single Sign on protocol [11] and PKCS#11 standard implementations [33], HeartBleed bug [66] in OpenSSL etc. All these attacks might have been known to attackers prior to their discovery by researchers.

2.6.1 TAMARIN PROVER Use Cases

Tamarin has emerged as a popular symbolic modelling and formal protocol analysis tool. It has been used to analyse a variety of protocols and schemes. It was the Tamarin modelling of TLS 1.3 [51], which made it possible to uncover a potential attack, with the adversary masquerading as a client, without being detected. This analysis helped improve the protocol by including additional safeguards which satisfied all the desired security guarantees [51]. Analysis of many other security protocols such as 5G authentication [13], secure Internet of Things (IoT) [88], and PKCS# 11 [90] have also been successfully performed using Tamarin. Other case studies include analysing the EMV(EuropayMasterCard-Visa) complaint payment protocol for mobile devices [47] and vehicular networking (V2X) security protocols [142]. Both the Direct Anonymous Attestation Schemes, i. e., TPM 2.0-based [141] and ECC-based [143] have also been analysed using the Tamarin.

The credentials above are enough to prove the versatility and efficacy of Tamarin for modelling and analysis across wide range of security protocols and schemes. All these studies mentioned above have modelled complete specifications of the protocols, or systems under study, to perform security analysis. They have used these comprehensive models to uncover new flaws and/or suggest fixes to the design. However, development of such models can be a time-consuming process taking many months, even for a large team [52].

2.7 Stealth Attacks in Various Context

Remaining undetected during attacks is one of the primary aim of an attacker. It empowers them to relaunch the attacks multiple times and is a serious risk for an information system. The concept of *stealth attack* was introduced by Jakobsson et al.. in 2003 as the class of attacks having minimised cost and less visibility [81]. The stealth attack described by them had mainly three phases viz. stealthiness of communication, execution, and propagation to remain stealthy throughout the life cycle. A stealthy attacker has also been defined as a special class of adversary who fine-tunes his actions to avoid being detected [38]. Similarly, every defence mechanism or countermeasure being designed against stealth attack must also study the environment of every system to be protected. An example of a perfect stealth attack could be Zero-click exploits [9, 107], developed by Pegasus, that are successful in taking full control of user's devices without their active participation.

Stealth attacks and their detection mechanisms have been studied mainly against Cyber-Physical system backdrops, e. g., SCADA, Smart Grids and similarly networked control systems. Some studies have analysed stealth attack on operating Systems using stealthy malware, rootkit etc. We, however, found no evidence of any attempt to analyse the stealthiness of security protocol traces, either using experimental or formal methods.

2.7.1 Stealth Attacks on Cyber-physical Systems

Many researchers have analysed the stealth attack scenario in industrial control systems (ICS) and cyber-physical systems [32, 38, 74]. A powerful attacker may mount a stealthy attack by driving the system to an arbitrary state while keeping the detection parameters under threshold. Such situations can be catastrophic and difficult to detect, as the attacker makes the detection difficult by keeping the parameters unchanged. Stealth attacks, analysed in the cyber physical system scenarios, have proposed hard to detect attacks against industrial control systems [32, 38]. Possibility of cyber stealth attack have been explored against critical information (CI) infrastructure [38], in which the powerful attackers launch very dangerous attacks with the objective of capturing sensitive information by remaining stealthy throughout their operation. Any remotely controlled cyberattacks on the critical infrastructures can prove disastrous, more so when the attack remains undetected or stealthy. When the attackers are primarily interested in remaining unnoticed, the system may remain exposed for longer duration without protection. This allows the attacker to train their attack vectors to other connected components of the CIs.

Cazorla et al. [38] conclude that stealth attack are very dangerous and extremely difficult to fully secure networks against them. After studying various mechanisms to prevent and thwart stealth attack they propose a combination of active and passive security mechanism. Cyber stealth attack in critical infrastructures [38] usually comprises stealthiness of communication, execution and propagation in that order. Every single attack, however, can be unique and may involve one or more phases out of three mentioned, but always in the same order.

Bopardikar et al. [32] has discussed the challenges of securing existing legacy cyber-physical control systems from stealth attack on the system components interaction. The proposed solutions focus mainly on the system reconfiguration to modify the attack effect, or by deploying additional virtual but secure measurements.

2.7.2 Stealth Attacks on Operating Systems

Stealth attacks on operating systems using Malware and Ad-hoc networks have been discussed in various research such as [73, 116, 139] etc. Mathias has proposed a platform HexPADS [112] to detect stealth attack based on the principle that the OS provides very limited software metric and is not suited to capture various memory based stealth attack. Attacks such as side channel and covert channel attacks are difficult to observe using conventional metrics. While analysing the attacks, it must be understood that the software attacks tend to modify the process environment or behaviour. Such changes can be captured at either ends, be it on the attacker's process or on the victim's process, and can be mapped to an attack. In memory based attacks, its detection is highly dependent on the precision of the measured runtime characteristics, along with the monitor's ability to distinguish between benign behaviour and attacks. So, the efficiency and success of any tool that detects a stealth attack would depend on its effectiveness in monitoring difference between a benign behaviour and a malicious behaviour [112].

2.7.3 Stealthy Denial of Service (DoS) attacks

There is a long history of Denial of Service (DoS) attacks on web services such as Amazon, MasterCard and PayPal, leading to heavy economic losses. Ficco et al. [69] have defined a low-rate DoS attack as a form of stealth attack. This low-rate DoS attack is carried out by directing packets flow at a rate low enough to avoid DoS detection. Its primary objective is to avoid detection mechanism by exploiting application level vulnerabilities. As most of the systems are designed to protect against normal high-rate DoS attacks, the attackers may choose to employ the low-rate DoS attacks repeatedly to avoid detection as well as cause enough damage similar to normal DoS attacks.

The use of Intrusion Detection or Prevention System (IDS or IPS) is ineffective to handle such attacks. To handle such attacks, an intrusion tolerant approach has been proposed by [69], designed to mitigate any service level unavailability resulting out of low-level DoS attack. Some of these stealth attacks may be launched by "querying a service using a very large request message resulting in high memory consumption" or "starting decryption of many encrypted messages resulting in high CPU load" [69].

Jakobsson [81] has, however, shown that when an adversary wants to disconnect the network, it can do so by launching a stealth version of the common DoS attack that modifies the behaviour of some nodes by fooling them into making illegal entries in their respective routing tables.

2.7.4 Miscellaneous Stealth Attacks Scenarios

BGP hijacking takes place when the Internet service providers (ISPs) forwards the IP prefixes without filtering them, leading to illegitimate takeover of some IP addresses. It has been demonstrated that HTTPS can be broken by BGP hijacking [70]. Birge-Lee et al. [25] further elaborated that the adversaries can use BGP to more stealthily hijack only part of the internet [70], but these attacks are also limited in that they require the adversary to have a specific location in the internet topology. Research [25] shows how an adversary in any location can perform a similarly stealthy attack. The authors have concluded that it is much easier to perform a stealthy attack against a certificate authority (CA) than previously anticipated.

Gruss et al. [73] have developed *Flush+Flush* attack on cache without making any memory access, contrary to any other cache attack. These attacks are claimed to be stealthy in nature as it cannot be detected via cache hits and misses. Similarly, Wagner and Soto [139] have presented a technique to bypass the detection from IDS by studying the pattern of IDS detection mechanism. They have come up with mimicry attacks that is able to evade detection by intrusion detection systems (IDS). These mimicry attacks can also be classified as a stealthy attack since the attack goes undetected.

Many researchers have extensively worked on stealth attack on security protocols on mobile ad hoc networks [81, 118]. The stealth attack have also been classified in two classes, first performing traffic analysis and second partitioning the network to reduce network throughput [118].

2.7.5 Detecting Stealth Attacks

Many studies have been undertaken to find out ways of detecting stealth attacks [4, 32]. In their work on log correlation for intrusion detection, Abad et al. [4] have shown structure of different attacks as reflected in various logs, based on data mining techniques. A wide range of Intrusion Detection or Prevention Systems (IDS or IPS) [87, 121] can detect attacks, and a lot of work has looked at ways to avoid these [105, 139]. Since a single log analysis is insufficient, as some attacks are likely to go unnoticed, it has been recommended that both the network and system logs act as the primary source of data for forensic analysis apart from intrusion detection and response [4].

The behaviour of any system can be analysed based on various metrics. With so many metrics, finding out a common reference behaviour may be a difficult task. This leads us to believe that rather than analysing the system behaviour, comparing the actual logs in traces generated by the protocols against the expected standard logs may be a better strategy for analysing the logs. While the efficiency of machine learning algorithm is dependent largely on the quality of model and training set data, it is not the case with automated analysis of logs in traces.

A stealthy attack may not necessarily belong to the class of severe attack, as the attacker may combine small unnoticed attacks multiple times to cause enough or even more damage than a single severe attack. Two interesting cases supporting this notion have been discussed by [69] and [146]. Analysis of data collected at US National Lab, the study found a recurring pattern of distributed brute-forcing attempts. These attacks were found to be targeting a wide range of machines, and it would have been easier to detect them using a detector with a universal view. At the same time, only aggregate view would be helpful in detecting some instances of stealthy attacks [82].

Bian et al. [23] have demonstrated a novel approach using machine learning to detect hosts in a network, also referred as target assets (TAs), that is victim of advanced persistent threats by analysing graph-based features such as network flows and host authentication logs among others. While their approach outperforms some other approaches, it is also limited by poor quality of the dataset. Such a limitation can be alleviated by using a framework independent of the dataset or a solution dealing with specific classes of interactions such as network security protocols and attacks thereon.

2.8 Notational Preliminaries

This section introduces various notations to be used throughout this thesis and provides background on *term rewriting* and *labeled multiset rewriting rules* used in Tamarin.

2.8.1 Term Rewriting

Rewrite systems perform a very simple task : perform computation by replacing equal terms in a given formula, and keep repeating till there are no terms left to be replaced in the set of directed equations. One of the most important properties for rewrite systems is that every term can be rewritten to a unique normal form. A rewrite rule provides a generic framework for basic actions executing parallel to each other in a concurrent system. Just like equational simplification, which can be performed in parallel independent of each other, rewriting can also be termed as concurrent rewriting [104]

In short, “Term Rewriting” provides a useful and powerful formalism tool that can be used in developing programming languages, automated deduction ad rewriting logic etc. Though equational deduction by undirected replacement of equals by equals can be very efficient, direct replacement (i. e., term rewriting) can be much easier and much faster [71]. Various standard notions related to “Term Rewriting” presented in this section follows from previous work presented in [62, 104, 122].More details on term rewriting is placed in section 5.4.

2.8.2 Labelled Multiset Rewriting

The equational theory E, used in a system, specifies all the cryptographic operations, protocol, and adversary capabilities in terms of what it can learn from the messages. Our system is a labelled multiset rewriting system $\mathcal{S}$ that supports generation of fresh names and persistent facts similar to one used in [122]. The verification tool Tamarin used in our work derives its input from the syntax and semantics of labelled multiset rewriting rules.

State and Facts The states of a transition system can be modelled as finite multisets of facts. Facts are constructed using terms over a fact signature. An unsorted signature $\Sigma_{\mathcal{F}}$ is categorised into linear and persistent fact symbols. The set of fact $\mathcal{F}$ consisting of all facts $F(t_1, ..., t_k)$ such $t_i \in \mathcal{T}$ and $F \in \Sigma^k_{\mathcal{F}}$. The set of ground facts, i. e., the set of facts not containing any variables, are denoted by $\mathcal{G}$. A fact $F(t_1, ..., t_k)$ is defined as linear or persistent, depending upon F either being linear or persistent, respectively. While the linear facts model resources that are consumable only once, persistent facts are used to model resources that can be consumed multiples times, still remaining in state.

Trace In our model, the traces (sequence of sets of facts) are defined by labelled multiset rewriting modulo E with $\mathcal{S}$. The security properties are verified by presence or absence of fact in traces at a certain time point. First-order formulas using predicate symbols are used to specify various security properties such as :

- $f@i$ means that the fact f is present in the trace at position i,$f \in_E tr[i]$ where i is a temporal variable and $tr[i]$ is the ith element of the trace tr.
- $i \prec j$ denotes that the time point i precedes the time point j.
- $i = j$ denotes that the time points i and j are equal.
- Similarly, $t \approx s$ denotes equality for terms t and s in equational theory E., i. e., $t =_E s$.

Multiset Rewrite Rule A labelled multiset rewriting rule is denoted by $l - [a] \rightarrow r$. It consists of a triple (l, a, r) with $l, a, r, \in \mathcal{F}^*$. Every rule of type $rule = l - [a] \rightarrow r$ has premises, actions, and conclusions with premises defined as $prems(rule) = l$, actions as $acts(rule) = a$ and conclusions as $concs(rule) = r$. A rule can be fired only when all the facts present in the premise of a rule are available in the system state. Upon firing of a rule, the facts present in the premise, i. e., l are consumed and replaced by those in conclusion, i. e., r. In the process, execution of a rule adds action a to the trace, which is an ordered list of $acts(rule)$ The set of such multiset label rewriting rules form a multiset rewriting system discussed in section 5.4.

2.9 Summary

In this chapter, we have introduced the notions of security properties along with attacks and their examples. This is followed by introducing the two models of formal protocol verification; *computational* and *symbolic* followed by a summary of automatic verification tool TAMARIN use cases. We found that, in most of the cases, a complete model of the protocol has been developed for security analysis. These analyses have uncovered many flaws and been helpful in improving the protocol design, however, the process has been time-consuming and complex. Hence, it begs the question whether it is possible to achieve a similar result by just modelling the attacks, and not the complete protocol? We would attempt to undertake such an experiment, of modelling just the attacks, and not the complete protocol specifications, and study its result in subsequent chapters.

Further, based on the overview of various scenarios presented in Section 2.7 explaining stealth attack and their detection mechanisms, we are now in position to summarise them and identify the research gaps. The stealth attack have mostly been defined as one which is difficult to detect and where the attacker focuses on avoidance of detection.

Our study found that most of the stealth attack studies have remained focused on industrial control system or cyber-physical systems, where the system states usually follow laws of Physics. Additionally, there are a variety of documented attacks on security protocols. Some studies have focused on man-in-the-middle attacks by proposing stealth attack on WPA2 encrypted Wi-Fi networks [8] and preventing such attacks [119].

As described in Section 2.7, many researchers have looked at attempting to discover from system logs that a stealthy attack is in progress, such as, using logs correlation in machine learning techniques. In our review of stealth attacks, we do not find any mention of the protocol trace analysis being used as a detection mechanism for the stealth attack. Further, here was no evidence of any formal modelling and analysis of the stealth attack, based on the protocol traces.

Since there has been no attempt on analysing the attacks on security protocols as stealth attack, we can consider the absence of any formal definition of *stealthiness of traces* as a research gap. Accordingly, in subsequent chapter, we propose to analyse the stealth attacks on security protocols based on the protocol trace analysis.

Part II

Modelling Attacks in a Formal Universe

Chapter Three

Modelling of Attacks on 802.11 4-Way Handshake

The usual approach of science of constructing a mathematical model cannot answer the questions of why there should be a universe for the model to describe. Why does the universe go to all the bother of existing?

Stephen Hawking

3.1 Motivation

There is an abundance of research related to formal analysis of protocols using automated protocol analysis tools such as Pro-Verif [29], Tamarin[103] etc. Such studies start by modelling a complete model of the protocol as per the standard specifications provided by the original authors or, in some cases, agencies such as IEEE etc. A comprehensive analysis of such protocols usually requires major effort and is a complex task [52, 53] considering all the possible combinations the model must be able to exhibit, in order to simulate a real-world execution. The security properties mandated by the protocols are then analysed in the presence of an attacker to see if the properties are satisfied or not.

We, however, rather than modelling the complete state machine, want to model only those actions sufficient to demonstrate an attack and analyse if such models can be used to analyse the security properties listed out in the protocol standard. We seek to test if there is merit in this technique, i.e., modelling of subset of actions enough to demonstrate an attack and analysis of security properties under the attack scenario. This chapter presents modelling of various attacks, such as KRACK [137] and Downgrade [131], on IEEE 802.11 *4-way handshake* as examples.

3.2 Contribution

The IEEE 802.11 standard defines a *4-way handshake* between a supplicant and authenticator for secure communication. Many attacks such as KRACK [137], cipher downgrades, and key recovery attacks have been recently discovered against it. These attacks raise the question whether the implementation violates one of the required security properties, or whether the security properties are insufficient. To the best of our knowledge, this is the first work that shows how to answer this question using formal methods by modelling only the attacks and not the complete protocol specification. We present in this chapter, modelling process for variety of these attacks and issues faced, using the Tamarin against the security properties mandated by the standard for the *4-way handshake.*

In this chapter, we present a novel methodology for modelling of attacks on 802.11 *4-way handshake* in order to verify its security properties as mandated in the standard. Our modelling approach is different from the normal use of formal methods for checking security protocols, which consists in defining a model of a protocol with its security properties to check for the existence of attacks. Instead, we use known attacks to model only a subset of protocol interactions which are sufficient to demonstrate such attacks. Such attack models will serve as foundation, to check if the security properties proposed in the standard are enough to ensure the security of the protocol, in the subsequent chapters.

3.3 Overview

Wireless networks are ubiquitous in domestic and corporate environments because they provide a convenient way to connect portable devices. However, security is a major concern, since a cybercriminal may not need physical access to eavesdrop or tamper with the communications between honest devices.

The original IEEE 802.11 standard [78], adopted in 1997, defined the Wired Equivalent Privacy (WEP) security algorithm, aimed at providing data confidentiality comparable to that of a traditional wired network. WEP was found vulnerable due to the use of a weak cipher, namely RC4, and the small size of its initialisation vector. Many attacks on WEP, such as key recovery attacks, have been published [132, 133]. Subsequently, WEP was replaced by Wi-Fi Protected Access (WPA) as an intermediate measure, before the final IEEE 802.11i amendment [79], commonly known as WPA2, was finalised in 2004.

The 802.11 standard [80] defines a *4-way handshake* as the key management protocol. It involves exchanging four messages between an access point (AP) and a client, or equivalently in 802.11 terminology, an authenticator and a supplicant. These messages enable parties to compute and share session/group keys for future unicast/multicast secure communication over the wireless medium. It also provides mutual authentication and session-key agreement.

This *4-way handshake* was proven formally secure [75, 76], and had no attacks published on it, until recently when the so-called Key Reinstallation Attack (KRACK) was uncovered by Vanhoef and Piessens in 2017 [137]. This attack exploits design and/or implementation flaws in the *4-way handshake* by reinstalling already in-use session or group keys. As a consequence, the adversary can break the security guarantees, even with a secure protocol for data confidentiality, such as the AES-based Counter Cipher Mode with Block Chaining Message Authentication Code Protocol (AES-CCMP), and decrypt or replay messages [137].

Moreover, various *4-way handshake* implementations have been found to be vulnerable to downgrade attacks in widely used routers [131], including models of Cisco and TP-Link. These attacks mostly affect the AP, when both the AP and the client support AES-CCMP and Temporal Key Integrity Protocol (TKIP) cipher suites. Although the client is always likely to choose the stronger AES-CCMP cipher suite over TKIP, an adversary can trick the AP into using TKIP.

We demonstrate the development models of *4-way handshake* using the security protocol verification tool TAMARIN. Our modelling focuses on the subset of functionalities and messages for successful execution of the attacks on *4-way handshake* and not building a complete model of the 802.11 state machines, thus enabling a Dolev-Yao adversary [64] to exploit the vulnerabilities. We show that TAMARIN can find the attacks mentioned above, and our models can formally verify that the suggested fixes to the vulnerabilities work as intended. The main contribution of this chapter is to present TAMARIN models of the 802.11 *4-way handshake* that exhibit several attacks [131, 137], that will be used later to suggest possible fixes, and demonstrate correctness of suggested fixes. We also show here how to use TAMARIN to encode the security property corresponding to a given attack. At the end, we also discuss some modelling issues that we have encountered.

3.4 Preliminaries

The IEEE 802.11 Standard.

This standard defines protocols for data confidentiality, mutual authentication, and key management, providing enhanced security at the medium access control (MAC) layer in wireless networks [80].

The original version of the standard [78] appeared in 1997, and defined the Wired Equivalent Privacy (WEP) security algorithm, based on the weak RC4 cipher. The

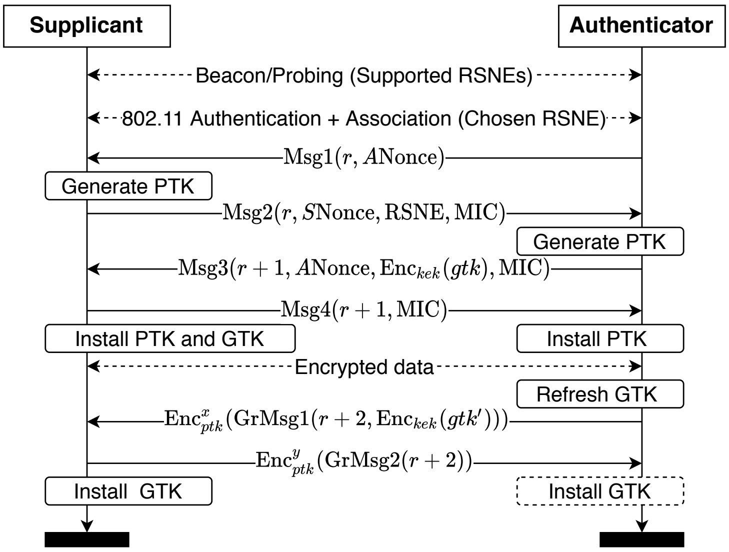

Supplicant
Authenticator
Beacon/Probing (Supported RSNEs)
802.11 Authentication + Association (Chosen RSNE)
$\text{Msg1}(r, A\text{Nonce})$
Generate PTK
$\text{Msg2}(r, S\text{Nonce}, \text{RSNE}, \text{MIC})$
Generate PTK
$\text{Msg3}(r+1, A\text{Nonce}, \text{Enc}_{kek}(gtk), \text{MIC})$
$\text{Msg4}(r+1, \text{MIC})$
Install PTK and GTK
Install PTK
Encrypted data
Refresh GTK
$\text{Enc}^{x}_{ptk}(\text{GrMsg1}(r+2, \text{Enc}_{kek}(gtk')))$
$\text{Enc}^{y}_{ptk}(\text{GrMsg2}(r+2))$
Install GTK
Install GTK

Figure 3.1: IEEE 802.11 standard *4-way handshake* and group key handshake

vulnerable WEP was replaced with Wi-Fi Protected Access (WPA), as an intermediate measure, before the IEEE 802.11i amendment (WPA2) [79] was released in 2004. WPA includes the use of a message authentication code algorithm, coined as Message Integrity Check (MIC), as well as the TKIP cipher suite, which allows a more secure per-packet key system compared to the fixed key system used by WEP. The Message Integrity Check (MIC) helps protect the authenticity of the EAP over LAN key frames (EAPOL-Key) [80]. The 802.11i amendment [79] and the current version of the standard [80] requires support of even more secure algorithm suites, discussed below. We summarise here the four stages of the 802.11 key generation process. We refer the reader to [80] for the full details.

- **Network Discovery.** In this stage, the clients search for available networks along with their parameters. Clients can either actively send and receive probes, or just observe the broadcast beacons passively to learn the supported cipher suites (e.g., TKIP and/or AES-CCMP), and version of WPA. This set of parameters is called a Robust Security Network Element (RSNE).
- **Authentication and Association.** In this step, the Pairwise Master Key (PMK) is derived at both ends. In WPA2-Personal mode, the PMK is derived using a Pre-Shared Key (PSK) with a length of 8 to 63 characters, the Service Set Identifier

(SSID), and the SSID length, while in WPA2-Enterprise mode, it is derived from a key generated by an Extensible Authentication Protocol (EAP), e.g., using 802.1X authentication [3]. The PMK is used later in the temporal keys generation. However, the real authentication is carried out during the *4-way handshake*. The client and the AP accept or reject the association request based on the AP agreeing to the client's choice of RSNE.

- **4-Way Handshake.** The *4-way handshake* takes place to agree on a fresh session key, namely the Pairwise Transient Key (PTK), and optionally the Group Temporal Key (GTK); see Fig. 3.1. PTK derivation [80, Sec. 12.7.1.7.5] uses the shared PMK, a supplicant nonce S_{Nonce}, an authenticator nonce A_{Nonce}, and both MAC addresses. The PTK can be refreshed after a fixed time interval, or at request from either party, by executing another *4-way handshake*. The PTK is split into a Key Confirmation Key (KCK), Key Encryption Key (KEK), and Temporal Key (TK). The KCK and KEK protect handshake messages, while the TK protects data frames through the data confidentiality protocol. The *4-way handshake* also transports the current GTK to the supplicant. The group key GTK is placed in the Key Data field encrypted with KEK. The frame is kept protected, using KCK and MIC, to preserve its authenticity [137]. Every message in the *4-way handshake* follows the layout of EAPOL-Key [80], and we use Msg$n$ to denote the nth message in the handshake. The authenticator starts the handshake and increments the replay counter on every message sent. The supplicant replies to messages using the received replay counter.

- **Group Key Handshake.** The standard allows for refreshing the GTK regularly, using a group key handshake, to ensure that only active clients are in possession of it. This process is initiated by the authenticator sending group message 1, denoted GrMsg1, to all clients. The clients reply, in turn, with group message 2, GrMsg2, with the received replay counter; see Fig. 3.1.

- **Data Confidentiality and Integrity Support.** The standard defines several data confidentiality suites such as AES-CCMP and AES-GCMP as mandatory, but also TKIP for backwards interoperability with WPA [80]. All suites include message integrity of the data frames. For brevity, we use the same notation as in [137] to denote an encrypted frame $\mathrm{Enc}_k^n()$, being n the nonce (replay counter) in use, and k the key, i.e., PTK for unicast and GTK for broadcast messages.

We note that our focus is mainly on the attacks to the *4-way handshake*. Therefore, the authentication and association stages are out of the scope of our work, and we will assume that the PMK is already available at both ends.

3.5 Fundamentals of TAMARIN PROVER

Tamarin is a state-of-the-art tool for symbolic verification and automated analysis of security properties in protocols, under the Dolev-Yao model [64], with respect to an unbounded number of sessions. In addition to the security protocol model specifying the actions taken by various agents participating in the protocol in various roles, Tamarin also expects specifications of adversary as well as protocol's security guarantees in the form of properties. It then tries to construct a proof demonstrating that the protocol meets the desired security properties, even in the presence of an adversary and, even when multiple arbitrary instances of the protocol runs are interleaved in parallel. The cryptographic messages in Tamarin are modelled as order-sorted term algebra.

To analyse a protocol using Tamarin, one has to start by using its specification language to build an abstract model of the protocol, honest participants and adversary who controls the network. Further, desired protocol security properties are encoded using the same specification language followed by constructing the proofs for the specified properties. Tamarin employs *multiset rewriting system* to represent the protocol executions where a security protocol is defined by a set of roles with each defined as a set of multiset rules.

Each rule represents an event in the protocol, such as sending or receiving of the message.

Tamarin can handle protocols with unrestricted global states and unbounded sessions. Sometimes, however, the user may have to provide auxiliary lemmas for complex protocols in order to help the tool terminate. It provides a specification language for the symbolic modelling and analysis of security protocols by facilitating the building of highly detailed models of security protocols, the security guarantees offered by the protocols, and all powerful Dolev-Yao adversary.

Though Tamarin is proven sound and complete, as the protocol security problem is undecidable [67], Tamarin may not terminate every time owing to the possible infinite number of messages, sessions, and nonces, and may require human interventions. To support such interventions, Tamarin offers a GUI based interactive mode for the users to guide the tool manually in its proof [89]. We provide a brief overview of Tamarin in this subsection, however we refer the reader to the Tamarin manual [55] for additional details.

Adversary Knowledge:

We consider an all powerful adversary called Dolev-Yao adversary [64] that controls the network. It can read, intercept and send all the messages on the network and is limited in its capability only by cryptographic constraints. Some researchers like Creese et al. [115] have argued that Dolev-Yao model presents a worst-case scenario which is very difficult if not impossible in real world assumptions. At the same time, they also agree that it is better for a protocol analysis to show failure in a strict environment rather than achieve their security goals in a restricted threat model [115].

Tamarin models the knowledge of an adversary (Dolev-Yao [64]) denoted by the persistent fact $!\mathsf{K}(t)$ specifying t is known to the adversary. A linear fact $Out(t)$ specifies message output by the protocol t to the public channel and adversary can read as well retain t in its knowledge with persistent fact $!\mathsf{K}(t)$ added to the trace. Another linear fact

$In(t)$ specifies sending of a message. All messages received by the protocol are assumed to be sent by the adversary, i.e., the adversary will change an *Out* from one protocol rule to an *In* to be consumed by another protocol rule using its knowledge !K. All the message deduction (MD) rules available to adversary are presented below [123]:

$$\begin{cases} \mathit{Receive}\frac{\mathit{Out}(t)}{!\mathsf{K}(t)} \quad \mathit{Send}\frac{!\mathsf{K}(t)}{\mathit{In}(t)}[!\mathsf{K}(t)] \\ \frac{\mathit{Fr}(n{:}\mathit{fresh})}{K(n{:}\mathit{fresh})} \quad \mathit{Public}\frac{}{K(t{:}\mathit{pub})} \quad \mathit{Fresh}\frac{}{\mathit{Fr}(n{:}\mathit{fresh})} \\ \mathit{Derive}\frac{K(t_1)...K(t_k)}{K(\mathsf{c}(t_1,...,t_k))} \text{ for all } (\mathsf{c} : k) \in \Sigma \end{cases}$$

TAMARIN allows expressing security properties as temporal, guarded first-order formulas, modelled as trace properties. The construct $F@i$ states the presence of the fact F at a time point i. A property can be specified as a *lemma* to be tested if it holds or not, and enforced as a *restriction*, while testing other lemmas in presence of this property [103]. a special feature of TAMARIN is its ability to exclude some traces from consideration using the restriction feature. The security properties, lemmas, and restrictions are expressed using first order logic formula and modelled as trace properties. The security claims are verified against the transition system traces result in proof constructed by TAMARIN either verifying or falsifying the same using the *Lemma* feature.

Security Lemma :

Security properties are described in TAMARIN using first-order logic and consist of observable actions present in the trace. To argue that the shared secret is not available to the attacker, the TAMARIN manual [55] suggests the following lemma verifying that it is not possible for somebody to claiming that they have set up a shared secret and for the adversary to know the key. Here, '#i' and '#j' refer to time-points 'i' and 'j' of the event.

```
lemma nonce_secrecy:
 "not( Ex A B s #i. Secret(A, B, s) @ i & (Ex #j. K(s) @ j))"
```

Restrictions on Traces :

Tamarin has the *restriction* feature, which allows a property to be enforced on the traces. This feature is essential for formal analysis of a protocol, to verify if enforcing particular security properties would stop an attack. To the best of our knowledge, other tools such as ProVerif do not offer this feature and hence are not suitable to our approach. The 'restriction' feature in Tamarin can restrict the set of traces to be considered in the protocol analysis, e.g. the restriction below will only allow action labels 'Eq(x,y)' satisfying the condition 'x=y', i. e., 'Eq(5,5)' can appear in the trace while 'Eq(5,6)' cannot.

```
restriction Equality: "All x y #i. Eq(x,y) @i ==> x = y"
```

From now on, we will be using the first-order logic notations to describe our lemmas and restrictions. E.g., the lemma nonce_secrecy will be written as:

$$\neg(\exists\ A, B, s, i.\ \ \mathrm{Secret}(A, B, s)@i \wedge (\exists\ j.\ \mathrm{K}(s)@j)). \tag{nonce\_secrecy}$$

Similarly, the restriction equality will be written as:

$$\forall\ x, y, i.\ \ \mathrm{Eq}(x, y)@i\ \Rightarrow\ x = y. \tag{equality}$$

3.5.1 The SAPiC Front End.

Tamarin provides a SAPiC front-end, which allows specifying Tamarin models using processes. We provide a brief overview of SAPiC here, but we refer the reader to [89, 103] for further reference. SAPiC parses descriptions of protocols in an extension of the applied pi-calculus [30], called *stateful applied pi-calculus*, and converts them into *(labelled) multiset rewriting rules* (MSRs) to be analysed by Tamarin.

$\langle P,Q \rangle ::=$	processes
$\mid 0$	terminal (null) process
$\mid P \mid Q$	parallel composition of P and Q
$\mid\ !P$	replication of P
$\mid \nu\ a;\ P$	binds a to a new fresh value in P
$\mid \mathsf{out}(m, t);\ P$	outputs message t on channel m
$\mid \mathsf{in}\,(m, t);\ P$	inputs of message t on channel m
$\mid \mathsf{if}\ \ Pred\ \mathsf{then}\ P\ [\mathsf{else}\ Q]$	P if predicate $Pred$ holds; otherwise Q
$\mid \mathsf{event}\ F;\ P$	executes event (action fact) F
$\mid P + Q$	non-deterministic choice
$\mid \mathsf{insert}\ \ m,\ t;\ P$	inserts t at memory cell m
$\mid \mathsf{delete}\ \ m;\ P$	deletes the content m
$\mid \mathsf{lookup}\ m\ \mathsf{as}\ x\ \mathsf{in}\ P\ [\mathsf{else}\ Q]$	if m exists, bind it to x in P; otw. Q
$\mid \mathsf{lock}\ \ m;\ P$	gain exclusive access to cell m
$\mid \mathsf{unlock}\ \ m;\ P$	waive exclusive access to m
$\mid [L] \ \text{–}[A]\!\rightarrow [R];\ P \quad (L, R, A \in \mathcal{F}^*)$	provides access to TAMARIN MSRs

Figure 3.2: SAPiC syntax ($a \in \mathcal{FN}$, $x \in \mathcal{V}$, $m,t \in \mathcal{T}$, $F \in \mathcal{F}$)

Fig. 3.2 describes the SAPiC syntax. The calculus comprises an *order-sorted term algebra* with infinite sets of publicly known names $\mathcal{PN}$, freshly generated names $\mathcal{FN}$, and variables $\mathcal{V}$. It also comprises a signature Σ, i.e., a set of function symbols, each with an arity. The messages are elements of a set of terms $\mathcal{T}$ over $\mathcal{PN}$, $\mathcal{FN}$, and $\mathcal{V}$, built by applying the function symbols in Σ.

The set of facts is defined as $\mathcal{F} = \{F(t_1, \ldots, t_n) \mid t_i \in \mathcal{T}, F \in \Sigma \text{ of arity } k\}$. The special fact $K(m)$ states that the term m is known to the adversary. For a set of roles, the TAMARIN MSRs define how the system, i.e., protocol, can make a transition to a new state. An MSR is a triple of the form $[L]$ –$[A]\!\rightarrow$ $[R]$, where L and R are the premise and conclusion of the rule, respectively, and A is a set of action facts, modelled by SAPiC events. For a process P, its trace $\mathrm{Tr}(P) = [F_1, \ldots, F_n]$ is an ordered sequence of action facts generated by firing the rules in order.

3.6 Formal Models of the 802.11 4-Way Handshake Attacks

We present some variants of the KRACK attacks, exploiting nonce reuse [137], and a downgrade attack from [131]. Along with the attack steps, we also highlight some relevant details of our SAPiC models for the attacks and for the security lemmas corresponding to each one. Some details, e.g., MIC and the usage of cipher suites in encryption are not relevant in the attack modelled by us and hence have been abstracted. The complete source for the models and mechanised proofs are available at [127]. For the sake of brevity, we have also placed the complete SAPiC code of KRACK attack discussed in Figure 3.3 in Appendix B.1.

3.6.1 KRACK Attacks

The KRACK attacks exploit vulnerabilities in the 802.11 key management protocols [137]. An adversary tricks a victim into reinstalling an already used key by dropping, delaying or altering the order of the *4-way handshake* messages between two honest principals. On every key installation, the standard mandates that the replay counter (nonce) of the data confidentiality protocol be reset. The adversary can collect different encrypted messages using the same key and nonce: messages sent after the initial key installation, and messages sent after the key reinstallation. The adversary can then use this information to attack the data confidentiality protocol. The practical implications of the attack may enable the adversary to replay, decrypt or even forge the data packets, depending on the choice of the cipher suite (e.g., TKIP, AES-CCMP,AES-GCMP). We refer the reader to [85, 137] for the detailed consequences of the attack.

The underlying causes of the attacks are the unclear standard specifications, such as the authenticator accepting any replay counter previously used in the *4-way handshake*,

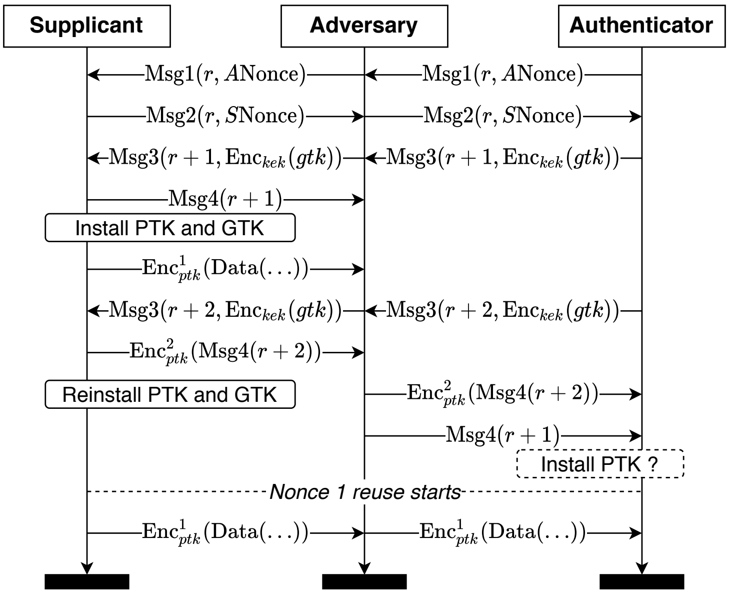

Supplicant
Adversary
Authenticator
Msg1(r, ANonce)
Msg1(r, ANonce)
Msg2(r, SNonce)
Msg2(r, SNonce)
Msg3($r+1$, $\mathrm{Enc}_{kek}(gtk)$)
Msg3($r+1$, $\mathrm{Enc}_{kek}(gtk)$)
Msg4($r+1$)
Install PTK and GTK
Enc^1_{ptk}(Data(...))
Msg3($r+2$, $\mathrm{Enc}_{kek}(gtk)$)
Msg3($r+2$, $\mathrm{Enc}_{kek}(gtk)$)
Enc^2_{ptk}(Msg4($r+2$))
Reinstall PTK and GTK
Enc^2_{ptk}(Msg4($r+2$))
Msg4($r+1$)
Install PTK ?
Nonce 1 reuse starts
Enc^1_{ptk}(Data(...))
Enc^1_{ptk}(Data(...))

Figure 3.3: KRACK - plaintext retransmission of message 3 after PTK install

not only the latest one [80, Sec. 12.7.6.5]. However, in practice, many APs fail to validate it, and imprudently accept an older replay counter. Subsequently, after discovery of vulnerabilities leading to KRACK attacks, the vendors were notified who have released patches to secure their older devices against these vulnerabilities. It may, however, be noted that the patches may themselves be flawed, or the attack can still succeed if the patched AP is connected to an unpatched vulnerable client [137].

We have successfully modelled several KRACK attacks exploiting the retransmission of message 3 and forcing nonce reuse [137]. We remark that the goal of our models is not to verify the compromise of the data confidentiality protocol. Instead, we aim at detecting the sufficient conditions that allow an adversary to exploit it, i.e., reinstallations of the same key.

Retransmission of Message 3 after PTK Install.

This variant of KRACK [137, Sec. 3.3] occurs when the supplicant accepts plaintext retransmission of message 3, even after a PTK has been installed. The message flow of the attack is shown in Fig. 3.3, and the outline of our model of the supplicant and

```
Supplicant :=
$\nu\ S_{\text{id}}$; out($S_{\text{id}}$);
!( in($A_{\text{id}}$);
  in(Msg1($r, A_{\text{Nonce}}$));
  $\nu\ S_{\text{Nonce}}$;
  let $ptk$ = CalcPtk($pmk, A_{\text{Nonce}}, \ldots$) in
  out(Msg2($r, S_{\text{Nonce}}$));
  in(Msg3($r + 1, A_{\text{Nonce}}, \text{Enc}_{kek}(gtk)$));
  event Running($S_{\text{id}}, A_{\text{id}}, pars$);
  out(Msg4($r + 1$));
  event S_InstallsPtk($S_{\text{id}}, ptk$);
  event S_InstallsGtk($S_{\text{id}}, gtk$);
  (( event Commit($S_{\text{id}}, A_{\text{id}}, pars$)
  ) +
  ( in(Msg3($r + 2, A_{\text{Nonce}}, \text{Enc}_{kek}(gtk)$));
  event Running($S_{\text{id}}, A_{\text{id}}, pars$);
  out($\text{Enc}_{ptk}$(Msg4($r + 2$)));
  event S_InstallsPtk($S_{\text{id}}, ptk$);
  event S_InstallsGtk($S_{\text{id}}, gtk$);
  event Commit($S_{\text{id}}, A_{\text{id}}, pars$)
  )))
```

```
Authenticator :=
$\nu\ A_{\text{id}}$; out($A_{\text{id}}$);
!( in($S_{\text{id}}$);
  $\nu\ r$;
  $\nu\ A_{\text{Nonce}}$;
  out(Msg1($r, A_{\text{Nonce}}$));
  let $ptk$ = CalcPtk($pmk, \ldots$) in
  in(Msg2($r, S_{\text{Nonce}}$));
  $\nu\ gtk$;
  event Running($A_{\text{id}}, S_{\text{id}}, pars$);
  event A_InstallsGtk($gtk$);
  out(Msg3($r + 1, A_{\text{Nonce}}, \text{Enc}_{kek}(gtk)$));
  (( in(Msg4($r + 1$));
  event A_InstallsPtk($ptk$);
  event Commit($A_{\text{id}}, S_{\text{id}}, pars$)
  ) +
  ( out(Msg3($r + 2, A_{\text{Nonce}}, \ldots$));
  in($\text{Enc}_{ptk}$(Msg4($r + 2$)));
  event A_InstallsPtk($ptk$);
  event Commit($A_{\text{id}}, S_{\text{id}}, pars$)
  )))
```

Figure 3.4: Model outline for supplicant and authenticator vulnerable to KRACK attack based on plaintext retransmission of message 3

authenticator are in Fig. 3.4. Note that we prepend 'S_' and 'A_' to the events executed at the supplicant and authenticator, respectively. The main process is defined as $\nu\ pmk$; (!Supplicant | Authenticator), instantiating an arbitrary number of supplicant processes. Our model computes the PTK [80, Sec. 12.7.1.7.5] with the identifiers A_{id}, S_{id} acting as the MAC addresses as follows:

$$ptk = \text{CalcPtk}(pmk, A_{\text{Nonce}}, S_{\text{Nonce}}, A_{\text{id}}, S_{\text{id}}).$$

The adversary sits between the supplicant and the authenticator to perform a man-in-the-middle (MITM) attack, and forwards messages 1-3 normally. Initial PTK install is captured by the event S_InstallsPtk(S_{id}, ptk), after which the supplicant can send encrypted frames using the encryption key TK associated to PTK. Message 4 is blocked from reaching the authenticator by the adversary. The model uses the non-deterministic choice in the authenticator process via the + operator from the SAPiC calculus. Therefore, it captures either the reception of message 4, and installs the PTK, or timeouts and retransmits message 3 with an updated replay counter, and waits again for the confirmation.

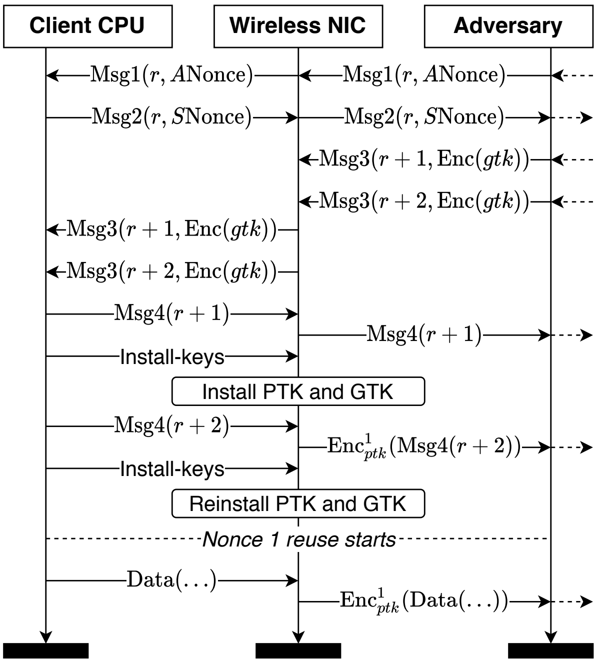
Similarly, in order to capture the fact that the state machine of the supplicant accepts plaintext retransmission of message 3, we also branch the supplicant process, in order to capture traces completing a normal run of the protocol, and traces with an adversary blocking message 4. This latter case matches the attack scenario, with the supplicant reinstalling an already in-use PTK (and GTK). It follows that the next data frames sent by the supplicant will be encrypted with a reused nonce. Our model, therefore, is aimed at capturing the traces with key reinstallations on the supplicant side using the same PTK already installed.

Retransmission of Message 3 before PTK Install.

This KRACK attack has two variants with the supplicant accepting either a plaintext or encrypted retransmission of message 3 with the PTK yet to be installed [137, Sec. 3.4].

The first case is shown in Fig. 3.5. This attack assumes that the authenticator performs its actions as expected. The first two messages are transmitted normally. However, the original message 3 is blocked by the adversary while he waits for the retransmission of message 3. Both messages are then forwarded to the supplicant. This triggers a race condition between the CPU and the network interface controller (NIC), which causes that the same key be reinstalled. In our model for this attack, Fig. 3.5b, the supplicant comprises both the NIC and the CPU, and it considers two branches in order to capture an implementation vulnerable to the attack: one where the *4-way handshake* follows the normal course, and another where the attacker is able to cause key reinstallation.

The second case of this attack is presented in Fig. 3.6. The main difference is

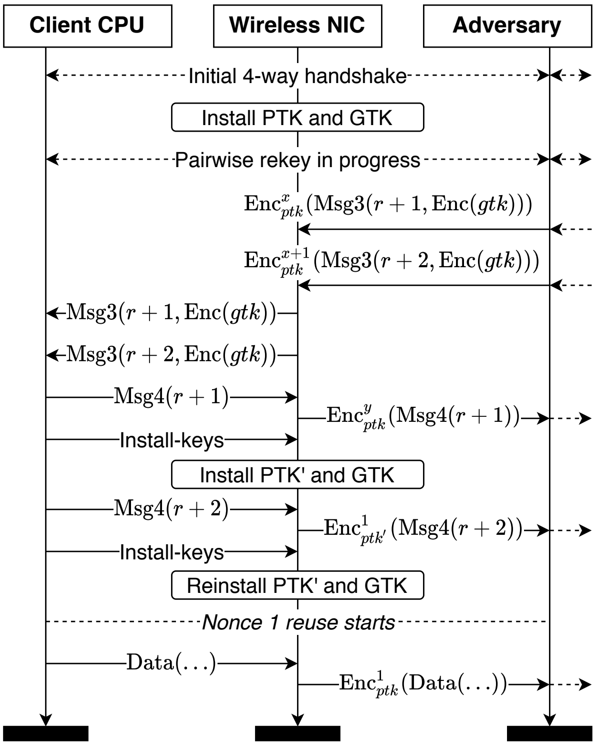


(a) Message sequence diagram

```
Supplicant =
!(
 // Messages 1-3 processed normally
 ((
 // No other msg. recv. in between
 out(Msg4($r + 1$));
 event S_InstallsPtk($S_{\mathrm{id}}$, $ptk$)
 ) + (
 // Retransm. message 3 recv.
 in(Msg3($r + 2$, $A_{\mathrm{Nonce}}$, Enc($gtk$)));
 out(Msg4($r + 1$));
 event S_InstallsPtk($S_{\mathrm{id}}$, $ptk$);
 out(Enc$_{ptk}$(Msg4($r + 2$)));
 event S_InstallsPtk($S_{\mathrm{id}}$, $ptk$)
 )))
```

(b) Model outline

Figure 3.5: KRACK - plaintext retransmission of message 3 before PTK install



(a) Message sequence diagram

```
Supplicant =
!( // Initial 4-way handshake
 event S_InstallsPtk($S_{\mathrm{id}}$, $ptk$);
 !( // PTK rekey
 // Messages 1-3 processed normally
 (( // No other msg. received
 out(Enc$_{ptk}$(Msg4($r + 1$)));
 event S_InstallsPtk($S_{\mathrm{id}}$, $ptk'$)
 ) + ( // Retransmitted msg. 3 recv.
 in(Enc$_{ptk}$(Msg3($r + 2, \ldots$)));
 out(Enc$_{ptk}$(Msg4($r + 1$)));
 event S_InstallsPtk($S_{\mathrm{id}}$, $ptk'$);
 out(Enc$_{ptk'}$(Msg4($r + 2$)));
 event S_InstallsPtk($S_{\mathrm{id}}$, $ptk'$)
 ))))
```

(b) Model outline

Figure 3.6: KRACK - encrypted retransmission of message 3 before PTK install

that it can only be executed during the PTK rekey phase. After an initial successful handshake, both principals install a PTK. During the PTK rekey process, the adversary follows the same strategy as above: it waits for a retransmission of message 3. This time, the messages are encrypted under the installed PTK, but the adversary is able to identify what particular message is being sent (e.g., by timeouts or message lengths). By appropriately delaying and forwarding the messages, the adversary causes a reinstall of the PTK being refreshed, ptk'. Our model (Fig. 3.6b) captures an arbitrary number of PTK rekey negotiations, and, again, it branches non-deterministically to capture the transitions of a supplicant state machine vulnerable to the attack.

For all three cases above (Figs. 3.3, 3.5 and 3.6), we query for the absence of KRACK attacks with lemma: "given an installation of PTK by the supplicant, it is not the case that there exists an earlier installation with the same PTK,"

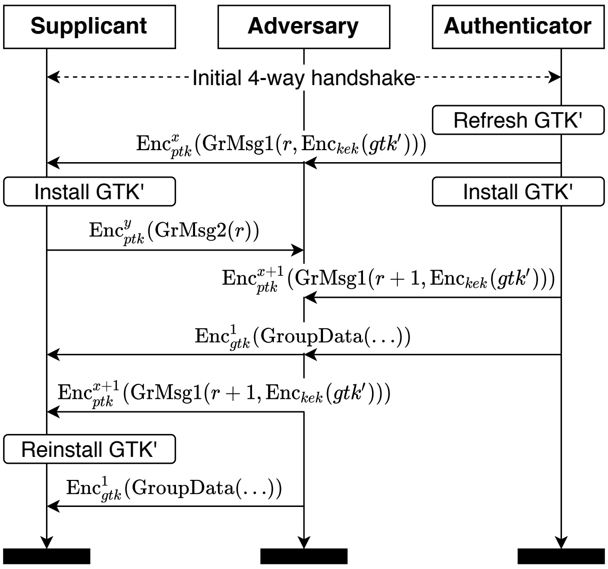
$$\begin{aligned} &\forall id, ptk, t_1.\ \text{S\_InstallsPtk}(id, ptk)@t_1 \Rightarrow \\ &\qquad \neg(\exists t_2.\ \text{S\_InstallsPtk}(id, ptk)@t_2 \wedge (t_2 < t_1)). \end{aligned} \tag{NoKrackPtk}$$

The events S_InstallsPtk are placed in the parts of the model where the primitive MLME-SETKEYS.request [80] is called, which causes nonce reset.

As expected, our TAMARIN models [127] falsify Lemma (NoKrackPtk), proving the existence of KRACK, allowing an adversary to cause key reinstall, nonce reuse and break the security guarantees of the data confidentiality protocol.

Attack Against the Group Key Handshake.

This variant of the KRACK attack targets the group key handshake, and tricks the supplicant into reinstalling a GTK, rather than a PTK [137, Sec. 4.1]. The attack is shown in Fig. 3.7. Note that the group key handshake runs encrypted by the already installed PTK. The standard requires that the supplicant install the GTK upon receipt of group message 1, regardless of whether it is a retransmission or not, and reply with


Supplicant
Adversary
Authenticator
Initial 4-way handshake
Refresh GTK'
$\mathrm{Enc}^{x}_{ptk}(\mathrm{GrMsg1}(r, \mathrm{Enc}_{kek}(gtk')))$
Install GTK'
Install GTK'
$\mathrm{Enc}^{y}_{ptk}(\mathrm{GrMsg2}(r))$
$\mathrm{Enc}^{x+1}_{ptk}(\mathrm{GrMsg1}(r+1, \mathrm{Enc}_{kek}(gtk')))$
$\mathrm{Enc}^{1}_{gtk}(\mathrm{GroupData}(\ldots))$
$\mathrm{Enc}^{x+1}_{ptk}(\mathrm{GrMsg1}(r+1, \mathrm{Enc}_{kek}(gtk')))$
Reinstall GTK'
$\mathrm{Enc}^{1}_{gtk}(\mathrm{GroupData}(\ldots))$

(a) Message sequence diagram

```
Supplicant =
!(
 // Initial 4-way handshake
 !(
 // Group key handshake
 in(Enc_ptk(GrMsg1(r',
 Enc_kek(gtk'))));
 // If r' already used, finish thread
 event S_InstallsGtk(S_id, gtk');
 out(Enc_ptk(GrMsg2(r')))
 ))
```

(b) Model outline

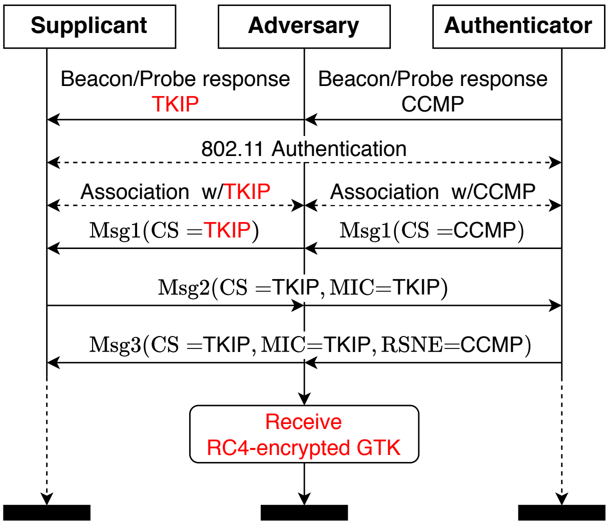
Figure 3.7: KRACK against group key handshake

group message 2. The adversary delays group message 2 from reaching the authenticator, triggering retransmission of group message 1. Now, the adversary forwards both versions of group message 1 to the supplicant, which causes a GTK install and subsequent reinstall. This will allow the attacker to replay group data frames to the supplicant [137].

To capture the reinstall of the GTK, Tamarin falsifies the following lemma stating that "given an installation of GTK by the supplicant, it is not the case that there exists an earlier installation with the same GTK,"

$$\forall id, gtk, t_1.\ \mathrm{S\_InstallsGtk}(id, gtk)@t_1 \Rightarrow$$
$$\neg(\exists t_2.\ \mathrm{S\_InstallsGtk}(id, gtk)@t_2 \wedge (t_2 < t_1)). \qquad \text{(NoKrackGtk)}$$

Our model (Fig. 3.7b) captures a scenario with a supplicant accepting arbitrary number of executions of the group key handshake, as long as the group message 1 has an increased replay counter. We note that for this model we assume an initial valid *4-way handshake* without exhibiting PTK reinstall. This modelling issue is discussed in Sec. 3.7.



(a) Message sequence diagram

```
Authenticator =
!( ν tid; // Thread ID
 in(⟨'AuthReq', $S_{\mathrm{id}}$⟩);
 out(⟨'AuthResp', $S_{\mathrm{id}}$, 'CCMP'⟩);
 in(⟨'AssocReq', $S_{\mathrm{id}}$, 'CCMP'⟩);
 out(⟨'AssocResp', $S_{\mathrm{id}}$, 'CCMP'⟩);
 // 4-way handshake
 event A_Starts($tid$, 'CCMP');
 out(Msg1($r$, $A_{\mathrm{Nonce}}$, 'CCMP'));
 in(Msg2($r$, $S_{\mathrm{Nonce}}$, $cs$));
 out(Msg3($r+1$, $A_{\mathrm{Nonce}}$,
 $\mathrm{Enc}_{\langle cs,kek\rangle}(gtk)$));
 event A_SentMsg3($tid, cs$);
 // (...)
)
```

(b) Model outline

Figure 3.8: Downgrade Attack on 802.11 (TP-Link WP841P)

3.6.2 Cipher Suite Downgrade

The downgrade attack we consider [131] is limited to the authenticator-side only. In a correct implementation, a client should be able to detect this attack easily by observing inconsistencies in the RSNE information. Recall from Sec. 3.4 that the RSNE information is selected in the association stage in plaintext, and subsequently encrypted and transmitted as part of message 3, as shown in Fig. 3.1. The supplicant must verify that the RSNE information observed in the association stage matches with the authenticated contents of message 3, and it should terminate the handshake otherwise.

In a downgrade attack, depicted in Fig. 3.8, the adversary forces GTK encryption with a weak cipher suite (RC4), rather than the intended strong cipher suite (AES-CCMP). The attack was discovered on the access point TP-Link WP841P [131, Sec. 5.2]. The authenticator advertises support for AES-CCMP during the association stage. However, it will follow the supplicant in switching the cipher suite in mid-handshake process, accepting the TKIP-based message 2.

An adversary acts as a MITM by negotiating the AES-CCMP suite with the authenticator, and TKIP with the supplicant, as message 1 is in plain. The supplicant calculates the PTK and replies with message 2 using the TKIP suite. The authenticator accepts the message, overrides its initial AES-CCMP selection, and responds with a well-formed TKIP message 3 containing the GTK encrypted with RC4. The adversary can now exploit the weakness of this cipher to recover the GTK [136]. The RSNE mismatch can be easily detected on forwarding of message 3 to the supplicant, which can drop the connection. Unfortunately, by this time, the adversary is already in possession of the RC4-encrypted GTK.

Encryption with different cipher suites can be modelled, e.g., with a signature Enc', Dec' indicating the cipher suite cs as an additional parameter. Then,

$$\forall m, k, cs.\ \mathrm{Dec}'_k(\mathrm{Enc}'_k(m, cs), cs) = m.$$

Note, that this theory is semantically equivalent to the usual symmetric encryption using as key the tuple $k' = \langle cs, k \rangle$, because $\mathrm{Dec}_{\langle cs,k \rangle}(\mathrm{Enc}_{\langle cs,k \rangle}(m)) = m$.

Our Tamarin model queries that "for each run of the protocol, the cipher suites used by them is the same," implying that a change of the cipher suite in between a run is impossible. As expected, the lemma below is falsified:

$$\forall tid, cs_1, cs_2, t_1, t_2.\ \mathrm{A\_SentMsg3}(tid, cs_1)@t_1 \wedge \mathrm{A\_Starts}(tid, cs_2)@t_2 \Rightarrow (cs_1 = cs_2). \qquad \text{(NoDowngrade)}$$

3.7 Modelling Issues

This section presents some modelling issues encountered and their handling in our models.

Data Confidentiality Protocol: Once a PTK has been agreed and installed, all traffic is symmetrically encrypted at the link-layer using the data confidentiality protocol [79]. This also applies to messages sent during PTK or GTK rekeying through the *4-way handshake* or the group key handshake. This can be easily modelled in TAMARIN with the symmetric encryption built-in theory. However, this has the undesired side effect that TAMARIN tries to derive arbitrary terms from the outputs of the protocol once the first encryption is sent to a process. That is, TAMARIN believes that it can use the honest processes as oracles for arbitrary encrypted terms using a key that the tool tries to derive unsuccessfully. Technically speaking, this causes that the tool has *partial deconstructions* left, complicating the proof of a lemma by either taking a very long time, or not terminating [102].

A *sources lemma* is one of the possibilities to guide the proof and help TAMARIN terminate. We define the following templates for inputs and outputs associated to the data confidentiality protocol:

// Template for outputs	*// Template for inputs*
event $\mathrm{DcpSend}(id, m, k)$;	in $(\mathrm{Enc}_k(m))$;
out$(\mathrm{Enc}_k(m))$;	event $\mathrm{DcpReceive}(id, m, k)$;

These events allow us to define the following sources lemma that eliminates the partial deconstructions left:

$$\forall id_1, m, k, t_1.\ \mathrm{DcpReceive}(id_1, m, k)@t_1 \Rightarrow$$
$$(\exists t_2.\ K(m)@t_2 \wedge (t_2 < t_1)) \vee (\exists id_2, t_2.\ \mathrm{DcpSend}(id_2, m, k)@t_2).$$

Indeed, the lemma states the obvious fact that whenever a party receives a message m encrypted with key k, then m was either known by adversary, or it was encrypted by an honest party. As honest parties do not leak secrets in our model, the lemma helps TAMARIN 'realising' that the outputs of the data confidentiality protocol can not be used by the adversary to gain any extra information.

Replay Counters: An important part to model in the 802.11 key management protocols is the use of replay counters. We model this feature for the attacks covering PTK and GTK rekeying (Fig. 3.6 and 3.7). The event ReplayCtr is placed at the locations where the supplicant should check the counter, e.g.,

$$\begin{aligned}&\mathsf{in}\,(\mathrm{Enc}_{tk}(\mathrm{Msg1}(r, A_{Nonce})));\\&\mathsf{event}\ \mathrm{ReplayCtr(r)};\end{aligned}$$

Then, the restriction below models a supplicant that accepts a message only if its replay counter is larger than the last one received.

$$\forall r_1, r_2, t_1, t_2.\ \mathrm{ReplayCtr}(r_1)@t_1 \wedge \mathrm{ReplayCtr}(r_2)@t_2 \wedge (t_1 < t_2) \Rightarrow (\exists x.\ r_1 + x = r_2).$$

We find that the restriction works as intended using the following sanity-check lemma.

$$\forall\ id, m, t_1, t_2.\ \mathrm{S\_AcceptsMessage}(id, m)@t_1 \wedge \mathrm{S\_AcceptsMessage}(id, m)@t_2 \Rightarrow (t_1 = t_2)). \tag{CorrectnessNoReplay}$$

3.8 Chapter Summary

In this chapter, we have introduced the automatic protocol verification tool Tamarin and its front-end SAPiC that allows modelling of processes. Subsequently, we have presented formal models of various KRACK attacks, using Tamarin, on the IEEE 802.11 *4-way handshake* and group key handshake, and downgrade attacks on implementations of the *4-way handshake.*The process of translating the security properties into lemmas has also been discussed in detail. Apart from modelling steps, we have also outlined modelling issues encountered by us during the process. The approach presented in this chapter can be helpful to model many new attacks and compare them with the complete protocol model. In the next chapter, we will present the analysis of security properties of 802.11 *4-way handshake* in our models that we have developed in this chapter.

Chapter Four

Analysis of 802.11 4-Way Handshake Attacks and Security Properties

Science cannot solve the ultimate mystery of nature. And that is because, in the last analysis, we ourselves are a part of the mystery that we are trying to solve.

Max Planck

4.1 Motivation

In chapter 3, we have argued that modelling of complete protocol specification using formal verification tools is a complex and time-intensive process. We have presented the Tamarin models of KRACK and Downgrade attacks on IEEE 802.11 *4-way handshake* for the purpose of testing the efficacy of modelling only the attacks, to test the security guarantees and their efficacy in capturing the attacks.

In this chapter, we proceed to check if the security properties proposed in the standard are enough to ensure the security of the protocol. we use our novel methodology for analysing the security properties as mandated in the standard using models of attacks on 802.11 *4-way handshake* developed in the previous chapter. As already explained, our modelling approach is different from the normal use of formal methods for checking security protocols, which consists in defining a model of a protocol with its security properties to check for the existence of attacks. Instead, we use our models and known attacks from previous works to check if the security properties proposed in the standard are enough to ensure the security of the protocol. Where they are not, we propose a new security property that could be added to the standard, encode it in TAMARIN, and use the tool to automatically show that it would be enough to stop a class of attacks, such as KRACK and Downgrade.

4.2 Contribution

This chapter, rather than modelling the complete state machine, presents the analysis of security properties based on the modelling of only those actions sufficient to demonstrate an attack. We have analysed if such models can be used to analyse the security properties listed out in the protocol standard. We also prove that there is merit in this technique, i. e., modelling of subset of actions is enough to demonstrate an attack and analysis of security properties under the attack scenario. This chapter presents one such case study using IEEE 802.11 *4-way handshake* as an example. Using our analysis, we have been able to prove that the security properties mandated in IEEE 802.11 are insufficient to capture the modelled attacks. Additionally, we have been able to come up with the security properties to be augmented to capture such attacks and verify them by proving that enforcing our new security properties is successful in stopping such attacks. In summary, we have successfully modelled and verified the process of our novel methodology of analysing security properties and testing their sufficiency by modelling just the attacks on any given protocol. Compared to use of traditional formal methods for the purpose, our method is much economical and easy to verify.

4.3 Overview

The IEEE 802.11 standard defines a list of security properties suggesting that it will lead to a secure *4-way handshake* (e. g., freshness of session keys, secrecy of session/group keys, authentication). The existence of the attacks described above raises serious questions about these security properties: Does the IEEE 802.11 specification or some implementations violate these properties, leading to these attacks? Or are these security properties insufficient to guarantee security? If so, what security properties would be sufficient to stop the attacks? In this chapter, we show how these questions can be formally answered using TAMARIN.

We encode the security properties from the standard using TAMARIN, and use the tool to see if any of these security properties are violated in the presence of the attacks. We find that the weaknesses that lead to the KRACK attacks [137] *do not* violate any of the required properties. This suggests that the security properties, as defined in the standard, are insufficient. We then propose new security properties, and by imposing them as restrictions in TAMARIN, we show that ensuring these new suggested properties is enough to stop these attacks.

4.4 Related Work

The IEEE 802.11 standard lists five properties, labelled from a) to e), for the *4-way handshake* [80, Sec. 12.6.14]. He *et al.* [76] aggregate four out of five of these security properties into *session authentication*, which can only be asserted when *key secrecy* is guaranteed. They formalise authentication in the cryptographic model using the notion of *matching conversations* [16], guaranteeing that the two entities have consistent views of the protocol runs. Using Protocol Composition Logic (PCL) [59], they verify that such properties hold. However, PCL has been subject of criticism by some authors such as [50], as it allows one to verify authentication protocols that rely on signing, but not those relying on decryption. More disconcertingly, there are no means to establish preceding actions in a thread. In contrast to matching conversations used in [76], we use standard notions of authentication from Lowe [95], e. g., mutual, injective agreement, to verify the security properties. Moreover, in their approach using PCL [76], the authors confirm that all their proofs were constructed manually. On the other hand, our verification using Tamarin is among the first attempts to verify security properties of 802.11 automatically.

Concurrent to our work, Cremers *et al.* [54] also developed a detailed Tamarin model of the WPA2 protocol capable of detecting KRACK attacks, among others. Their work, as ours, verifies the effectiveness of the patched protocol, post-discovery of the KRACK attacks, in stopping all the attacks, including the KRACK attacks. However, our goals are different; our focus is on developing a framework to test the adequacy of the required security properties in spotting the attacks. Therefore, we only model the functionalities required to demonstrate the attacks (KRACK and downgrade), rather than the whole protocol.

4.5 Methodology for Analysing Security Properties

Fig. 4.1 summarises our process of analysing the security properties, and augmenting additional security properties if needed. We start by building a model of a protocol with known attacks in Sec. 3.6. Subsequently, we verify all the security properties listed in the standard to see if they are satisfied or violated in Sec. 4.6. A violated security property will signify relation of the attack with the property. However, to establish that the security property and attack has a two-way relationship, the violated security property should then be enforced as a restriction to check if it would stop the attacks. If it does, it will indicate an implementation issue. Alternatively, if all the security properties are verified, but the attack still exists, we can conclude that the security properties required by the standard

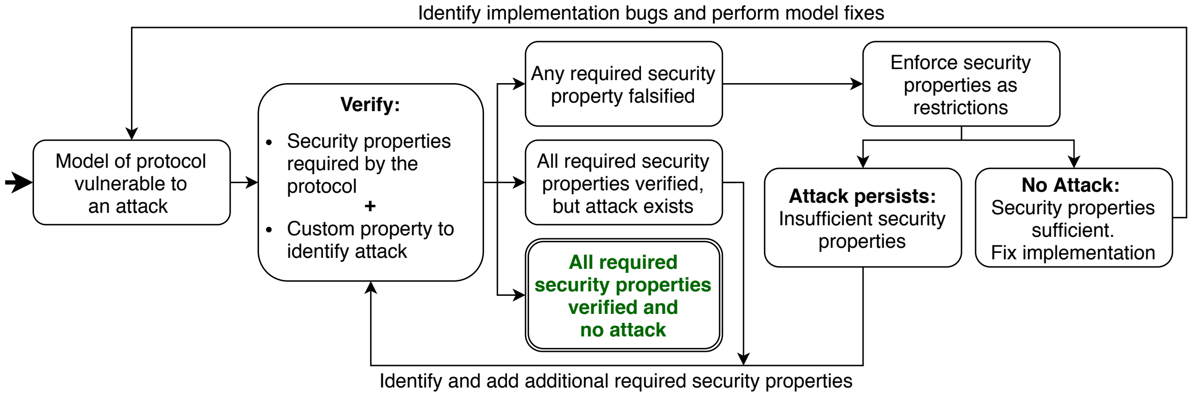

Identify implementation bugs and perform model fixes
Model of protocol vulnerable to an attack
Verify:
• Security properties required by the protocol
+
• Custom property to identify attack
Any required security property falsified
All required security properties verified, but attack exists
All required security properties verified and no attack
Enforce security properties as restrictions
Attack persists: Insufficient security properties
No Attack: Security properties sufficient. Fix implementation
Identify and add additional required security properties

Figure 4.1: Flow diagram for verifying security properties, identifying new ones, and fixing the model against an attack

are insufficient and need to be augmented. After analysing the attacks, we propose a security property corresponding to the attack, shown below in Sec. 4.7. To test that the new property is successful in stopping the attack, we first place it as a lemma in the model and expect it to be falsified. Then, we enforce this property as a restriction in the model, expecting that it stops the attack. This helps us to verify if the attack corresponds to the new proposed security property. Finally, we execute the protocol model after fixing the vulnerability, to verify the absence of the attack. The verification of our newly proposed security properties and the fixes proves both the adequacy of the final set of properties, and correctness of the fixes in the protocol. We discuss this in Secs. 4.7 and 4.8.

4.6 Analysis of IEEE 802.11 Security Properties

In this section, we list the five properties a)-e) specified for the *4-way handshake* in the 802.11 standard [80, Sec. 12.6.14]. These properties overlap with each other and cannot be easily encoded into conventional queries, e. g., secrecy or authentication. Therefore, we sometimes define multiple security lemmas that jointly satisfy a given property. Moreover, the standard is unclear about what properties are satisfied by the group key handshake. In that case, we consider an extension of property c) below for GTK. We recall that we prepend ‘S_’ and ‘A_’ to the supplicant and authenticator events, respectively.

a) **Confirm the existence of the PMK at the peer.** As stated in Sec. 3.4, our model treats this property as a premise. However, to confirm this property, we use the following lemma:

$$\begin{aligned}\forall id_1, id_2, pmk_1, pmk_2, t_1, t_2.\ &\text{A\_HasPmk}(id_1, pmk_1)@t_1 \wedge\\ &\text{S\_HasPmk}(id_2, pmk_2)@t_2 \Rightarrow (pmk_1 = pmk_2).\end{aligned} \tag{ConfPmk}$$

b) **Ensure that the security association keys (PTK/GTK) are fresh.** This security property states that at every run (thread *tid*) of the protocol it must generate a fresh PTK/GTK. We verify this property at the supplicant side through lemma

$$\begin{aligned}\forall id_1, id_2, ptk, t_1, t_2.\ &\text{S\_ComputesPtk}(id_1, ptk)@t_1 \wedge\\ &\text{S\_ComputesPtk}(id_2, ptk)@t_2 \Rightarrow (id_1 = id_2).\end{aligned} \tag{FreshPtk}$$

Similarly, we define the following Lemma FreshGtk for the case of GTK.

$$\begin{aligned}\forall id_1, id_2, gtk, t_1, t_2.\ &\text{S\_ComputesGtk}(id_1, gtk)@t_1 \wedge\\ &\text{S\_ComputesGtk}(id_2, gtk)@t_2 \Rightarrow (id_1 = id_2).\end{aligned} \tag{FreshGtk}$$

c) **Synchronise the installation of temporal keys into the MAC.** We consider the strongest authentication property from Lowe's hierarchy [95], namely, injective *agreement*. For the case of PTK, we verify that: "for each S_CommitPtk event executed by the supplicant S_{id}, the associated authenticator A_{id} executed the corresponding A_RunningPtk earlier, and for each run of the protocol there is a unique S_CommitPtk for each A_RunningPtk,"

$$
\begin{aligned}
&(\forall S_{\text{id}}, A_{\text{id}}, pars, t_1.\ \text{A\_CommitPtk}(S_{\text{id}}, A_{\text{id}}, pars)@t_1 \Rightarrow\\
&\qquad((\exists t_2.\ \text{S\_RunningPtk}(A_{\text{id}}, S_{\text{id}}, pars)@t_2 \wedge (t_2 < t_1))\\
&\qquad \wedge \neg(\exists S'_{\text{id}}, A'_{\text{id}}, t_3.\ \text{A\_CommitPtk}(S'_{\text{id}}, A'_{\text{id}}, pars)@t_3 \wedge \neg(t_3 = t_1)))\\
&\quad \wedge\\
&\forall S_{\text{id}}, A_{\text{id}}, pars, t_1.\ \text{S\_CommitPtk}(S_{\text{id}}, A_{\text{id}}, pars)@t_1 \Rightarrow\\
&\qquad((\exists t_2.\ \text{A\_RunningPtk}(A_{\text{id}}, S_{\text{id}}, pars)@t_2 \wedge (t_2 < t_1))\\
&\qquad \wedge \neg(\exists S'_{\text{id}}, A'_{\text{id}}, t_3.\ \text{S\_CommitPtk}(S'_{\text{id}}, A'_{\text{id}}, pars)@t_3 \wedge \neg(t_3 = t_1)))).
\end{aligned}
\tag{AgreePtk}
$$

Obviously, the set of parameters *pars* must contain the value of the PTK. The commit events (e. g., S_CommitPtk) are placed as late as possible on the supplicant side. Accordingly, the running events (e. g., A_RunningPtk) are executed as early as possible, when all the parameters to agree are available to the authenticator. In order to capture mutual agreement, as shown above, the lemma also includes the case when the roles of the authenticator and supplicant are reversed.

As customary, authentication requires key secrecy to be asserted. We verify this using the following lemma for PTK:

$$\forall id, ptk, t_1.\ \text{S\_InstallsPtk}(id, ptk)@t_1 \Rightarrow \neg(\exists t_2.\ K(ptk)@t_2). \tag{SecretPtk}$$

Again, S_InstallsPtk models the primitive MLME-SETKEYS.request [80], and we require that any installed PTK is unknown to the adversary.

For GTK, we define the Lemmas (AgreeGtk) and (SecretGtk) equivalently. Moreover, we also need to capture *weak agreement* [95] of GTK in the group key handshake, through the lemma

$$\forall S_{\mathrm{id}}, A_{\mathrm{id}}, pars, t_1.\ \mathrm{S\_WCommitGtk}(S_{\mathrm{id}}, A_{\mathrm{id}}, pars)@t_1 \Rightarrow \\ ((\exists t_2.\ \mathrm{A\_WRunningGtk}(A_{\mathrm{id}}, S_{\mathrm{id}}, pars)@t_2 \wedge (t_2 < t_1)), \qquad \text{(WeakAgreeGtk)}$$

which includes the GTK in *pars*. As opposed to (AgreeGtk) in the *4-way handshake*, the agreement in the group key handshake is not injective, because multiple retransmissions of the same GTK are allowed.

d) **Transfer the GTK from the Authenticator to the Supplicant.** We verify if the GTK received by the supplicant is the same GTK calculated and forwarded by the authenticator using lemma

$$\forall id, gtk, t_1.\ \mathrm{S\_InstallsGtk}(id, gtk)@t_1 \Rightarrow \\ (\exists t_2.\ \mathrm{A\_GeneratesGtk}(gtk)@t_2 \wedge (t_2 < t_1)). \qquad \text{(SameGtk)}$$

This property provides assurance to the supplicant that the GTK received by it is the same which was generated by the authenticator in the current handshake cycle.

e) **Confirm the selection of cipher suites.** We capture injective agreement of the cipher suite with Lemma (AgreeCs), similar to (AgreePtk) above, by using the cipher suite within the parameters *pars*.

We queried the lemmas defined for the above five properties in the Tamarin models presented in Sec. 3.6, in order to verify them in presence of KRACK and downgrade attacks. Unexpectedly, all the lemmas were reported as verified when KRACK attacks were present, as shown in Table 4.1. In the case of the downgrade attack, however, Tamarin reported expected violation of Lemma (AgreeCs) only.

Table 4.1: TAMARIN results of testing properties a)-e) from the 802.11 standard in Sec. 4.6. *No*[*Attack*] refers to (NoKrackPtk), (NoKrackGtk) or (NoDowngrade) accordingly. (✓ verified; ✗ falsified; – n/a)

Security Properties:		a) ConfPmk	b) FreshKeys		c) SynchronisedKeys					d) SameGTK	e) ConfCiphers
Lemmas	*No*[*Attack*]	(ConfPmk)	(FreshPtk)	(FreshGtk)	(AgreePtk)	(AgreeGtk)	(WeakAgreeGtk)	(SecretPtk)	(SecretGtk)	(SameGtk)	(AgreeCs)
Case Study:											
PTK reinst. Figs. 3.3, 3.4	✗	✓	✓	✓	✓	✓	–	✓	✓	✓	✓
PTK reinst. Fig. 3.5	✗	✓	✓	✓	✓	✓	–	✓	✓	✓	✓
PTK reinst. Fig. 3.6	✗	✓	✓	✓	✓	✓	–	✓	✓	✓	✓
GTK reinst. Fig. 3.7	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Downgrade Fig. 3.8	✗	✓	✓	✓	✓	✓	–	✓	✓	✓	✗

4.7 Proposing New Security Properties

Security Property for KRACK Attack: Table 4.1 from Section 4.6 clearly establishes the inadequacy of set of security properties mandated by the IEEE 802.11 standard to capture security violation by KRACK attacks reviewed in Sec. 3.6. Though IEEE has since addressed the issue of nonce reuse in 802.11 implementations [1], and the Wi-Fi Alliance tests the devices before certifying them for WPA2/3 [2], there is no mention of security properties being added to the standard that could capture various KRACK variants such as the ones presented by Lemmas (NoKrackPtk) and (NoKrackGtk). What it means that if someone were to formally verify these standards against attacks such as KRACK and Downgrade, then the documented security properties present in the standard specifications will still be unable to capture these attacks. To bridge the gap between the 802.11 standard and its implementation, we propose an additional security property to capture such vulnerabilities:

f) Ensure that the security association keys are not used more than once.

The security property f) is encoded, using the following lemma, in Tamarin.

$$\begin{aligned}\forall id, ptk, t_1, t_2.\ & \text{S\_InstallsPtk}(id, ptk)@t_1 \wedge \\ & \text{S\_InstallsPtk}(id, ptk)@t_2 \Rightarrow (t_1 = t_2). \qquad \text{(NoPtkReuse)}\end{aligned}$$

Equivalently, we define the following Lemma using GTK in place of PTK.

$$\begin{aligned}\forall id, gtk, t_1, t_2.\ & \text{S\_InstallsGtk}(id, gtk)@t_1 \wedge \\ & \text{S\_InstallsGtk}(id, gtk)@t_2 \Rightarrow (t_1 = t_2). \qquad \text{(NoGtkReuse)}\end{aligned}$$

We expect this security property to capture the KRACK attack, i. e., this security property should be falsified by the KRACK attack. Table 4.2 shows that all the KRACK attack models from Sec. 3.6 violate either one or both properties, i. e., the KRACK attacks are now captured by our new security property f).

Security Property for Downgrade Attack: The downgrade attack from Fig. 3.8 violates property e) through the Lemma (AgreeCs). Surprisingly, the attack continue to exist even after enforcing this property as restriction. Since enforcing the agreement property on the cipher suite does not stop the attack, it is violating a property not present in the standard. A detailed analysis of property e) along with the downgrade attack suggests that though the standard guarantees authentication w.r.t. other party, it does not perform agreement with itself. Accordingly, we suggest the following additional security property g), as Lemma (ValidCipherSuite), to the model of Fig. 3.8b that captures this attack (See Table 4.2).

g) The cipher suite that the authenticator started with is the cipher suite that the authenticator finishes with, and is the strongest one from the available choices.

As expected, the downgrade attack from Sec. 3.6 is captured by property g), which is encoded in Tamarin using lemma

$$\begin{aligned}\forall tid, cs_1, cs_2, t_1, t_2.\ &\text{A\_SentMsg3}(tid, cs_1)@t_1 \wedge \\ &\text{A\_Starts}(tid, cs_2)@t_2 \Rightarrow (cs_1 = cs_2). \qquad \text{(ValidCipherSuite)}\end{aligned}$$

Verification of New Security Properties: To verify that our proposed security properties f) and g) correspond to respective attacks, we fix the respective Tamarin models of Sec. 3.6 by enforcing (NoPtkReuse), (NoGtkReuse) and (ValidCipherSuite) as *restrictions.* On testing the Lemmas (NoKrackPtk), (NoKrackGtk), and (NoDowngrade), i. e., security properties from Sec. 3.6, Tamarin verifies them in the fixed model, proving that the proposed security properties are successful in stopping these attacks.

Table 4.2: TAMARIN results of testing properties a)-e) from the 802.11 standard and proposed properties f) and g) in Sec. 4.6. *No*[*Attack*] refers to (NoKrackPtk), (NoKrackGtk) or (NoDowngrade) accordingly. (✓ verified; ✗ falsified; – n/a)

Security Properties:		a) ConfPmk	b) FreshKeys		c) SynchronisedKeys					d) SameGTK	e) ConfCiphers	f) NoKeyReuse		g) ValidCipher
Lemmas	*No*[*Attack*]	(ConfPmk)	(FreshPtk)	(FreshGtk)	(AgreePtk)	(AgreeGtk)	(WeakAgreeGtk)	(SecretPtk)	(SecretGtk)	(SameGtk)	(AgreeCs)	(NoPtkReuse)	(NoGtkReuse)	(ValidCipherSuite)
Case Study:														
PTK reinst. Figs. 3.3, 3.4	✗	✓	✓	✓	✓	✓	–	✓	✓	✓	✓	✗	✗	✓
PTK reinst. Fig. 3.5	✗	✓	✓	✓	✓	✓	–	✓	✓	✓	✓	✗	✗	✓
PTK reinst. Fig. 3.6	✗	✓	✓	✓	✓	✓	–	✓	✓	✓	✓	✗	✗	✓
GTK reinst. Fig. 3.7	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	✗	✓
Downgrade Fig. 3.8	✗	✓	✓	✓	✓	✓	–	✓	✓	✓	✗	✓	✓	✗

Table 4.3 shows the results of testing the original security properties mandated in the IEEE 802.11 standard while enforcing the new security properties (f) and g)) proposed in the Section 4.7. We find that, on enforcing the lemmas NoPtkReuse,NoGtkReuse and ValidCipherSuite as restrictions, the all the attacks under analysis, KRACK described by lemmas (NoKrackPtk,NoKrackGtk) and the Downgrade attack described by the lemma NoDowngrade, are stopped while satisfying all the security properties. This proves a two-way correspondence between the lemmas and attacks.

It is, therefore, strongly recommended that our novel security properties f) and g) should be added to the 802.11 standard to be able to capture not only the known attacks such as KRACK and Downgrade, but possibly also other attacks belonging to a similar class.

4.8 Verifying the Mitigations to the Models

Finally, we fix the KRACK models, from Sec. 3.6, making sure that they follow the newly proposed security property f), i. e., disconnect if there is an attempt to install with the same PTK or GTK, and then execute the model again. After the fix, both of the attack lemmas, i. e., Lemmas (NoKrackPtk) and (NoKrackGtk), along with the security properties (NoPtkReuse) and (NoGtkReuse) are verified. The absence of the attack, with the new security properties verified, shows the validity of the proposed fix. This result is also a verification of the proposed countermeasure for KRACK by [137].

Similarly, the downgrade attack from Fig. 3.8 can be easily detected at the supplicant side [131], and can be stopped if the authenticator implementation disallows the change of cipher suites mid-handshake.

Table 4.3: TAMARIN results of testing original security properties a)-e) from the 802.11 standard and proposed property f) and g) enforced as restrictions in Sec. 4.6. *No*[*Attack*] refers to (NoKrackPtk), (NoKrackGtk) or (NoDowngrade) accordingly. (✓ verified; ✗ falsified; – n/a)

Security Properties:		a) ConfPmk	b) FreshKeys		c) SynchronisedKeys					d) SameGTK	e) ConfCiphers
Lemmas	*No*[*Attack*]	(ConfPmk)	(FreshPtk)	(FreshGtk)	(AgreePtk)	(AgreeGtk)	(WeakAgreeGtk)	(SecretPtk)	(SecretGtk)	(SameGtk)	(AgreeCs)
Case Study:											
PTK reinst. Figs. 3.3, 3.4	✓	✓	✓	✓	✓	✓	–	✓	✓	✓	✓
PTK reinst. Fig. 3.5	✓	✓	✓	✓	✓	✓	–	✓	✓	✓	✓
PTK reinst. Fig. 3.6	✓	✓	✓	✓	✓	✓	–	✓	✓	✓	✓
GTK reinst. Fig. 3.7	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Downgrade Fig. 3.8	✓	✓	✓	✓	✓	✓	–	✓	✓	✓	✓

Table 4.4: TAMARIN results of analysing original security properties a)-e) from the 802.11 standard in Sec. 4.6 in the fixed model described in Sec. 4.8. *No*[*Attack*] refers to (NoKrackPtk), (NoKrackGtk) or (NoDowngrade) accordingly. (✓ verified; ✗ falsified; – n/a)

Security Properties:		a) ConfPmk	b) FreshKeys		c) SynchronisedKeys					d) SameGTK	e) ConfCiphers
Lemmas	*No*[*Attack*]	(ConfPmk)	(FreshPtk)	(FreshGtk)	(AgreePtk)	(AgreeGtk)	(WeakAgreeGtk)	(SecretPtk)	(SecretGtk)	(SameGtk)	(AgreeCs)
Case Study:											
PTK reinst. Figs. 3.3, 3.4	✓	✓	✓	✓	✓	✓	–	✓	✓	✓	✓
PTK reinst. Fig. 3.5	✓	✓	✓	✓	✓	✓	–	✓	✓	✓	✓
PTK reinst. Fig. 3.6	✓	✓	✓	✓	✓	✓	–	✓	✓	✓	✓
GTK reinst. Fig. 3.7	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Downgrade Fig. 3.8	✓	✓	✓	✓	✓	✓	–	✓	✓	✓	✓

Accordingly, we fix the model ensuring that it rejects a connection where the authenticator does not start and finish with the same cipher suite. After fixing it, Tamarin reports the attack Lemma (NoDowngrade) as verified, i. e., the Downgrade attack no longer exists, and that the mitigation is valid. Both the fixed Tamarin models, of KRACK and downgrade attacks, are publicly available at [127]. The results of our analysis, in the fixed model as per Section 4.5, are available in the Table 4.4.

4.9 Chapter Summary

In Chapter 3, we presented modelling of various KRACK attacks on the IEEE 802.11 *4-way handshake* and group key handshake, and Downgrade attacks on implementations of the *4-way handshake.* Using the tool Tamarin in this chapter, we verified all the security properties of the *4-way handshake* mandated by the 802.11 standard, in the presence of KRACK and Downgrade vulnerabilities. We found that KRACK attacks did not violate any of the required security properties. We concluded that the set of properties, mandated in the standard, was inadequate to capture these attacks.

Using a novel approach, we have proposed additional security properties to be added to the 802.11 standard, enabling it to capture them. We also demonstrate that enforcing these security properties in our model successfully stops these attacks. Accordingly, we fix the models with countermeasures to mitigate the attacks and verify all the security properties, providing a formal proof of correctness of the recommended countermeasures. We have demonstrated use of our novel technique in strengthening the 802.11 *4-way handshake* protocol specifications, by testing the adequacy of the set of required security properties against many known attacks, and by augmenting them with new properties, as required. This approach is useful to strengthen other protocol specifications in a similar fashion by testing the adequacy of the set of required security properties against known or newly discovered attacks, and by augmenting them with new properties, if required.

Part III

Defining Stealthiness in a Trace Model

Chapter Five

Formal Model of Stealthiness

"The greatest pleasure I know, is to do a good action by stealth, and to have it found out by accident."

Charles Lamb

5.1 Motivation

In Chapter 3, we have already established that there is merit in modelling just the attacks and not the complete protocol. Our objective now is to establish a set of properties, an attack must satisfy, in order to be classified as a stealth attack. As introduced earlier, '*Stealth Attacks'* are those where an attacker is successful in camouflaging its network activities with message traffic appearing as standard, as defined by the protocols for non-attack messaging. As the message traffic appears to be indistinguishable from a standard run of a protocol, such attacks are difficult to capture.

There have been many studies attempting to analyse the system logs to find out if a stealthy attack is in progress, mainly using data mining techniques [4, 106, 124]. The formal analysis of security protocols has mostly remained focused on proving the security properties mandated by the standard, or finding an attack in the case of any security property being falsified. Study and classification of various attacks on security protocols have not attracted much attention, especially in terms of their equivalence to the expected behaviour. Our primary motivation to study the attacks that follow the expected behaviour emanates from the fact that such attacks may be harder to capture, and a standard framework may help identify such attacks. Subsequently, by logging suitable parameters, we aim to convert the highly potent stealth attacks to non-stealth attacks, which we can capture using our framework.

5.2 Contribution

This chapter presents modelling of protocols using a labelled multiset rewriting (MSR) system, similar to one used by Tamarin. We extend the labelled MSR to allow for required functionalities in modelling the protocol run in our framework. In order to develop a framework to formalise the detection of such attacks on security protocols, we propose analysing logs present in the traces generated by the protocol execution and their order using formal methods. This can be achieved by analysing the logs from the actual protocol traces and comparing them to the expected protocol traces based on standard protocol behaviour. To verify the results, automated protocol verification tools can be used.

We present a trace based definition of what is a stealth attack and show how stealth attacks can be detected, or their absence proved, using formal notions developed by us such as standard trace, standard looking trace and attack run. We attempt to capture such stealth attacks by defining stealthiness in a trace generated by the actions of the protocol. We examine if, given the protocols that the message interaction should be following, can the attacker successfully hide its attack in the underlying network traffic? Using the standard framework of multiset-rewriting, we have developed a formal notion of protocol specification and traces generated thereof.

5.3 Overview

We give an intuitive definition of a "standard looking run", i. e., a trace in which the logged actions could have been produced by some interleaving of some number of correct runs of the protocol. A stealth attack is then defined as an attack that breaches some security condition and produces a standard looking run. We model the protocol execution using labelled transition rules with one rule for every action and each rule logging action labels in the trace with some parameters. Our proposal is to use well-known techniques for specifying security protocols, the framework of multiset-rewriting, to provide the specification of the protocols of interest. Finally, we have introduced parlance and notations related to term rewriting and labelled multiset rewriting, forming the basis of our formal analysis in subsequent chapters. We do not consider aspects of network traffic, such as timing and packet size, or the probability of a particular message, which could be used to indicate attacks, we leave such an extension as a future work.

Using the above principles, we develop the notion of stealthiness and define it in a trace model. We use well-known techniques for specifying security protocols, the framework of multiset-rewriting, to provide the specification of the protocols of interest. We then systematically augment these protocol specifications with logging specifications, recording what parts of those protocols are logged with an objective of detecting stealthy attacks. We then apply protocol analysis to determine when such a protocol admits stealthy attacks.

In order to capture the inherent complexity of security protocols, mathematical notions based formal methods are employed to reason their correctness and verify the security guarantees provided by the protocols [12]. The formal verification can detect issues leading to protocol failure, e. g., design flaws and other weaknesses due to poor implementation [43]. By addressing design issues and exploiting weaknesses of the proposed protocols, these methods can provide high-level definitions of security properties and their proofs. The formal methods provide a framework for modelling messages, transition rules etc. for the protocol. Certain assumptions about cryptographic primitives and the power of the adversary have to be made right at the initial phase of modelling. It also accounts for the fact that the network is under control of an all powerful adversary.

We analyse execution patterns of protocols in order to ascertain if the execution corresponds to the expected standard execution of a given protocol. We call all such executions stealthy executions and traces generated thereof, a stealthy trace. Further, we pass the attack traces under stealthy conditions, resulting in either the attack or the non-attack situation. If an attack is able to execute itself while generating only stealthy traces, we label it as a stealth attack, otherwise as a non-stealth attack. It is important to mention here that we are not interested in finding new attacks on protocols. We are more interested in verifying whether the attacks given by protocol verification tools confirm our assumptions regarding stealthiness of an attack.

5.4 Modelling Protocols in our Framework

We use term rewriting along with a labelled multiset rewriting (MSR) system to model security protocols, similar to that used by the Tamarin tool [103, 123], which is already described briefly in Section 2.8.

Terms and equational theories: We consider a signature, i. e., a set of function symbols $\Sigma = \{(\mathsf{c}_i, k_i) \mid i = 1, \ldots, m\}$ that defines a set of *constructors* c_i each with an arity k_i. A signature, written as f/n, represents a function f of arity n. Assume a countable set of variables $x \in \mathcal{V}$ and a set of names $n \in \mathcal{N}$, divided further in sets of public and fresh names $\mathcal{PN}$ and $\mathcal{FN}$. $\mathcal{T}_\Sigma$ denotes the set of valid terms built over Σ, $\mathcal{PN}$, $\mathcal{FN}$, and $\mathcal{V}$. The set of (finite) trees or *terms* $t \in \mathcal{T}_\Sigma(\mathcal{V}, \mathcal{N})$ is the smallest set satisfying the following: (1) Any $x \in \mathcal{V}$ or $n \in \mathcal{N}$ is an element of $\mathcal{T}_\Sigma(\mathcal{V}, \mathcal{N})$; and (2) If $(\mathsf{c}, k) \in \Sigma$ then

$\mathsf{c}(t_1, \ldots, t_k) \in \mathcal{T}_\Sigma(\mathcal{V}, \mathcal{N})$ for any $t_1, \ldots, t_k \in \mathcal{T}_\Sigma(\mathcal{V}, \mathcal{N})$. An *annotated term* $s \in \mathcal{S}_\Sigma(\mathcal{V}, \mathcal{N})$ is of the form $\circ t$ (a *linear term*) or $!t$ (a *persistent term*). A *ground term* is an element of $\mathcal{T}_\Sigma(\{\}, \mathcal{N})$, i. e., constructed without any variables.

We will omit explicit mention of Σ, $\mathcal{V}$ and $\mathcal{N}$ in what follows, just denoting the set of terms and annotated terms as $\mathcal{T}$ and $\mathcal{S}$, respectively. A *ground term* is constructed without any variables. In our modelling, we use terms to represent messages and log entries, while names are used to represent keys and nonces.

The term algebra is supported by a fixed equational theory E, where the signature and equational theory supports equations for both pairing and projection, i. e., $\{\langle ., . \rangle, \mathsf{fst}, \mathsf{snd}\} \subseteq \Sigma$ and equations $\mathsf{fst}(\langle x, y \rangle) = x$ and $\mathsf{snd}(\langle x, y \rangle) = y$ are in E.

Sets, Sequences, and Multisets The symbol $\mathbb{N}_n$ represents the set $\{1, \ldots, n\}$. The domain and range of a binary relation $\mathcal{R}$ are denoted by $domain(\mathcal{R})$ and $range(\mathcal{R})$ respectively. A set of sequences over $\mathcal{S}$ is denoted by $\mathcal{S}^*$. The i-th element of the sequence s is represented by s_i, $|s|$ stands for the length of s, and $idx(s) = \{1, 2, ..., |s|\}$ for a set of indices of s. The empty sequence is denoted by [], $[s_1, s_2...., s_k]$ denoted the sequence s with $|s| = k$. The concatenation of sequences s and s' is denoted by $s.s'$. For a sequence s, $mset(s)$ and $set(s)$ represent their corresponding multiset and set respectively, where a multiset is a generalised version of a set that allows multiple instances of the multisets elements. We denote multiset using symbol $S^\sharp$ for set S. The operations on multisets also contain the symbol $\sharp$ as superscript, e. g., $m_1 \cup^\sharp m_2$ is used to describe union of two multisets m_1 and m_2. Finally, $set(m)$ represents a set of all the elements of a multiset m [89].

Substitution A *substitution* $\theta \in \Theta$ is a finite mapping from variables to terms. We sometimes denote this as $\{t_1/x_1, \ldots, t_k/x_k\}$ for a mapping where $\theta(x_i) = t_i$ for $i = 1, \ldots, k$ and $\theta(x) = x$ if $x \notin \{x_1, \ldots, x_n\}$. We define the *domain* of a substitution as $dom(\theta) = \{x \mid$

$\theta(x) \neq x\}$. Substitutions can be applied on terms as $f(t_1, ..., t_n)\theta = f(t_1\theta, ..., t_n\theta)$. For example, a substitution $\theta : \mathcal{X} \rightarrow \mathcal{T}_\Sigma(\mathcal{X})$ can be applied as $\theta = \{X \mapsto \mathit{senc}(Msg, Key)\}$ can be applied to a term $t = \mathit{sdec}(X, Key)$ resulting in $t\theta = \mathit{sdec}(\mathit{senc}(Msg, Key), Key$. We can also perform composition $\theta\tau$ over substitutions θ and τ denoted as $\theta \circ \tau$. For substitutions $\theta = [x \mapsto f(y), y \mapsto f(z)]$ and $\tau = [y \mapsto g(a), z \mapsto g(b)]$, the composition will produce $\theta \circ \tau = [x \mapsto f(g(a)), y \mapsto f(g(b)), z \mapsto g(b)]$ [89].

We extend the notion of substitutions to terms and annotated terms in the natural way. We also assume the metafunctions $\mathit{fn}(\_)$ and $\mathit{fv}(\_)$ for the free names and free variables respectively in an (annotated) term. A *ground substitution* maps any variables in its domain to a ground term; we denote the set of ground substitutions as Θ^{Gnd}.

A *renaming substitution* $\varphi \in \Phi$ is an injective finite mapping from names to names, and we denote this as $\{n'_1/n_1, \ldots, n'_k/n_k\}$. For brevity, we will sometimes denote the sequence $n_1, \ldots, n_k$ by $\vec{n_k}$. We denote the domain and range by $\mathit{dom}(\varphi)$ and $\mathit{ran}(\varphi)$, respectively. The substitution φ is a *renaming away from* $\{\vec{n_k}\}$ if $\mathit{ran}(\varphi) \cap \{\vec{n_k}\} = \{\}$.

Let $\mathcal{T}^\sharp$ and $\mathcal{S}^\sharp$ denote the set of multisets of terms and annotated terms, respectively. Each transition rule *rule* is provided as a 4-tuple

$$\mathit{rn} : l \;\; - [a] \rightarrow r$$

where *rn* is the unique name to identify the rule and the multiset of annotated terms $l \in \mathcal{S}^\sharp$ denotes the *premises* of the rule whereas the multisets of annotated terms $a, r \in \mathcal{S}^\sharp$ denote the *actions* and *conclusions* of the rule, respectively.

We use sequences as a meta-notation in several places. We denote a list of k items (rules, rule lists etc) as $[X_1, \ldots, X_k]$. We denote sequence concatenation by $[X_1, \ldots, X_j] \;@\; [Y_1, \ldots, Y_k] = [X_1, \ldots, X_j, Y_1, \ldots, Y_k]$. We denote addition to the left of a sequence by $X :: [X_1, \ldots, X_k] = [X] \;@\; [X_1, \ldots, X_k]$.

We refer to the terms in a rewrite rule as *facts*. The premises contain both linear facts and persistent facts. Let $\mathit{lfacts}(l) = \{t \mid \circ t \in l\}$ denote the former and $\mathit{pfacts}(l) = \{t \mid !t \in l\}$ denote the latter. A rule list $\mathit{RuleList} = [\mathit{rule}_1, \ldots, \mathit{rule}_n]$ denotes a sequence of such rules. We extend the application of substitutions to rules and rule lists in the obvious way.

The cryptographic messages are modelled using the sort *msg* and two incomparable subsorts: *pub* and *fresh* for public and fresh names. We assume two countably infinite sets $\mathcal{PN}$ and $\mathcal{FN}$ of public names *pub* and fresh names *fresh*. The rules can either use public names directly or variables with substitutions mapping some of these variables to public names, such as a participant name, and others to fresh names, such as nonces, freshly generated keys etc. For the application of a substitution to a rule, the main complication is to rename apart the "fresh" names in the rule so that they do not appear in the premises after the instantiation of the rule. The fresh names can only be generated by instances of the *Fresh* rule, and it guarantees that the same fresh name is never generated twice. We generate all such names by applying the special 'fresh' rule for generating instances of $Fr(x)$ facts:

$$Fresh : [] \ -[] \rightarrow [Fr(x : fresh)] \mid \forall i, j \in \{1, \ldots, k\} \text{ and } n \in \mathcal{FN} \text{ where}$$

$$rn_i : l_i \ -[a_i] \rightarrow r_i = rn_j : l_j \ -[a_j] \rightarrow r_j = Fresh : [] \ -[] \rightarrow [Fr(n)], \text{ and}$$

$$\text{it holds that } i = j.$$

Rules may contain free variables, and we denote the set of all ground instantiations of the rules in a rule list by $\mathit{ginst}(\mathit{RuleList})$:

$$
\begin{aligned}
&\mathit{ginst}([\mathit{rule}_1, \ldots, \mathit{rule}_n]) = \\
&\quad \{[\theta(\mathit{rule}_1), \ldots, \theta(\mathit{rule}_n)] \ \mid \\
&\quad \theta \in \Theta^{\mathrm{Gnd}} \ \wedge \ \mathit{dom}(\theta) \supseteq \bigcup_i \mathit{fv}(\mathit{rule}_i)\}.
\end{aligned}
$$

A linear fact models a resource that is available for consumption at most once. A persistent fact, on the other hand, is checked without consuming it in the firing of a rule and is never removed from the state. For a rule to be fired, all the facts contained in its premise must be present in the current global state. The execution of the rule consumes linear facts from the state and produces the corresponding facts present on the conclusions of the rule that are added to the global state.

5.4.1 Labelled Transition relation

The global system state is denoted by *State* $\subseteq^\sharp (\mathcal{T}^\sharp \cup^\sharp \mathcal{S}^\sharp)$, a multiset containing the state information. The state of labelled transition system consists of protocol state, fresh generated names $\mathcal{FN}$, the messages on the network, and attacker's knowledge MD specified in Sec. 3.5. The transition relation [123] below models *runs of the system* by rewriting the state with a ground instance of a rule list *RuleList*:

$$
\frac{(\mathit{rn} : l \ -[a] \rightarrow r) \in \mathit{ginst}(\mathit{RuleList} \cup MD \cup \{\mathcal{FN}\}) \qquad \mathit{lfacts}(l) \subseteq^\sharp \mathit{State} \quad \mathit{pfacts}(l) \subseteq \mathit{State}}{\mathit{State} \xrightarrow{\theta_1(a)} (((\mathit{State} \setminus^\sharp \mathit{lfacts}(l)) \cup^\sharp \theta(r))}
$$

where $\subseteq^\sharp$, $\setminus^\sharp$ and $\cup^\sharp$ denote operations for multisets. The renaming away substitution ensures that there is no confusion between "fresh" names introduced in the actions and conclusions, and the names that already exist in the state. The action label on the transition rule records the actions $\theta(a)$ generated by the firing of the rule.

5.4.2 Protocol Run and Trace

A *run* of a protocol is specified as a sequence of some ground instantiated rule (setup and protocol) applications:

$$\boxed{State_i, \quad (rn_i : l_i \ -[a_i] \rightarrow r_i), \quad State'_i}$$

with $State_0 = \{\}$ and each subsequent state is the result of the application of the corresponding rule:

$$\boxed{State_i \longrightarrow (a_i) State'_i}$$

where $State'_i$ is related to $State_{i+1}$ in some systematic way, $State_{i+1}$ is derived from $State'_i$ by replacing every output term $Out(t_1, \ldots, t_k)$ with the input term $In(t_1, \ldots, t_k)$. Since these are special terms representing interaction with an adversary in reasoning about the protocol, this corresponds to the case where there is no interference from an adversary.

A *trace* consists of the sequence of ordered ground actions resulting from a run, denoted as a trace as tr.

$$\boxed{tr = [(a_1), \ldots, (a_k)]}$$

Similarly, a *template* consists of the sequence of ordered actions containing variables, extracted from rules in their order present in the rule list, denoted as *template*. We discuss templates, and their applications, in detail in the next chapter.

5.5 Extensions to Labelled MSR in our Framework

To analyse the stealthiness of protocol traces, we extend the notions of *Fresh Names*, *Labels*, *Rules* and *Traces* in the Labelled Multiset Rewriting (MSR) System :

Fresh Names - Setup and Protocol: We divide the set of $fresh \in \mathcal{FN}$ into two namespaces : $fresh_{setup} \in \mathcal{FN}_{setup}$ such that $\mathcal{FN}_{setup} \subset \mathcal{FN}$ for fresh names generated by the setup rules and $fresh_{proto} \in \mathcal{FN}_{proto}$ for fresh names generated by the protocol rules such that $\mathcal{FN}_{proto} \subset \mathcal{FN}$. The setup generated fresh names and protocol generated fresh names must be disjoint, i. e., $\mathcal{FN}_{proto} \cap \mathcal{FN}_{setup} = \{\}$. The function $FrNames(tr)$ is used to extract all the fresh names present in the trace tr.

Labels - Basic and Logged: Various logging schemes may differ in what they log for a protocol. Net flow logs record the sender, receiver and time for each message. More in-depth logs will also include message types and values. Logs will usually never include key values or other security-sensitive information. Additionally, in some cases, it is not possible to log all the protocol steps. To allow flexibility and modelling of all such logging schemes, we extend the notion of labels in our model.

As stated earlier, the actions a are the labels in a labelled transition system defined using the rules $rn : l \ -[a] \rightarrow r$. These labels in turn are used to construct the trace of an execution. In our framework, the analysis of trace is based on the analysis of logs present in these labels. We assume that actions of all the rules by default contain *basic labels* that may or may not contain enough logging information.

We assume an auxiliary signature Σ_{log} that specifies, for each rule named rn, a constructor[1] c_{rn} for the log entry for the firing of that rule. The arity of that constructor is the number of parameters specified by $LogParams(rule)$. The custom log $\mathsf{c}_{rn} LogParams(rule)$ is then added to the action of each rule, where rn is the rule name for the $rule$.

[1] In the examples, for a rule named XXX, we name the log constructor $LogXXX$.

To analyse the protocols based on the logging information, the logs must contain the same set of parameters, wherever possible, in order to derive the meaningful conclusion. To facilitate such logging, we allow the users to add logs $Log \in \mathcal{L}^\sharp$ to the front of actions(labels) of protocol rules (defined in section 5.5). This enables the extension of labels by dividing them in two categories namely; *basic labels* $a_{Basic} \in \mathcal{T}^\sharp$ and *logged labels* $a_{Logged} \in \mathcal{T}_L^\sharp$ and $\mathcal{T}_L = \mathcal{T} \cup^\sharp \mathcal{L}$.

For the *Log*, any user-defined name along with any set of chosen parameters is permissible. Also, it is straight forward to extract a *Log* from *logged label*, i. e., extracting the first element from the action a, wherever $a \in \mathcal{L}$.

$$getLog(a) = getLog(ah :: as) = \begin{cases} ah & \text{if } a \in \mathcal{L} \\ \{\} & \text{if } a \in \mathcal{T} \end{cases}$$

In the tool based on our framework for detecting stealth attacks, presented in section 8.5, the process of adding the logs is undertaken by the user prior to stealthiness analysis.

Rules - Setup and Protocol: Some protocols require mandatory information such as details about public key infrastructure, fixed domain names etc. during their initial phase, in absence of which the protocol execution cannot be started. To facilitate such scenarios, we classify the labelled multiset rewriting rules in two categories, namely *Setup Rules* and *Protocol Rules.*

In our model, we allow modelling of processes such as distribution of public/private key pair to the agents, setting up symmetric key or session key etc. by the setup rules, while the actual protocol steps are executed using the protocol rules. The actions of these rules are defined using the basic and logged labels from 5.5. Since we need only protocol steps to generate and analyse its trace for stealthiness, we allow annotation of logs only on the *protocol rules.* So, while the actions of *Setup Rules* consists only of *basic labels*, the

Protocol Rules are allowed to have both *basic labels* and *logged labels.*

Our model expects a rule list consisting of both setup and protocol rules for any protocol with basic labels. Thereafter, based on analysis of the rules and parameters to be logged, our system mandates adding logs to the actions of all the *protocol rules* making them rules with *logged labels.* Our transition rules can now be partitioned in two sets of rules : $Rule_{Setup}$ and $Rule_{Proto}$ for setup and protocol rules respectively with:

Setup Rule defined as : $Rule_{Setup} = \{rn : l \ -[a_{Basic}] \rightarrow r \mid a_{Basic} \in \mathcal{T}^{\sharp}\}$

Protocol Rule defined as : $Rule_{Proto} = \{rn : l \ -[a_{Logged}] \rightarrow r \mid a_{Logged} \in \mathcal{T}_L^{\sharp}\}$

Traces - Protocol, Setup and Logged: When the trace is generated using the *ProtoRules*, we call such a trace a *Protocol Trace*, denoted by tr_{Proto}, containing both the basic and logged actions. Additionally, the *Logged Traces* $tr_{\log}$ are generated by extracting only the logs from the protocol traces generated earlier:

$$tr_{\log} = getLogs(tr_{proto})$$

$$= getLogs([(a_1), \ldots, (a_k)])$$

$$= [getLog((a_1), \ldots, getLog(a_k)]$$

While the notation tr is used to denote the trace generated by a given rule list, we use tr_{Setup} and tr_{Proto} to denote the traces generated by only the *SetupRules* and, *ProtoRules* respectively.

5.5.1 Restrictions on Setup and Protocol Rules

We now define the restrictions that every rule and rule list must follow. These restrictions ensure that the rule list is one that can be reduced using labelled transition relation defined in Sec. 5.4.1, and produce a run.

Definition 5.1 (Setup/Protocol Rule Restriction)**.** Both the **Setup Rule** and **Protocol Rule** must adhere to the following restrictions:

1. Apart from the user-defined $STATE$ facts, the premise can use only the input *In* fact and fresh *Fr* fact to generate only setup fresh names or protocol fresh names depending on it being either the setup rule or protocol rule respectively.
2. Apart from the user-defined $STATE$ facts, the conclusion can use only the output *Out* fact.

On the lines of $\mathit{RuleList} = [\mathit{rule}_1, \ldots, \mathit{rule}_n]$ denoting a sequence of rules, from now on, we will use the notations $\mathit{SetupRules} = [\mathit{Srule}_1, \ldots, \mathit{Srule}_n]$ to denote a sequence of setup rules, and $\mathit{ProtoRules} = [\mathit{Prule}_1, \ldots, \mathit{Prule}_n]$ to denote a sequence of protocol rules.

Based on the restrictions on rules as defined in Def. 5.1, we are now in position to list out the restrictions for the rule list.

Definition 5.2 (Setup/Protocol Rule List Restriction)**.** Both the sequence of setup rules *SetupRules* and sequence of protocol rules *ProtoRules* must satisfy, in addition to following the definition 5.1, all the following restrictions:

1. All the $STATE$ facts sharing same name will always have the same arity.
2. All $STATE$ facts must be unique, i. e., the same $STATE$ must not appear in more than once in the same premise or conclusion.
3. All the FRESH facts in the rule list must have different parameters.

We assume that the *setup rules* e. g., generation of public keys etc. can be executed in any order as long as they are executed before protocol rules dependent on them. This does not lead to loss of any generality as we expect all the required keys by the protocol rules to be already present in the state. This would be made possible by executing the setup rules enough number of times in advance.

The intuition behind these restrictions is drawn from TAMARIN semantics. These restrictions assure that the TAMARIN model written following them will always be parsed without errors and accepted for verification.

5.6 Running Example

Consider a simple protocol ($A \rightarrow B : A, \{nA\}_{pkB}$) where A sends to B its name with a *secret* nA encrypted with the public key of B, pkB. All the rules presented here are *protocol rules*. As discussed, we assume that the *setup rules*, for generation of public keys, are executed before protocol execution starts. This assumption has no side effect on the trace generated by the protocol, as the *setup rules* do not appear in the trace as they do not contain any logging information. This protocol can be modelled using a setup rule list $\mathit{SetupRules}_{\text{stealth}}$ comprising one rule *RegisterPK*

$$\mathit{RegisterPK} \ : \ [Fr(ltkB)] - [\,] \rightarrow [!Ltk(B, ltkB), !Pk(B, pk(ltkB)), Out(pk(ltkB))]$$

and a protocol rule list $\mathit{ProtoRules}_{\text{stealth}}$ consisting of the following three rules.

Example 5.1.

$$
\begin{aligned}
\mathit{A1} \ &: [\mathit{Fr}(nA), !\mathit{Pk}(B, pkB)] - [\mathit{LogA1}(A, nA), \mathit{OutA1}(A, nA)] \\
&\rightarrow [\mathit{Out}(A, aenc\{nA\}pkB)] \\
\mathit{B2} \ &: [\mathit{In}(A, aenc\{nA\}pkB), !\mathit{Ltk}(B, ltkB)]- \\
&[\mathit{LogB2}(A, B, nA), \mathit{InB2}(A, B, nA)] \rightarrow [\mathit{MSec}(A, B, nA)] \\
\mathit{X3} \ &: [\mathit{MSec}(A, B, nA)] - [\mathit{LogX3}(A, B, nA), \mathit{Secret}(A, B, nA)] \rightarrow []
\end{aligned}
$$

The first rule *A1* creates a fresh name (nonce) nA and sends it encrypted with the known public key pkB of B, along with the sender's identity on the public channel. The constructor *Out* denotes an output of a tuple of values (the identity of A and the encryption of the nonce nA under the public key of B). The label $\mathit{OutA1}(A, nA)$ records or logs the output from A.

The second rule *B2* models the receiver B receiving the message (using the constructor *In* to receive a tuple of values), and decrypting it using its private key $ltkB$. The *Out* and *In* constructors play a special role in protocol modelling, the former denoting a new fact that is known to the "adversary," who is then responsible for converting that to an input fact that is received later in the protocol[2].

The final rule *X3* models the value nA as shared secret between A and B. This rule is helpful for writing a secrecy lemma, using action label $\mathit{Secret}(A, B, nA)$, to verify the security guarantee enforced by the protocol.

All the three rules in rule list $\mathit{ProtoRules}_{\text{stealth}}$ contain logged labels as some custom logs (highlighted with *red*) have been added to the basic labels making them logged labels. In this example, for a rule named XXX, we have named the log constructor LogXXX. E. g., for the Prule_{A1}, the chosen log name is *LogA1* with $\mathit{LogParams}(\mathit{Prule}_{A1})$ chosen from parameters used in the rule such as nA, B, pkB etc. In our example, the logged labels

[2] We assume one of these constructors for each arity, but leave the arity implicit.

contain the logs store the principal names and nonces being used in the protocol run.

5.6.1 Protocol and Logged Traces

For the protocol presented in Example 5.1, if the rules *A1*, *B2* and *X3* are executed in given order, the protocol and logged traces thus formed will be :

$$\begin{aligned} tr_{proto} = \quad & [(\mathit{LogA1}(A,\ nA), \mathit{OutA1}(A,\ nA)), (\mathit{LogB2}(A, B, nA), \\ & \mathit{InB2}(A, B, nA)), (\mathit{LogX3}(A, B, nA), \mathit{Secret}(A, B, nA))] \\ tr_{log} = \quad & [(\mathit{LogA1}(A,\ nA)), (\mathit{LogB2}(A, B, nA)), (\mathit{LogX3}(A, B, nA))] \end{aligned}$$

5.7 Allowed Sequences of Protocol Rules

To say what the log of standard runs of the protocol should look like, we need to specify what is logged and the order in which we would expect the protocol steps to occur. An important novelty of our work is adding this to the standard labelled multiset rewriting system described above. Some protocols may allow more than one execution sequence, e. g., by skipping some protocol rules. For such cases, simply placing the protocol rules in order may not be sufficient, as there would be no way of knowing the correct sequence. At the same time, such restrictions are not required for the *SetupRules* and they can be executed in any order as long as they are generated before any protocol rules dependent on them. Therefore, we only define a function $\mathit{AllowedSequences}(\mathit{ProtoRules})$:

Definition 5.3 (Allowed Sequences)**.** For a rule list of protocol rules $\mathit{ProtoRules} = [\mathit{Prule}_1, ..., \mathit{Prule}_n]$, $\mathit{AllowedSequences}(\mathit{ProtoRules})$ is a function such that $\mathit{AllowedSequences}(\mathit{ProtoRules}) \subseteq \mathcal{P}(\mathit{ProtoRules})$ and for any $[\mathit{Prule}_{i_1}, \ldots, \mathit{Prule}_{i_m}] \in \mathit{AllowedSequences}(\mathit{ProtoRules})$, we have :

1. $i_j < i_k$ for $1 \leq j < k \leq m$, and
2. $[\mathit{Prule}_{i_1}, \ldots, \mathit{Prule}_{i_k}] \in \mathit{AllowedSequences}(\mathit{ProtoRules})$ for all $k \leq m$.

We require that all the allowed sequences, present in $\mathit{AllowedSequences}(\mathit{ProtoRules})$, can be executed using labelled transition relation defined in Sec. 5.4.1 with every rule in $\mathit{AllowedSequences}(\mathit{ProtoRules})$ executed *only once* for a single execution which is required to ensure that the logs in generated trace can be identified with their respective rules.

In other words, the allowable sequences are any set of sublists of *ProtoRules*, with the restriction that the order of the rules remain unchanged. The restriction on *AllowedSequences* that every rule list present must be able to execute ensures that the successive rules have a construction such that the conclusion of previous rules allow the premise of the next rule to execute. It is worth highlighting that the $\mathit{AllowedSequences}(\mathit{ProtoRules})$ in our model are prefix-closed only, although in a practical setting, they can be any order of rules allowed by the protocol specifications. We will write *AllowedSequences* when the rule list is clear.

Cases of *AllowedSequences*: The allowed sequences can generate multiple possible sequences of rules supporting various executions. We discuss some of them here.

Complete Execution: A complete run of the protocol is where all the protocol steps are fired in a pre-specified order as per protocol specification. E. g., for a sample protocol with rule list $\mathit{ProtoRules}_{\text{sample}} = [\mathit{Prule}_1, \mathit{Prule}_2, \mathit{Prule}_3, \mathit{Prule}_4]$, the allowed sequence will have the complete rule list as its member, i. e., $[\mathit{Prule}_1, \mathit{Prule}_2, \mathit{Prule}_3, \mathit{Prule}_4] \in \mathit{AllowedSequences}(\mathit{ProtoRules}_{\text{sample}})$, and hence will support complete execution.

Partial Execution: Some protocols may allow partial execution where the execution must start from the first step but may terminate at any stage such as in case of incomplete termination. Alternatively, the system might record a mix of complete and incomplete runs of the protocol. Such executions may be modelled by allowing prefix sublists of the complete run of the protocol. The allowed sequence, in such cases, can be represented by :

$$\mathit{AllowedSequences}(\mathit{ProtoRules}_{\text{sample}}) = \{[\mathit{Prule}_1], [\mathit{Prule}_1, \mathit{Prule}_2], [\mathit{Prule}_1, \mathit{Prule}_2, \mathit{Prule}_3], [\mathit{Prule}_1, \mathit{Prule}_2, \mathit{Prule}_3, \mathit{Prule}_4]\}$$

Prohibited Cases: For the above rule list $\mathit{ProtoRules}_{\text{sample}}$, though our model can generate various *AllowedSequences* such as Case 1 and Case 2 below. At the same time, Case 3 can never be generated, as *AllowedSequences* never places a rule more than once in any sequence.

Case 1 : $[\mathit{Prule}_1, \mathit{Prule}_2, \mathit{Prule}_3, \mathit{Prule}_4]$

Case 2 : $[\mathit{Prule}_1, \mathit{Prule}_2, \mathit{Prule}_4]$

Case 3 : $[\mathit{Prule}_1, \mathit{Prule}_2, \mathit{Prule}_3, \mathit{Prule}_2, \mathit{Prule}_4]$

We note that these orderings are expected only for the protocol rules modelling the protocol steps and not on the setup rules, such as those modelling key generations or key reveal events. It must be noted that any modification or annotations of the rules discussed from now on refer to only such rules performing protocol steps and not setup rules.

Every rule in an *AllowedSequences* is allowed to be fired only once. However, some protocols may allow few rule(s) to be fired more than once in a single run, e. g., in the third case, i. e., *prohibited cases* of *AllowedSequences* above. Such executions will generate traces with the same logs at different time points. Analysing these traces may be difficult, as the presence of more than one predecessor to a log is likely to cause conflicts. Additionally, allowing these *prohibited cases* of *AllowedSequences* will also violate the Def. 5.3. To handle such cases, we tweak the rule list by placing those protocol rules, appearing more than once in the protocol rule list, as many times as necessary. Each recurrence of any such rule must, however, is given a distinct rule names and distinct log names.

Original: $[Prule_1, Prule_2, Prule_3, Prule_2, Prule_4]$

Modified: $[Prule_1, Prule_2, Prule_3, Prule'_2, Prule_4]$

Such a modification allows us to support protocols firing some rule(s) more than once in a single run, while still adhering to our restriction of each rule being fired at most once. Since a Dolev-Yao attacker can execute any protocol rules in any order, the above modification makes such an execution possible and generate traces with logs appearing in different order. Furthermore, it is also possible to generate multiple unique traces by interleaving these traces.

5.8 Defining ‘Standard Trace’

As already explained, our model requires the setup rules to be executed, in any order, before executing the protocol rules in a specified order. These executions can be used to construct a *SetupState* comprising information about public and private keys of the prospective protocol participants. The protocol rules, executed subsequently, will make use of this *SetupState* to generate different traces. However, before proceeding to analyse these traces, we would like to see what does a trace of the system look like when the protocol rules are executed in the order of $AllowedSequences(\_)$ after the execution of setup rules.

Given a trace of a system, and the logs generated from this, there is no evidence of an attack if the log entries could have been generated by some interleaving of standard runs of the system. We now define this formally, first for a single run producing a *standard trace*, and then for any interleaving of allowed sequences, a *standard looking trace* produced by merging of multiple standard traces. For all the definitions in this section, we will also use the notation ASR to denote $AllowedSequences(ProtoRules)$.

Definition 5.4 (Standard Trace $\mathit{StdTrace}(\mathit{AllowedSequence}, \mathit{SetupState}, \theta, tr_{log})$)**.** Given an allowed sequence *AllowedSequence*, a setup state *SetupState*, and a substitutions list θ, tr_{log} is a *Standard Trace* if it can be generated by executing all the rules *AllowedSequence* in the order they appear in the list, from the *SetupState*, with the substitutions from θ in the labelled transition relation defined in subsection 5.4.1 where the only attacker rules allowed is passing the outputs to inputs and no modification.

We will just write this predicate as *StdTrace* if its arguments are implicit. Additionally, we will call a run of the system as a *'Standard Run'* if it produces a standard trace of the form *StdTrace* as per Def. 5.4. It is worth re-emphasising that while the *run* refers to the firing of rules, a *trace* refers to the sequence of logged labels.

It is also useful to be able to extract the log entries from a trace:

$$\mathit{Logs}(a) = \{\mathsf{c}(t_1, \ldots, t_k) \in a \;\mid\; (\mathsf{c} : k) \in \Sigma_{\text{log}}\}$$

$$\mathit{Logs}([a_1, \ldots, a_k]) = [\mathit{Logs}(a_1), \ldots, \mathit{Logs}(a_k)]$$

$$\mathit{Log}([a_1, \ldots, a_k], i) = \mathit{Logs}(a_i)$$

In what follows, we will write tr_{log} for a trace of a system with logs, i. e., a logged trace. It is pertinent to note that the logged trace tr_{log} is produced using only the logged labels of protocol rules containing ground instantiations for all the parameters in the logs generated during the protocol rule(s) execution. We assume a function $\mathit{merge}(tr_1, \ldots, tr_k)$ that computes the set of all possible traces resulting from interleaving $tr_1, \ldots, tr_k$ and is available in Appendix A.

In our system, the rules can use public names, fresh names, and variables. While the public names and variables can take any value, the *fresh names* must be generated every time using a new value. In order to test if the system adheres to this specification, we define the following functions, in order to extract the fresh variables and names used in the rule list and trace.

Definition 5.5 (*freshInRL*(RL))**.** Given a rule list RL, *freshInRL*(RL) returns all the arguments used in a fresh fact of the form $Fr(\_)$, and present in the premise of any rule in the rule list RL.

Depending on the rule list RL containing either the variables or the names, as a result of applying substitution to the rule list, the function may either return a set of variables used in the $Fr(\_)$ fact or all the elements of a fresh name set respectively. Subsequently, for a rule list RL partitioned into RL_{Setup} and RL_{Proto} for setup and protocol rules respectively, the functions *freshInRL*(RL_{Setup}) and *freshInRL*(RL_{Proto}) will return the arguments of facts present in the RL_{Setup} and RL_{Proto} respectively.

Definition 5.6 (*FrFromRLinTr*(θ, RL, tr))**.** Given a list of substitutions $\theta = [\sigma_1, \ldots, \sigma_i]$, rule list $RL = [R_1, \ldots, R_i]$, and a trace tr, *FrFromRLinTr*$(\theta, RL, tr) =$ *freshInRL*$(\theta(RL)) \cap FrNames(tr)$ where $\theta(RL) = [\sigma_1(R_1), \ldots, \sigma_i(R_i)]$

5.9 'Standard Looking Trace' and 'Stealth Attack'

The *standard trace* from the definition 5.4 produces what looks like a trace, generated from a single run of the system, based on execution of an allowed sequence of protocol rules. In a realistic setting, however, the system should be able to not only identify and analyse the trace produced by a single run, but also all such traces that are the product of multiple concurrent runs. We define a *standard looking trace* as a trace which is produced by execution of interleaving of multiple allowed sequences, i. e., product of merging of multiple standard traces, each of which is generated using Def. 5.4.

Definition 5.7 (Standard Looking Trace)**.** Given a set of allowed sequences ASR representing a protocol and a setup rule list *SetupRules*, the test trace tr_{test} is a **standard looking trace** if there exists a *SetupState* generated by executing the setup rule list *SetupRules* in any order and any number of times, allowed sequences $ASR_1, \ldots, ASR_k \in ASR$, traces $tr_1, \ldots, tr_k$ and lists of substitutions $\theta_1, ..., \theta_k$ such that for all tr_i:

$$\mathit{StdTrace}(ASR_i, \mathit{SetupState}, \theta_i, tr_i) \text{ holds, and}$$

$$\mathit{Logs}(tr_{\text{test}}) \in \mathit{merge}(\mathit{Logs}(tr_1), \ldots, \mathit{Logs}(tr_k)) \text{ and}$$

$$\mathit{FrFromRLinTr}(\theta_i, ASR_i, \mathit{Logs}(tr_i)) \cap \mathit{FrFromRLinTr}(\theta_j, ASR_j, \mathit{Logs}(tr_j)) = \{\} \text{ for } i \neq j$$

The first condition ensures that every trace, contained in the *Standard Looking Trace*, is a *Standard Trace.* The second condition requires that the logged trace tr_{log} is a combination of some collection of standard looking single traces containing only the logged labels, and the third condition ensures that all fresh names used in the traces must be unique, i. e., they should be generated new and fresh for each trace.

Additionally, the same *SetupRules* must be used to generate a unique setup state *SetupState* to be used by the protocol rules in advance. The *ProtoRules* may, however, be allowed to fire in any order, as mandated by *AllowedSequences*(*ProtoRules*), during the protocol execution.

Definition 5.8 (Attack Trace)**.** If a trace generated by a protocol execution demonstrates successful violation of any pre-defined security properties, with the pre-defined security properties being the guaranteed security properties by the protocol specification, then we call it an **Attack Trace**.

Definition 5.9 (Stealth Attack)**.** If a trace tr_{attack} is an attack trace of a protocol as per definition 5.8 and a standard looking trace for the protocol as per the definition 5.7, then the corresponding attack is a **Stealth Attack**.

We present application of these definitions, on a standard looking trace generated from Ex. 5.1, in the next chapter in Sec. 6.6.1 along with the challenges encountered in the manual analysis of such traces.

In the subsequent chapters, the terms “standard looking” and “normal looking” are sometimes used interchangeably in context of runs and traces.

5.10 Chapter Summary

In this chapter, we have developed a notion of standard trace, standard looking trace, and attack trace to define a *‘Stealth Attack’* using a formal model using a multiset-rewriting (MSR) system. We have presented multiple extension to the standard MSR, such as adding setup and protocol fresh names, basic and logged labels, and setup and protocol rules and traces etc. Using the definitions of standard trace, we have been able to define a standard looking trace which can be used to identify any given attack trace as *stealthy* or *non-stealthy*.

The protocol traces may contain any log multiple times. Though our system is able to verify the stealthiness of a protocol run by analysing traces, yet the challenge that remains is to map any given log to a particular instance of a protocol execution. We consider its solution in subsequent chapters. It is worth noting that in all our examples, the trace containing the logs should not store sensitive parameters as plaintext, instead, logs should only store hashed values of all such parameters.

Chapter Six

TAMARIN Model of Stealthiness

> *"Everything must be taken into account. If the fact will not fit the theory-let the theory go."*
>
> Agatha Christie

6.1 Motivation

In Chapter 5, we have presented formalised notions of stealthy attacks in protocols modelled as MSRs. The goal of this formalisation is to identify, if a protocol is vulnerable to stealthy attacks, particularly attacks that may be 'hidden' in the interleaving of several runs of the protocol. While using these notions may not be tricky, in a real-world implementation, exponential blow-up of multiple runs and their possible interleaving may make the analysis a complex endeavour. The main motivation of this chapter is to be able to test the stealthiness of attacks based on definitions presented in the previous chapter using the automatic protocol verification tool Tamarin. To solve the exponential blow-up of multiple runs, we propose to introduce an identifier, such as *session identifier* (session ID), to the logs in order to uniquely identify and map them to their respective runs. Subsequently, we also need to identify and formalise the restrictions to be placed on the traces generated by the Tamarin tool in order to make the traces equivalent to standard looking traces defined using Def. 5.7.

The notion of ‘stealthiness’ presented in Chapter 5, using Def. 5.7, and its implementation using the Tamarin presented in this chapter, using two restrictions of Correspondence and Uniqueness, differ in one aspect, i. e., the introduction of session identifiers in each log entry. Intuitively, there is nothing wrong in adding an identifier. However, it needs to be proved that annotating the protocol rules with session identifiers does not alter its behaviour. Accordingly, we attempt to provide a proof for Theorem 6.3 which states that there is an attack present in a trace with session identifiers, under two restrictions of Correspondence and Uniqueness, if and only if a stealth attack is present in a trace without session identifiers.

6.2 Contributions

This chapter presents our approach of implementing the formal stealthiness framework, presented in Chapter 5, using the automatic protocol verification tool Tamarin. Based on our introduction of session ID to logs, we identify equivalent restrictions to be imposed on protocol traces generated by Tamarin, in order to make Tamarin definitions of stealthiness equivalent to our formal model presented earlier in the previous chapter.

6.3 Overview

This chapter starts with the definition of well-formedness of rules followed by definition of validity of facts, variables, and rules. A Tamarin model usually allows any number of inputs or any number of outputs in the premise and conclusion of the rules. We, however, require a well-formed rule to have both input and output to a maximum of one. This well-formedness, along with validity conditions defined further, guarantees that multiple occurrences of the same variable always takes the same value. In the absence of well-formedness and validity conditions, we may have a trace with different substitutions for different occurrences of any variable. This will end up generating multiple possible traces thereby making analysis of traces almost an impossible task. It is worth mentioning here that both the well-formedness and validity conditions are already followed by most Tamarin examples.

Further, we introduce the notion of 'Standard Templates' as an ordered list of parameterised logs to generate multiple valid traces using different substitutions. We also introduce a unique identifier in the form of a 'session ID' to make a distinction among traces generated by the protocol runs. We add these session IDs to the rules, i. e., in the premise, conclusion and labels. We remark that the addition of session ID does not stop any rule from firing, hence the addition of session ID does not, in any way, alter the generation of any standard traces as per Def. 5.4. The only change would be the presence of session IDs in the traces.

We remark that the session IDs do not enforce anything on the system and adding the session IDs to the traces does not, in any way, guarantee that the traces will be stealthy or otherwise. The standard looking trace using Def. 5.7 is a result of interleaving many single standard traces. Though this merging is straightforward, extracting the original standard traces from a standard looking trace is a complex and an error-prone task. Similarly, if the session IDs are not added carefully to these traces, it may be possible that the modified traces do not follow stealthiness conditions.

To handle such situations, our system also mandates that if there is a trace generated using the rules without the session IDs, e. g., $AllowedSequences(ProtoRules)$, and if we add session IDs to the rules making them, there will exist a trace which we can un-annotate, and it will be the same. This is in line with our central theoretical result, i. e., an attack is present in a trace with session identifiers, under certain restrictions, if and only if a stealth attack is present in a trace without session identifiers. So, if a trace is stealthy with session ID under two restrictions, namely Correspondence and Uniqueness as per Def.6.22 and Def. 6.23, it should also be stealthy under Def. 5.7 after removing the session

IDs. This is also demonstrated in the Fig. 6.1 below.

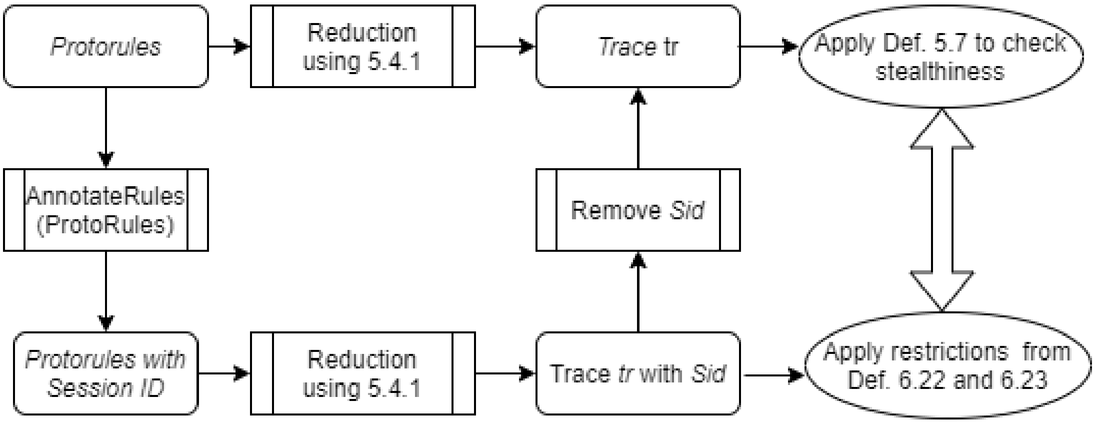

Protorules
Reduction using 5.4.1
Trace tr
Apply Def. 5.7 to check stealthiness
AnnotateRules (ProtoRules)
Remove Sid
Protorules with Session ID
Reduction using 5.4.1
Trace tr with Sid
Apply restrictions from Def. 6.22 and 6.23

Figure 6.1: Reduction of *ProtoRules* to Traces with and without session IDs

Using these fundamental tools, this chapter defines two restrictions; namely Correspondence and Uniqueness, to be enforced in a Tamarin model, so that all the traces generated by the system are always stealthy. If, under these restrictions, an attack is successful, it is termed as Stealth Attack else non-Stealth Attack.

Finally, using these definitions followed by propositions and theorems on templates and traces, we have provided proof for our central theorem that an attack is present in a trace with session identifiers, under certain restrictions, if and only if a stealth attack is present in a trace without session identifiers.

6.4 Well-formedness of Rules and Rule Lists

In general, Tamarin examples may have more than one input or more than one output in a rule. Modelling of such rules may result in wrong pair of output-input matching with each other. It is also possible for a protocol rule to not always produce an output. In such cases, a following rule with an input will have nothing to receive from a previous rule and the execution may abruptly come to a halt. Executing such protocol rules could, however, be made possible if we are able to match two such rules where the first rule does not produce an output and the second rule does not accept any input.

We define well-formedness of rules and rule lists to ensure that the situations defined above are handled. We place the restriction to allow every rule with at most one input or output. We also make matching of two rules possible, using the *STATE* facts, even when the first rule does not produce an output and the second rule does not accept any input. These conditions provide flexibility to our model by ensuring that we are able to match the input of a rule to the correct previous output from any rule, and not necessary the output of a previous rule only.

We enlist the above well-formedness conditions, for a rule and rule list in this section, by allowing possible construction for the rules, as defined below. It may be noted that these restrictions are in addition to ones already defined earlier in Definitions 5.1 and 5.2. Together, all these restrictions ensure that the rule list can be executed using the labelled transition relation present in Section 5.4.1. These definitions require all the Tamarin rules to have at most one output and one input, with no loss of generality.

Definition 6.1 (Well-Formed Rule)**.** Any well-formed setup and protocol rule, while following all the restrictions as defined in Def. 5.1, can have one of the following construction:

(a) *In* **to** *Out* **Rule:** A rule with single input in premise and single output in conclusion.

(b) *In* **to** *STATE* **Rule:** A rule with single input in premise and no output in conclusion.

(c) *STATE* **to** *Out* **Rule:** A rule with no input in premise and single output in conclusion.

(d) *STATE* **to** *STATE* **Rule:** A rule with no input in premise and no output in conclusion.

Definition 6.2 (Well-Formed Rule List)**.** Any well-formed setup and protocol rule, while following all the restrictions as defined in Def. 5.2, can have one of the following three forms:

1. **Rule List using no** ***In*****, i. e.,** *STATE* **to** *Out***:**

- A rule of the form (c) from Def. 6.1 and rule list of the form 2 in this definition.
- A rule of the form (d) from Def. 6.1 and rule list of the form 1 in this definition.

2. **Rule List using *In* and *Out*:**

   - A rule of the form (b) from Def. 6.1 and rule list of the form 1 in this definition.
   - A rule of the form (c) from Def. 6.1 and rule list of the form 2 in this definition.
   - A rule of the form (a) from Def. 6.1 and rule list of the form 2 in this definition.

3. **Rule List using *STATE*:**

   - A rule with one *In* or *Out* in its facts and rule list of the form 2 in this definition.
   - A rule with neither *In* nor *Out* in its facts and rule list of the form 1 in this definition.

6.5 Validity of Facts, Variables, and Rules

For our definitions to work, it is essential that all the rules present in the TAMARIN model of a protocol are valid as per definitions presented later in this section. We assume any construction of a rule and the rule list, which is parsed by TAMARIN without any error, as valid. The validity conditions ensure that all variables in the rule list always map to the same value. It is to be noted that most of the publicly available TAMARIN models already follow our validity conditions. It may be recalled that a *template* of a protocol is thttps://www.overleaf.com/project/5eb5eed7a084e00001ff411dhe sequence of actions containing variables that appear in the execution. A variable can appear multiple times in a rule list and hence in a template. However, to generate a valid run from a template, it is imperative that each variable has at most one value i. e., every valid variable must have the same value for all its occurrence in a rule list. This is necessary to ensure that any generated trace after substitution on a template is always a valid one.

We present below the formal treatment of validity of rule lists based on the validity of facts and variables used by them.

Motivating Example

In order to make it easier to understand our approach, we present the working of our definitions with the help of the following motivating example, considering the following rule list:

Example 6.1.

$$R1 : [Fr(ltkA)] - [] \rightarrow [Ltk('A', ltkA), Pk('A', pk(ltkA))]$$

$$R2 : [Fr(ltkB)] - [] \rightarrow [Ltk('B', ltkB), Pk('B', pk(ltkB))]$$

$$R3 : [Pk(R, pkR)] - [Log3(I, R, pkR)] \rightarrow [Out(aenc(< I, R >, pkR), Step(I, R, `Init')]$$

$$R4 : [Ltk(S, ltkS), In(aenc(< I, S >, pk(ltkS)), Step(I, S, `Init')] - [Log4(I, S, pk(ltkS))] \rightarrow []$$

Definition 6.3 (PN : Set of Public names)**.** Given a rule list RL, the set of public names PN consists of all the public names appearing in the rule list RL.

Applying Def. 6.3 on the Ex. 6.1: $PN = \{'A', 'B'\}$.

Definition 6.4 (Allowed fact types)**.** All the facts present in the premise and conclusion of every rule must belong to either of the following types of facts :

- **Fresh, Output and Input fact:** As already defined earlier, the facts, fresh Fr() and input In(), are used in premise of a rule for a new variable and receive a message. The output fact Out() is used in the conclusion of a rule to send the message.
- **Unique state fact (USF):** A unique state fact is a fact for which the state name only appears in the conclusion of a single rule. The USFs are used to maintain

temporary protocol state information and can be used to store the system memory of a transition system at any specific point.

- **Unique Argument state fact (UAF)** Unique Argument state facts are facts that appear in the setup rules such that all arguments of the fact are fresh names or constants, where the state name-constant pair is unique. The fresh and constant arguments of UAF should always remain fresh and constant respectively across all their instances. Since a fresh name is never reused whereas a constant value might, allowing them to mix might violate the correctness conditions A set of all such generated UAFs is referred to as UAF_{gen}. UAF can also appear in the protocol rules if they appear in the premise and conclusion with the same state name and arguments. All such instances form a set of UAFs referred to as UAF_{used}. The UAFs help in setting up the initial knowledge of a system, such as long term, private, and public keys etc.

  All the occurrences of the UAF must use either fresh names *or* public names in the same positions. The following **restrictions** are, therefore, applied on the UAFs :

  - Whenever a UAF uses a public name, this fact name also appears with all the possible public names in the same position.
  - Whenever a UAF uses a function application in a position, then all facts with the same state name have the same function application or a variable or a public name in the same position, but never a fresh name.
  - Whenever two UAFs with the same state name use a nested function application in the same position, then these function applications at each nesting level must also follow the above two restrictions.

Applying Def. 6.4 on the Ex. 6.1 We can classify all the facts as follows:

- Fresh fact: $Fr(ltkA), Fr(ltkB)$

- Output and Input fact:$Out(aenc(< I, R >, pkR)), In(aenc(< I, S >, pk(ltkS)))$
- USF: $Step(I, R, `Init')$
- UAF:

$$Ltk('A', ltkA), Ltk('B', ltkB), Pk('A', pk(ltkA))$$

$$, Pk('B', pk(ltkB)), Pk(R, pkR), Uaf1(R, z), Ltk(S, ltkS)$$

As already outlined, the validity of a rule list, composed of valid rules, is defined to ensure that every valid variable with the same name receive the same value. Additionally, we also want to make sure that all the variables which are being used for the same purpose also receive the same value after substitution, even though they may have different names. It may be recalled that a *template* of a protocol is the sequence of actions containing variables that appear in the execution.

Our aim here is to make sure that no two rules should be allowed to use the same variable for more than one purpose. This is to ensure that they are never substituted with the same value resulting in an invalid trace. For example, if a variable pkR denotes the public key of R in one place, everywhere it appears it still denotes the public key of R and nothing else. This restriction in enforced by making sure that they are "directly" or "indirectly" linked, as defined below.

Definition 6.5 (Directly Linked Variable)**.** A variable v is directly linked in a rule R w.r.t. the rule list RL, if either:

- variable v is present in the input fact in the premise of R and it is also present in the output fact in the conclusion of the most recent rule of RL, containing an output fact, at the same position.
- Or variable v is present in a USF in the premise of R and there exists a USF with the same state name present in the conclusion of some earlier rule in RL, with v in the same position.

Applying Def. 6.5 on the Ex. 6.1: We can extract the sets of directly linked variables from each rule as follows:
V_{D1} in $R1$ w.r.t. $RL = [] = \{\}$, V_{D2} in $R2$ w.r.t. $RL = [R1] = \{\}$, V_{D3} in $R3$ w.r.t. $RL = [R1, R2] = \{\}$, and V_{D4} in $R4$ w.r.t. $RL = [R1, R2, R3] = \{I\}$

While the definition 6.5 helps in establishing links among the variables present in input/output facts and unique state facts, it does not take into account variables linked indirectly via Unique Argument state facts (UAFs).

As the UAFs have unique arguments, if one argument for two different UAFs is the same, then we can conclude that the others are also all the same. E. g., we could consider the UAF $Pkey(X, pkX)$ that denotes that the public key of principal X is pkX. Given two occurrences of $Pkey(I, pkI)$ in a rule set, if we can show that both of the variables I are linked then, due to the uniqueness of the arguments, we can conclude that both of the pkIs also take the same value. So, following on from our earlier definition, we define:

Definition 6.6 (Indirectly Linked Variable)**.** A variable v is indirectly linked in a rule R w.r.t. the rule list RL, if there exist UAFs with the same state name in both the premise of the rule R and in the conclusion of a rule in RL, with variable v at the same position in both, and there exists a term t that is in the same position in both UAFs and t a variable directly linked[1] in R w.r.t. RL or t is a constant.

We note that these definitions could be extended to capture more complex links between variables, e. g., strings of indirect and direct links in different combinations. However, the definitions above are sufficient to capture everything we need for standard protocol models.

Applying Def. 6.6 on the Ex. 6.1: We find that there are no indirectly linked variables present in the rule list. V_{ID1} in $R1$ w.r.t. $RL = [] = \{\}$, V_{ID2} in $R2$ w.r.t. $RL = [R1] = \{\}$, V_{ID3} in $R3$ w.r.t. $RL = [R1, R2] = \{\}$, and V_{ID4} in $R4$ w.r.t. $RL = [R1, R2, R3] = \{\}$

[1] We could extend this definition to also allow t to be an indirectly linked variable, however, this is not needed for any of the examples we have looked at.

Sets of New Variables from Each Rule in Ex. 6.1: V_{New1} in $R1 = \{ltkA\}$, V_{New2} in $R2 = \{ltkB\}$, V_{New3} in $R3 = \{I, R, pkR, s\}$, and V_{New4} in $R4 = \{S, ltkS\}$

Based on the definitions 6.5 and 6.6, we are now in the position to define a valid variable in a rule with respect to the rule list. It is worth reminding that all valid variables would only take a single value in a normal run.

Definition 6.7 (Valid Variable)**.** A variable v in a rule R is valid w.r.t. a rule list RL if either:

- v is new, i. e., not used in RL
- or v is directly linked
- or v is indirectly linked

Applying Def. 6.7 on the Ex. 6.1 to test validity of variables: Set of All Variables in $R1 = \{ltkA\}$, Set of All Variables in $R2 = \{ltkB\}$, Set of All Variables in $R3$ $= \{I, R, pkR\}$, and Set of All Variables in $R4 = \{I, S, ltkS\}$.

- Variable $ltkA$ in $R1$ is valid w.r.t. $RL = []$ as $ltkA \in V_{New1}$
- Variable $ltkB$ in $R2$ is valid w.r.t. $RL = [R1]$ as $ltkB \in V_{New2}$
- Variable I in $R3$ is valid w.r.t. $RL = [R1, R2]$ as $I \in V_{New3}$
- Variable R in $R3$ is valid w.r.t. $RL = [R1, R2]$ as $R \in V_{New3}$
- Variable pkR in $R3$ is valid w.r.t. $RL = [R1, R2]$ as $pkR \in V_{New3}$
- Variable I in $R4$ is valid w.r.t. $RL = [R1, R2, R3]$ as $I \in V_{D4}$
- Variable S in $R4$ is valid w.r.t. $RL = [R1, R2, R3]$ as $S \in V_{New4}$
- Variable $ltkS$ in $R4$ is valid w.r.t. $RL = [R1, R2, R3]$ as $ltkS \in V_{New4}$

Definition 6.7 defines the validity of one variable in one rule w.r.t. one rule list. We further define validity of all the variables in a rule by applying this definition on all variables of a rule. Subsequently, a valid rule list is defined as a rule list with each rule having all its variables as valid.

Definition 6.8 (Valid Rule)**.** A given rule R is a valid rule w.r.t. a rule list RL if all the facts in R and RL are allowed facts as per definition 6.4 and all of its variables are valid variables as per definition 6.7.

Applying Def. 6.8 on the Ex. 6.1 to test validity of rules:

- The rule $R4$ is valid w.r.t. $RL = [R1, R2, R3]$
- The rule $R3$ is valid w.r.t. $RL = [R1, R2]$
- The rule $R2$ is valid w.r.t. $RL = [R1]$
- The rule $R1$ is valid w.r.t. $RL = []$

Definition 6.8.1 (Valid Rule List)**.** A rule list $RL = [R_1, R_2....R_n]$ is a valid rule list if :

- the rule list RL is empty.
- or, rule R_n is a valid rule w.r.t. the rule list $[R_1, ...R_{n-1}]$ with $[R_1, ...R_{n-1}]$ a valid rule list.

Applying Def. 6.8.1 on the Ex. 6.1 to test validity of rule list: The rule list $RL = [R1, R2, R3, R4]$ is a valid rule list.

6.5.1 Valid and Well-Formed Rule List

Based on the above definitions, we define a valid and well-formed rule list as below.

Definition 6.9 ($WFAndValidRLSetupProto(\mathit{SetupRules}, \mathit{ProtoRules})$)**.** Given setup and protocol rule lists namely *SetupRules* and *ProtoRules*, $WFAndValidRLSetupProto(\mathit{SetupRules}, \mathit{ProtoRules})$] holds if:

- *SetupRules* contains only basic logs and no fresh proto names.
- *ProtoRules* is valid as per Def. 6.8.1.
- *SetupRules ProtoRules* are both well-formed as per Def. 6.2.

Definition 6.9.1 (WellFormedAndValidRuleList(RL))**.** A given rule list *RL* is valid if there exists two rule lists namely *SetupRules* and *ProtoRules* such that *RL* is a result of concatenation of *SetupRules* with *ProtoRules* and $WFAndValidRLSetupProto(\mathit{SetupRules}, \mathit{ProtoRules})$ holds.

From now on, all references to the *SetupRules* and *ProtoRules* shall imply well-formed and valid setup rule list and well-formed and valid protocol rule list, respectively. Additionally, we assume that the outputs from the setup rules are never received as input by the protocol rules. The information sharing among the setup rules and protocol rules always takes place using the persistent facts (modelled as $STATE$ facts in our model).

6.6 Introducing Session Identifier

In order to test our model using Tamarin, we start by defining a *Standard Template* based on the definition of *template* defined in 5.4.2 and subsequently applying our 'Formal Stealth Model', from the previous chapter 5, on the running example presented in 5.6. It may be recalled that the *AllowedSequences* in all our examples are prefix-closed, although in a practical setting, they can be any order of protocol rules as per the protocol specifications.

Definition 6.10 (Standard Template)**.** Given the allowed sequences $\mathit{AllowedSequences}(\mathit{ProtoRules})$ for the protocol rule list *ProtoRules* with logs, a *Stan-*

dard Template, denoted by $\mathit{StdTemplate}(\mathit{ProtoRules})$, can be generated by extracting the logs with variables in the order of any allowed sequences.

The set of all such *Templates* constitute the set of *Standard Templates*, denoted by *StdTemplates*. A standard template is, therefore, an ordered list of parameterised logs ordered by the time point of their occurrence. As a protocol may have more than one allowed execution sequence, and therefore more than one standard template, there may be multiple templates for a protocol.

As already explained earlier, the logged trace tr_{log} is produced using only the logged labels of protocol rules containing ground instantiations for all the parameters in the logs generated during the protocol rule(s) execution, a standard template $tr_{template}$ will only contain variables. A standard template $tr_{template}$ can be used to generate different traces by applying different substitutions, with each substitution generating of traces by substituting the parameters of $tr_{template}$ with correctly typed terms.

A trace, generated by the Tamarin rules execution, is the result of applying a specific substitution on the action labels. The standard templates are helpful in order to be able to generate multiple traces from any list of logged labels. We will use standard templates in subsequent sections to develop our Tamarin model of stealthiness by adding the session IDs to develop the notion of standard session templates which, as we will see later, will help us define the two required restrictions for stealthiness.

6.6.1 Applying Formal Stealth Model to Running Example

We will now use our framework to analyse the modified sample protocol presented in the Ex. 5.1. We assume that the protocol execution is prefix-closed. The set of standard templates for this protocol, i. e., *StdTemplates* will then have the following form :

$$\begin{aligned}&\mathit{StdTemplates}(\mathit{ProtoRules}_{\text{stealth}}) =\\ &\quad \{[\mathit{LogA1}(A, nA)],\ [\mathit{LogA1}(A, nA), \mathit{LogB2}(A, B, nA))],\\ &\quad [\mathit{LogA1}(A, nA), \mathit{LogB2}(A, B, nA), \mathit{LogX3}(A, B, nA)]\}\end{aligned}$$

Based on our formal stealth model definitions from Section 5.9, we analyse the following two test traces for their stealthiness :

$$\begin{aligned}tr_{\text{test1}} =& [\mathit{LogA1}(\mathit{Aly}, \mathit{n1}), \mathit{LogA1}(\mathit{Aly}, n2), \mathit{LogB2}(\mathit{Aly}, \mathit{Rob}, \mathit{n2}),\\ & \mathit{LogA1}(\mathit{Kim}, \mathit{n3}), \mathit{LogB2}(\mathit{Kim}, \mathit{Rob}, \mathit{n3})]\end{aligned}$$

It is trivial to prove that $tr_{\text{test1}} \in \mathit{merge}(tr_1, tr_2, tr_3)$ with $tr_i \in \theta_i(\mathit{StdTemplates})$, $i = 1, 2, 3$, with $\theta_1 = \{\mathit{Aly}/A, \mathit{Bob}/B, \mathit{n1}/nA\}$, $\theta_2 = \{\mathit{Aly}/A, \mathit{Rob}/B, n2/nA\}$ and $\theta_3 = \{\mathit{Kim}/A, \mathit{Rob}/B, n3/nA\}$, classifying tr_{test1} as a standard looking trace. On the other hand, if we have the following test trace:

$$\begin{aligned}tr_{\text{test2}} =& [\mathit{LogA1}(\mathit{Aly}, \mathit{n1}), \mathit{LogB2}(\mathit{Aly}, \mathit{Bob}, \mathit{n1}),\\ & \mathit{LogB2}(\mathit{Aly}, \mathit{Rob}, \mathit{n2}),\\ & \mathit{LogA1}(\mathit{Kim}, \mathit{n3}], \mathit{LogB2}(\mathit{Kim}, \mathit{Rob}, \mathit{n3})]\end{aligned}$$

We find that this trace cannot be generated by the formal stealth model definitions for any combination of traces, as $[\mathit{LogB2}(\mathit{Aly}, \mathit{Rob}, n2)]$ can not be generated by from templates by applying $\sigma(\mathit{StdTemplates})$, for any substitution $\sigma = [\theta_1, \ldots, \theta_i]$, and is therefore *not* a standard looking trace.

While this analysis is straight forward, the difficulty arises due to the exponential blow-up in the complexity of checking interleaving of several runs, and the lack of an existing support for this in any protocol checking tool. Our approach to managing this complexity is to introduce the notion of *session identifiers* to disambiguate, for each log entry, which run of the protocol that log entry corresponds to. We define a function $\mathit{AnnotatedRules}(\mathit{ProtoRules})$ to add session identifiers to the rule list $\mathit{ProtoRules}$.

Definition 6.11 ($\mathit{Sessions}(\mathit{ProtoRules})$)**.** Given the allowed sequences $\mathit{AllowedSequences}(\mathit{ProtoRules})$ for the protocol rule list $\mathit{ProtoRules}$ with logs, we define the augmentation of the allowed sequences annotated with session ID as $\mathit{Sessions}(\mathit{ProtoRules}) = \mathit{AnnotatedRules}(\mathit{AllowedSequences}(\mathit{ProtoRules}))$ where

$$\begin{aligned}
&\mathit{AnnotatedRules}([\mathit{ProtoRules}_1, \dots, \mathit{ProtoRules}_k]) = \\
&\qquad \mathit{AnnotatedRules}(\mathit{ProtoRules}_1)@\cdots@\mathit{AnnotatedRules}(\mathit{ProtoRules}_k) \\
&\mathit{AnnotatedRules}((rn : l\ -[a]\to r) :: \mathit{ProtoRules}) = \\
&\qquad (rn : l'\ -[a']\to r') :: \mathit{AnnotatedRules}(\mathrm{sid}, \mathit{ProtoRules})
\end{aligned}$$

where $l' = l \cup^\sharp \mathit{Fr}(\mathrm{sid})$, $r' = r \cup^\sharp \mathit{Out}(\mathrm{sid})$,
$a = \mathsf{c}_{rn}(t_1, \dots, t_k) \cup^\sharp a_0$, $a' = \mathsf{c}_{rn}(\mathrm{sid}, t_1, \dots, t_k) \cup^\sharp a_0$ and

$$\begin{aligned}
&\mathit{AnnotatedRules}(\mathrm{sid}, [rn : l\ -[a]\to r]) = \\
&\qquad [rn : \mathrm{sid}, l'\ -[a']\to r] \\
&\mathit{AnnotatedRules}(\mathrm{sid}, (rn : l\ -[a]\to r) :: \mathit{ProtoRules}) = \\
&\qquad (rn : l'\ -[a']\to r') :: \mathit{AnnotatedRules}(\mathrm{sid}, \mathit{ProtoRules})
\end{aligned}$$

where $l' = l \cup^\sharp \mathit{In}(\mathrm{sid})$ and $r' = r \cup^\sharp \mathit{Out}(\mathrm{sid})$.

For each of the possible allowed sequences of protocol rules, the annotation works by generating a fresh session identifier, firing the fresh rule $Fr(sid)$ with variable name sid), in the premise of first rule followed by the same being output by every rule except the last one and accepted as input by every subsequent rule. $\mathit{AnnotateRules}(\_)$ also adds this session ID to each log entry. Hence, a single correct run of the protocol will be tagged with a unique session ID. We require that sid is not used, as a variable name, in the original rules *ProtoRules*.

In line with the function $\mathit{AnnotatedRules}(\mathit{ProtoRules})$ used to add session IDs to the rule list, we also assume a corresponding function $\mathit{UnAnnotateRules}(\mathit{ProtoRules})$ which can be used to remove the session IDs such that:

$$\mathit{ProtoRules} = \mathit{UnAnnotateRules}(\mathit{AnnotatedRules}(\mathit{ProtoRules}))$$

We will just write *UnAnnotateRules* when the rule list is implicit.

We note that these annotated rules do not enforce any restrictions on when a rule can fire, therefore any sequence of possible rules using $\mathit{AllowedSequences}(\mathit{ProtoRules})$ is also possible using $\mathit{Sessions}(\mathit{ProtoRules})$. However, this lack of restriction also means that when multiple runs of the protocol happen at the same time, it is quite possible for a rule to pick up the wrong session ID from an environment. It is also possible for the attacker to interfere with the session IDs, replaying or altering them. In other words, there are no restrictions enforced by our annotated rules that mean the session IDs are used correctly.

This means that it is not the case that multiple runs of the protocol, with no interference from the attacker, will necessarily produce a trace in which the session IDs correctly tag standard looking single runs. However, we will show below that it is possible to find a protocol attack in which session IDs tag standard looking runs if, and only if, there is a stealth attack as per Definition 5.9. I. e., the addition of session IDs makes checking for the existence of stealth attacks for the original protocol rules tractable.

Definition 6.12 (Standard Session Templates)**.** For a given rule list *ProtoRules*, we define the *standard session templates* as

$$\begin{aligned}\mathit{Templates}_{\text{Sessions}}(\mathit{ProtoRules}) = \{&\mathit{StdTemplate}(\mathit{ProtoRules}') \mid \\ &\mathit{ProtoRules}' \in \mathit{Sessions}(\mathit{ProtoRules})\}\end{aligned}$$

We will write just $\mathit{Templates}_{\text{Sessions}}$ when the rule list is implicit.

Lemma 6.1. Give any protocol rule list *ProtoRules*, for all traces generated using the normal rule list *AllowedSequences*(*ProtoRules*), there exists a trace with session IDs generated from the rule list of the form *Sessions*(*ProtoRules*) such that if we remove the session IDs using *UnAnnotateRules*, the traces generated are equal.

Proof. As can be seen in Fig. 6.1, the labels present in the original trace will be generated by the firing of rules from *ProtoRules* in sequence. All the rules, after annotation, can still fire, in the same sequence, as long as there is no change in the conclusion premise relationship among the neighbouring rules pair. The firing of the rules is dependent on the conclusion and premise relation which does not get changed in any way by Def. 6.11 as it only adds session identifiers.

Hence, addition or removal of session identifiers using *AnnotatedRules*((*ProtoRules*)) and *UnAnnotateRules*((*ProtoRules*)) does nothing to stop any rules from firing, implying that the Lemma holds. □

Lemma 6.2. Give any protocol rule list *ProtoRules*, for all traces generated using rule list of the form *Sessions*(*ProtoRules*), there exists a matching trace without session IDs.

Proof. Follows directly from Lemma 6.1. □

6.7 Adding Sessions to Running Example:

Applying Definition 6.11 to our rule list $\mathit{ProtoRules}_{\text{stealth}}$ from the Ex. 5.1, we get the following rule list $\mathit{Sessions}(\mathit{ProtoRules})$ with the changes highlighted in *red*.

$$\begin{aligned}
A1\ &: [Fr(nA), Fr(sid), !Pk(B, pkB)]- \\
&[LogA1(sid, A, nA), OutA1(A, nA)] \\
&\rightarrow [Out(A, aenc\{nA\}pkB), Out(sid)] \\
B2\ &: [In(sid), In(A, aenc\{nA\}pkB), Ltk(B, ltkB)]- \\
&[LogB2(sid, A, B, nA), InB2(A, B, nA)] \\
&\rightarrow [MSec(A, B, nA), Out(sid)] \\
X3\ &: [In(sid), MSec(A, B, nA)]- \\
&[LogX3(sid, A, B, nA), Secret(A, B, nA)] \rightarrow []
\end{aligned}$$

Applying Def. 6.12, we get the *standard session templates* as:

$$\begin{aligned}
&\mathit{Templates}_{\text{Sessions}} = \\
&\quad \{[LogA1(sid, A, nA)],\ [LogA1(sid, A, nA), LogB2(sid, A, B, nA))], \\
&\quad [LogA1(sid, A, nA), LogB2(sid, A, B, nA), LogX3(sid, A, B, nA)]\}
\end{aligned}$$

6.8 Substitutions on Templates and Standard Traces

Our model produces a Template from a set of rules that is made up of the logged action we expect to see. For example, the template might look like the following:

$$[Msg1(I, R, pkI, Na), Msg2(R, I, pkR, Na, Nb), Msg3(I, R, pkI, Nb)]$$

where I, R, pkI, pkR, Na and Nb are all variables.

A run of the protocol will produce a trace in which these variables all take values, e. g.:

$$[Msg1(`A', `B', pk(15244), 6385), Msg2(`B', `A', pk(9647), 6385, 7421), Msg3(`A', `B', pk(15244), 7421)]$$

We want to be sure that '1') every standard run of a system will match the template for some substitution and '2') if we substitute values into the template we get a trace that can be produced by a standard run. However, without additional conditions, these two requirements will not hold in all cases.

For '1', we must have that every occurrence of a variable anywhere in the template will take the same value at run time. E. g., in the template above, the Na variable in $Msg1$ must always take the same value as the Na variable in $Msg3$. If a run of the protocol exists in which these two variables take different values, then there will not exist a substitution that will make the template match that run.

To ensure that '1' does hold we restricted, in the Sec. 6.5, our valid protocol rules to ensure that there is a link between all occurrences of variables with the same name (e. g., the variable is used as an output in one and a matching input in the following rule). It is worth reminding that all example protocol models already follow these restrictions.

'2' is more complex. We may have two variables in the template that represent the same value, e. g., if one rule calls the two principals A and B and another rule uses I and R, then we will need to make sure that A and I take the same value, and B and R take the same value. Also, the protocol rules can enforce different restrictions on different variables. E. g., in the template above I and R will be public names, Na and Nb fresh names, whereas pkA must be the function $'Pk'$ applied to a fresh name. Therefore, when going from a template to a trace generated by a run, we need to ensure that any substitution respects these restrictions. To do this we first build equivalence classes of names that will take the same value, i. e., those names that are linked by the rules. We then restrict the possible facts the protocol rules must use, e. g., the facts must always use public names in the same positions and fresh names in the same positions.

We then look to see if each variable is linked to a public name, a fresh name or a function application, and we restrict the substitution to only consider those values. We will then try to prove these restrictions are enough to prove '1' and '2'.

6.8.1 Substitutions Properties on Variables and Rule lists

For valid rules, the substitutions used in each step of the reduction of the normal or standard run must use the same values for variables with the same name. We first prove this for a single directly linked variable:

Proposition 1. Given a valid rule list $RL = [R_1, \ldots, R_j]$ and a standard run of the protocol with rules using $\sigma_1, \ldots, \sigma_j$ as substitutions for the reductions in each step of the run. If the variable v is directly linked as per Def 6.5 in the rule R_j w.r.t. $[R_1, \ldots, R_{j-1}]$ then either v does not appear in $[R_1, \ldots, R_{j-1}]$ or there exists some σ_i such that $\sigma_i(v) = \sigma_j(v)$ and $i \neq j$.

Proof. We have to prove Prop 1 that for each directly linked variable as per Def. 6.5, the substitution being used in the current rule is the one that has already been used earlier for this variable.

We can use induction on length of rule list with the following two cases:

- Base case: We have $RL = []$, for a single rule R_1, all the variables are new, i. e., do not appear in RL, hence Prop 1 holds.

- Step case: Assume Prop 1 holds for the valid rule list $RL = [R_1, \dots, R_j]$ and the variable v is directly linked as per Def 6.5 in the rule R_j w.r.t. $[R_1, \dots, R_{j-1}]$ then we have two cases:

  – New Variables: Since a variable does not appear earlier, the substitution used for this is new and hence Prop 1 holds.

  – Directly Linked variables: Use Def. 6.5 which will have further two cases: **Variable in successive Input Output:** Using Def.6.4, we can show that an input fact consumes the output fact of the previous rule and in order to do that, all of its variables will always need to have same value, should the output fact need to be consumed during the normal run, hence the current substitution σ_j and σ_i, with $i = 1, \dots, (j-1)$, must agree on all the variables used at same positions in successive input output.

    Variable in Unique State Fact: Using Def.6.4, we can show that as USF with a state name is produced only once, all of its variables will always need to have the same value, should this USF need to be consumed during the normal run, hence the current substitution σ_j and σ_i, with $i = 1, \dots, (j-1)$, must agree on all the variables used at same positions in USF.

□

Next, we show that indirectly linked variables must also take the same value as a previous variable:

Proposition 2. Given a valid rule list $RL = [R_1, \dots, R_j]$ and a normal run of the protocol with rules using $\sigma_1, \dots, \sigma_j$ as substitutions for the reductions in each step of the run. If the variable v is indirectly linked as per Def 6.6 in the rule R_j w.r.t. $[R_1, \dots, R_{j-1}]$ then either v does not appear in $[R_1, \dots, R_{j-1}]$ or there exists some σ_i such that $\sigma_i(v) = \sigma_j(v)$ and $i \neq j$.

Proof. We have to prove Prop 2 that for each indirectly linked variable as per Def. 6.6, the substitution being used in the current rule is the one that has already been used earlier for this variable. We can use induction on length of rule list with the following two cases:

- Base case: We have $RL = []$, for a single rule R_1, all the variables are new, i. e., do not appear in RL, hence Prop 2 holds.
- Step case: Assume Prop 2 holds for the valid rule list $RL = [R_1, \ldots, R_{j-1}]$ and the variable v is indirectly linked as per Def 6.6 in the rule R_j w.r.t. $[R_1, \ldots, R_{j-1}]$ then we have two cases:
  - New Variables: It does not appear in the rule list $[R_1, \ldots, R_{j-1}]$ so far, therefore the Prop 2 holds by definition.
  - Indirectly Linked variables: Use Def. 6.6 which will have further two cases:
    - Variable in another UAF that are directly linked: We assume the UAF is in R_i, Prop 1 tells us that this variable, say v_d, takes the same value such that $\sigma_j(v_d) = \sigma_i(v_d)$ for $i = 1, \ldots, (j-1)$. By Def. 6.4, all the arguments for UAFs with the same name must be unique (because the arguments are either fresh names, or unique constants). Therefore, the only possible match for the UAF in R_j is the UAF in R_i. By the definition allow sequence the rule can fire, therefore we know that the UAFs do match, and σ_j and σ_i agree on all variables in the UAF, i. e., $\sigma_j(v) = \sigma_i(v)$.
    - Constant c at the same position: Using Def.6.4, we can show that the generated UAF in the setup rules UAF_{gen} with a state name has a unique state name-constant pair. For this UAF to be consumed in the protocol rules, it will use the same substitution during the normal run, hence the current substitution σ_j must have been present earlier, i. e., in $\sigma_1, \ldots, \sigma_{j-1}$.

□

We now use these two propositions to show that all variables in a template for valid rules must take the same value:

Proposition 3. Given a valid rule list RL and a normal run of the protocol, let $\sigma_1, \dots \sigma_n$ be the substitutions used for the reductions in each step of the run for the variables being used in the RL, then there exists a substitution σ such for all σ_i and $v \in \mathit{domain}(\sigma_i), \sigma(v) = \sigma_i(v)$.

Proof. We have to prove Prop 3 for each valid variable, i. e., for all σ_i and $v \in \mathit{domain}(\sigma_i)$, defined by Def. 6.7 in the rule list $[R_1, \dots, R_{j-1}]$, showing that all the individual substitutions $\sigma_1, \dots, \sigma_n$ can be replaced by a single σ.

We can use induction on length of rule list with the following two cases:

- Base case (length 1): We have a single rule R_1 and a single substitution σ_1, hence $\sigma_1 = \sigma$,

- Step case: Assume Prop 3 holds for substitutions $\sigma_1, \dots, \sigma_{n-1}$ and we have to show it is true for σ_n. To prove this, we replace all $\sigma_1, \dots, \sigma_{n-1}$ with a single substitution σ and prove that σ is also compatible with σ_n. This will require proving the following three cases:

  – New Variables: We can assume a new σ_n and add this to σ as σ_n has not been seen before, so does not conflict with existing substitutions.

  – Directly Linked variables as per Def. 6.5: Apply Prop 1

  – Indirectly Linked variables as per Def. 6.6: Apply Prop 2

□

6.8.2 Analysing Templates and Traces

Proposition 4. Given a rule R, and a template $tr_{template}$ generated from R, there exists a trace tr_{log}, generated from the normal run of R, and a substitution σ such that $\sigma(tr_{template}) = tr_{log}$.

Proof.

- Using the rule R and Def. 5.4, we get a trace tr_{log} and a substitution σ.
- Applying σ on the template $tr_{template}$ gives us the trace tr_{log}, i. e., $\sigma(tr_{template}) = tr_{log}$.

□

We can now use these proposition to show that the following Theorem 6.1, specifying relationship between standard trace, template and substitutions, holds.

Theorem 6.1. Given a valid rule list RL, for every standard trace with logs tr_{log}, there exists a template $tr_{template}$ of RL and a substitution σ such as $\sigma(tr_{template}) = tr_{log}$.

Proof. We can use induction on length of rule list with the following two cases:

- Base case (length 1): For a single rule R_1 and a single substitution σ_1, using Prop. 4, we can prove $\sigma_1 = \sigma$,
- Step case: Assume Theorem 6.1 holds for the valid rule list $RL' = [R_1, \ldots, R_{n-1}]$ and a standard trace, say $tr_{log'}$, generated by standard run of RL', then there exists a template $tr_{template'}$ and substitution σ' such that $\sigma'(tr_{template'}) = tr_{log'}$. We then need to prove that for the valid rule list $RL = [R_1, \ldots, R_n]$ with corresponding standard trace, say tr_{log}, there will exist a template $tr_{template}$ and substitution σ such that $\sigma(tr_{template}) = tr_{log}$. We have two cases:
  - If R_n has no log, the standard run using the valid rule list RL will also produce the trace tr_{log} exactly the same as $tr_{log'}$. Hence, we can assume an

unchanged template $tr_{template} = tr_{template'}$ using the same substitution as the earlier substitution $\sigma = \sigma'$, such that $\sigma(tr_{template}) = tr_{log}$.

- If R_n has a log, we can build a substitution, say $\sigma' = \{\sigma_1, \ldots, \sigma_{n-1}\}$ generating a trace $tr_{log'}$ from a normal run of a valid rule list $RL' = [R_1, \ldots, R_{n-1}]$. We know that the rule list $RL = [R_1, \ldots, R_n]$, will add log with values to the generated trace tr_{log}. Using Prop. 3, with RL' using σ', followed by R_n using σ_n, we can prove that, for the trace tr_{log}, there exists a single substitution σ on RL and hence there exists a template $tr_{template}$ such that $\sigma(tr_{template}) = tr_{log}$.

□

6.9 Rewriting Templates using Equivalence of Names

Sec. 6.5 defines the validity of variables and subsequently validity of rule lists based on them. It is evident from these definitions that the facts can use different names for identifying the same principal/message/key etc. leading the templates to use different names for the same value. It is possible for a public key to be referred to by two different names, e. g., pkS and $pk(ltkS)$. It means that the *Template* generated from the example rule list, i. e., $[Log3(I, R, pkR), Log4(I, S, pk(ltkS))]$ can have multiple substitutions to its terms. To make sure that the substitutions applied to the *Template* are valid, all such variables denoting the same principals/messages/keys etc. must be substituted with the same values. We present the equivalence definition that finds the names in the rule list that must be the same, e. g., pkS and $pk(ltkS)$. We start by scanning the protocol rule list, to identify all names that are to be treated as equivalent, in the following relation $Equivalent(n_1, n_2)$:

Definition 6.13 ($Equivalent(n_1, n_2)$)**.** Given a valid rule list RL, the relation $Equivalent(n_1, n_2)$, for two terms (variables, public names and function applications)

n_1 and n_2, is the smallest symmetric, reflexive, transitive relation such that:

- if n_1 and n_2 are present in the same position in an output and then the first input fact appearing in one of the subsequent rules then $\mathit{Equivalent}(n_1, n_2)$.
- if n_1 and n_2 are arguments in the same position of unique state facts in RL with the same state name then $\mathit{Equivalent}(n_1, n_2)$.
- if n_1 and n_2 are arguments in the same position of UAFs in RL with the same state name, and there exists n_3 and n_4, both in the same positions, and none of them equal to either n_1 or n_2,with n_3 not in the same position as n_1 and n_4 not in the same position as n_2, and $\mathit{Equivalent}(n_3, n_4)$ then $\mathit{Equivalent}(n_1, n_2)$.
- $\mathit{Equivalent}(n_1, n_2)$ with $n_1 = [x_1, \ldots, x_i]$ and $n_2 = [y_1, \ldots, y_i]$ holds if $\mathit{EquivalentList}(n_1, n_2)$.

Definition 6.14 ($\mathit{EquivalentList}(l_1, l_2)$)**.** Given a valid rule list RL, the relation $\mathit{EquivalentList}(l_1, l_2)$, for two lists l_1 and l_2 with $l_1 = [x_1, \ldots, x_i]$ and $l_2 = [y_1, \ldots, y_i]$ holds if $\mathit{Equivalent}(x_1, y_1)$ and $\mathit{EquivalentList}([x_2, \ldots, x_i], [y_2, \ldots, y_i,])$

Applying Def. 6.13 to check Equivalence in Ex. 6.1: In the rule list of Ex. 6.1, we have a set of all the variables in the rule list = $\{ltkA, ltkB, I, R, S, pkR, ltkS\}$, a set of all public names in the rule list = $\{'A',' B'\}$, and a set of all function applications in the rule list = $\{aenc(< I, R >, pkR), aenc(< I, S >, pk(ltkS)), pk(ltkS)\}$

Applying the Def. 6.13, we find all the names equivalent to itself, such as $\mathit{Equivalent}(I, I)$ etc. Additionally, $\mathit{Equivalent}(R, S)$ and $\mathit{Equivalent}(pkR, pk(ltkS))$ also hold.

Definition 6.15 (Equivalence Classes)**.** Given a valid rule list RL, an equivalence class E_n for a term n is the set of all the names equivalent to it using $\mathit{Equivalent}(n, \_)$ as per Def 6.13, i. e., $E_n = \{x \mid \mathit{Equivalent}(n, x)\}$.

Applying Def. 6.15 to build Equivalence class from Ex. 6.1: As per Def 6.15, all the Equivalence classes contain itself such as $E_{ltkA} = \{ltkA\}$,$E_{ltkB} = \{ltkB\}$ etc. Additionally, we also have $E_R = \{R, S\}$ and $E_{pkR} = \{pkR, pk(ltkS)\}$.

Clearly the equivalence class of terms should not contain terms that can never be equal, such as two different public names, two different function applications, or a public name and a function application.

Furthermore, we want to be sure that any standard run will always map the variables in the equivalence class to the same values:

Proposition 5. Given a valid allowed sequence list RL and a standard run of the protocol, let σ be a substitution that can be used for all reductions, then for all terms n in the rule list we have that $\forall\ n_1, n_2 \in E_n$ implies $\sigma(n_1) = \sigma(n_2)$.

Proof. From the Def. 6.15, $n_1, n_2 \in E_n$ implies $Equivalent(n_1, n_2)$ as per Def. 6.13. To prove that $\sigma(n_1) = \sigma(n_2)$, we use proof by induction on derivation of equivalence:

- **Base Case 1:** n_1 and n_2 are present in the same position in an output and then the first input fact appearing in one of the subsequent rules. The standard run of the protocol using valid rule list RL is possible only if previous output with n_1 matches the input with n_2 implying $\sigma(n_1) = \sigma(n_2)$.

- **Base Case 2:** n_1 and n_2 are arguments in the same position of USF in RL with the same state name. USFs are generated only in a single place therefore n_1 and n_2 must take the same value, as they are generated by the standard run hence must match, implying $\sigma(n_1) = \sigma(n_2)$.

- **Base Case 3:** Given $Equivalent(n_1, n_2)$, $Equivalent(n_1, n_2)$ follows directly from induction on equivalence being a symmetric relation if $n_1 = n_2$, hence $\sigma(n_1) = \sigma(n_2)$ must be true.

- **Step Case 1:** Given *Equivalent*(n_2, n_1), *Equivalent*(n_1, n_2) follows directly from induction on equivalence being a reflexive relation, therefore $\sigma(n_1) = \sigma(n_2)$ must be true.

- **Step Case 2:** Given *Equivalent*(n_1, n_3) and *Equivalent*(n_3, n_2), *Equivalent*(n_1, n_2) follows directly from induction on equivalence being a transitive relation, therefore $\sigma(n_1) = \sigma(n_2)$ must be true.

- **Step Case 3:** n_1 and n_2 are arguments in the same position of UAFs in RL with the same state name., and there exists n_3 and n_4, both in the same positions, and none of them equal to either n_1 or n_2, with n_3 not in the same position as n_1 and n_4 not in the same position as n_2 and *Equivalent*(n_3, n_4). By the induction hypothesis, we have $\sigma(n_3) = \sigma(n_4)$ and since the UAF, as per Def. 6.4, requires for the state name-constant pair to be unique, and we know that as ASR, the RL will execute and $\sigma(n_1) = \sigma(n_2)$ must be true.

- **Step Case 4:** *Equivalent*(n_1, n_2) with $n_1 = [x_1, \ldots, x_i]$ and $n_2 = [y_1, \ldots, y_i]$ holds, as the induction hypothesis tells us *Equivalent*(x_1, y_1), *Equivalent*$(x_2, y_2), \ldots,$ *Equivalent*(x_2, y_2) hold, implying *EquivalentList*(n_1, n_2) also holds and therefore $\sigma(n_1) = \sigma(n_2)$ must be true.

□

Lemma 6.3. For all valid rule list RL and equivalences classes of term from that rule list E_n, for all $n_1, n_2 \in E_n$:

- if n_1 is a public name then $n_1 = n_2$ or n_2 is a variable.

- if $n_1 = f(x_1, \ldots, x_n)$ then $n_2 = f(y_1, \ldots, y_n)$ or is a variable.

Proof. Follows directly from Prop 5 above. □

6.9.1 Classification of Equivalence Class

The generation of a trace of a normal or standard run from a template requires that each member of an equivalence class generated by the application of Def. 6.15 be substituted with the same value/name. This substitution has to not only make sure that each appearance of a given term is substituted with the same value, but also make sure that all the terms appearing in the template are substituted together. In order to achieve this, we need to determine all such equivalence classes that appear together in any UAF and can be considered linked to each other. E. g., a principal 'A' and its public key pkA must be linked together. This is followed by computing the minimal equivalent set, providing us with the smallest possible set of all the linked equivalent classes. The following functions help us establish such relationships.

Two equivalent classes are said to be linked to each other if at least one member from both equivalent classes is used in any UAF, present in the rule list, as its arguments.

Definition 6.16 ($\mathit{LinkedEqClasses}_{RL}(E_1, E_2)$)**.** Given a valid rule list RL, $\mathit{LinkedEqClasses}_{RL}(E, E')$ is the smallest reflexive, transitive, symmetric relation that includes: For all E_1 and E_2 such that there exists $arg_{e1} \in E_1$ and $arg_{e2} \in E_2$ and there exists a uaf in RL that includes arg_{e1} and arg_{e2} arguments then $\mathit{LinkedEqClasses}_{RL}(E_1, E_2)$.

Applying Def. 6.16 in Ex. 6.1: We have $\mathit{LinkedEqClasses}_{RL}(E_R, E_{pkR})$ and $\mathit{LinkedEqClasses}_{RL}(E_R, E_{ltkS})$.

Definition 6.17 (EquivEqClasses(RL))**.** Given a valid rule list RL, EquivEqClasses(RL) are the equivalence classes produced by the $\mathit{LinkedEqClasses}_{RL}(E, E')$ relation.

It may be recalled that the *SetupState* is generated by executing the setup rules in any order any number of times. This *SetupState* is then used by the protocol rules to receive the knowledge about long term keys, private and public keys etc. It may also be recalled that such information is stored in UAFs of the rules, with $UAF \in UAF_{gen}$ generating them in setup rules and $UAF \in UAF_{used}$ consuming them in the protocol rules. E. g., $Pk(`A', pkA)$ has the principal name $`A'$ appearing together with its public key pkA. Therefore, to ensure the validity of substitutions, we need to make sure that $`A'$ and pkA are always substituted together. We write the following function returning all unique arguments facts, from the protocol rule list, that use members of an equivalence class.

Definition 6.18 ($ProtocolUAFs(RL, E_q)$)**.** Given a valid rule list RL and an equivalence class of names, $E_q \in EquivEqClasses(RL)$ then $ProtocolUAFs(RL, E_q)$ equals a list of all UAFs in protocol rule of RL which contain a member of E_q as an argument.

Applying Def. 6.18 in Ex. 6.1: We have the list of UAFs in protocol rule list as $[Pk(R, pkR), Ltk(S, ltkS)]$.

The list of UAFs $[Pk(R, pkR), Ltk(S, ltkS)]$ tells us that R and pkR are the names that go together and so must be substituted together, and so do S and $ltkS$. However, we still need to find out what we can replace these terms with. For this purpose, we write the function $MatchingSetupUAFs(RL, E_q)$ to extract all the possible substitutions for the list produced by $ProtocolUAFs(RL, E_q)$.

Definition 6.19 ($MatchingSetupUAFs(RL, E_q)$)**.** Given a valid rule list RL and an equivalence class of equivalence classes of names E_q, $MatchingSetupUAFs(RL, E_q)$ returns a list of UAFs from setup rules matching the list of UAFs returned by $ProtocolUAFs(RL, E_q)$ as per Def. 6.18 such that

- All members of an equivalence class present as the arguments of the UAFs in $ProtocolUAFs(RL, E_q)$ must match to the same value present as arguments in the $MatchingSetupUAFs(RL, E_q)$ while sharing the same UAF names and appearing in the same position as an argument.

Applying Def. 6.19 in Ex. 6.1: We have the list matching UAFs in setup rule list extracted as: [[Pk('A',pk(ltkA)),Ltk('A',ltkA)], [Pk('B',pk(ltkB)),Ltk('B',ltkB)]]

6.9.2 Template Rewriting

Sec. 6.9 presents the equivalence definition, $\mathit{Equivalent}(n_1, n_2)$, to finds the names in the rule list that must be same. While the template generated can use different names for the same value, e. g., pkS and $pk(ltkS)$, it is important that the both names, such as pkS and $pk(ltkS)$, must be substituted with same values.

In order to make this process simple, rather than trying to find out how to substitute multiple names with the same value, we rewrite the template itself to achieve the same result.The Def. 6.20.3, presented below, makes sure that all terms, marked as equivalent by the Def. 6.15, are rewritten with a single term. This rewritten template ensures that there is no ambiguity in applying substitution on templates, and that it will always produce the same rewritten template. E. g., the names pkS and $pk(ltkS)$, in the template will always be rewritten as $pk(ltkS)$. The issue is that whenever we have a template such as $Log1(S, pkS), Log2(S, Pk(ltkS))$ producing a trace $Log1(5, 123), Log2(5, Pk(456))$, it would be left to the trace analysis to establish if the value 123 is same as $pk(456)$. However, a rewritten template of the form $Log1(S, pk(ltkS)), Log2(S, Pk(ltkS))$ solves this issue by generating a trace such as $Log1(5, pk(456)), Log2(5, Pk(456))$, and removing the ambiguity altogether.

Definition 6.20 ($PreRewriteTemplate(T, RL)$)**.** Given a valid rule list RL and a template T for this rule list, let $E_{t1}, \ldots, E_{tm}$ be the equivalence classes of all terms in T. $PreRewriteTemplate(T, RL)$ can generate a rewritten template T' based on the following rules until we have a template which satisfies normal form definition as per Def. 6.20.1:

rule A: if there exists $f(t_1, \ldots t_i) \in E_{ti}$ then we rewrite all members of E_{ti} with the function and apply these rewrite rules to each of the arguments, i. e., $\mathit{Rewrite}(E_{ti}) = f(\mathit{Rewrite}(E_{t1}), \ldots, \mathit{Rewrite}(E_{ti}))$.

rule B: if there exists a public name in E_{ti}, we rewrite all members of E_{ti} in T with this public name.

rule C: if there are no public names, function applications or UAF variables in E_{ti}, then we pick one variable from E_{ti} and replace all members of E_{ti} in T with this.

We make $Rewrite(\_)$ and therefore $PreRewriteTemplate(\_,\_)$ deterministic such that; for each equivalence class containing function, public name or variable, only the first such term is chosen to represent the whole class.

Furthermore, the template rewriting is an iterative process and hence needs to have a terminating condition. For this purpose, we define the following predicate $TempNormalForm(T, RL)$ to test if the template T is in its normal form w.r.t. the rule list RL.

Definition 6.20.1 ($TempNormalForm(T, RL)$)**.** Given a valid rule list RL and a template T for this rule list, $TempNormalForm(T, RL)$ holds if, for every term $t1$ present in T, all the following conditions hold:

- There does not exist any other term, e.g., t, such that both t and $t1$ appear in the same equivalence class.
- If there exists a function term tf such that $t1$ and tf are in the same equivalence class then $t1$ and tf have the same function name.
- If there exists a public name tp such that $t1$ and tp are in the same equivalence class then both the terms $t1$ and tp are equal to each other.

After rewriting the template as per Def. 6.20 such that it satisfies all the conditions present in Def. 6.20.1, we need to instantiate the terms used in UAF as per the following definition.

Definition 6.20.2 ($InstantiateTemp(T, RL, T')$)**.** Given a template T generated from a valid rule list RL, for all protocol UAFs Uaf_p appearing in RL, there exists a setup UAF Uaf_s such that $Uaf_s = \sigma(Uaf_p)$ and $T' = \sigma(T)$.

Lemma 6.4. Given a valid rule list RL and a template T, there always exists a template T' such that T' is a rewritten template using $PreRewriteTemplate(T, RL)$ from Def 6.20, and T' is in normal form as per Def. 6.20.1.

Proof. Follows directly from the Def. 6.20 and Def. 6.20.1. □

Definition 6.20.3 ($RewriteTemplate(T, RL)$)**.** Given a valid rule list RL and a template T for this rule list, T' is the rewritten template of T such that:

- There exists a template T_{tmp} which is produced using Def. 6.20 on T.
- The template T_{tmp} is in normal form as per Def. 6.20.1.
- The template T' is an instantiated template of T_{tmp} by applying Def. 6.20.2.

Template Rewriting in Ex. 6.1: Using Def 6.20.3, the equivalence classes used in the *Template* generated from the rule list $[Log3(I, R, pkR), Log4(I, S, pk(ltkS))]$ are: E_I, E_R, E_{pkR}.

- Using rule A, $pk(ltkS) \in E_{pkR}$, hence $Rewrite(pkR) = Rewrite(E_{pk(ltkS)}) = pk(Rewrite(E_{ltkS})) = pk(ltkS)$ making the rewritten template $T' = [Log3(I, R, pk(ltkS))$ $, Log4(I, S, pk(ltkS)]$.
- The rule B does not apply.
- As per rule C, variables R and S are $\in E_R$, so all their occurrences will be replaced with R making the rewritten template $T' = [Log3(I, R, pk(ltkS)), Log4(I, R, pk(ltkS)]$.

The rewritten template $T' = [Log3(I, R, pk(ltkS)), Log4(I, R, pk(ltkS)]$. satisfies Def. 6.20.1 and hence in a normal form.

Instantiating Rewritten Template in Ex. 6.1: Using Def 6.20.2, the instantiation of template $T' = [Log3(I, R, pk(ltkS)), Log4(I, R, pk(ltkS)]$ takes the following form.

- Taking into account $[Pk(R, pkR), Ltk(R, ltkR)]$ as $ProtocolUAFs(RL, E_q)$, and $[Pk('A', pk(ltkA)), Ltk('A', ltkA]$ as $MatchingSetupUAFs(RL, E_q)$, we have a substitution σ matching $Pk(R, pkR)$ with $Pk('A', pk(ltkA))$, hence instantiating R with $'A'$ and pkR with $pk(ltkA)$ and producing $T'' = [Log3(I,' A', pk(ltkA)), Log4(I,' A', pk(ltkA)]$ as a rewritten and instantiated template.

As described at the start of this section, the process of rewriting and instantiating the template ensures that the system receives a template which is easy to substitute and therefore comparing traces is easier than before. E. g., both R and S used in $Log3(I, R, pkR)$ and $Log4(I, S, pk(ltkS))$ appear as $'A'$ in the rewritten template, thereby removing any possibility of ambiguity in the generated traces from this rewritten template.

Lemma 6.5. Given a valid rule list RL and a template T, the rewritten template using $RewriteTemplate(T, RL)$ from Def 6.20.3 does not contain any variable v with a function equivalent representation present in any equivalence class, i. e., $\forall\ v \nexists\ f, Equivalent(v, f(\_))$.

Proof. Follows directly from the rule A of Def. 6.20. □

Theorem 6.2. Given a well-formed and valid rule list RL, a template T generated using RL, the substitution σ applied on all the variables of the rewritten templates $tr_{template}$ generated using $RewriteTemplate(T, RL)$ from Def 6.20.3, there exists a standard run of the system using RL which generates the same trace tr_{log} matching this substitution on $tr_{template}$.

Proof. We can use induction on length of rule list with the following two cases:

- Base case (length 1): For a single rule R_1 and a single substitution σ_1, using Prop. 4, we can prove that $\sigma(tr_{template}) = tr_{log}$.
- Step case: Assume that for a rewritten template $tr_{template'}$ generated from a valid rule list $RL' = [R_1, \ldots, R_{n-1}]$ there exists a standard trace for RL', say $tr_{log'}$, and substitution σ' such that $\sigma'(tr_{template'}) = tr_{log'}$, we then need to prove that for all rules R_n such that $RL = [R_1, \ldots, R_{n-1}, R_n]$ is a valid rule list, and for all rewritten templates $tr_{template}$ generated from RL there exists a standard trace for RL say tr_{log} and substitution σ such that $\sigma(tr_{template}) = tr_{log}$. We have two cases:
  - If R_n has no log, the length of rewritten templates $tr_{template'}$ and $tr_{template}$ will be same. However, $tr_{template}$ may be a differently rewritten template compared with $tr_{template'}$ with only difference between them being variable names (i. e., there exists a function ($f : var \rightarrow var$) which can rewrite $tr_{template'}$ to $tr_{template}$. Let σ' and σ be the substitutions used in $tr_{template'}$ and $tr_{template}$ respectively, to generate the trace $tr_{log} = tr_{log'}$. We can then use the function f to generate both the new rewritten template $tr_{template}$ and substitution $sigma$ such that $\sigma = f \cdot \sigma'$ with $\sigma(tr_{template}) = tr_{log}$
  - If R_n has a log, we can build a substitution, say $\sigma' = \{\sigma_1, \ldots, \sigma_{n-1}\}$ generating a trace $tr_{log'}$ from a normal run of a valid rule list $RL' = [R_1, \ldots, R_{n-1}]$. The traces $tr_{log'}$, generated from RL', and tr_{log} generated from RL will be same except the last element added by R_n. As we already have a σ' for the rule list $RL' = [R_1, \ldots, R_{n-1}]$, following on from the above case, we can use a substitution $\sigma'' = f \cdot \sigma'$ on template $tr_{template}$ for variables from $[R_1, \ldots, R_{n-1}]$. Let σ_n be the substitution applied on the variables used in R_n, then Prop. 3 tells us that there exists a single substitution σ replacing all individual substitutions $f \cdot \sigma_1, \ldots, f \cdot \sigma_{n-1}, \sigma_n$, for all the variables used in RL, and $\sigma(tr_{template}) = tr_{log}$.

□

6.10 Enforcing Stealthiness in TAMARIN

After ensuring, through Theorem 6.2, that all the runs of the system can be generated by substituting the rewritten templates, in this section we present two restrictions on traces of a protocol with session IDs, which together force protocol runs with session IDs to be 'standard looking'. We show below that this is equivalent to standard looking for protocols without session IDs (as defined in Definition 5.7). Importantly, the restrictions below can be automatically checked in TAMARIN, whereas Definition 5.7 cannot be directly checked.

From now on, in this chapter, all the reference to the *StdTemplates* will imply a template to be of the form of a standard session template $\mathit{Templates}_{\text{Sessions}}(\_)$ rewritten using Def. 6.20.3.

A trace could be made up of many runs of a protocol, some of which have not yet finished. To define when the logs in a particular trace appear in the expected order, we define the following helper function, which tells us when the m-th element of a trace with session IDs (tr_{log}) could have been generated by a single run of a protocol matching the template tr_{template} rewritten using Def. 6.20.3:

Definition 6.21 ($\mathit{Correspondence}(tr_{\text{template}}, tr_{\text{log}}, m, \sigma)$)**.** Given a trace tr_{log}, an index m, a substitution σ, and a rewritten template tr_{template}, $\mathit{Correspondence}(tr_{\text{template}}, tr_{\text{log}}, m, \sigma)$ holds if there exists indexes $i_1, \ldots, i_l$ such that $i_j < i_{j+1} < m$ for $j \in \{1, \ldots, (l-1)\}$ and it holds that

- $\sigma(tr_{\text{template}})$ is a subsequence of tr_{log}
- last label present in $\sigma(tr_{\text{template}})$ is equal to the label present in tr_{log} at position m, i.e., $tr_{\text{log}}m = \mathit{Log}(\sigma(tr_{\text{template}}), l+1)$ with length of tr_{template} being (l+1)

- and $Log(tr_{\text{log}}, i_k)$ (for all k) and $Log(tr_{\text{log}}, m)$ use a single session ID that appears nowhere else in the trace before m.

This definition means that $Correspondence(tr_{\text{template}}, tr_{\text{log}}, m, \sigma)$ is true if m is the index of the last action of what looks like a correct single run of the protocol following the template tr_{template}. I.e, the m-th action does not look suspicious. We also note that we are using a single substitution for all elements of the trace, while the reduction rule for traces allows different substitutions for each step. We have already shown, using Theorem 6.1 in Sec. 6.8.2, that given the restrictions on our rules, that such a single substitution exists.

It is worth highlighting that $AllowedSequences(ProtoRules)$ presents protocol rules and their possible valid ordering of execution. Hence, execution of each allowed sequence it will generate a valid trace, and hence a valid template. Also, we use rewritten templates, using $RewriteTemplate(T, RL)$ as per Def. 6.20.3, to generate the following restriction.

Definition 6.22 (Correspondence Restriction)**.** Given a logged template $tr_{\text{log}} = [a_1, \ldots, a_n]$, a rule list *ProtoRules* and $Templates_{\text{Sessions}}(ProtoRules)$ as per Def. 6.12, we say that the *Correspondence Restriction* holds if, for $i = 1, \ldots, n$ for each log entry a_i present in the logged template, there exists a substitution σ and a rewritten standard session template $tr_{template'}$ such that $tr_{template'} = RewriteTemplate(tr_{\text{template}}, ProtoRules)$ as per Def. 6.20.3, generated from one of the Allowed Sequences, such that all the previous log entries are present in the same order, i. e., $Correspondence(tr_{\text{template}'}, tr_{\text{log}}, i, \sigma)$ holds for the generated trace.

We remark that the above restriction can be formulated using any rewritten templates produced by $RewriteTemplate(tr_{\text{template}}, ProtoRules)$, as per Def. 6.20.3, as they differ only in the variable names, and so they will restrict the traces in same way.

Lemma 6.6. The Correspondence Restriction, as per Def. 6.22, holds for every standard looking trace as defined in the Def. 5.7.

Proof. Every standard looking trace is a result of merging one or more traces, each of which is a result of normal run producing a standard trace as per Def. 5.4. Every standard trace guarantees that the log labels will be present in the trace in the order of rules provided by *AllowedSequences*.

However, the major difference in the traces being considered here is that the session IDs are present in the logs used by Def. 6.22 whereas there is no session IDs in the logs in Def. 5.7.

In Fig 6.1, for every standard looking trace of the form tr, there exists a trace of the form tr_{sid} with session IDs, following correspondence restrictions, such that if we remove the session IDs from tr_{sid}, this trace is same as tr. As the correspondence restriction holds for tr_{sid}, it will hold for tr also.

Also, the Correspondence Restriction may not hold for any other trace of the form $tr \notin StdTraces$. Since a normal run can never produce any trace of the form $tr \notin StdTraces$, hence by contradiction, the Correspondence restriction must hold for every standard looking trace. □

Lemma 6.7. If the Correspondence Restriction as per Def. 6.22 holds for a trace tr, then every element of standard looking trace tr will match a standard template as per Def. 6.10.

Proof. Follows from Def. 6.22 and Lemma 6.6. □

As the allowed sequences must be prefix-closed, for any element of the trace we wish to test that happens to be the final element of one of the possible patterns of standard protocol runs, there is a template for a standard run which ends on this element of the trace and for which all other required log actions appear in the correct order in the test trace.

We additionally require that, for a run to generate a standard looking trace, fresh names generated by the protocol rules are different between sessions. Recall that each run of a protocol involves the introduction of a new session identifier (the same name is used in the protocol specification, but the semantics of rule application ensures that this is renamed apart from the rest of the trace) for each run. Similarly, every other fresh name being used in the trace must have a unique value for each running instance of the protocol. We define a *Uniqueness Restriction* implying that any fresh value, generated by a protocol rule, is never shared across protocol runs.

Definition 6.23 (Uniqueness Restriction)**.** Given a logged template of the form $tr_{\text{log}} = [a_1, \ldots, a_n]$ containing all the logs 'a_i' of the form $Log_i(sid_i, pars_i)$, a generated by the protocol rule list *ProtoRules*, the *Uniqueness Restriction* enforces that a fresh name, generated by a protocol rule, will be present in any of the two different logs *if and only if* they use the same session identifier. Accordingly, we define the following restriction using tr_{log} and apply it to the trace tr:

$$\forall\ Log_i, Log_j, x \in \mathit{freshInRL}(RL_{proto}) \text{ as per Def. 5.5}, sid_i, sid_j.$$

$$Log_i(sid_i, pars_i) \wedge Log_j(sid_j, pars_j) \wedge (x \in (pars_i \cap pars_j)) \Rightarrow (sid_1 = sid_2)$$

Lemma 6.8. The Uniqueness Restriction, as per Def. 6.23, holds for a standard looking trace as defined in the Def. 5.7.

Proof. Third condition in Def. 5.7 for a standard looking trace tr mandates that $\forall\ tr_i \in$ *StdTraces*, it must be true that:

$$\forall\ i, j.\ \mathit{FrFromRLinTr}(\theta_i, ASR_i, Log(tr_{\text{log}}, i)) \cap$$

$$\mathit{FrFromRLinTr}(\theta_j, ASR_j, Log(tr_{\text{log}}, j)) \ = \{\}\text{for } i \neq j$$

Hence, the Uniqueness Restriction defined in Def. 6.23 should hold by Def. 5.7. However, due to presence of session IDs in the Uniqueness Restriction (Def. 6.23), we can show that this Lemma follows from Lemma 6.6. □

Lemma 6.9. If a trace tr_{log} matches a template from tr_{template}, both the Correspondence and Uniqueness Restrictions hold.

Proof. The proof follows directly from Def. 6.22 and 6.23. □

The standard looking trace from Def. 5.7 works even without the session IDs whereas the Correspondence and Uniqueness restrictions from Def. 6.22 and 6.23, required for a stealthy trace in this chapter, works only with the session IDs. To prove that they are equivalent, we can now state our main theorem which tells us that the correspondence and uniqueness restrictions together are equivalent to our formal definition of stealth from Chapter 5.

Theorem 6.3. Given a valid and well-formed *SetupRules*, and a valid and well-formed protocol rule set with custom logs *ProtoRules* as per Def. 6.9, *AllowedSequences*(*ProtoRules*), and a given security property there is a stealth attack against *ProtoRules* with trace tr if, and only if, there is an attack trace, exhibiting the same attack, with session ID tr' using the rules in *Sessions*(*ProtoRules*)) that conforms to the Correspondence and Uniqueness Restrictions, such that tr equals the trace tr' with session ID removed.

Proof. In this proof, we only consider the stealth attack traces, i. e., the traces that lead to attacks and look normal. So, assuming an attack trace, it will look normal, if and only if our stealthiness restrictions hold. If there is no attack trace, we ignore the trace and do not claim anything.

In the 'if' direction, we assume that for *ProtoRules*, *AllowedSequences*(*ProtoRules*), and a given security property, there is a stealth attack against *ProtoRules* with trace tr_{att}. Definitions 5.7 uses individual substitutions corresponding to each step of a trace. E. g., the trace tr_1 may use multiple substitutions $\sigma_{1_1}, \ldots, \sigma_{1_i}$ corresponding to each of its parts

$tr_{1_1}, \ldots, tr_{1_i}$. As it follows from Theorem 6.1, all these substitutions may be replaced by a single substitution, i. e., σ in order to generate the trace tr_1. Therefore, Definitions 5.7 and 6.10 along with Theorem 6.1 then tell us that there exists $tr_1, \ldots, tr_k$ and $\sigma_1, \ldots, \sigma_k$ such that $\forall i \; tr_i \in \sigma_i(\mathit{StdTemplates})$ and $tr_{\text{att}} \in \mathit{merge}(tr_1, \ldots, tr_k)$ with each tr_i matching a template as per Lemma 6.9 therefore matching both the Correspondence and Uniqueness restrictions as per Lemmas 6.6 and 6.8. Subsequently, a trace produced using merging of all such traces $\mathit{merge}(tr_1, \ldots, tr_k)$ will also follow both the restrictions.

We wish to explicitly state that we have two sets of rule lists, first without the session ID of the form *AllowedSequences*(*ProtoRules*) and another using the session ID, i. e., *Sessions*(*ProtoRules*). Accordingly, we can generate two sets of traces. While the trace $tr_1, \ldots, tr_k$ generated using substitutions $\sigma_1, \ldots, \sigma_k$ from *AllowedSequences*(*ProtoRules*), its corresponding trace with session IDs $tr'_1, \ldots, tr'_k$ can be generated using substitutions $\sigma'_1, \ldots, \sigma'_k$ from *Sessions*(*ProtoRules*). The Lemma 6.1 establishes equivalence among these two sets of traces.

Each tr'_i will start with an initial rule from one of *Sessions*(*ProtoRules*), or with a message from the attacker, either way a fresh session ID can be used. The following messages, i. e., the second element of tr'_i can reuse this session ID. Therefore, for each of the *AllowedSequences*(*ProtoRules*) rules that was used to generate tr_{att} the corresponding rule in *Sessions*(*ProtoRules*) can be used to generate a trace tr'_{att} that is the same as tr_{att} but with a unique session ID added for all tr_i, therefore the Correspondence Restriction holds. Theorem 6.1 also makes sure that all the variables in the trace have the same value.

Def. 5.7 also tells us that $\bigcap_{i=1}^{n} \mathit{FrFromRLinTr}(\sigma_i, ASR_i, \mathit{Logs}(tr_i)) \bigcap_{i=1}^{n} (\mathit{fresh}(RL))\sigma_i = \{\}$ and as each σ_i are the substitutions used for different session IDs, it follows that the Uniqueness Restriction holds which is also supported by Lemma 6.8.

Finally, as tr_{att} will be equal to tr'_{att} with the session IDs removed, then if the tr'_{att} will invalidate the same properties as tr_{att} and will therefore also be an attack trace.

In the 'only if' direction, using Lemma 6.2, we assume there is an attack trace tr'_{att} generated by the rules in $Sessions(ProtoRules)$ which violates some security property and fulfils the Correspondence and Uniqueness Restrictions. We define tr_{att} to be the trace tr'_{att} with all session IDs removed.

The Correspondence Restriction, along with Lemma 6.7, tells us that for every element of the trace tr'_{att} there is some rewritten template that matches it. Subsequently, Theorem 6.2, tells us that for every such template, there exists a matching trace with all other necessary elements also in the trace. The condition on the session IDs in the definition of the correspondence predicate, defined in the Def. 6.21, ensure that each of these log actions share a unique session ID, and therefore, they are either a prefix of the other or don't overlap with each other.

We may then take the templates used to match all the elements of the trace as t_1 to t_k, and the Uniqueness Restrictions then tells us that we can find substitutions σ_i such that $\bigcap_{i=1}^{k} FrFromRLinTr(\sigma_i, ASR_i, Logs(tr_i)) = \{\}$ and $tr_i = \sigma_i(t_i)$ is a sub-trace of tr_{att}. Therefore, using Theorem 6.2, tr_{att} will be a member of $merge(\{tr_1, \ldots, tr_k)$, and will be a 'standard looking' trace by Def. 5.7.

By the same reasoning as above, tr_{att} will also be an attack trace, making it a stealthy attack for the rule set $ProtoRules$. □

6.11 Chapter Summary

In Chapter 5, the formal definitions of *'standard looking trace'* and *'stealth attack'* was developed. Subsequently, in this chapter, we have presented the addition of session IDs to the logs to distinguish individual runs of the protocol. The session IDs do not, in any way, modify the execution of the protocol. However, they do help with development and addition of two restrictions; *Correspondence* and *Uniqueness*, for a valid rule list. These restrictions can be added to the Tamarin models, to verify the stealthiness of the attack traces, and hence, of the attacks.

We have also analysed the transition of a template to trace produced by a standard run of the protocol. Generation of a standard trace by firing the valid rules in the order of any allowed sequence seems straight forward. However, due to the complexity of various facts and variables used by them, the generated templates are likely to have different variables in them. We needed to ensure that, given a template and a substitution, the generated traces always take the same form.

For this purpose, we defined equivalence among the names used in the rule list to rewrite the templates before converting them to traces by applying substitutions on them. We also use various definitions to help rewrite the templates with suitable examples.

We have also outlined many restrictions, presented as propositions, and theorems in this chapter, and using them, provided proof for our central theorem that an attack is present in a trace with session identifiers, under certain restrictions, if and only if a stealth attack is present in a trace without session identifiers.

Chapter Seven

Modelling TAMARIN Semantics and Stealthiness using Coq

"There are only two things in the world: nothing and semantics"

Werner Erhard

7.1 Motivation

In Chapters 5 and 6, we have presented formal and Tamarin models of stealthiness. Tamarin provides support for specifying the protocols and powers of adversaries where the security properties proofs can be generated either manually or interactively. However, to prove the equivalence among models or prove or disprove the assumptions, Tamarin alone is not sufficient. We use *Coq*, an interactive theorem prover, to model the Tamarin semantics, and encode various system properties, in order to understand the underlying Tamarin semantics and underpinning of concepts. The assumption behind our encoding is that it will provide a solid mathematical foundation to our assumptions apart from developing a newer class of case study using Coq.

7.2 Contributions

This chapter starts with encoding of basic Tamarin syntax and semantics using Coq. This encoding helps us model and understand the working of Tamarin which is used to develop further assumptions for modelling our stealth framework. Subsequently, we present Coq based models of both our formal model and Tamarin model of stealthiness. These models help us verify the correctness of our system assumptions. A selected set of type checked Coq code of definitions used throughout the thesis have been presented in this chapter. Our complete Coq model is available at [126].

We have used Coq to encode almost all the definitions presented in Chapters 5 and 6. However, in this chapter, we supplement these definitions by adding many useful definitions and lemmas about well-formedness and behaviour of the system in order to help understand the equivalence among our formal and Tamarin based definition of a stealthy trace. Our exercise of encoding Tamarin semantics has been helpful in many ways, such as making the definitions stronger and understanding the Tamarin semantics better. We also present a snapshot of our approach in this chapter to encourage and inspire the research community to adopt such encoding for other similar use-cases.

To the best of our knowledge, ours is the first attempt in modelling Tamarin using Coq which can be helpful for the academic community, to understand Tamarin better and, in testing and verification of theorems on system behaviour.

7.3 Introduction to Coq

Coq is a proof assistant used to model specifications, and verify if a given system adheres to them. The underlying system allows development of proofs in an interactive manner with the help of expressive higher-order logic [18].

Coq provides a general purpose environment for developing formal mathematical proofs. Its formal language supports everything from defining objects, their behaviours and finally writing proofs [110]. Some salient features of Coq include, but are not limited to, providing support for functions, definitions and writing interactive proofs using multiple tactics. The type system of Coq helps express very precise specifications and as such, we use it to model the multiset rewrite system used in TAMARIN along with its semantics. There are numerous recent examples of Coq being used successfully for modelling and verification in various contexts such as in model transformation verification [40, 134], verifying security protocols [109, 113], performing formal reasoning about security of various web services such as AWS (Amazon Web Service) [45] to name a few.

We have developed our Coq model in many sub-sections with each of them dependent on all the previous sub-sections. We will now explain these sub-sections and their importance.

7.4 Modelling of TAMARIN Semantics using Coq

The Coq model developed by us is not only a simulation of TAMARIN semantics, but also augmentation of numerous functions, predicates, and definitions to aid in understanding the correctness of our assumptions in developing our *Stealth* framework. Other than defining TAMARIN semantics, to support our proof of correctness and equivalence, we have written more than 200 lemmas and proved many of them. We leave the proof of all other lemmas as a possible future work.

In this section, we present Coq encoding of standard TAMARIN semantics with some of our extensions added to it. We use these semantics to define equality and membership functions. Subsequently, we define how the substitutions take place in TAMARIN and how ground terms are defined. Finally, to prove that all the Coq semantics hold together, we apply our model on Needham Schroeder protocol, which is a success proving efficacy of our model.

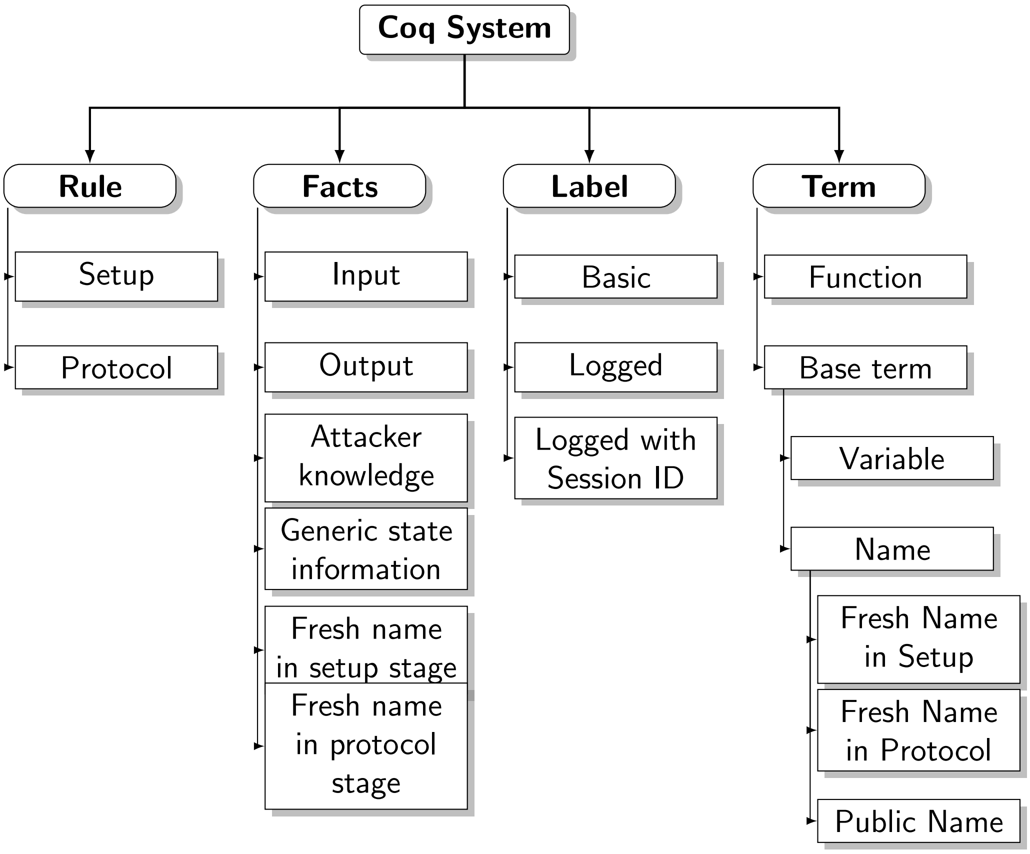


Figure 7.1: Coq Taxonomy

7.4.1 Basic Syntax Definitions

We provide a brief snapshot of some syntax used in our Coq model. We start by defining *actionLabel* to model the facts present in the *actions* of Tamarin rules. We extend the notion of action labels present in Tamarin by defining three categories of labels to model Basic and Logged labels, as described in Section 5.5. We also define a Logged label with Session ID in order to uniquely identify each action label to its protocol execution instance. The relevance and details of this mechanism is discussed in Chapter 6. Finally, a *trace* is defined as a list of *labels*.

Subsequently, we define a *fact* to represent various facts present in the *premise*

```
Inductive actionLabel : Type :=
 | ActLabel : name → terms → actionLabel.
Inductive label : Type :=
 | BasicLabel : list actionLabel → label
 | LoggedLabel : actionLabel → list actionLabel → label
 | LoggedSIDLabel : baseTerm → actionLabel → list actionLabel → label.
Definition trace : Type := list label.

Inductive fact : Type :=
  | IN : terms → fact
  | OUT : terms → fact
  | K : terms → fact
  | STATE : name → terms → fact
  | FR_S : baseTerm → fact
  | FR_P : baseTerm → fact.
Inductive rule : Type :=
  | RuleSetup : facts → label → facts → rule
  | RuleProto : facts → label → facts → rule.
Definition ruleList : Type := list rule.
```

and *conclusions* of a Tamarin *rule*. The facts *IN* and *OUT*, present in the *premise* and *conclusion* of a rule, represents an input to and output from a rule, respectively. The facts *K* and *STATE* represent knowledge of an attacker and general protocol state information. We define two types of *fresh* names, i. e., *FR_S* and *FR_P* to denote fresh values generated in the setup rules and protocol rules respectively. As we will see, the *fresh* names generated in the setup rules such as public/private keys etc. could be reused by the protocol rules. However, our model requires the *fresh* names generated by the protocol rules to be unique. We define the 4-tuple Tamarin rule, defined in Section 5.4, as a triple leaving out the rule name rn. These rules have two subtypes, i. e., Setup and Protocol rules as per Section 5.5.

7.4.2 Equality and Membership Functions

We define equality and membership functions based on the syntax defined in previous sections. The development of equality functions, though self-explanatory, helps to understand the structure of each data type. Similarly, membership functions help us in understanding the underlying hierarchy.

7.4.3 Defining Substitutions and Ground terms

As defined in Chapter 5, a substitution is a mapping from variable(s) to name(s). Using this substitution, we define various operations such as *compose* and how these substitutions are applied on various data types such as labels, action labels, facts etc. We have also added suitable *lemmas* to enforce the commutative property on compose operations when applied to any of the data types.

Subsequently, we define a ground base term as a base term not of type variable. This definition is then used to define ground terms, facts, rules etc. We also define the grounding of composition of substitution followed by adding *lemmas* to enforce the commutative property.

7.4.4 TAMARIN Reduction

This is the most important section, where we start by defining reduction of a TAMARIN rule *r* converting facts such as *f1* to facts *f2* while producing label *l*.

```
Definition reduceWithRule (r1:rule) (f1:facts) (l:label) (f2:facts) : Prop :=
    ∃ r2, (ginst r2 r1) ∧ reduceWithGroundRule r2 f1 l f2.
```

Subsequently, we define the predicates *run* and *runSigma* which emulate the labelled transition relation present in section 5.4.1.

```
Inductive run : ruleList → facts → trace → facts → Prop :=
  | Run_Stop : ∀ rl f1, run rl f1 [] f1
  | Run_Step : ∀ rl f1 t f2 f3 l, (reduceWithRules rl f1 l f2)
                                   → (run rl f2 t f3) → run rl f1 (l::t) f3
  | Run_Step_Silent : ∀ rl f1 t f2 f3, (reduceWithRules rl f1 emptyLabel f2)
                                   → (run rl f2 t f3) → run rl f1 t f3
```

```
| Run_Fresh : ∀ rl f1 t f2 f3 fn, (reduceWithRule (freshRuleP fn) f1 emptyLabel f2)
        → (run rl f2 t f3) → run rl f1 t f3.
```

7.4.5 TAMARIN Example

Before moving on to encoding and establishing proofs of complex system behaviour, we proceeded to encode the publicly available Tamarin model of the classic three message version of the Needham-Schroeder-Lowe Public Key Protocol [55]. Using this example, we demonstrated that our model can generate the public and private keys for participating entities apart from executing the rules. We were able to achieve the correctness guarantee by not only writing down the correctness *lemmas* but also proving them. An example lemma is presented here. For detailed working of this example, we refer the reader to our Coq model available at [126].

```
Lemma protocolRun2to3Step : run SetupAndProtocolRules
            [FR_P (Name (Fresh_P ”ltkB”)); FR_P (Name (Fresh_P ”ltkA”))]
            []
            [STATE LtkFact [A; (Bterm (Name (Fresh_P ”ltkA”)))];
             STATE PkFact [A; Func PkName [(Name (Fresh_P ”ltkA”))]];
             OUT [Func PkName [(Name (Fresh_P ”ltkA”))]];
             FR_P (Name (Fresh_P ”ltkB”))].
```

All the definitions and functions developed up to this section relate to standard Tamarin semantics, and can easily be reused in any other Tamarin case study with very little modifications.

7.5 Modelling of Stealthiness in Coq

In this section, we present the Coq code snippets related closely to our stealthiness definitions. We present the extensions provided by us in Tamarin semantics and their encoding in Coq. To help the reader understand the context, code snippets from our Coq model of Tamarin semantics and our stealthiness definitions from Chapters 5 and 6 are also presented here. This section also talks about rationale and process behind the selection of specific Tamarin semantics to be modelled.

7.5.1 New TAMARIN Reduction

Recall that while the *SetupRules* in our system can be executed in any order, the *ProtoRules* needs to be executed only in order mandated by any of the members of *AllowedSequences*. Both the above predicates can be used to execute any rule in any order and hence used for execution of setup rules in our model. However, to execute *ProtoRules*, we use the predicate *runInOrderSigmaP* defined as below:

```
Inductive runInOrderSigmaP : ruleList → substitutions → facts → trace → facts →
Prop :=
    | Sigma_Run_Stop_P : ∀ f1 sigma, runInOrderSigmaP [] sigma f1 [] f1
    | Sigma_Run_Step_P : ∀ r rlp sigmaNow sigmaPr f1 t f2 f3 l,
        groundRule (applySubRule sigmaNow r)
          → reduceWithGroundRule (applySubRule sigmaNow r) f1 l f2
            → (runInOrderSigmaP rlp sigmaPr f2 t f3)
              → runInOrderSigmaP (r::rlp) (sigmaNow::sigmaPr) f1 (l::t) f3
    | Sigma_Run_Step_Silent_P : ∀ r rlp sigmaNow sigmaPr f1 t f2 f3,
        groundRule (applySubRule sigmaNow r)
          → reduceWithGroundRule (applySubRule sigmaNow r) f1 emptyLabel f2
```

```
            → (runInOrderSigmaP rlp sigmaPr f2 t f3)
              → runInOrderSigmaP (r::rlp) (sigmaNow::sigmaPr)f1 t f3
    | Sigma_Run_Comm_P : ∀ rlp sigmaPr f1 t f2 trm,
     (subList [OUT trm] f1)
     → runInOrderSigmaP rlp sigmaPr ((IN trm)::(removeFacts [OUT trm] f1)) t f2
       → runInOrderSigmaP rlp sigmaPr f1 t f2.
```

Using these definitions and predicates, we have encoded one of the most important definitions of our model, i. e., Standard Looking Trace from Def. 5.7 as below.

```
Definition Def5_7_Standard_Looking_Trace (allowSeq:list ruleList) (setupRules:ruleList)
(testTrace:trace) : Prop :=
 ∃ σ  sometraces, (Subset_of_Standard_Traces allowSeq setupRules σ sometraces)
        ∧ (merge (map getLogTrace sometraces) (getLogTrace testTrace))
            ∧ ConditionOnFreshNames σ allowSeq (map getLogTrace sometraces).
```

7.5.2 Well-formedness and Validity Definitions

Moving on to start of Chapter 6, we start by Coq encoding of well-formedness definitions by translating the conditions for well-formedness of rule lists already described in the Section 6.4. The well-formedness of a protocol rule list is defined using the *wellformedRulesFromState* definition below.

```
Definition wellformedRulesFromState (fs:facts) (rs:ruleList) : Prop :=
  (oneInputXorOutput fs ∧ wellFormedProtoRuleFromInOut rs)
    ∨ (noInputOrOutput fs ∧ wellFormedProtoRuleNoInOut rs).
```

Similarly, the well-formedness of the setup rule list is defined using the *wellformedRulesFromStateSetup* definition. Subsequently, we encode the definitions presented in the section 6.5, for validity of facts, rules, rule list, as below.

```
Definition wellformedRulesFromStateSetup (fs:facts) (rs:ruleList) : Prop :=
  (oneInputXorOutput fs ∧ wellFormedSetupRuleFromInOut rs)
    ∨ (noInputOrOutput fs ∧ wellFormedSetupRuleNoInOut rs).
```

```
Definition Def6_4_allowedFactType (f:fact) (r: rule) (rs:list rule) : Prop :=
  (∃ t, (member (FR_P t) (premiseFacts r)) ∧ (eq_fact f (FR_P t))) ∨
     (∃ ts, (member (IN ts) (premiseFacts r) ∧ (eq_fact f (IN ts)) ∨
           (member (OUT ts) (conclFacts r)) ∧ (eq_fact f (OUT ts)))) ∨
              (uniqueStateFact f (r::rs)) ∨
                  (uniqueArgumentsFact f r rs).
```

```
Definition Def6_5_variableDirectlyLinked (var: variable) (newRule:rule) (pastRules: list
rule) : Prop :=
   (variableLinkedInputOutput var newRule pastRules)
      ∨ (variableInUniqueStateFact var newRule pastRules).
```

```
Fixpoint Def6_6_variableIndirectlyLinked (v:variable) (newRule:rule) (pastRules: list
rule) : Prop :=
  ∃ n ts, uniqueArgumentsFact (STATE n ts) newRule pastRules
    ∧ member (STATE n ts) (premiseFacts newRule)
      ∧ (∃ RuleA, member (STATE n ts) (conclFacts RuleA) ∧ (member RuleA
pastRules))
        ∧ member v (variablesInTerms ts)
          ∧ (∃ t, member t (variablesInTerms ts)
            ∧ Def6_3_variableDirectlyLinked t newRule pastRules)
              ∧ (∃ c, member c (constantsInTerms ts)).
```

```
Definition Def6_7_validVariable (newRule:rule) (pastRules: list rule) (var: variable) :
Prop :=
   (variableIsNew var pastRules) ∨ (Def6_3_variableDirectlyLinked var newRule
pastRules)
        ∨ (Def6_6_variableIndirectlyLinked var newRule pastRules).
```

```
Definition Def6_8_validRule (newRule:rule) (pastRules: list rule): Prop :=
        (∀ f, (member f (allFactsRL (appendAny pastRules [newRule])))
             ∧ (Def6_4_allowedFactType f newRule pastRules))
                 ∧ (∀ vs, trueForAll (Def6_7_validVariable newRule pastRules) vs).
```

```
Fixpoint Def6_9_validRuleList (rs : list rule) : Prop :=
match rs with
| [] ⇒ True
| r::rst ⇒ (Def6_7_validRuleList rst) ∧ (Def6_7_validVariables r rst (variablesInRule r))
end.
```

The Coq definitions used here along with the complete model is available online at [126].

7.5.3 TAMARIN definitions of Stealthiness

As discussed earlier, the major difference between the formal and TAMARIN definitions of stealthiness is the presence of session ID in the labels and traces in TAMARIN framework. Therefore, we start this section with annotation of rules and traces with session IDs, in Coq, using the following functions.

```
Fixpoint annotateRules (rl : ruleList) : ruleList :=
  match rl with
  | [] ⇒ []
  | r::t ⇒ (annotateFirstRule r) ::(annotateOtherRules t)
  end.
Definition AddSID :(list ruleList) → (list ruleList) := map annotateRules.
Definition AddSIDTrace: trace → trace := map updateLabel.
Definition Def6_11_SessionsRL (allowSeq:list ruleList) : (list trace) :=
               Def_StdTemplates (AddSID allowSeq).
Definition Def6_12_StdSessionTemplate (allowSeq:list ruleList) (thisTemp:trace) : Prop
:=
                        (In thisTemp (Def6_11_SessionsRL allowSeq)).
```

Similarly, we have also defined a function to remove the session IDs from a trace.

```
Fixpoint RemoveSIDfromLabel (l1 : label) : label :=
  match l1 with
    | (LoggedSIDLabel t al lb) ⇒ (LoggedLabel al lb)
    | (LoggedLabel al lb) ⇒ (LoggedLabel al lb)
    | (BasicLabel als) ⇒ (BasicLabel als)
  end.
Definition RemoveSID : trace → trace := map RemoveSIDfromLabel.
```

Finally, we encode the main restrictions namely *Correspondence* and *Uniqueness* as per Def. 6.22 and 6.23.

```
Definition Def6_21_Correspondence (tr_temp:trace) (tr_log:trace) (m:nat) (sigma:substitution)
: Prop :=
 ∀ temp_m lbl,
   (subSequenceTrace (applySubTrace sigma tr_temp) tr_log) ∧
     (eq_label (lastElement temp_m (applySubTrace sigma tr_temp)) (LabelAt m tr_log
lbl)).
```

```
Definition Def6_22_CorrespondenceRestriction (RL_proto:ruleList) (allowSeq: list ruleList)
(tr_log:trace) (tr_tempsessions: trace) : Prop :=
  ∀ m, (indexExists m tr_log) ∧
      Def6_12_StdSessionTemplate allowSeq tr_tempsessions →
        ∃ sigma tr_rewrittenTemp ,
          (Def6_20_3_RewriteTemplate tr_tempsessions RL_proto tr_rewrittenTemp) →
             Def6_21_Correspondence tr_rewrittenTemp tr_log m sigma.
```

```
Definition Def6_23_UniquenessRestriction (RL_proto:ruleList) (tr_log:trace) : Prop :=
  ∀ Log_i Log_j x sid_1 sid_2 pars_i pars_j ,
       (member x (Def5_5_freshInRLProto RL_proto))
        ∧ (LoggedLbl_in_Trace Log_i tr_log)
        ∧ (LoggedLbl_in_Trace Log_j tr_log)
         ∧ (getSID Log_i sid_1)
         ∧ (getSID Log_j sid_2)
          ∧ (getParsFromLog Log_i pars_i)
           ∧ (getParsFromLog Log_j pars_j)
            ∧ (member x (commonFreshNames (getFreshProtoNamesinLabel Log_i) (getFreshProtoNamesinLabel Log_j)))
               → (eq_baseTerm sid_1 sid_2).
```

7.5.4 Correctness Lemmas and Axioms on System Behaviour

After encoding all the definitions of stealthiness, we write many *axioms* and prove several *lemmas* which helps us establish the correctness behaviour of the system. We have written over 100 lemmas and axioms and proved many of them to establish the correctness of our assumptions. A selection of lemmas and axioms is placed below.

```
Lemma StateToStateOneOutRule : ∀ r sigma f1 l f2,
  oneOutput f1 → StateToStateProtocolRule r →
    reduceWithRuleSigma r sigma f1 l f2 → oneOutput f2.

Axiom InputToOutputRemovesInput : ∀ r sigma f1 l f2,
  oneInputXorOutput f1 → InputToOutputProtocolRule r →
    reduceWithRuleSigma r sigma f1 l f2 → noInputs f2.

Axiom FreshRuleDoesNotChangeOutputs : ∀ f1 f2 fn sigma,
  reduceWithRuleSigma (freshRuleP fn) sigma f1 emptyLabel f2
    → oneOrNoOutput f1 → oneOrNoOutput f2.

Axiom SetupRuleFromInOutDoesNotChangeInOut : ∀ r f1 l f2,
  oneInputXorOutput f1 → wellFormedSetupRuleFromInOut r →
    reduceWithRules r f1 l f2 → oneInputXorOutput f2.

Axiom outputInRuleWillMatchRule : ∀ rule1 rules sigma f1 l f2 vars,
  reduceWithGroundRule (applySubRule sigma rule1) f1 l f2
    → groundRule (applySubRule sigma rule1)
      → Def6_2_wellformedRulesFromState f1 (rule1::rules)
        → oneOutputAndNoInputIs (conclFacts rule1) (OUT vars)
          → member (OUT (applySubTerms sigma vars)) f2.
```

We have proved many Lemmas in this section and written some of very obvious ones as Axioms. We aim to prove them in our possible future work.

7.5.5 Correctness Lemmas and Propositions

In this section, we present lemmas and propositions from Chapter 6. These propositions help us in establishing various properties which are self-explanatory using their names. E.g., *Prop1_DirectlyLinkedVarsHaveSameValues* helps establish that all directly linked variables will have same values and so on.

```
Lemma Prop1_DirectlyLinkedVarsHaveSameValues: ∀ fs1 fs2 σ_j tr v r rlProto rlSetup σ,
  ((Def6_9_validRuleList (appendAny rlProto [r])
       ∧ runInOrderSigma (appendAny rlProto [r]) rlSetup (appendAny σ [σ_j]) fs1 tr
fs2))
            → ((Def6_5_variableDirectlyLinked v r rlProto)
                 → (variableIsNew v rlProto)
                      (∃ σ_i, (eq_name (σ_i v) (σ_j v))
                                 ∧ ((member σ_i σ))))

Lemma Prop2_IndirectlyLinkedVarsHaveSameVal: ∀ fs1 fs2 σ_j tr v r rlProto rlSetup σ,
  ((Def6_9_validRuleList (appendAny rlProto [r])
       ∧ runInOrderSigma (appendAny rlProto [r]) rlSetup (appendAny σ [σ_j]) fs1 tr
fs2))
            → ((Def6_6_variableIndirectlyLinked v r rlProto)
                 → (variableIsNew v rlProto) ∨
                           (∃ σ_i, (eq_name (σ_i v) (σ_j v))
                                 ∧ ((member σ_i σ)))).

Lemma Prop3_AllSigmasCanBeReplacedbyASingleSigma: ∀ fs1 fs2 tr r rlProto rlSetup σ,
          (Def6_9_validRuleList (appendAny rlProto [r])
               ∧ runInOrderSigma (appendAny rlProto [r]) rlSetup σ fs1 tr fs2)
                    → (∃ σ, ∀ v σ_i, (Def6_7_validVariable r rlProto v)
                                ∧ (member σ_i σ)
                              → (eq_name (σ_i v) (σ v)))

Lemma Prop4_ExistsTraceAndSigmaForTemplate: ∀ allowSeq rlSetup rlp tr_temp,
          (Def6_9_validRuleList rlp ∧ Def2_1_Valid_AllowSequences allowSeq rlp)
            → ∃ tr_log σ σ_i, Def5_4_Standard_Trace allowSeq rlSetup σ, tr_log
                  → (Def6_10_StdTemplate allowSeq [tr_temp])
                              ∧ (eq (applySubTrace σ [tr_temp]) tr_log)
                                    ∧ (member σ_i σ)

Theorem Thm__ExistsTemplateAndSigmaForTrace: ∀ allowSeq rlSetup rlp tr_log,
          (Def6_9_validRuleList rlp ∧ Def2_1_Valid_AllowSequences allowSeq rlp)
            → ∃ tr_template σ, Def5_4_Standard_Trace allowSeq rlSetup σ tr_log
                  → (Def6_10_StdTemplate allowSeq tr_template)
                                          ∧ (eq (applySubTrace σ tr_template) tr_log).
```

7.6 Equivalence of Two Models

Our main Theorem 6.3 from Chapter 6 and its manual proof-sketch established the equivalence among two set of attack traces, one with the session ID and other without it.

Though we have not been able to complete the proof of equivalence using Coq, we have encoded the theorem as below and left its proof as a future work.

```
Theorem Thm_6_3_StealthAttackImpliesBothRestrictionsOnTraces : ∀ allowSeq rlSetup
attackTrace rlProto,
  ((WellFormedAndValidRLSetupProto rlSetup rlProto
   ∧ (eq allowSeq (allPrefixs rlProto)))
    →
     (Def5_9_Stealth_Attack allowSeq rlSetup (RemoveSID attackTrace)))
    ↔
      (∃ tmplt_sessions tmplt_Rewritten,
     Def6_12_StdSessionTemplate allowSeq tmplt_sessions ∧
      Def6_20_3_RewriteTemplate tmplt_sessions rlProto tmplt_Rewritten ∧
       Def6_22_CorrespondenceRestriction rlProto allowSeq tmplt_Rewritten attackTrace
        ∧
         Def6_23_UniquenessRestriction rlProto attackTrace).
```

7.7 Importance of Coq Encoding and Learnings

Our Coq model consisting of definitions and semantics presented in previous chapters are not merely encoding of formal or Tamarin definitions to Coq but more than that in many ways. The Coq encoding helped us in improving our definitions at more than one occasion. E.g., initially, the definition of template rewriting, i. e., Def. 6.20.3 was written as a non-deterministic combination of some rewrite rules. These rules could have been fired in any sequence sometimes leading to an incorrect rewritten template. The Coq encoding of this definition helped us see the problem, and we broke this definition into smaller ones making them deterministic.

```
Inductive Def6_20_PreRewriteTemplate: trace → ruleList → trace → Prop :=
  | Case_Base : ∀ RL , Def6_20_PreRewriteTemplate [] RL []
  | case_Step : ∀ l1 l2 RL t1 t2, (rewriteWithRules l1 RL l2)
                                   → Def6_20_PreRewriteTemplate t1 RL t2
                                     → Def6_20_PreRewriteTemplate (l1::t1) RL (l2::t2).

Definition Def_6_20_1_TemplateNormalForm (tmp:trace) (RL:ruleList) :=
    ∀ t1, (In t1 (termsFromTemplate tmp))
          → (noOtherEquivalentTermsPresent t1 tmp RL)
    ∧ ∃ tf, (In tf (singleEqClass RL t1)) ∧ (termIsFunc tf)
          → (eq_termFuncName t1 tf)
    ∧ ∃ tp, (In tp (singleEqClass RL t1) ∧ (termIsPub tp)
          → (eq_term t1 tp)).

Axiom normalFormExistsForEveryTmp : ∀ tmp f2 RL,
    Def6_9_validRuleList RL ∧
          Def6_2_wellformedRulesFromState f2 RL ∧
            Def6_10_StdTemplate [RL] tmp
              → ∃ tmp1, Def6_20_PreRewriteTemplate tmp RL tmp1
                                    ∧ Def_6_20_1_TemplateNormalForm tmp1 RL.

Definition Def_6_20_2_InstantiateTemplate (tmp1:trace) (RL:ruleList) (tmp2:trace):
Prop := ∃ sigma, ∀ uafP, (In uafP (ProtocolUAFsFromRL RL))
            → ∃ uafS, (In uafS (SetupUAFsFromRL RL)) ∧
                    (eq_fact uafS (applySubFact sigma uafP)) ∧
                       eq_trace tmp2 (applySubTrace sigma tmp1).
```

```
Definition Def6_20_3_RewriteTemplate (tmp1:trace) (RL:ruleList) (tmp2:trace): Prop
:=
  ∃ tmp, (Def6_20_PreRewriteTemplate tmp1 RL tmp) ∧
                              Def_6_20_1_TemplateNormalForm tmp RL ∧
                                Def_6_20_2_InstantiateTemplate tmp RL tmp2.
```

Similarly, Coq helped us type check and correct errors arising out of incompatibility between many predicates or definitions when used in subsequent definitions.

7.8 Chapter Summary

This chapter starts with an introduction to Coq and its use in modelling standard Tamarin semantics. One of the main contributions of our work present in this chapter is the possible reusability of our Coq code of Tamarin semantics for anyone looking to verify their work using Tamarin.

We have also presented the process and importance of Coq encoding of our formal and Tamarin model and definitions of stealthiness. The Coq model and encoding is helpful to any reader for better understanding of our system with the help of multiple type checked definitions and propositions from our Coq model. Using these type checked definitions, we have been able to prove many lemmas and propositions in our Coq model.

Though we have not been able to provide machine checked proofs for all the lemmas and theorem, we have provided a proof sketch to establish correctness of our system in section 6.3. We leave developing Coq proofs, for all the lemmas and theorems, as a future work.

Chapter Eight

Case Studies: Testing for Stealthiness in TAMARIN

The man of science has learned to believe in justification, not by faith, but by verification.

THOMAS H. HUXLEY

8.1 Motivation

After presenting the theoretical framework for stealthy traces in previous chapters, the next step is to try to implement the theory by testing the stealthiness of attacks using the automatic protocol verification tool TAMARIN. As already explained, to solve the exponential blow-up of multiple runs, we have introduced an identifier, such as $Session-ID$ to the logs, in order to uniquely identify and map them to their respective runs. The publicly available TAMARIN models do not follow any such annotation, hence they need to be modified in order to be used by our framework. Subsequently, we also need to generate the correct restrictions, as per Def. 6.22 and 6.23, to be placed on the traces generated by the TAMARIN tool in order to make the traces equivalent to standard looking traces defined using Def. 5.7.

Various attacks on protocols present a challenge to both the protocol designers and users alike. In order to develop resilient security protocols, protocol designers must look for correctness of cryptographic scheme used in the protocol along with proofs of various security guarantees claimed by it. They should also be aware of various attacks against a specific class of protocol. Analysing the attacks and their detection mechanism can be very helpful while designing the attack-resilient protocols and systems. Detecting a stealth attack is most likely to involve more effort than a non-stealth attack, hence it is important to study and analyse the stealth attacks and their behaviour. It is with this objective that we seek to test the stealthiness of various known attacks on security protocols.

There is an abundance of Tamarin based protocol models publicly available. Most of these models simply provide proof of correctness and that the security properties hold for those protocols. However, some of them do provide evidence of an attack, and are useful for us to test these attacks for their stealthiness. For our analysis, we want to use the protocol models from Tamarin repository, exhibiting an attack on any of the security property, along with the attack models developed by us. Our objective is to test the stealthiness of the attacks against logs composed of a combination of parameters.

8.2 Contributions

This chapter introduces a Python-based tool *StealthCheck*, developed by us, to automatically add the stealthiness restrictions to any Tamarin source of a protocol. We have used *StealthCheck* to compare different logging strategies for relative effectiveness at recording evidence of attacks. This approach has been successful to specify protocols and analyse them for vulnerability to stealthy attacks. Finally, we present a summary of our stealthiness analysis of various known attacks, using our framework, on both the publicly available Tamarin models [55] and on a Tamarin model of a real-life attack, i.e., the KRACK attack [137] on IEEE 802.11 *4-way handshake*. Based on our stealthiness test on the attack traces, we have presented a minimal set of parameters, to be logged to convert a stealth attack into a non-stealth attack and vice-versa.

8.3 Overview

The Formal Stealth Model presented in the Chapter 5 guarantees that, in the Def. 5.7 of “standard looking trace”, all the logs appear in sequence as per the protocol standard template with no intermediate logs missing. Further, in Chapter 6, we developed two restrictions, namely Correspondence and Uniqueness, to ensure that the traces generated under these restrictions are always stealthy. However, there is currently no tool that can directly, and automatically check this definition.

As discussed in Sec. 3.5, Tamarin can demonstrate an attack on any vulnerable protocol by falsifying one of the security lemmas provided in the model, signifying that a counter example has been found violating the conditions stated in the lemma, and generate an attack trace in the process. Further, Tamarin[103] can also restrict generation of traces using *restriction* feature. To automatically analyse the attack traces for stealthiness, we use Tamarin to allow generation of only standard looking traces of a protocol by executing the vulnerable Tamarin models under Correspondence and Uniqueness restrictions.

In order to automatically verify such attack traces for stealthiness, we outline the development process of Python-based tool *StealthCheck* in this chapter. The chapter also explains the steps taken by *StealthCheck* to generate the Tamarin restrictions equivalent to stealthiness definitions presented in previous chapters. We start with a user adding custom logs and selected parameters to the Tamarin model of a vulnerable protocol that exhibits an attack by falsifying one of the security lemmas. Subsequently, *StealthCheck* scans the Tamarin model and extracts list of logs, in order of allowed sequences, and a list of fresh names, to construct two restrictions, and add them to the original model. It further verifies if the attack exists in the restricted environment, classifying them as either *stealth attack* or non-*stealth attack*. This chapter outlines a step-by-step process of the tool development.The *StealthCheck* tool is available online and can be at [126].

To test various attacks for their stealthiness, we have applied *StealthCheck* to both publicly available Tamarin models from [55] and Tamarin model of KRACK attack [137] developed by us. Our analysis has shown that, in most of the cases, a correct combination of logged parameters is able to detect the attacks and convert them to non-stealthy. However, some attacks are still able to bypass the detection completely, and remain stealthy.

8.4 Algorithm used by *StealthCheck*

We start with a Tamarin model of a protocol containing a rule list of the forms $\mathit{SetupRules}_{\text{stealth}}$ and $\mathit{ProtoRules}_{\text{stealth}}$ similar to the running example from Sec. 5.6. The $\mathit{ProtoRules}_{\text{stealth}}$ is annotated with stealth logs prior to its processing by *StealthCheck*. We call the new rule list as $\mathit{RuleList}_{proto}$.

The Tamarin model presented in Chapter 6 mandates applying the two restrictions on traces, namely Correspondence and Uniqueness, formed on standard session templates of the form $\mathit{Templates}_{\text{Sessions}}$, as per Def. 6.12 and rewritten using Def. 6.20.3. We, however, did not need to rewrite any of the templates generated by Tamarin protocol models. This is because in both public, and custom models developed by us, we did not encounter a situation when a specific term, present in the generated restrictions, could receive two distinct substitutions. We, however, intend to incorporate this feature in *StealthCheck*, and leave this extension as a future work.

StealthCheck starts by scanning $\mathit{RuleList}_{proto}$ for Fr fact for collecting a list $frList$ of fresh names and the logs that use them. Subsequently, it adds session IDs to $\mathit{RuleList}_{proto}$

resulting in a modified rule list of the form $\mathit{Sessions}(\mathit{RuleList}_{\mathit{proto}})$ similar to one presented in Sec. 6.7. A list of all the log labels present, $\mathit{LogList}$, in the $\mathit{Sessions}(\mathit{ProtoRules})$ is also prepared. Based on these two lists, i. e., list of fresh names and the log labels, $StealthCheck$ proceeds to generate $Correspondence$ and $Uniqueness$ restrictions. In order to add $Uniqueness$ restrictions to the model, $StealthCheck$ uses the lists $frList$ and $LogList$. These restrictions are generated, for each pair of logs using any fresh names from list $frList$ using Def. 6.23, to guarantee that different fresh names are used in different sessions.

Subsequently, $Correspondence$ restrictions equivalent to Def. 6.22 are generated by looking at each of the standard session templates from $\mathit{Templates}_{\text{Sessions}}$, generated using $\mathit{AllowedSequences}(\mathit{RuleList}_{\mathit{proto}})$, verifying that, for every last element of template, the trace corresponds to at least one of these templates. To generate these restrictions, $StealthCheck$ looks at the last log label and its parameters and a list of log labels preceding it in the template and their parameters. The restriction mandates that for every occurrence of this last log label, there must exist a previous occurrence of all the preceding log labels present in the template in the same order. The complexity, however, arises at the point of classifying the log label parameters as either existential or universal quantifiers in the first-order formula of restriction. All the common parameters among all the log labels present are placed as universal quantifiers at the front of the formula whereas the remaining parameters from the log labels are placed before them as existential quantifiers while also specifying a time-point based ordering among the log labels as defined by the template. Apart from maintaining this ordering, the restriction also ensures that there is at most a single occurrence of each log label with respect to a single session identifier. Finally, if any log label appears as the last element in more than one template, the system generates one restriction per such occurrence, and all these restrictions are then combined in form of a disjunction.

The $StealthCheck$ tool follows the syntax of 'restriction' feature of TAMARIN while

Algorithm 8.1 Steps and Main Functions used in *StealthCheck*

Input: TAMARIN model with $RuleList_{proto}$ annotated with Stealth logged labels and security lemmas

Output: TAMARIN input model added with Session IDs in $RuleList_{proto}$ and appended with Stealthiness restrictions

```
function CollectFreshFromPremise($RuleList_{proto}$)
    frList=[]
    for $ruleN \in RuleList_{proto}$ do
        if Fr(x) fact present in premise(rule) then
            idx="Log"+N
            frList.setdefault(idx,[]).append(x)
        end if
    end for
    return frList
end function
function SelectLogLabels($RuleList_{proto}$)
    LogList=[]
    for $ruleN \in RuleList_{proto}$ do
        LogList.append($getLog(Action(ruleN))$)
    end for
    return LogList
end function
function GenUniquenessRestrictions(FrList,LogList)
    for fr1, fr2 in FrList: do
        for  log1, log2 in LogList: do
            if log1 and log2 using fr1, fr2 then
                Return restriction from Def. 6.23 using log1,log2,fr1, and fr2.
            end if
        end for
    end for
    return UniqueRest
end function
function GenCorrespondenceRestrictions(LogList)
    for lastAL in len(LogList)-1, len(LogList)-2,...,1): do
        currentLog=LogList[lastAL]
        allprevLogs=LogList[:lastAL]
        Return restriction using Def. 6.22 based on allprevLogs and currentLog
    end for
end function
function Main
    frList= CollectFreshFromPremise($RuleList_{proto}$)
    $SidRuleList_{proto} = Sessions(RuleList_{proto})$ using Def. 6.11
    LogList=selectLogLabels($SidRuleList_{proto}$)
return(GenUniquenessRestrictions(FrList,LogList)+ GenCorrespondenceRestrictions(LogList))
end function
```

generating all such restrictions. For detailed working of these algorithms, we refer the reader to *StealthCheck* python code available at [126].

8.4.1 Applying TAMARIN Stealth Model to Example 5.1

As explained above, *StealthCheck* starts by collecting the list of fresh names and log labels. For the example rule list *Sessions*(*ProtoRules*) as shown in the Sec. 6.7, *StealthCheck* will generate the lists as:

$$frList = [nA]$$

$$LogList = [LogA1(sid, A, nA), LogB2(sid, A, B, nA), LogX3(sid, A, B, nA)]$$

It would also need to take into consideration the standard session templates.

$$\begin{aligned} &\mathit{Templates}_{\text{Sessions}}(\mathit{ProtoRules}) = \\ &\quad \{[\ [\mathit{LogA1}(sid, A, nA)],\ [\mathit{LogA1}(sid, A, nA), \mathit{LogB2}(sid, A, B, nA)], \\ &\quad [\mathit{LogA1}(sid, A, nA), \mathit{LogB2}(sid, A, B, nA), \mathit{LogX3}(sid, A, B, nA)]\} \end{aligned}$$

Accordingly, *StealthCheck* automatically generates the following restrictions and adds them to the Tamarin model.

Correspondence Restrictions:

$$restriction\ EveryALBeforeLogX3andUniqueLogs :$$

$$\text{``}(\forall A, B, nA, sid, t_1.\ \text{LogX3}(sid, A, B, nA)@t_1 \Rightarrow$$

$$(\exists\ t_2.\text{LogB2}(sid, A, B, nA)@t_2 \wedge (t_2 < t_1) \wedge$$

$$(\exists\ t_3.\text{LogA1}(sid, A, B, nA)@t_3 \wedge (t_3 < t_2) \wedge$$

$$(\forall A1, B1, nA1, sid, t_4.\text{LogX3}(sid, A1, B1, nA1)@t_4 \Rightarrow (t_4 = t_1))\wedge$$

$$(\forall A2, B2, nA2, sid, t_5.\text{LogB2}(sid, A2, B2, nA2)@t_5 \Rightarrow (t_5 = t_2))\wedge$$

$$(\forall A3, B3, nA3, sid, t_6.\text{LogA1}(sid, A3, B3, nA3)@t_6 \Rightarrow (t_6 = t_3))))\text{''}$$

$restriction\ EveryALBeforeLogB2andUniqueLogs$: ....

The *correspondence* restrictions, generated by applying Def. 6.22, enforce that for each log present in any of the standard template, all the previous logs present in the same template must be present in the trace in the same order. It also restricts use of a single session ID, at most once for each of the log entry. These restrictions are generated for every possible log entry, with a single restriction generated per log entry. To generate these restrictions, we follow the general steps outlined in the Algorithm 8.1.

It may be noted that while generating these restrictions, a universal quantifier is used with the parameters for the present log entry, whereas existential quantifiers are used for parameters exclusive to each of the previous log entries. E. g., the example above uses $\forall A, B, nA, sid, t_1.$ for the $LogX3$, whereas for other logs such as $LogA1$ and $LogB2$, we use $\exists$ for time point parameters only.

$$\mathit{Templates}_{\text{Sessions}}(\mathit{ProtoRules}) =$$

$$\{\ [\ldots],\ [\mathit{LogA1}(sid, A, nA), \mathit{LogX3}(sid, A, B, nA)]$$

$$[\mathit{LogA1}(sid, A, nA), \mathit{LogB2}(sid, A, B, nA), \mathit{LogX3}(sid, A, B, nA)]\}$$

As shown above, in the case of a log entry appearing as the last member in more than one standard template, there would be multiple restrictions possible for such a log entry, i.e., one restriction per template to be precise. However, to make things simpler, all such restrictions, generated for a log entry, will be combined in a single restriction. It is worth emphasising that for all of our case studies presented later in this chapter, this situation does not arise as the *AllowedSequences*(RL) are prefix-closed, hence none of the log entry appear with a different set of prefix ever in the standard template. Therefore, all our Correspondence restrictions take the form as described earlier.

However, this may not always be true, and we can have some cases where the templates are not prefix-closed. Our system will still be able to handle such cases. E. g., assuming standard session templates $\mathit{Templates}_{\text{Sessions}}$ with *LogX3*(A, B, nA) appearing in more than one standard template, the restriction for *LogX3* will have the following form:

$$
\begin{aligned}
&restriction\ EveryALBeforeLogX3andUniqueLogs :\\
&\quad\text{“}(\forall A, B, nA, sid, t_1.\ \text{LogX3}(sid, A, B, nA)@t_1 \Rightarrow\\
&\quad\ (\exists\ t_2.\text{LogB2}(sid, A, B, nA)@t_2 \wedge (t_2 < t_1)\wedge\\
&\quad\ (\exists\ t_3.\text{LogA1}(sid, A, B, nA)@t_3 \wedge (t_3 < t_2)\wedge\\
&\quad\ (\forall A1, B1, nA1, sid, t_4.\text{LogX3}(sid, A1, B1, nA1)@t_4 \Rightarrow (t_4 = t_1))\wedge\\
&\quad\ (\forall A2, B2, nA2, sid, t_5.\text{LogB2}(sid, A2, B2, nA2)@t_5 \Rightarrow (t_5 = t_2))\wedge\\
&\quad\ (\forall A3, B3, nA3, sid, t_6.\text{LogA1}(sid, A3, B3, nA3)@t_6 \Rightarrow (t_6 = t_3))\\
&\quad\ \vee\\
&\quad\ (\exists\ t_7.\text{LogA1}(sid, A, B, nA)@t_7 \wedge (t_7 < t_1))\wedge\\
&\quad\ (\forall A1, B1, nA1, sid, t_8.\text{LogX3}(sid, A1, B1, nA1)@t_8 \Rightarrow (t_8 = t_1))\wedge\\
&\quad\ (\forall A3, B3, nA3, sid, t_9.\text{LogA1}(sid, A3, B3, nA3)@t_9 \Rightarrow (t_9 = t_7))))\text{”}
\end{aligned}
$$

Lemma 8.1. Every element of a logged trace $tr_{\log}$ generated from well-formed and valid rules will match one and only one of the above generated Tamarin restrictions.

Proof. The proof follows directly from Def. 6.22 and 6.23. □

Lemma 8.2. Given a trace tr_{log}, if the TAMARIN Correspondence restriction is true for a log entry present in tr_{log}, then there is a template that matches all the logged entries that make the restriction true.

Proof. The proof follows directly from Def. 6.22. □

Uniqueness Restrictions: The *uniqueness* restrictions, generated by applying Def. 6.23, enforce that all the fresh names used in the protocol run are unique for each instance, i.e., corresponding to each session ID. This restriction enforces the uniqueness of all the log parameters that are fresh names, except for the session ID, for which the uniqueness is enforced in the *correspondence* restriction. Following Algorithm 8.1, *StealthCheck* scans for all the usages of Fresh fact in protocol rules, e.g., $Fr(x)$, and keeps adding restrictions for each fresh value generated. E.g., our Running Example 5.1 uses the nonce nA in all the protocol rules namely $A1$, $B2$, and $X3$. The following restriction enforces that a specific value of nA only be used in the same run of the protocol and never reused in any other run. These restrictions are generated for all the fresh variables used by a pair of log entries present in the template. In the event of no fresh variables used in the log entries, these restrictions do not need to be enforced and accordingly, the *StealthCheck* skips generating them and adding to the TAMARIN model.

$$
\begin{aligned}
&restriction\ Unique\_na\_forLogA1andLogB2 :\\
&\qquad \text{“}(\forall A1, nA, sid1, A2, B2, sid2, t_1, t_2.\\
&\qquad\qquad \text{LogA1}(sid1, A1, nA)@t_1 \wedge \text{LogB2}(sid2, A2, B2, nA)@t_2 \Rightarrow (sid1 = sid2))\text{”}\\
&restriction\ Unique\_na\_forLogA1andLogX3 : ....
\end{aligned}
$$

StealthCheck verifies the stealthiness of any attack against logs of individual, as well as all possible combination of, participant logs with different sets of parameters. For more details, we refer the reader to *StealthCheck* user manual in Appendix C.

8.5 Testing Stealthiness of Attacks

In order to verify if an attack exists on a security protocol, we execute the Tamarin model of a protocol with security lemmas with each lemma ideally representing a security property. Tamarin may terminate with the lemmas either verified or falsifies meaning the security property being preserved or violated respectively. We take into account only those situations where a security lemma is falsified, providing proof of existence of an attack. This attack model can then be subjected to be tested in a controlled scenario, using the *restriction* feature of Tamarin with the attacker only allowed to perform stealthy actions, i. e., the attack traces conform to both *Correspondence* and *Uniqueness* restrictions. In case the attack still exists, we call such attacks a stealth attack.

As already stated, we have selected Tamarin sources showing violation of any security property demonstrating an attack from its public repository [55]. This has been done to demonstrate that our tool is compatible with the publicly available sources. We divide this section into two subsections, with the first section summarising the stealthiness of attacks on existing public models, followed by a demonstration of our tool for our new KRACK attack model, shown in Fig. 3.4.

8.5.1 Stealthiness of Public TAMARIN models

The *StealthCheck* tool [126] parses the Tamarin source code, adds session identifiers to the rules followed by adding all the required restrictions to check stealthiness of the generated attack traces. It checks the stealthiness of attacks for all possible combinations of logs of participants. As most of the protocols have an initiator I and a responder R, with some of them also using an authenticator or a server S, many combinations of logs are possible. Our analysis results in most of the example attacks being labelled as *stealthy* when analysed against logs of a single participant. *StealthCheck* outputs Tamarin source codes for all the possible combinations of participants, and also outputs the results after executing them. To help the user understand the working on *StealthCheck*, Appendix D.1 lists TAMARIN code of the Needham-Schroeder Public Key Protocol [93], taken from [55], and modified as valid input for *StealthCheck*. It is followed by the output code generated by *StealthCheck* ready to be tested stealthiness of attack on this protocol, placed in Appendix D.2.

Combined logs: Initiator, Responder and Server		**Logged Parameters**			
Vulnerable Protocol	**Security Property Verified**	**Names**	**Names & Nonces**	**Names, Nonces & Keys**	**Names & Message**
Denning-Sacco with ‘prefix property’ [46]	Session Matching	✗	✓	✓	✓
Classic Needham-Schroeder Public Key Protocol [93]	Nonce Secrecy	✗	✓	✓	✓
KAS2 (eCK) Protocol [39]	Key Secrecy	✗	✗	✗	✗
The Station-To-Station Protocol (MAC version) [26]	Perfect Forward Secrecy	✗	✗	✓	✓
The MTI/C0 protocol [91]	Key Secrecy	✗	✗	✗	✗
One-round AKE (JKL_TS2_2004) [83]	Key Secrecy	✗	✗	✓	✓
One-round AKE (JKL_TS1_2004) [83]	Key Secrecy	✗	✓	✓	✓
One-round AKE (JKL_TS1_2008) [84]	Key Secrecy	✗	✗	✗	✗
BAN concrete Andrew Secure RPC [36]	Entity Authentication	✗	✓	✓	✓

Table 8.1: Stealth Attack Analysis against Global parameterised logs

Table 8.1 summarises the stealthiness of known attacks on vulnerable protocols using our Tamarin framework, for a range of logging strategies. For our analysis, we have used four logging strategies as evident in the table; logging only partner names, logging names and nonces, logging names, nonces, and keys and logging names and messages.

After applying the restrictions discussed in Section 6.10 to the models, some attacks are stopped, i. e., become non-stealthy. This is denoted by ✓. The attacks that can bypass the restrictions, i. e., stealth attacks are denoted by ✗. The results of Table 8.1 are based on the combined logs of all the participants. However, it is also possible to verify the stealthiness of these attacks against various combinations of participant's logs.

Next, we analyse stealthiness of these attacks against the logs of only the initiator, the result of which is shown in the Table 8.2. We find that this log is not capable of capturing any attack, and all the attacks are stealthy against such logs. Accordingly, all the attacks are labelled as stealth attacks, denoted by ✗.

Individual participant log (Initiator only)		**Logged Parameters**			
Vulnerable Protocol	**Security Property Verified**	**Names**	**Names & Nonces**	**Names, Nonces & Keys**	**Names & Message**
Denning-Sacco with 'prefix property' [46]	Session Matching	✗	✗	✗	✗
Classic Needham-Schroeder Public Key Protocol [93]	Nonce Secrecy	✗	✗	✗	✗
KAS2 (eCK) Protocol [39]	Key Secrecy	✗	✗	✗	✗
The Station-To-Station Protocol (MAC version) [26]	Perfect Forward Secrecy	✗	✗	✗	✗
The MTI/C0 protocol [91]	Key Secrecy	✗	✗	✗	✗
One-round AKE (JKL_TS2_2004) [83]	Key Secrecy	✗	✗	✗	✗
One-round AKE (JKL_TS1_2004) [83]	Key Secrecy	✗	✗	✗	✗
One-round AKE (JKL_TS1_2008) [84]	Key Secrecy	✗	✗	✗	✗
BAN concrete Andrew Secure RPC [36]	Entity Authentication	✗	✗	✗	✗

Table 8.2: Stealth Attack Analysis against Initiator logs only

We further analyse stealthiness of these attacks against the logs of responder only, the result of which is shown in the Table 8.3. We find that even this log alone, just like the previous case, is not capable of capturing any of these attacks, and all the attacks are stealthy against such logs. Accordingly, all the attacks are labelled as stealth attacks, denoted by ✗.

Individual participant log (Responder only)		**Logged Parameters**			
Vulnerable Protocol	**Security Property Verified**	**Names**	**Names & Nonces**	**Names, Nonces & Keys**	**Names & Message**
Denning-Sacco with 'prefix property' [46]	Session Matching	✗	✗	✗	✗
Classic Needham-Schroeder Public Key Protocol [93]	Nonce Secrecy	✗	✗	✗	✗
KAS2 (eCK) Protocol [39]	Key Secrecy	✗	✗	✗	✗
The Station-To-Station Protocol (MAC version) [26]	Perfect Forward Secrecy	✗	✗	✗	✗
The MTI/C0 protocol [91]	Key Secrecy	✗	✗	✗	✗
One-round AKE (JKL_TS2_2004) [83]	Key Secrecy	✗	✗	✗	✗
One-round AKE (JKL_TS1_2004) [83]	Key Secrecy	✗	✗	✗	✗
One-round AKE (JKL_TS1_2008) [84]	Key Secrecy	✗	✗	✗	✗
BAN concrete Andrew Secure RPC [36]	Entity Authentication	✗	✗	✗	✗

Table 8.3: Stealth Attack Analysis against Responder logs only

We have seen that the analysis based on the individual logs only results in all the attacks getting marked as stealthy. However, it is not the case for a combination of participant logs. Analysis suggests that combined logs of initiator and responder is enough to capture the stealthiness of the attack, even without the server log. E. g., the attack on Denning-Sacco protocol with prefix property is non-stealthy against combined logs of initiator and responder only, even after excluding the server logs. This is an important result as, apart from suggesting the optimal parameters from the logs, we can also figure

Combined log of Initiator and Responder		**Logged Parameters**			
Vulnerable Protocol	**Security Property Verified**	**Names**	**Names & Nonces**	**Names, Nonces & Keys**	**Names & Message**
Denning-Sacco with 'prefix property' [46]	Session Matching	✗	✓	✓	✓
Classic Needham-Schroeder Public Key Protocol [93]	Nonce Secrecy	✗	✓	✓	✓
KAS2 (eCK) Protocol [39]	Key Secrecy	✗	✗	✗	✗
The Station-To-Station Protocol (MAC version) [26]	Perfect Forward Secrecy	✗	✗	✓	✓
The MTI/C0 protocol [91]	Key Secrecy	✗	✗	✗	✗
One-round AKE (JKL_TS2_2004) [83]	Key Secrecy	✗	✗	✓	✓
One-round AKE (JKL_TS1_2004) [83]	Key Secrecy	✗	✓	✓	✓
One-round AKE (JKL_TS1_2008) [84]	Key Secrecy	✗	✗	✗	✗
BAN concrete Andrew Secure RPC [36]	Entity Authentication	✗	✓	✓	✓

Table 8.4: Stealth Attack Analysis against Initiator and Responder combined logs

out the optimal combination of logs to detect the attacks.

Usefulness of Stealthiness

A protocol may have various participants and similarly many attacks exploiting the design vulnerabilities. In order to detect the attacks, the protocol designers would always be interested in finding out the most suitable partner, or a set of partners, capable of providing the most useful analysis results. Testing the stealthiness of an attack w.r.t. various combination of logging parameters helps us in identifying the logging partners and parameters, to be used in the logs, in order to detect the stealth attacks. As we can see from the tables above, there are cases where even a subset of participant's logs are sufficient to detect the attacks.

Analysis of Attacks Stealthiness w.r.t. Logs parameters

The analysis clearly establishes the fact that only logging the names may not be a good idea, as the attacker can easily mount stealth attacks in such cases. It is highly recommended that in addition to the participant names, nonces, and keys (both hashed) should be added to the logs. After trying out various combination of logs and a mix of parameters in the logs, we have three attacks namely, key secrecy attacks on KAS2 (eCK) Protocol

[39], The MTI/C0 protocol [91] and One-round AKE (JKL_TS1_2008) [84], that could not be detected, and hence are stealthy against all combination of logs. Analysis of the stealthiness of various attacks as shown in the tables above, in case of combination of participant logs, also presents us with the following findings:

Logs with Names: None of the attacks is captured by the logs containing only the participant identifiers, i.e., names and hence all our attacks are stealthy against the logs containing only names.

Logs with Names and Nonces: Some attacks are captured by the logs containing the participant identifier and nonces used in the protocol, i.e., names and nonces combination. In fact, out of nine attacks, only five attacks remain stealthy against the logs containing the names and nonces while the rest are captured by the logs and hence non-stealthy.

Logs with Names, Nonces and Keys: In this case, some more attacks are captured by the logs containing the participant identifier, nonces, and keys used in the protocol, i.e., names, nonces, and keys combination. Now, out of Nine (09) attacks, only Three (03) attacks remain stealthy against the logs containing the names, nonces, and keys while the rest are captured by the logs and hence non-stealthy.

Logs with Names and Messages: The results in this case are exactly the same as per the logs with names, nonces, and keys combination. It implies that logs with names and messages are not able to capture any more attacks, and hence adding the whole messages to the logs does not have any qualitative effect.

8.5.2 Stealthiness of KRACK attack

In the previous section, most of the protocols used as case studies are theoretical, very old, or rarely implemented. This begs the question whether our tool can be used to learn something new, a real-world protocol attack. With this objective, we have modelled the KRACK attack presented in Fig. 3.4 using TAMARIN. Using the *StealthCheck* tool, we receive the following results:

Initiator log (Authenticator only)		**Logged Parameters**		
Vulnerable Protocol	**Attack Modelled**	**Names**	**Names & Nonces**	**Names, Nonces & Keys**
IEEE 802.11 *4-way handshake* [137]	KRACK attack	✗	✗	✗

Table 8.5: Stealthiness of KRACK attack against Authenticator logs only

Responder log (Supplicant only)		**Logged Parameters**		
Vulnerable Protocol	**Attack Modelled**	**Names**	**Names & Nonces**	**Names, Nonces & Keys**
IEEE 802.11 *4-way handshake* [137]	KRACK attack	✗	✓	✓

Table 8.6: Stealthiness of KRACK attack against Supplicant logs only

Combined logs (Authenticator and Supplicant)		**Logged Parameters**		
Vulnerable Protocol	**Attack Modelled**	**Names**	**Names & Nonces**	**Names, Nonces & Keys**
IEEE 802.11 *4-way handshake* [137]	KRACK attack	NT	✓	✓

Table 8.7: Stealthiness of KRACK attack against combined logs (NT:Non-Termination)

Analysis of KRACK attack

The KRACK attack is captured by the Supplicant using its logs alone, with the logs containing 'Names and Nonces' and 'Names, Nonces and Keys'. Intuitively, a combined log of Supplicant and Authenticator messages should also be able to capture this attack. However, the TAMARIN model does not terminate in the case of testing KRACK against the combined logs. We would like to handle the non-termination as future work. However, the non-termination in TAMARIN, we can either prove the security property in interactive mode or prove only a specific lemma. All the TAMARIN models of protocol examples used in this section, ready to be analysed by the *StealthCheck* tool, and the tool manual is present at [126]. The *StealthCheck* tool manual is also made available in Appendix C.

8.6 Chapter Summary

Based on the analysis of attacks and their stealthiness, we can summarise our results to safely make some conclusions. To start with, the presence or absence of logged values does have an effect on the stealthiness of the attack. Needless to say, as expected, all the attacks with empty logs are always stealthy. The attacks are also stealthy against logs with only names. In some cases, adding additional parameters to logs also does not help in capturing the attacks. However, a universal log is always more effective than individual local logs in detecting stealth attacks. We define a universal log as the combination of logs of all the parties, e.g., both clients or client(s) and server, whereas the local log will belong to only one participant such as only the client or server.

There is an important distinction between how industry looks at a single participant log compared to our framework. Lots of industry look at a single log to look for attacks and find it sufficient to detect attacks, such as scanning, brute-force attacks, and buffer overflow, just by looking at them. A single participant log, in such cases, can be sufficient for an IDS to flag that an attack is in progress.

However, our analysis of attacks is different in the sense that we are looking for protocol attacks under Dolev-Yao [64] attacker model where a single logs usually is insufficient to show attacks. The attacks, analysed by our framework, under Dolev-Yao model [64], are difficult to be detected using a single participant log alone as we need a group of logs to look for correspondence among the participant's activity. As can be seen in our results, a single participant log alone has mostly been unable to capture any attack, barring the case of KRACK attack captured by the Responder log alone.

All the other cases of non-stealth attacks have involved some combination of logs. It is possible to convert a non-stealthy attack to a stealthy one or vice-versa by adding parameters to a combined log of protocol participants.

An important future result could be testing the impact of logging certain protocol steps, and leaving out others, using our framework. An attack may be ‘stealthy’ if a specific step is not logged, as it allows the attacker to perform an activity without getting noticed, and vice versa.

While we have been able to analyse the stealthiness of the attacks under study, we did not have enough time to look at the attacks and their behaviour in detail. We leave it as future work to analyse the attack and investigate why some attacks could not be captured by any combination of logs.

Part IV

Closing Statements

Chapter Nine

Conclusions and Future Directions

In literature and in life we ultimately pursue, not conclusions, but beginnings.

Sam Tanenhaus

During the course of this thesis, we have attempted to answer three research questions presented in Section 1.2 relating to security properties and stealthiness in the attacks on security protocols.

We studied various examples of complete protocol modelling, which used formal verification to test if the security guarantees claimed by the protocols hold or not. The literature study showed that such modelling is, more often than not, a complex task, and are both time and effort-intensive process. Keeping in mind that an attack on the protocol should be expected to violate one of its security guarantees, we decided to explore modelling of an attack, as compared to the full protocol model, and test the security properties. Our methodology involved modelling only a subset of interactions, leading to an attack. By modelling KRACK and Downgrade attacks on WPA2 *4-way handshake*, we successfully established that there is enough merit in modelling the attacks. We proved that the set of security properties mandated is not sufficient to capture these attacks. We were also able to demonstrate the technique of augmenting additional security properties to capture the attacks. The guiding philosophy of this work was to uncover the insufficiency of security properties and identify new ones by modelling various attack scenarios that can be used to stop such class of attacks in future.

Subsequently, in order to classify the attacks as stealth and non-stealth ones, we provided a novel definition of stealthiness for attacks against a protocol by comparing the attack trace to templates generated by single normal runs of the protocol. We also formally defined the formation of a normal looking trace by merging multiple valid single traces. Subsequently, we successfully established that a formal definition of a standard run of a protocol is equivalent to imposing some restrictions on a version of the protocol annotated with session IDs. Using our *StealthCheck* tool, we have shown how this definition can be automatically checked using the TAMARIN tool. Our framework can be used to identify stealth attacks on any vulnerable protocol and generate the minimal set of parameters to be logged in order to convert the stealth attacks to non-stealth attacks.

Our analysis of multiple vulnerable protocols suggested that it is possible to identify most of the attacks using a correct logging strategy. Our technique and results can be used to recommend the minimal set of parameters, which if logged, could help detect many attacks on security protocols. The results of experiments from the case studies presented in Chapter 6 not only help one understand the details of stealth attacks on protocols, but also help evaluate the efficacy of various detection mechanisms adopted.

The results of checking stealthiness of various attacks helped us to form some conclusions. We found that the presence or absence of logged values was directly related with stealthiness of the attack. The attacks with empty logs were always stealthy. None of the attacks were captured by stealthy logs containing only the names, and sometimes adding additional parameters was also not helpful. However, a universal log, i.e., a combined log of various participants, was always more effective than individual local logs in detecting stealth attacks. We found that we can convert a non-stealthy attack to a stealthy one or vice-versa by adding parameters to a combined log of protocol participants.

9.1 Contributions and Reflections

The research work presented in this thesis can be helpful in many ways to augment, and test, the design of resilient systems. While the attack modelling technique can be used to improve the protocol specification by augmenting additional security properties, the stealth framework can be used to improve the protocol design, developing a system to analyse logs during an attack to raise the flag in case of an attack, and handling zero-day attacks etc.

Additionally, this thesis has demonstrated use of both pen-and-paper and automatic verification techniques of protocols, their security requirements, and attacks on them using Tamarin [103] to verify the security guarantees based on protocol traces. Modelling of Tamarin using theorem prover Coq is a novel contribution and has the potential of opening up new frontiers in research by verifying the correctness of formal results. We outline a summarised contribution of our work presented in this thesis, followed by their possible applications, in this section.

Modelling of Attacks, and not the Complete Protocol: The formal modelling of attacks presented in Chapter 3 stands in contrast with earlier formal models of protocols which usually involved modelling of the complete specifications of the protocols. Our attempt was prompted by the fact that modelling a complete protocol is a time and effort intensive process. The earlier complete models also needed to be accurate and require understanding of intricate details of every state of protocol state machine. In contrast to this, our model comprising a subset of these state machine states has proved to be cost-effective and helpful in improving the protocol specification. We have successfully modelled various attacks, on IEEE 802.11 4-way handshake protocols, such as KRACK [137] and Downgrade attacks [131], and found that it's much easier to produce a smaller model to study attacks compared to modelling the complete protocol specifications. This contribution aligns with our research questions **RQ1** presented in Section 1.2.

Testing Adequacy of Set of Security Properties: Chapter 4 made an important contribution by proposing a new paradigm in protocol analysis where, given a known attack on a protocol along with its set of security properties, the analysis evaluated the adequacy of this set of security properties in capturing or missing this attack. This methodology was also able to check if the set of security properties needed to be augmented with additional security properties. After analysing various attacks on IEEE 802.11 4-way handshake protocols, such as KRACK [137] and Downgrade attacks [131], we found that the set of security properties were insufficient. We then successfully suggested additional security properties to enable capturing such classes of attacks. This contribution aligns with our research questions **RQ1** presented in Section 1.2.

This is an important result as it can not only analyse the existing security properties, but also recommend an additional set of security properties that should be added to the protocol standards to make them more robust. Finally, our methodology also allows to test the correctness of augmented security properties in stopping the attacks.

Stealth Attack Framework: Using the definition of a 'standard looking trace' presented in Chapter 5, we provided a novel definition of stealthiness for attacks against a protocol by comparing the attack traces against standard looking runs of the protocol. We started by developing a formal model to represent the protocol interaction employing multiset rewriting rules that allows labelled logging of parameters during protocol run. Then, by using formal notions, we presented definitions of a stealthy run and stealth attack.

Subsequently, in Chapter 6, we showed how we can implement this formal definition using automatic protocol verification tool Tamarin by annotating each run of the protocol with session IDs. The standard protocol model was executed under stealthy restrictions, allowing production of traces corresponding to only the stealthy runs of a protocol.

Many protocols can be modelled against known attacks on them, and an optimal set of logging parameters can be put together to convert a stealth attack to a non-stealth one. This information, i.e., optimal set of logging parameters, can be useful while designing similar classes of protocols in future and such parameters may be verified by the communicating parties as part of the protocol interaction. The analysis of multiple vulnerable protocols suggested that it is possible to detect most of the attacks using a correct logging strategy.

Using *StealthCheck* framework, the results of experiments presented in Chapter 8 not only verify the stealthiness of stealth attacks on protocols, but also compare the efficacy of various detection mechanisms adopted. This contribution aligns with our research questions **RQ2** and **RQ3** presented in Section 1.2.

Verification using Coq: Presented in Chapter 7, to prove that the formal model for stealth check and TAMARIN model for stealth check are equivalent, we used Coq, a formal proof management system. This approach included modelling of TAMARIN semantics in Coq, which is a novel research contribution and can be helpful for the research community to understand the working of TAMARIN better. While we have not been able to provide a complete machine-checked proof, using Coq has helped us type check our definitions, lemmas, and theorems etc. We look forward to completing the manual proofs of theorems, presented in this thesis, as a future work.

The methodologies presented in this thesis have the potential of opening up new frontiers in formal verification. We have demonstrated that modelling of just the attack traces against the complete protocol specification is indeed economical as well effective effort, helpful in improving the protocol specification. The testing of stealthiness of attacks on security protocols can be used to improve the protocols and also convert stealth attacks to non-stealth attacks. We discuss some possible applications of our work below.

While this thesis has made some key research contributions, we should not lose sight of the fact that, in terms of their real-world implementations, our research work still needs further attention. Both our *key* contributions; novel methodology of verifying the set of security properties, and testing stealthiness in the protocol trace, have been performed following the principles of formal verification methods, more specifically using symbolic analysis of protocols. As such, the results suffer from all the usual limitations and drawbacks of formal verification and symbolic analysis techniques. We have not been able to model many network parameters such as timing of messages, packet size, and network load etc. to name a few. While our results hold theoretically, we have not been able to test them in a practical setting.

In the following section, we list out some possible future research directions related to our research presented in this thesis. Our proposals consist of both the possible extensions to the current work, and future adaptations based on our framework.

9.2 Future Scope and Directions

The work presented in the thesis has potential of being adopted for possible future research directions in the area of modelling and verification of the protocols.

The methodology of modelling only the attacks against the complete protocol specification can be easily applied to other use cases. It can be easily applied to test the set of required security properties for other protocols, such as TLS, against known attacks on them. It could be interesting to see if the results suggest additional security properties to be augmented to the protocol specifications. Some ideal candidates could be modelling of attacks on TLS [128] such as BEAST [120], LOGJAM [7]), Triple handshake [21], etc. in order to test the adequacy of set of security properties listed in the TLS standard.

Similarly, the stealth framework presented in this thesis can also be used to improve the initial protocol design and their robustness. Formal analysis of a protocol has the potential to uncover possible attacks. Once such an attack is found, stealthiness of this attack can be tested using our framework, and if found stealthy, we can find out the set of parameters to convert this attack to non-stealth attack.

Our stealth framework can model any protocol, including those that are victim of a zero-day attack. Using various combination of logs, our framework can evaluate stealthiness of the zero-day attack, and suggest a possible set of parameters to detect the attack as a non-stealthy one.

Possible Extensions to Current Framework

Extending notion of 'Stealthiness' In this thesis, while modelling the protocol attacks and verifying their stealthiness, we do not consider aspects of network traffic, such as timing and packet size, or the probability of a particular message, which could be used to indicate attacks. It should be an interesting study to model these attributes and to see the results, we leave such an extension as future work.

Another interesting study could be verifying the impact of logging certain protocol steps on stealthiness of the attacks. An attack may be 'stealthy' if a specific step is not logged, as it allows the attacker to perform an activity without getting noticed, and vice versa. Such experiments will help both the protocol designers and attackers to identify and compare various protocol steps in terms of their relative vulnerability.

Completing 'Coq' framework We have shown the proof sketch of our central theorem based on a mix of manual and 'Coq' based proofs of propositions and lemmas presented in Chapter 6. We would like to convert the manual proofs of all the lemmas and propositions to develop a machine-checked proof for our central theorem showing that the attack trace generated by the rule list *ProtoRules*, using the Def. 5.9 and another attack trace generated using same rule list with session identifiers, are equivalent if the latter conforms to Correspondence and Uniqueness restrictions.

Future Adaptations Based on Our Work

Log-based IDS Our stealth framework can be implemented to develop an intrusion detection system (IDS), where the communicating parties may share their logs, containing the logged values and timestamp, with a trusted server. Since this server will have logs from all the protocol participants, it will essentially have access to global view or universal logs. The server, capable of testing stealthiness of protocol traces using our framework, would raise an alert, to all the parties, as soon as it detects a deviation from the standard expected run. Subsequently, all the parties can take appropriate steps to stop the attack.

Such a system could also take the form of a forensic suite capable of performing stealthiness of traces. By fine-tuning the parameters to be logged, these IDS should be able to capture and convert stealthy attacks to non-stealthy, making them easy to capture.

Handling Zero-day Attacks Zero-click exploits [107], such as by Pegasus [98, 99], are successful in taking over complete control of the devices, mainly mobile phones, without active participation of the users. These attacks exploit the bugs present in popular messaging applications such as iMessage, FaceTime, and WhatsApp etc. As per the literature available [98, 99], its working has remained stealthy, and it took a while for the community to verify presence of this spyware on victim phones. Our 'stealthiness' framework can be employed for modelling such attacks, and try to capture these attacks by observing possible changes in system logs, if any.

Studying Stealth attacks on IDS A stealthy attacker may not try to break the whole system in one go, rather focusing on gaining useful information in a stealthy manner, i. e., without getting noticed. Wagner and Soto [139] have shown that the IDS can be susceptible to mimicry attacks, which disguise an attack appearing normal to the IDS. Lightweight stealthy attacks may be dangerous, and a small attack can be carried out multiple times to mount severe attacks. Sufatrio et al., [144] have also presented an efficient algorithm for automated mimicry attack construction, which can be useful for evaluating the robustness of the IDS to attacks.

Modelling these attacks using our framework can be an interesting research problem to test if such actions can be captured by putting the successive logged information together. A mimicry attack is able to cloak a basic attack subtrace into a stealthy attack subtrace which the IDS classify as being normal. The important point to note here is the 'subtrace'. Recall that, in our stealth model, the allowed sequences of protocols are prefix-closed and therefore capable of modelling both the complete and incomplete runs of a protocol. On similar lines, our framework may be tweaked to model subtraces, as a possible extension to our work.

Stealth attack on users Humans can be modelled as part of the system, based on their knowledge level, characterised as Naive, Intermediate, and Expert [14]. Every level of user would be expected to know about some safeguards. During the interaction with the machine, a human or automated program would respond to a protocol step following the protocol specifications. A dishonest user or an adversary controlled user would be able to launch successful attacks.

Stealth attacks on humans, e. g., phishing or social engineering attacks, can then be defined, assuming that any attack on an expert user would be unstoppable by a user at intermediate or beginner level. At the same time, an attack on a beginner would be detected by the other two groups and will be a non-stealth attack to them. These results can be used to classify the stealth attacks on human or automated users based on their capability levels.

We can use the results of stealth or non-stealth attacks on various level of users in identifying the weaknesses among users. Subsequently, these results can be helpful in training the users to handle various social engineering and phishing attacks.

Analysing attacks on a combination of secure protocols Similar to the use of Tamarin in formal analysis of combination of secure protocols presented in [31], extending our stealth framework to handle multiprotocol traces can be an interesting research problem. Research [49] has shown that many protocols are vulnerable to multi-protocol attacks, and simply detecting attacks against a single protocol alone may not be sufficient to stop the attacks. Constructing various possible standard traces based on multiple protocols running simultaneously is likely to be a challenging task. However, by analysing the combination of protocol traces, we can use our stealth framework to develop a notion of 'standard trace' in a multi-protocol execution setting. Such a research study can also be helpful in understanding multi-protocol attack traces [49], and analysing their stealthiness based on our framework.

Bibliography

[1] IEEE 802.11 WLANs WG Group Information. Available at https://mentor.ieee.org/802.11/dcn/17/11-17-1602-03-000m-nonce-reuse-prevention.docx (Accessed : July 2020).

[2] Wi-Fi Alliance : Security Update October 2017. Available at https://www.wi-fi.org/security-update-october-2017 (Accessed : July 2020).

[3] IEEE Standard for Local and Metropolitan Area Networks–Port-Based Network Access Control. IEEE Std. 802.1X-2020, Feb. 2020.

[4] Abad, C., Taylor, J., Sengul, C., Yurcik, W., Zhou, Y., and Rowe, K. Log correlation for intrusion detection: A proof of concept. In *Computer Security Applications Conference, 2003. Proceedings. 19th Annual* (2003), IEEE, pp. 255–264.

[5] Abadi, M., and Needham, R. Prudent engineering practice for cryptographic protocols. In *Research in Security and Privacy, 1994. Proceedings., 1994 IEEE Computer Society Symposium on* (1994), IEEE, pp. 122–136.

[6] Abadi, M., and Rogaway, P. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of cryptology 15*, 2 (2002), 103–127.

[7] Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J. A., Heninger, N., Springall, D., Thomé, E., Valenta, L.,

et al. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 5–17.

[8] Agarwal, M., Biswas, S., and Nandi, S. Advanced Stealth Man-in-The-Middle Attack in WPA2 Encrypted Wi-Fi Networks. *IEEE Communications Letters 19*, 4 (2015), 581–584.

[9] AlEroud, A., and Karabatis, G. A Contextual Anomaly Detection Approach to Discover Zero-Day Attacks. In *2012 International Conference on Cyber Security* (2012), pp. 40–45.

[10] Armando, A., Basin, D., Boichut, Y., Chevalier, Y., Compagna, L., Cuéllar, J., Drielsma, P. H., Héam, P.-C., Kouchnarenko, O., Mantovani, J., et al. The AVISPA tool for the automated validation of internet security protocols and applications. In *International conference on computer aided verification* (2005), Springer, pp. 281–285.

[11] Armando, A., Carbone, R., Compagna, L., Cuellar, J., and Tobarra, L. Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for Google apps. In *Proceedings of the 6th ACM workshop on Formal methods in security engineering* (2008), ACM, pp. 1–10.

[12] Avalle, M., Pironti, A., and Sisto, R. Formal verification of security protocol implementations: a survey. *Formal Aspects of Computing 26*, 1 (2014), 99–123.

[13] Basin, D., Dreier, J., Hirschi, L., Radomirovic, S., Sasse, R., and Stettler, V. A formal analysis of 5G authentication. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), pp. 1383–1396.

[14] Basin, D., Radomirovic, S., and Schmid, L. Modeling human errors in security protocols. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)* (2016), IEEE, pp. 325–340.

[15] Bella, G., Massacci, F., and Paulson, L. C. Verifying the SET purchase protocols. *Journal of Automated Reasoning 36*, 1-2 (2006), 5–37.

[16] Bellare, M., and Rogaway, P. Entity authentication and key distribution. In *Annual international cryptology conference* (1993), Springer, pp. 232–249.

[17] Bertot, Y. A short presentation of Coq. In *International Conference on Theorem Proving in Higher Order Logics* (2008), Springer, pp. 12–16.

[18] Bertot, Y., and Castéran, P. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions.* Springer Science & Business Media, 2013.

[19] Bhargavan, K., Blanchet, B., and Kobeissi, N. Verified models and reference implementations for the TLS 1.3 standard candidate. In *2017 IEEE Symposium on Security and Privacy (SP)* (2017), IEEE, pp. 483–502.

[20] Bhargavan, K., Fournet, C., and Kohlweiss, M. mitls: Verifying protocol implementations against real-world attacks. *IEEE Security & Privacy 14*, 6 (2016), 18–25.

[21] Bhargavan, K., Lavaud, A. D., Fournet, C., Pironti, A., and Strub, P. Y. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *2014 IEEE Symposium on Security and Privacy* (2014), IEEE, pp. 98–113.

[22] Bhargavan, K., and Leurent, G. Transcript collision attacks: Breaking authentication in TLS, IKE, and SSH. In *Network and Distributed System Security Symposium – NDSS 2016* (2016).

[23] Bian, H., Bai, T., Salahuddin, M. A., Limam, N., Abou Daya, A., and Boutaba, R. Host in danger? detecting network intrusions from authentication logs. In *2019 15th International Conference on Network and Service Management (CNSM)* (2019), IEEE, pp. 1–9.

[24] Bilge, L., and Dumitraş, T. Before we knew it: an empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 833–844.

[25] Birge-Lee, H., Sun, Y., Edmundson, A., Rexford, J., and Mittal, P. Using BGP to acquire bogus TLS certificates. In *Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2017), Minneapolis, MN, USA* (2017).

[26] Blake-Wilson, S., and Menezes, A. Unknown key-share attacks on the station-to-station (STS) protocol. In *International Workshop on Public Key Cryptography* (1999), Springer, pp. 154–170.

[27] Blanchet, B. Automatic proof of strong secrecy for security protocols. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004* (2004), IEEE, pp. 86–100.

[28] Blanchet, B. Security protocol verification: Symbolic and computational models. In *Proceedings of the First international conference on Principles of Security and Trust* (2012), Springer-Verlag, pp. 3–29.

[29] Blanchet, B., Cheval, V., Allamigeon, X., and Smyth, B. Proverif: Cryptographic protocol verifier in the formal model. *URL http://prosecco. gforge. inria. fr/personal/bblanche/proverif* (2010).

[30] Blanchet, B., et al. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends® in Privacy and Security 1*, 1-2 (2016), 1–135.

[31] Blot, E., Dreier, J., and Lafourcade, P. Formal Analysis of Combinations of Secure Protocols. In *International Symposium on Foundations and Practice of Security* (2017), Springer, pp. 53–67.

[32] Bopardikar, S. D., and Speranzon, A. On analysis and design of stealth-resilient control systems. In *Resilient Control Systems (ISRCS), 2013 6th International Symposium on* (2013), IEEE, pp. 48–53.

[33] Bortolozzo, M., Centenaro, M., Focardi, R., and Steel, G. Attacking and fixing PKCS# 11 security tokens. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), ACM, pp. 260–269.

[34] Boyd, C., and Mathuria, A. *Protocols for authentication and key establishment.* Springer Science & Business Media, 2013.

[35] Bryans, J., and Ryan, P. Security and Trust in a Voter-Verifiable Voting Scheme. In *Fast* (2003), pp. 113–120.

[36] Burrows, M., Abadi, M., and Needham, R. M. A logic of authentication. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* (1989), vol. 426, The Royal Society, pp. 233–271.

[37] Callegati, F., Cerroni, W., and Ramilli, M. Man-in-the-Middle Attack to the HTTPS Protocol. *IEEE Security & Privacy 7*, 1 (2009), 78–81.

[38] Cazorla, L., Alcaraz, C., and Lopez, J. Cyber Stealth Attacks in Critical Information Infrastructures. *IEEE Systems Journal* (2016).

[39] Chatterjee, S., Menezes, A., and Ustaoglu, B. A generic variant of NIST's KAS2 key agreement protocol. In *Australasian Conference on Information Security and Privacy* (2011), Springer, pp. 353–370.

[40] Cheng, Z., Tisi, M., and Douence, R. CoqTL: a Coq DSL for rule-based model transformation. *Software and Systems Modeling 19*, 2 (2020), 425–439.

[41] Chrétien, R., Cortier, V., and Delaune, S. Decidability of trace equivalence for protocols with nonces. In *2015 IEEE 28th Computer Security Foundations Symposium* (2015), IEEE, pp. 170–184.

[42] Clark, J., and Jacob, J. A survey of authentication protocol literature, 1997.

[43] Coffey, T., Dojen, R., and Flanagan, T. Formal verification: an imperative step in the design of security protocols. *Computer Networks 43*, 5 (2003), 601–618.

[44] Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., and Stebila, D. A formal security analysis of the signal messaging protocol. *Journal of Cryptology 33*, 4 (2020), 1914–1983.

[45] Cook, B. Formal reasoning about the security of Amazon Web Services. In *International Conference on Computer Aided Verification* (2018), Springer, pp. 38–47.

[46] Cortier, V., Delaune, S., and Lafourcade, P. A survey of algebraic properties used in cryptographic protocols. *Journal of Computer Security 14*, 1 (2006), 1–43.

[47] Cortier, V., Filipiak, A., Florent, J., Gharout, S., and Traoré, J. Designing and proving an EMV-compliant payment protocol for mobile devices. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)* (2017), IEEE, pp. 467–480.

[48] Cortier, V., Grimm, N., Lallemand, J., and Maffei, M. A type system for privacy properties. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), pp. 409–423.

[49] Cremers, C. Feasibility of multi-protocol attacks. In *First International Conference on Availability, Reliability and Security (ARES'06)* (2006), IEEE, pp. 8–pp.

[50] Cremers, C. On the protocol composition logic PCL. In *Proc. ACM Symp. Inf., Comput. Commun. Secur. (ASIACCS)* (Tokyo, Japan, Mar. 2008), ACM, pp. 66–77.

[51] Cremers, C., Horvat, M., Hoyland, J., Scott, S., and van der Merwe, T. A comprehensive symbolic analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), pp. 1773–1788.

[52] Cremers, C., Horvat, M., Scott, S., and van der Merwe, T. Analysis of TLS 1.3: 0-RTT, Resumption and Delayed Authentication. `https://tls13tamarin.github.io/TLS13Tamarin/tls13-draft10.html`. (Accessed on 14/08/2021).

[53] Cremers, C., Horvat, M., Scott, S., and van der Merwe, T. Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In *Security and Privacy (SP), 2016 IEEE Symposium on* (2016), IEEE, pp. 470–485.

[54] Cremers, C., Kiesl, B., and Medinger, N. A Formal Analysis of {IEEE} 802.11's WPA2: Countering the Kracks Caused by Cracking the Counters. In *29th {USENIX} Security Symposium ({USENIX} Security 20)* (2020), pp. 1–17.

[55] Cremers, C., and Schmidt, B. Github - tamarin-prover/tamarin-prover: Main source code repository of the tamarin prover for security protocol verification. `https://github.com/tamarin-prover/tamarin-prover`. (Accessed on 11/09/2020).

[56] Cremers, C. J., Mauw, S., and De Vink, E. Defining authentication in a trace model. In *Fast* (2003), pp. 131–145.

[57] Cremers, C. J., Mauw, S., and de Vink, E. P. A syntactic criterion for injectivity of authentication protocols. *Electronic Notes in Theoretical Computer Science 135*, 1 (2005), 23–38.

[58] Cremers, C. J. F. *Scyther: Semantics and verification of security protocols.* Eindhoven University of Technology Eindhoven, Netherlands, 2006.

[59] Datta, A., Derek, A., Mitchell, J. C., and Roy, A. Protocol composition logic (PCL). *Electronic Notes in Theoretical Computer Science 172* (2007), 311–358.

[60] De Renesse, F., and Aghvami, A. Formal verification of ad-hoc routing protocols using SPIN model checker. In *Electrotechnical Conference, 2004. MELECON 2004. Proceedings of the 12th IEEE Mediterranean* (2004), vol. 3, IEEE, pp. 1177–1182.

[61] Delaune, S., Kremer, S., and Ryan, M. Verifying privacy-type properties of electronic voting protocols: A taster. In *Towards trustworthy elections*. Springer, 2010, pp. 289–309.

[62] Dershowitz, N., Kaplan, S., and Plaisted, D. A. Rewrite, rewrite, rewrite, rewrite, rewrite,. . . . *Theoretical Computer Science 83*, 1 (1991), 71–96.

[63] Diffie, W., Van Oorschot, P. C., and Wiener, M. J. Authentication and authenticated key exchanges. *Designs, Codes and cryptography 2*, 2 (1992), 107–125.

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

[65] Dowling, B., Fischlin, M., Günther, F., and Stebila, D. A Cryptographic Analysis of the TLS 1.3 draft-10 Full and Pre-shared Key Handshake Protocol. *IACR Cryptol. ePrint Arch. 2016* (2016), 81.

[66] Durumeric, Z., Kasten, J., Adrian, D., Halderman, J. A., Bailey, M., Li, F., Weaver, N., Amann, J., Beekman, J., Payer, M., et al. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (2014), ACM, pp. 475–488.

[67] Even, S., Goldreich, O., and Shamir, A. On the security of ping-pong protocols when implemented using the RSA. In *Conference on the Theory and Application of Cryptographic Techniques* (1985), Springer, pp. 58–72.

[68] Fábrega, F. J. T., Herzog, J. C., and Guttman, J. D. Strand spaces: Why is a security protocol correct? In *Proceedings. 1998 IEEE Symposium on Security and Privacy (Cat. No. 98CB36186)* (1998), IEEE, pp. 160–171.

[69] Ficco, M., and Rak, M. Intrusion tolerance of stealth DoS attacks to web services. In *IFIP International Information Security Conference* (2012), Springer, pp. 579–584.

[70] Gavrichenkov, A. Breaking https with bgp hijacking. *Black Hat. Briefings* (2015).

[71] Goguen, J. A., and Meseguer, J. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science 105*, 2 (1992), 217–273.

[72] Goldwasser, S., and Micali, S. Probabilistic encryption. *Journal of computer and system sciences 28*, 2 (1984), 270–299.

[73] Gruss, D., Maurice, C., Wagner, K., and Mangard, S. Flush+ Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2016), Springer, pp. 279–299.

[74] Guo, Z., Shi, D., Johansson, K. H., and Shi, L. Worst-case stealthy innovation-based linear attack on remote state estimation. *Automatica 89* (2018), 117–124.

[75] He, C., and Mitchell, J. C. Analysis of the 802.11 i 4-Way Handshake. In *Proceedings of the 3rd ACM workshop on Wireless security* (2004), pp. 43–50.

[76] He, C., Sundararajan, M., Datta, A., Derek, A., and Mitchell, J. C. A modular correctness proof of IEEE 802.11 i and TLS. In *Proceedings of the 12th ACM conference on Computer and communications security* (2005), pp. 2–15.

[77] Hirschi, L., Baelde, D., and Delaune, S. A method for verifying privacy-type properties: the unbounded case. In *2016 IEEE Symposium on Security and Privacy (SP)* (2016), IEEE, pp. 564–581.

[78] IEEE. IEEE standard for Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specs. *IEEE Std 802.11-1997* (1997), 1–445.

[79] IEEE. IEEE Standard for information technology-Telecommunications and information exchange between systems-Local and metropolitan area networks-Specific requirements-Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Amendment 6: Medium Access Control (MAC) Security Enhancements. *IEEE Std 802.11i-2004* (2004), 1–190.

[80] IEEE. IEEE standard for Information technology—Telecommunications and information exchange between systems Local and metropolitan area networks—Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. *IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012)* (2016), 1–3534.

[81] Jakobsson, M., Wetzel, S., and Yener, B. Stealth attacks on ad-hoc wireless networks. In *Vehicular Technology Conference, 2003. VTC 2003-Fall. 2003 IEEE 58th* (2003), vol. 3, IEEE, pp. 2103–2111.

[82] Javed, M., and Paxson, V. Detecting stealthy, distributed SSH brute-forcing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 85–96.

[83] Jeong, I. R., Katz, J., and Lee, D. H. One-round protocols for two-party authenticated key exchange. In *International Conference on Applied Cryptography and Network Security* (2004), Springer, pp. 220–232.

[84] Jeong, I. R., Katz, J., and Lee, D. H. One-round protocols for two-party authenticated key exchange. *Journal of KIISE: Computer Systems and Theory 33* (2008).

[85] Joux, A. Authentication failures in NIST version of GCM. *NIST Comment* (2006), 3.

[86] Kemmerer, R. A. Analyzing encryption protocols using formal verification techniques. *IEEE Journal on Selected areas in Communications 7*, 4 (1989), 448–457.

[87] Kenkre, P. S., Pai, A., and Colaco, L. Real time intrusion detection and prevention system. In *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014* (2015), Springer, pp. 405–411.

[88] Kim, J. Y., Holz, R., Hu, W., and Jha, S. Automated Analysis of Secure Internet of Things Protocols. In *Proceedings of the 33rd Annual Computer Security Applications Conference* (2017), ACM, pp. 238–249.

[89] Kremer, S., and Künnemann, R. Automated analysis of security protocols with global state. *Journal of Computer Security 24*, 5 (2016), 583–616.

[90] Künnemann, R. Automated backward analysis of PKCS# 11 v2. 20. In *International Conference on Principles of Security and Trust* (2015), Springer, pp. 219–238.

[91] Kunz-Jacques, S., and Pointcheval, D. About the Security of MTI/C0 and MQV. In *International Conference on Security and Cryptography for Networks* (2006), Springer, pp. 156–172.

[92] Lowe, G. An Attack on the Needham- Schroeder Public- Key Authentication Protocol. *Information processing letters 56*, 3 (1995).

[93] Lowe, G. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems* (1996), Springer, pp. 147–166.

[94] Lowe, G. Casper: A compiler for the analysis of security protocols. In *Proceedings 10th Computer Security Foundations Workshop* (1997), IEEE, pp. 18–30.

[95] Lowe, G. A hierarchy of authentication specifications. In *Computer security foundations workshop, 1997. Proceedings., 10th* (1997), IEEE, pp. 31–43.

[96] Malladi, S., Alves-Foss, J., and Heckendorn, R. B. On preventing replay attacks on security protocols. Tech. rep., IDAHO UNIV MOSCOW DEPT OF COMPUTER SCIENCE, 2002.

[97] Mao, W., and Boyd, C. Towards formal analysis of security protocols. In *[1993] Proceedings Computer Security Foundations Workshop VI* (1993), IEEE, pp. 147–158.

[98] Marczak, B., Anstis, S., Crete-Nishihata, M., Scott-Railton, J., and Deibert, R. Stopping the press: New York Times journalist targeted by Saudi-linked Pegasus spyware operator. Tech. rep., 2020.

[99] Marczak, B., Scott-Railton, J., McKune, S., Abdul Razzak, B., and Deibert, R. HIDE AND SEEK: Tracking NSO Group's Pegasus Spyware to operations in 45 countries. Tech. rep., 2018.

[100] Marrero, W., Clarke, E., and Jha, S. Model checking for security protocols. Tech. rep., CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1997.

[101] Meadows, C. A model of computation for the NRL protocol analyzer. In *Proceedings The Computer Security Foundations Workshop VII* (1994), IEEE, pp. 84–89.

[102] Meier, S. *Advancing automated security protocol verification.* PhD thesis, ETH Zurich, 2013.

[103] Meier, S., Schmidt, B., Cremers, C., and Basin, D. The TAMARIN prover for the symbolic analysis of security protocols. In *International Conference on Computer Aided Verification* (2013), Springer, pp. 696–701.

[104] Meseguer, J. Conditional rewriting logic as a unified model of concurrency. *Theoretical computer science 96*, 1 (1992), 73–155.

[105] Moh, M., Pininti, S., Doddapaneni, S., and Moh, T.-S. Detecting web attacks using multi-stage log analysis. In *2016 IEEE 6th International Conference on Advanced Computing (IACC)* (2016), IEEE, pp. 733–738.

[106] Naik, N., and Jenkins, P. Discovering hackers by stealth: Predicting fingerprinting attacks on honeypot systems. In *2018 IEEE International Systems Engineering Symposium (ISSE)* (2018), IEEE, pp. 1–8.

[107] OCCRP. How Does Pegasus Work. `https://www.occrp.org/en/the-pegasus-project/how-does-pegasus-work`, July 2021. (Accessed on 19/07/2021).

[108] Otway, D., and Rees, O. Efficient and timely mutual authentication. *ACM SIGOPS Operating Systems Review 21*, 1 (1987), 8–10.

[109] Palombo, H. M., Zheng, H., and Ligatti, J. POSTER: Towards precise and automated verification of security protocols in Coq. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), pp. 2567–2569.

[110] Paulin-Mohring, C. Introduction to the coq proof-assistant for practical software verification. In *LASER Summer School on Software Engineering* (2011), Springer, pp. 45–95.

[111] Paulson, L. C. Proving properties of security protocols by induction. In *Proceedings 10th Computer Security Foundations Workshop* (1997), IEEE, pp. 70–83.

[112] Payer, M. HexPADS: a platform to detect "stealth" attacks. In *International Symposium on Engineering Secure Software and Systems* (2016), Springer, pp. 138–154.

[113] Rahli, V., Vukotic, I., Völp, M., and Esteves-Verissimo, P. Velisarios: Byzantine fault-tolerant protocols powered by Coq. In *European Symposium on Programming* (2018), Springer, pp. 619–650.

[114] Rescorla, E., and Dierks, T. The transport layer security (TLS) protocol version 1.3, 2018.

[115] Roscoe, A., Creese, S., Goldsmith, M., and Zakiuddin, I. The attacker in ubiquitous computing environments: Formalising the threat model, 2003.

[116] Rudd, E., Rozsa, A., Gunther, M., and Boult, T. A Survey of Stealth Malware: Attacks, Mitigation Measures, and Steps Toward Autonomous Open World Solutions. *IEEE Communications Surveys & Tutorials* (2016).

[117] Ryan, P., Schneider, S. A., Goldsmith, M., Lowe, G., and Roscoe, B. *The modelling and analysis of security protocols: the CSP approach.* Addison-Wesley Professional, 2001.

[118] Sahadevaiah, K., and PVGD, P. R. Impact of security attacks on a new security protocol for mobile ad hoc networks. *Network Protocols and Algorithms 3*, 4 (2011), 122–140.

[119] Sarjana, F. W., Yuliar Arif, T., Adriman, R., and Munadi, R. Simple Prevention of Advanced Stealth Man-in-The-Middle Attack in WPA2 Wi-Fi Networks. In *2019 International Conference on Electrical Engineering and Computer Science (ICECOS)* (2019), pp. 349–353.

[120] Sarkar, P. G., and Fitzgerald, S. Attacks on ssl a comprehensive study of beast, crime, time, breach, lucky 13 & rc4 biases. *Internet: https://www. isecpartners. com/media/106031/ssl_attacks_survey. pdf [June, 2014]* (2013).

[121] Scarfone, K., and Mell, P. Guide to intrusion detection and prevention systems (idps). Tech. rep., National Institute of Standards and Technology, 2012.

[122] Schmidt, B. *Formal analysis of key exchange protocols and physical protocols.* PhD thesis, ETH, Zurich, 2012.

[123] Schmidt, B., Meier, S., Cremers, C., and Basin, D. Automated analysis of Diffie-Hellman protocols and advanced security properties. In *2012 IEEE 25th Computer Security Foundations Symposium* (2012), IEEE, pp. 78–94.

[124] Shittu, R. O. *Mining intrusion detection alert logs to minimise false positives & gain attack insight.* PhD thesis, City University London, 2016.

[125] Shu, G., and Lee, D. Testing security properties of protocol implementations-a machine learning based approach. In *27th International Conference on Distributed Computing Systems (ICDCS'07)* (2007), IEEE, pp. 25–25.

[126] Singh, R. R. StealthCheck Tool Webpage: Python based automated Stealth Verification Tool using Tamarin-Prover (Source code, Manual and Examples). `http://people.du.ac.in/~rrsingh/StealthCheck/`. (Accessed on 14/08/2021).

[127] Singh, R. R., Moreira, J., Chothia, T., and Ryan, M. D. Tamarin prover models of the 802.11 4-way handshake Attacks and Security Properties (Source code and Proof Results). `http://people.du.ac.in/~rrsingh/wpa2models/`, 2020.

[128] Sirohi, P., Agarwal, A., and Tyagi, S. A comprehensive study on security attacks on ssl/tls protocol. In *2016 2nd International Conference on Next Generation Computing Technologies (NGCT)* (2016), IEEE, pp. 893–898.

[129] Song, D. X. Athena: a new efficient automatic checker for security protocol analysis. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop* (1999), IEEE, pp. 192–202.

[130] Song, J., Kim, H., and Park, S. Enhancing conformance testing using symbolic execution for network protocols. *IEEE Transactions on Reliability 64*, 3 (2015), 1024–1037.

[131] Stone, C. M., Chothia, T., and de Ruiter, J. Extending automated protocol state learning for the 802.11 4-way handshake. In *European Symposium on Research in Computer Security* (2018), Springer, pp. 325–345.

[132] Stubblefield, A., Ioannidis, J., and Rubin, A. D. A key recovery attack on the 802.11 b wired equivalent privacy protocol (WEP). *ACM transactions on information and system security (TISSEC) 7*, 2 (2004), 319–332.

[133] Tews, E., and Beck, M. Practical attacks against WEP and WPA. In *Proceedings of the second ACM conference on Wireless network security* (2009), pp. 79–86.

[134] Tisi, M., and Cheng, Z. Coqtl: An internal DSL for model transformation in COQ. In *International Conference on Theory and Practice of Model Transformations* (2018), Springer, pp. 142–156.

[135] van der Merwe, T. An Analysis of the Transport Layer Security Protocol, 2018.

[136] Vanhoef, M., and Piessens, F. Predicting, decrypting, and abusing WPA2/802.11 group keys. In *25th {USENIX} Security Symposium ({USENIX} Security 16)* (2016), pp. 673–688.

[137] Vanhoef, M., and Piessens, F. Key reinstallation attacks: Forcing nonce reuse in WPA2. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), pp. 1313–1328.

[138] Vigano, L. Automated security protocol analysis with the AVISPA tool. *Electronic Notes in Theoretical Computer Science 155* (2006), 61–86.

[139] Wagner, D., and Soto, P. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security* (2002), ACM, pp. 255–264.

[140] Wang, J., Zhang, J., and Zhang, H. Type flaw attacks and prevention in security protocols. In *Software Engineering, Artificial Intelligence, Networking,*

and Parallel/Distributed Computing, 2008. SNPD'08. Ninth ACIS International Conference on (2008), IEEE, pp. 340–343.

[141] Wesemeyer, S., Newton, C. J., Treharne, H., Chen, L., Sasse, R., and Whitefield, J. Formal Analysis and Implementation of a TPM 2.0-based Direct Anonymous Attestation Scheme. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security* (2020), pp. 784–798.

[142] Whitefield, J., Chen, L., Kargl, F., Paverd, A., Schneider, S., Treharne, H., and Wesemeyer, S. Formal analysis of V2X revocation protocols. In *International Workshop on Security and Trust Management* (2017), Springer, pp. 147–163.

[143] Whitefield, J., Chen, L., Sasse, R., Schneider, S., Treharne, H., and Wesemeyer, S. A symbolic analysis of ecc-based direct anonymous attestation. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)* (2019), IEEE, pp. 127–141.

[144] Yap, R. H., et al. Improving host-based IDS with argument abstraction to prevent mimicry attacks. In *International Workshop on Recent Advances in Intrusion Detection* (2005), Springer, pp. 146–164.

[145] Zhang, M., Wang, L., Jajodia, S., Singhal, A., and Albanese, M. Network diversity: a security metric for evaluating the resilience of networks against zero-day attacks. *IEEE Transactions on Information Forensics and Security 11*, 5 (2016), 1071–1086.

[146] Zlomislić, V., Fertalj, K., and Sruk, V. Denial of service attacks, defences and research challenges. *Cluster Computing 20*, 1 (2017), 661–671.

Appendices

Appendix One

Functions used in Def. 5.7

MergeList : $List \; \times \; List \rightarrow Set[List]$

$$MergeList(x : xs, y : ys) = \{x : MergeList(xs, y : ys), y : MergeList(x : xs, ys)\}$$

$$mergeList([\,], ys) = \{ys\}$$

$$mergeList(xs, [\,]) = \{xs\}$$

mergeSet : $Set[List] \; \times \; List \rightarrow Set[List]$

$$mergeSet(\{S1\} \cup S, L) = mergeList(S1, L) \cup mergeSet(S, L)$$

$$mergeSet(\{\}, L) = \{L\}$$

merge : $MultiSet[List] \; \rightarrow \; Set[List]$

$$merge(\{L\} \cup S) = mergeSet(merge(S), L)$$

$$merge(\{\}) = \{\}$$

Appendix Two

SAPiC Code for Attacks on 802.11 *4-way handshake*

B.1 Code for KRACK Attack of Figure 3.3

```
/*****************************************************************************
        AUTHORS : R R Singh, J Moreira, Tom Chothia, M. D. Ryan
        Attack against the 4-way handshake, when the supplicant when (victim) still
        accepts plaintext retransmissions of message 3 if a PTK is installed.
        Notation:
        pmk = Pairwise Master Key,
        ptk = Pairwise Transient Key, composed of:
                kck = Key Confirmation Key
                kek = Key Encryption Key
                tk = Temporal Key
        gtk = Group Temporal Key
        A_ = Events only in authenticator process
        S_ = Events only in supplicant process
*****************************************************************************/
theory Krack_fig4
begin
 builtins :
        symmetric-encryption,
        multiset
functions:
        true/0, CalcPtk/1,      mic/2,  kck/1,  kek/1,  tk/1,   verifyMic/3
```

```
equations:
        verifyMic(mic(m,k), m, k) = true


let Supplicant =
        ν ~S_id;        out(~S_id);
        !(
                out(⟨'AuthRequest', ~S_id⟩);
                in(⟨'AuthResponse', ~S_id, cs⟩);
                out(⟨'AssociationRequest', ~S_id, cs⟩);
                in(⟨'AssociationResponse', ~S_id, cs⟩);
                event S_HasPmk(~S_id, ~pmk);
                // Initial 4-way handshake
                ν ~tid;
                in(A_id);
                let pat_msg1_contents = ⟨'Header1', r, ANonce, cs⟩ in
                in(pat_msg1_contents);
                ν ~SNonce;
                let tptk = CalcPtk(⟨~pmk, ANonce, ~SNonce, A_id, ~S_id⟩) in
                event S_ComputesPtk(~S_id, ~tid, tptk);
                let msg2_contents = ⟨'Header2', r, ~SNonce, cs⟩ in
                out(⟨msg2_contents, mic(msg2_contents, ⟨cs, kck(tptk)⟩)⟩);
                let msg3_contents =
                    ⟨'Header3', r+'1', ANonce, cs, senc(gtk, ⟨cs, kek(tptk)⟩)⟩ in
                in(⟨msg3_contents, mic(msg3_contents, ⟨cs, kck(tptk)⟩)⟩);
                event S_ReceivesGtk(~S_id, ~tid, gtk);

                let ptk = tptk in
                event S_RunningPtk(~S_id, A_id, ptk);
                event S_RunningGtk(~S_id, A_id, ⟨ANonce, ~SNonce, gtk⟩);
                event S_RunningCs(~S_id, A_id, ⟨ANonce, ~SNonce, cs⟩);

                let msg4_contents = ⟨'Header4', r+'1', cs⟩ in
                out(⟨msg4_contents, mic(msg4_contents, ⟨cs, kck(tptk)⟩)⟩);

                event S_InstallsPtk(~S_id, ptk); // MLME-SETKEYS.request
                event S_InstallsGtk(~S_id, gtk); // MLME-SETKEYS.request

                ((
                        event S_CommitPtk(~S_id, A_id, ptk);
                        event S_CommitGtk(~S_id, A_id, ⟨ANonce, ~SNonce, gtk⟩);
                        event S_CommitCs(~S_id, A_id, ⟨ANonce, ~SNonce, cs⟩);
                        event S_Branch1()
        ) +
        (
            let msg3_rtx_contents =
              ⟨'Header3', r+'1'+'1', ANonce, cs, senc(gtk2, ⟨cs, kek(tptk)⟩)⟩ in
            in(⟨msg3_rtx_contents, mic(msg3_rtx_contents, ⟨cs, kck(tptk)⟩)⟩);

            let ptk = tptk in
            event S_RunningPtk(~S_id, A_id, ptk);
            event S_RunningGtk(~S_id, A_id, ⟨ANonce, ~SNonce, gtk⟩);
            event S_RunningCs(~S_id, A_id, ⟨ANonce, ~SNonce, cs⟩);
            let msg4_rtx_contents = ⟨'Header4', r+'1'+'1', cs⟩ in
            out(senc(⟨msg4_rtx_contents, mic(msg4_rtx_contents,
                                              ⟨cs, kck(tptk)⟩)⟩, tk(ptk)));

            event S_InstallsPtk(~S_id, ptk); // MLME-SETKEYS.request
            event S_InstallsGtk(~S_id, gtk2); // MLME-SETKEYS.request

            event S_CommitPtk(~S_id, A_id, ptk);
            event S_CommitGtk(~S_id, A_id, ⟨ANonce, ~SNonce, gtk⟩);
            event S_CommitCs(~S_id, A_id, ⟨ANonce, ~SNonce, cs⟩);
            event S_Branch2()
        )))
let Authenticator =
    ν ~A_id;        out(~A_id);
    !(     ν ~tid;
        in(⟨'AuthRequest', S_id⟩);
        out(⟨'AuthResponse', S_id, 'CCMP'⟩);

        let cs = 'CCMP' in
        in(⟨'AssociationRequest', S_id, cs⟩);
        out(⟨'AssociationResponse', S_id, cs⟩);
        event A_HasPmk(~A_id, ~pmk);

        // Initial 4-way handshake
        ν ~r;
        ν ~ANonce;
        in(S_id);
        let pat_msg1_contents = ⟨'Header1', ~r, ~ANonce, cs⟩ in
        out(pat_msg1_contents);
        let ptk = CalcPtk(⟨~pmk, ~ANonce, SNonce, ~A_id, S_id⟩) in
        let msg2_contents = ⟨'Header2', ~r, SNonce, cs⟩ in
        in(⟨msg2_contents, mic(msg2_contents, ⟨cs, kck(ptk)⟩)⟩);
        ν ~gtk;
        event A_GeneratesGtk(~gtk);
        event A_RunningPtk(~A_id, S_id, ptk);
        event A_RunningGtk(~A_id, S_id, ⟨~ANonce, SNonce, ~gtk⟩);
        event A_RunningCs(~A_id, S_id, ⟨~ANonce, SNonce, cs⟩);
        event A_InstallsGtk(~gtk); // MLME-SETKEYS.request

        let msg3_contents =
          ⟨'Header3', ~r+'1', ~ANonce, cs, senc(~gtk, ⟨cs, kek(ptk)⟩)⟩ in
        out(⟨msg3_contents, mic(msg3_contents, ⟨cs, kck(ptk)⟩)⟩);
        ((
            let msg4_contents = ⟨'Header4', ~r+'1', cs⟩ in
            in(⟨msg4_contents, mic(msg4_contents, ⟨cs, kck(ptk)⟩)⟩);
            event A_InstallsPtk(ptk); // MLME-SETKEYS.request
            event A_CommitPtk(~A_id, S_id, ptk);
            event A_CommitGtk(~A_id, S_id, ⟨~ANonce, SNonce, ~gtk⟩);
            event A_CommitCs(~A_id, S_id, ⟨~ANonce, SNonce, cs⟩);
            event A_Branch1()
        )+ (
            let msg3_rtx_contents =
              ⟨'Header3', ~r+'1'+'1', ~ANonce, cs, senc(~gtk, ⟨cs, kek(ptk)⟩)⟩ in
            out(⟨msg3_rtx_contents, mic(msg3_rtx_contents, ⟨cs, kck(ptk)⟩)⟩);

            // Msg4(r+1) could have been received here (ignored)
            let msg4_rtx_contents = ⟨'Header4', ~r+'1'+'1', cs⟩ in
            in(senc(⟨msg4_rtx_contents, mic(msg4_rtx_contents,
                                                  ⟨cs, kck(ptk)⟩)⟩, tk(ptk)));

            // Msg4(r+1) could have been received here (ignored)
            event A_InstallsPtk(ptk); // MLME-SETKEYS.request
            event A_CommitPtk(~A_id, S_id, ptk);
            event A_CommitGtk(~A_id, S_id, ⟨~ANonce, SNonce, ~gtk⟩);
            event A_CommitCs(~A_id, S_id, ⟨~ANonce, SNonce, cs⟩);
            event A_Branch2()
        )))
// Main process starts here
ν ~pmk; (!Supplicant | Authenticator)
==== omitted some lemmas=====
// The KRACK attack does not exist.
lemma NoKrackPtk:
    "All id ptk #i. S_InstallsPtk(id, ptk)@i ==>
```

```
                        not(Ex #j. S_InstallsPtk(id, ptk)@j & (j⟨i))"
// Security properties specified in [IEEE 802.11, Sec. 12.6.14]
// a) Confirm the existence of the PMK at the peer.
lemma ConfirmPmk:
all-traces
    "All id1 id2 pmk1 pmk2 #t1 #t2.
      A_HasPmk(id1, pmk1)@t1 & S_HasPmk(id2, pmk2)@t2 ==⟩ (pmk1=pmk2)"

// b) Ensure that the security association keys are fresh.
lemma FreshPtk:
all-traces
    "All id tid1 tid2 ptk #t1 #t2.
      S_ComputesPtk(id, tid1, ptk)@t1 & S_ComputesPtk(id, tid2, ptk)@t2 ==⟩ (tid1=tid2)"

lemma FreshGtk:
all-traces
    "All id tid1 tid2 gtk #t1 #t2.
      S_ReceivesGtk(id, tid1, gtk)@t1 & S_ReceivesGtk(id, tid2, gtk)@t2 ==⟩ (tid1=tid2)"

// c) Synchronize the installation of one or more temporal keys into the MAC.
lemma AgreementPtk:
all-traces
    "(All X Y ptk #i. A_CommitPtk(X, Y, ptk)@i ==⟩
        ((Ex #j. S_RunningPtk(Y, X, ptk)@j & (j⟨i))
        & not(Ex X2 Y2 #i2. A_CommitPtk(X2, Y2, ptk)@i2 & not(#i2=#i))
        ))
     & (All X Y ptk #i. S_CommitPtk(X, Y, ptk)@i ==⟩
        ((Ex #j. A_RunningPtk(Y, X, ptk)@j & (j⟨i))
        & not(Ex X2 Y2 #i2. S_CommitPtk(X2, Y2, ptk)@i2 & not(#i2=#i))
        ))"
lemma AgreementGtk:
all-traces
    "(All X Y gtk #i. A_CommitGtk(X, Y, gtk)@i ==⟩
        ((Ex #j. S_RunningGtk(Y, X, gtk)@j & (j⟨i))
        & not(Ex X2 Y2 #i2. A_CommitGtk(X2, Y2, gtk)@i2 & not(#i2=#i))
        ))
     & (All X Y gtk #i. S_CommitGtk(X, Y, gtk)@i ==⟩
        ((Ex #j. A_RunningGtk(Y, X, gtk)@j & (j⟨i))
        & not(Ex X2 Y2 #i2. S_CommitGtk(X2, Y2, gtk)@i2 & not(#i2=#i))
        ))"

lemma SecretPtk:
```

```
all-traces
        ”All id ptk #i. S_InstallsPtk(id, ptk)@i ==⟩ (not (Ex #j. K(ptk)@j))”
lemma SecretGtk:
all-traces
        ”All id gtk #i. S_InstallsGtk(id, gtk)@i ==⟩ (not (Ex #j. K(gtk)@j))”

// d) Transfer the GTK from the Authenticator to the Supplicant.
lemma SameGtk:
all-traces
        ”All S gtk #i. S_InstallsGtk(S, gtk)@i ==⟩ (Ex #j. A_GeneratesGtk(gtk)@j & (j⟨i))”
// e) Confirm the selection of cipher suites.
lemma AgreementCs:
all-traces
        ”(All X Y cs #i. A_CommitCs(X, Y, cs)@i ==⟩
                ((Ex #j. S_RunningCs(Y, X, cs)@j & (j⟨i))
                & not(Ex X2 Y2 #i2. A_CommitCs(X2, Y2, cs)@i2 & not(#i2=#i))
                ))
         &      (All X Y cs #i. S_CommitCs(X, Y, cs)@i ==⟩
                ((Ex #j. A_RunningCs(Y, X, cs)@j & (j⟨i))
                & not(Ex X2 Y2 #i2. S_CommitCs(X2, Y2, cs)@i2 & not(#i2=#i))
                ))”

// f) Security Property to capture KRACK attack
lemma NoPtkReuse:
all-traces
        ”All id ptk #i #j. S_InstallsPtk(id, ptk)@i & S_InstallsPtk(id, ptk)@j ==⟩ (#i=#j)”

lemma NoGtkReuse:
all-traces
        ”All id gtk #i #j. S_InstallsGtk(id, gtk)@i & S_InstallsGtk(id, gtk)@j ==⟩ (#i=#j)”

// Uncomment the following to enforce restrictions and verify the proposed as well as standard properties
/*
restriction RestNoPtkReuse:
        ”All id ptk #i #j. S_InstallsPtk(id, ptk)@i & S_InstallsPtk(id, ptk)@j ==⟩ (#i=#j)”
restriction RestNoGtkReuse:
        ”All id gtk #i #j. S_InstallsGtk(id, gtk)@i & S_InstallsGtk(id, gtk)@j ==⟩ (#i=#j)”
*/
end
```

Appendix Three

StealthCheck user manual

The *StealthCheck* tool, developed in Python, verifies the the stealthiness of attacks against the various combinations of logged parameters of participants as well as individual logs, e.g., while testing attack on Needham- Shroeder Public Key protocol with two participants, the tool will generate three output Tamarin files, one each using logs of I and R and third using combined logs of I and R.

Rules for TAMARIN file to be processed : In order to be verified for stealthiness, a model should follow the conventions listed below for rules/facts/action labels:

- Prefix logs to the action labels of all the rules of the form : LogRuleName(Pars(RuleName)),.....with parameters to be logged as required chosen from those being used in the rule.
- Rulename specification : String of the form ID (a letter) followed by SEQ (a number), where ID is for identifying participant ID uniformly used in model such as I, R, A, B etc.
- All protocol rules must be numbered 1,2,3,..,n and placed in a sequence in Tamarin file. This is to help the tool differentiate between protocol rules and auxiliary rules.
- All the logs must have distinct names of the form Log[RuleName]. Fact and action

label names must also be distinct.

- Comments should be avoided within the rules. If required, must be made either before the start of the rule or after the rule ends.
- ‘sid’ must not be used by any rule in the TAMARIN source. If present, rename ’sid’ to avoid confusion.
- None of the security lemmas, originally present in the model, should be modified to use any of the stealth logs added above.

StealthCheck **features :** The python based *StealthCheck* utility stealthcheck.py modifies the TAMARIN source by adding session IDs to premise, conclusions and stealth logs, stealth Logs in rules, if not present, with no parameters, and finally constructing and adding the two stealthiness restrictions : Correspondence and Uniqueness. Finally, it automatically verifies the stealthiness of an attack by verifying / falsifying the security lemmas present in the model.

Usage : Execute TAMARIN to check presence of any attack on the protocol. Execute ./stcheck_singlefile with this TAMARIN source file name or ./stcheck_directory with a directory name containing multiple such TAMARIN source file names.
e.g., to check one protocol, execute
$ ./stcheck_singlefile NamesMessages/densac_sym_cbc.spthy
and, to check all protocols in a single directory, execute
$ ./stcheck_directory NamesMessages

It is also possible for TAMARIN to not terminate as the *StealthCheck* utility tries to prove all the files in one go. In such cases, the models should be passed individually to TAMARIN to verify the stealthiness of attacks. Additionally, lemma proofs could also be attempted manually using *‘auto-sources’* feature. $ tamarin-prover –prove –auto-sources filename.spthy

Appendix Four

TAMARIN code for the case studies

D.1 TAMARIN Codes of NSPK protocol [55]

Original TAMARIN Code snippets annotated with Logs, containing Partner names and Nonces, given as input to the StealthCheck tool

```
theory NSPK3
begin
 builtins : asymmetric-encryption
// Public key infrastructure
rule Register_pk:
  [ Fr(~ltkA) ]
  --⟩
  [ !Ltk(A, ltkA),!Pk(A, pk(~ltkA)), Out(pk(~ltkA)) ]

rule Reveal_ltk:
  [ !Ltk(A, ltkA) ] --[ RevLtk(A)    ]-⟩ [ Out(ltkA) ]
rule I1:
  let m1 = aenc{'1', ~ni, I\}pkR
  in
    [ Fr(~ni) , !Pk(R,pkR)] − −[OUT_I_1(m1), LogI1(I,R, ni)]− > [Out(m1), St_I1(I,R, ~ni)]

rule R2:
  let m1 = aenc{'1', ni, I}pk(ltkR)
  m2 = aenc{'2', ni, ~nr}pkI
  in
```

```
    [ !Ltk(R,ltkR),In(m1),!Pk(I,pkI),Fr( nr)] − −[IN_R1_ni(ni,m1),OUT_R_1(m2),Running(I,R, ⟨'init',ni,~nr⟩)
   , LogR2(I,R,ni, nr)]− > [Out(m2),St_R1(R, I, ni, ~nr)]


rule I3:
  let m2 = aenc{'2', ni, nr}pk(ltkI)
      m3 = aenc{'3', nr}pkR
  in
   [ St_I1(I, R, ni), !Ltk(I, ltkI) , In( m2 )
   , !Pk(R, pkR)]
  --[ IN_I_2_nr( nr, m2), LogI3(I,R,ni,nr)
   , Commit (I, R, ⟨'init',ni,nr⟩)
   , Running(R, I, ⟨'resp',ni,nr⟩) ]-⟩
   [ Out( m3 ), Secrecy(I,R,nr), Secrecy(I,R,ni) ]


rule R4:
   [ St_R_1(R, I, ni, nr) , !Ltk(R, ltkR)
   , In( aenc{'3', nr}pk(ltkR) ) ]
  --[ Commit (R, I, ⟨'resp',ni,nr⟩), LogR4(I,R,ni,nr)
   ]-⟩
   [ Secrecy(R,I,nr) , Secrecy(R,I,ni)   ]


rule X5:
  [ Secrecy(A, B, m) ] --[ Secret(A, B, m), LogX5(A,B,m) ]-⟩ []
// Nonce secrecy from the perspective of both the initiator and the responder.
lemma nonce_secrecy:
    not(Ex A B s #i.
          Secret(A, B, s) @ i
        & (Ex #j. K(s) @ j)
        & not (Ex #r. RevLtk(A) @ r)& not (Ex #r. RevLtk(B) @ r)
      )

// Injective agreement from the perspective of both the initiator and the responder.
lemma injective_agree:
    All actor peer params #i. Commit(actor, peer, params) @ i
      ==⟩   (Ex #j. Running(actor, peer, params) @ j & j ⟨ i
              & not(Ex actor2 peer2 #i2.
                  Commit(actor2, peer2, params) @ i2 & not(#i = #i2)
            ) )
        | (Ex #r. RevLtk(actor) @ r) | (Ex #r. RevLtk(peer) @ r)
end
```

D.2 TAMARIN Code of NSPK modified by *StealthCheck*

Modified TAMARIN Code snippets by StealthCheck tool for stealthiness test based on combined logs of (I)nitiator and (R)esponder

```
theory NSPK3
begin
 builtins : asymmetric-encryption
rule  Register_pk:  [Fr(~ltkA)]  --⟩
                         [!Ltk(A, ltkA),!Pk(A,pk(~ltkA)),Out(pk(~ltkA))]
rule  Reveal_ltk :  [!Ltk(A,ltkA)]
                          --[RevLtk(A)]-⟩ [Out(ltkA)]
rule  I1 :  let m1 = aenc{'1', ~ni, I}pkR in[Fr( sid),Fr( ni),!Pk(R,pkR)]
           --[OUT\_I\_1(m1),LogI1(~sid,I,R,ni)]-⟩
               [Out(~sid),Out(m1),St_I1(I,R,~ni)]



rule  R2:  let m1 = aenc{'1', ni, I}pk(ltkR)
               m2 = aenc{'2', ni, ~nr}pkI in
       [In(sid),!Ltk(R,ltkR),In(m1),!Pk(I,pkI),Fr( nr)] − −[IN_R1_ni(ni,m1),OUT_R_1(m2),Running(I,R,⟨'init'
    ,ni,~nr⟩),
               LogR2(sid,I,R,ni, nr)]− > [Out(m2),St_R1(R,I,ni,~nr)]



rule  I3:  let m2 = aenc{'2', ni, nr}pk(ltkI)
               m3 = aenc{'3', nr}pkR in
       [In(sid),St\_I1(I,R,ni),!Ltk(I, ltkI),In(m2),!Pk(R,pkR)]
           --[IN_I_2_nr(nr,m2),LogI3(sid,I,R,ni,nr),
               Commit(I,R,⟨'init',ni,nr⟩),
               Running(R,I,⟨'resp',ni,nr⟩)]-⟩
                   [Out(m3),Secrecy(I,R,nr),Secrecy(I,R,ni)]



rule  R4:     [In(sid),St_R_1(R,I,ni,nr),!Ltk(R,ltkR),In(aenc{'3',nr}pk(ltkR))]
               --[Commit(R,I,⟨'resp',ni,nr⟩),LogR4(sid,I,R,ni,nr)]-⟩
                   [Secrecy(R,I,nr),Secrecy(R,I,ni)]
rule  X5:  [In(sid),Secrecy(A,B,m)] --[Secret(A,B,m),LogX5(sid,A,B)]-⟩ []
```

```
lemma  nonce_secrecy:
        ” not(    Ex A B s #i. Secret(A, B, s)  @ i
                  & (Ex #j. K(s) @ j) & not (Ex #r. RevLtk(A) @ r)
                  & not (Ex #r. RevLtk(B) @ r)
                  )”

lemma  injective_agree :
”    All  actor  peer  params #i.
        Commit(actor, peer, params) @ i        ==⟩
           (Ex #j. Running(actor, peer, params) @ j & j ⟨ i
              & not(Ex actor2 peer2 #i2.
                  Commit(actor2, peer2, params) @ i2 & not(#i = #i2)))
              |  (Ex #r. RevLtk(actor) @ r) | (Ex #r. RevLtk(peer)  @ r) ”

restriction  Unique_ni_forLogI1andLogI1:
”All  I2  sid2  ni  I1  R1 sid1  R2 #i #j. LogI1(sid1,I1,R1,ni )  @i & LogI1(sid2,I2,R2,ni )  @j ==⟩ (sid1 = sid2)”

restriction  Unique_ni_forLogI1andLogR2:
”All  I2  nr2 sid2  ni  I1  R1 sid1  R2 #i #j. LogI1(sid1,I1,R1,ni )  @i & LogR2(sid2,I2,R2,ni,nr2 ) @j ==⟩ (sid1 = sid2)”

restriction  Unique_ni_forLogI1andLogR2_1:
”All  I2  ni  sid2  ni2  I1  R1 sid1  R2 #i #j. LogI1(sid1,I1,R1,ni )  @i & LogR2(sid2,I2,R2,ni2,ni ) @j ==⟩ (sid1 = sid2)”

restriction  Unique_ni_forLogI1andLogI3:
”All  I2  nr2 sid2  ni  I1  R1 sid1  R2 #i #j. LogI1(sid1,I1,R1,ni )  @i & LogI3(sid2,I2,R2,ni,nr2 )  @j ==⟩ (sid1 = sid2)”

restriction  Unique_ni_forLogI1andLogI3_1:
”All  I2  sid2  ni2  I1  R1 sid1  ni  R2 #i #j. LogI1(sid1,I1,R1,ni )  @i & LogI3(sid2,I2,R2,ni2,ni  )  @j ==⟩ (sid1 = sid2)”

restriction  Unique_ni_forLogI1andLogR4:
”All  I2  nr2 sid2  ni  I1  R1 sid1  R2 #i #j. LogI1(sid1,I1,R1,ni )  @i & LogR4(sid2,I2,R2,ni,nr2 ) @j ==⟩ (sid1 = sid2)”

restriction  Unique_ni_forLogI1andLogR4_1:
”All  I2  sid2  ni2  I1  R1 sid1  ni  R2 #i #j. LogI1(sid1,I1,R1,ni )  @i & LogR4(sid2,I2,R2,ni2,ni ) @j ==⟩ (sid1 = sid2)”

restriction  Unique_nr_forLogR2andLogR2:
”All  I2  nr2 sid2  ni  I1  R1 sid1  nr1  R2 #i #j. LogR2(sid1,I1,R1,ni,nr1 ) @i & LogR2(sid2,I2,R2,ni,nr2 ) @j
==⟩ (sid1 = sid2)”

restriction  Unique_nr_forLogR2andLogR2_1:
”All  I2  ni  sid2  ni2  I1  R1 sid1  nr1  R2 #i #j. LogR2(sid1,I1,R1,ni,nr1 ) @i & LogR2(sid2,I2,R2,ni2,ni ) @j
==⟩ (sid1 = sid2)”
```

```
restriction Unique_nr_forLogR2andLogR2_2:
”All I2 nr2 sid2 nr I1 R1 sid1 ni1 R2 #i #j. LogR2(sid1,I1,R1,ni1,nr ) @i & LogR2(sid2,I2,R2,nr,nr2 ) @j
==⟩ (sid1 = sid2)”

restriction Unique_nr_forLogR2andLogR2_3:
”All I2 nr sid2 ni2 I1 R1 sid1 ni1 R2 #i #j. LogR2(sid1,I1,R1,ni1,nr ) @i & LogR2(sid2,I2,R2,ni2,nr ) @j
==⟩ (sid1 = sid2)”

restriction Unique_nr_forLogR2andLogI3:
”All I2 nr2 sid2 ni I1 R1 sid1 nr1 R2 #i #j. LogR2(sid1,I1,R1,ni,nr1 ) @i & LogI3(sid2,I2,R2,ni,nr2 ) @j
==⟩ (sid1 = sid2)”

restriction Unique_nr_forLogR2andLogI3_1:
”All I2 ni sid2 ni2 I1 R1 sid1 nr1 R2 #i #j. LogR2(sid1,I1,R1,ni,nr1 ) @i & LogI3(sid2,I2,R2,ni2,ni ) @j
==⟩ (sid1 = sid2)”

restriction Unique_nr_forLogR2andLogI3_2:
”All I2 nr2 sid2 nr I1 R1 sid1 ni1 R2 #i #j. LogR2(sid1,I1,R1,ni1,nr ) @i & LogI3(sid2,I2,R2,nr,nr2 ) @j
==⟩ (sid1 = sid2)”

restriction Unique_nr_forLogR2andLogI3_3:
”All I2 nr sid2 ni2 I1 R1 sid1 ni1 R2 #i #j. LogR2(sid1,I1,R1,ni1,nr ) @i & LogI3(sid2,I2,R2,ni2,nr ) @j
==⟩ (sid1 = sid2)”

restriction Unique_nr_forLogR2andLogR4:
”All I2 nr2 sid2 ni I1 R1 sid1 nr1 R2 #i #j. LogR2(sid1,I1,R1,ni,nr1 ) @i & LogR4(sid2,I2,R2,ni,nr2 ) @j
==⟩ (sid1 = sid2)”

restriction Unique_nr_forLogR2andLogR4_1:
”All I2 ni sid2 ni2 I1 R1 sid1 nr1 R2 #i #j. LogR2(sid1,I1,R1,ni,nr1 ) @i & LogR4(sid2,I2,R2,ni2,ni ) @j
==⟩ (sid1 = sid2)”

restriction Unique_nr_forLogR2andLogR4_2:
”All I2 nr2 sid2 nr I1 R1 sid1 ni1 R2 #i #j. LogR2(sid1,I1,R1,ni1,nr ) @i & LogR4(sid2,I2,R2,nr,nr2 ) @j
==⟩ (sid1 = sid2)”

restriction Unique_nr_forLogR2andLogR4_3:
”All I2 nr sid2 ni2 I1 R1 sid1 ni1 R2 #i #j. LogR2(sid1,I1,R1,ni1,nr ) @i & LogR4(sid2,I2,R2,ni2,nr ) @j
==⟩ (sid1 = sid2)”

restriction Unique_nr_forLogR2andLogR4:
```

```
”All sid1 I1 nr ni1 I2 R2 sid2 R1 ni2 #i #j. LogR2(sid1,I1,R1,ni1,nr ) @i & LogR4(sid2,I2,R2,ni2,nr ) @j ==⟩ (sid1 =
    sid2)”

restriction EveryALBeforeLogR4_1:
”All sid I R ni nr #i1. LogR4(sid,I,R,ni,nr )@i1 ==⟩
( Ex #i2. LogI3(sid,I,R,ni,nr ) @i2 & (i2 ⟨ i1)
   & Ex #i3. LogR2(sid,I,R,ni,nr ) @i3 & (i3 ⟨ i2)
   & Ex #i4. LogI1(sid,I,R,ni ) @i4 & (i4 ⟨ i3)
&
    ( All I1 R1 ni1 nr1 #i5. LogR4(sid,I1,R1,ni1,nr1 ) @i5 ==⟩ (#i5 = #i1))
    &
        ( All I2 R2 ni2 nr2 #i6. LogI3(sid,I2,R2,ni2,nr2 ) @i6 ==⟩ (#i6 = #i2))
        &
        ( All I3 R3 ni3 nr3 #i7. LogR2(sid,I3,R3,ni3,nr3 ) @i7 ==⟩ (#i7 = #i3))
        &
        ( All I4 R4 ni4 #i8. LogI1(sid,I4,R4,ni4 ) @i8 ==⟩ (#i8 = #i4)) )”

restriction EveryALBeforeLogI3_2:
”All sid I R ni nr #i1. LogI3(sid,I,R,ni,nr )@i1 ==⟩
( Ex #i2. LogR2(sid,I,R,ni,nr ) @i2 & (i2 ⟨ i1)
   & Ex #i3. LogI1(sid,I,R,ni ) @i3 & (i3 ⟨ i2)
   & (
        All I1 R1 ni1 nr1 #i4. LogI3(sid,I1,R1,ni1,nr1 ) @i4 ==⟩ (#i4 = #i1)) &
        ( All I2 R2 ni2 nr2 #i5. LogR2(sid,I2,R2,ni2,nr2 ) @i5 ==⟩ (#i5 = #i2))
        &
        ( All I3 R3 ni3 #i6. LogI1(sid,I3,R3,ni3 ) @i6 ==⟩ (#i6 = #i3)) )”

restriction EveryALBeforeLogR2_3:
”All sid I R ni nr #i1. LogR2(sid,I,R,ni,nr )@i1 ==⟩
( Ex #i2. LogI1(sid,I,R,ni ) @i2 & (i2 ⟨ i1)
   &
        ( All I1 R1 ni1 nr1 #i3. LogR2(sid,I1,R1,ni1,nr1 ) @i3 ==⟩ (#i3 = #i1))
        &
        ( All I2 R2 ni2 #i4. LogI1(sid,I2,R2,ni2 ) @i4 ==⟩ (#i4 = #i2)) )”
end
```