Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Adhirath Kabra

Automatic Security Analysis of Lightweight Authentication and Key Exchange Protocols

Master's thesis in Security and Cloud Computing (SECCLO)
Supervisor: Colin Boyd
Co-supervisor: Lise Millerjord, Chris Brzuska
June 2022

NTNU
Norwegian University of
Science and Technology

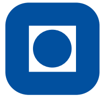
Adhirath Kabra

Automatic Security Analysis of Lightweight Authentication and Key Exchange Protocols

Master’s thesis in Security and Cloud Computing (SECCLO)
Supervisor: Colin Boyd
Co-supervisor: Lise Millerjord, Chris Brzuska
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Norwegian University of
Science and Technology



NTNU – Trondheim
Norwegian University of
Science and Technology

Automatic Security Analysis of Lightweight Authentication and Key Exchange Protocols

Adhirath Kabra

Submission date: June 2022
Supervisor: Colin Boyd, NTNU, IIK
Co-supervisor: Lise Millerjord, NTNU, IIK & Chris Brzuska, Aalto University

Norwegian University of Science and Technology
Department of Information Security and Communication Technology

Title: Automatic Security Analysis of Lightweight Authentication and Key Exchange Protocols

Student: Adhirath Kabra

Problem description:

Perfect forward secrecy (PFS) is integral in modern-day authenticated key exchange (AKE) protocols. Traditional methods to achieve forward secrecy, however, involves computationally heavy public key cryptography, rendering them useless for constrained devices. Consequently, lightweight AKE protocols have been proposed to provide PFS with only symmetric primitives. Formal verification of these protocols will reliably confirm claimed security guarantees prior to deployment in constrained environments, which are particularly vulnerable to compromise.

The thesis aims to formally analyze security properties of such recently published and standardized AKE protocols employing symmetric key cryptography, using the automatic verification tool Tamarin Prover. With a specific focus on PFS, we are also interested in comparing the efficiency and verifying the provision of essential security notions typically expected of an AKE protocol, such as secrecy and authentication.

Date approved: 2022-03-01

Responsible professor: Colin Boyd

Supervisor(s): Lise Millerjord

Abstract

Perfect Forward Secrecy (PFS) is vital in contemporary authenticated key exchange (AKE) protocols. Typically attained using public key cryptography, achieving forward secrecy is infeasible for communication in resource-constrained environment. Consequently, lightweight AKE protocols that offer PFS with only symmetric primitives are recently proposed. Formal analysis of these protocols can help in providing credibility prior to their deployment, and also reliably serve as a universally understood proof for the corresponding security properties.

To this end, we perform the formal verification of the SAKE protocol using an automatic verification tool Tamarin. In addition to proving the claimed security properties of session key secrecy, authentication and forward security, through Tamarin analysis, we also illustrate an attack that breaks the synchronization robustness of the protocol, resulting in de-synchronization of the internal states of the communicating parties. Furthermore, we have cogently presented a comprehensive guide to using Tamarin as a verification tool, detailing its key features, software usage and the foundational logic behind its analysis.

Contents

List of Figures **v**

List of Algorithms **vii**

1 Introduction **1**

1.1 Motivation 1

1.2 Research Focus 2

1.2.1 Security Properties of Interest 2

1.3 Contribution 3

2 Background **5**

2.1 Key Exchange Protocols 5

2.2 Modeling Protocols 6

2.2.1 Formal Verification Today 7

2.3 Achieving PFS with Symmetric Primitives 8

3 Tamarin Prover: The Essentials **11**

3.1 Tamarin Logic 11

3.2 Building Blocks 12

3.2.1 Variables, functions and equations 12

3.2.2 Rules and restrictions 14

3.2.3 Security properties and lemmas 17

3.3 Tamarin Analysis 18

4 Formal Analysis of SAKE **25**

4.1 SAKE: Review 25

4.1.1 Design Ideas 25

4.1.2 Protocol Flow 26

4.2 Modelling in Tamarin 27

4.2.1 Rules 27

4.2.2 Lemmas 32

4.2.3 Discussion 39

5 Conclusion and Future Work 41
5.1 Remarks 41
5.2 Achieving Full Synchronization Robustness 41

References 43

List of Figures

2.1 DH key exchange 6

3.1 An artificial DH protocol 13
3.2 Tamarin interactive mode landing page 19
3.3 Model loaded from `Example` theory file 20
3.4 Computed sources of the fact `AgStA3` 21
3.5 Methods to prove a lemma 22
3.6 Counter-example trace found 22
3.7 Counter-example attack diagram 23
3.8 Session key secrecy holds *wrt.* B 24

4.1 Key evolution and session key derivation in SAKE (adapted from [ACF20]) 26
4.2 SAKE protocol flow for an arbitrary epoch j (adapted from [ACF20]) . 28
4.3 Knowledge of initiator and responder before the target protocol run . . 29
4.4 Knowledge of initiator and responder after the target protocol run . . . 33
4.5 Tamarin proof: overview of the executability lemma 34
4.6 Tamarin proof: session key secrecy lemma *wrt.* the initiator 36
4.7 Tamarin proof: session key secrecy lemma *wrt.* the responder 36
4.8 Tamarin proof: session key PFS lemma *wrt.* the initiator 38
4.9 Tamarin proof: session key PFS lemma *wrt.* the responder 39
4.10 Tamarin proof: existence of an attack against synchronization robustness 40

List of Algorithms

3.1 Initialization, functions and equational theory 13
3.2 Public Key Infrastructure for DH-based protocols 15
3.3 Setup initial key knowledge with participants 15
3.4 First message: *A* sends the public key and corresponding MAC . . . 15
3.5 Second message: *B* processes `m1` and derives the session key 16
3.6 *A* receives `m2` and derives the session key 17
3.7 Session key secrecy lemmas 18
3.8 An example deconstruction rule 19
4.1 Functions and equational theory 29
4.2 Producing facts that represent knowledge of participants 29
4.3 First Message: initiator sends a nonce 30
4.4 Second Message: responder sends a nonce and divulges its internal state 30
4.5 Initiator evolves and derives a session key 31
4.6 Responder evolves and derives a session key 31
4.7 Initiator accepts the target run 32
4.8 Responder accepts the target run 32
4.9 Executablity check for the model 33
4.10 Session key secrecy *wrt.* the initiator and the responder 35
4.11 Session key PFS *wrt.* the initiator and the responder 37
4.12 Synchronization robustness of the SAKE protocol 38

Chapter 1
Introduction

1.1 Motivation

Internet of Things (IoT) influences many facets of our everyday life. The use cases of these proliferating objects include, but are not limited to, healthcare, industries, wearables, smart homes and critical infrastructure, offering us automatization, increased accessibility and convenience to achieve desired functionalities [AIM10]. With such a diverse set of application scenarios, it is vital to safeguard the data transmitted by these devices. Security flaws can result in breach of privacy, breakdown of production lines, or even escalate to disruption of economy. Consequently, the scientific community has seen a significant increase in research focus for IoT security.

IoT devices are, however, characterized by limited power, computational and storage resources. As a result, new and application-specific protocols need to be formulated, or present-day Internet protocols securing communications require alteration to accommodate these constraints. Amongst these cryptographic protocols, key exchange schemes form the basis of a trusted exchange of information. The primary goal of a key exchange protocol is to share a common secret key between the participants communicating over an untrusted network. Such protocols essentially allow symmetric-key cryptography to be used for secure transmission of data between the parties after the key agreement, even in absence of prior shared knowledge.

Over time, employment of key exchange protocols in diverse application scenarios and increasing adversarial capabilities resulted in evolution of the required security guarantees. For the IoT scenario, achieving these guarantees also becomes more complicated by the need to maintain compatibility with a vast number of vendors and service providers, as well as change in bandwidth, energy availability and deployment environment [KHHJ17]. Out of these requirements, one established security goal in modern key exchange schemes is the authentication of communicating parties, signifying that the key agreement indeed completed with the intended partners. Consequently, many Authenticated Key Exchange (AKE) protocols with varied

application purposes have been lately proposed for constrained environment [HS20]. Ever since, public review and security analysis of these protocols has proved to be crucially important in their adoption and standardization.

1.2 Research Focus

As the protocols get complex, and niche, application-specific security properties are desired, manual analysis turns out to be impractically tedious and error-prone. This is worsened by frequent changes in implementation details and adversarial models [CHSvdM16]. Therefore, computer-aided verification of protocols has become more important in the last decade. Many tools are developed to perform an automatic formal security analysis using mathematical and logical methods, resulting in flaw detection or security affirmation for widely deployed protocols. Some prominent analysis tools, including AVISPA [ABB+05], ProVerif [Bla16] and Tamarin [BCDS17], and their usability in the context of lightweight AKE protocols are discussed in [HS20; KHHJ17], eventually demonstrating the rationale to opt for Tamarin Prover for formal verification of the protocols in this work. Tamarin, as a tool, not only offers a more expressive modeling language as compared to many of its counterparts, but also provides a comprehensive set of analysis features along with guaranteeing termination of the analysis (Chapter 3). Formal analysis of these AKE protocols using Tamarin will help in providing credibility prior to their deployment, and also serve as a universally understood proof for the corresponding security properties.

1.2.1 Security Properties of Interest

Now, we will brief on the major security properties that we intend to verify in an AKE protocol. Considering a typical Dolev-Yao adversary [DY83] that fully controls the network, all protocols must fulfil two properties to secure the communication. First, confidentiality, which implies that protected information is not made available to unauthorized entities. For an AKE protocol, it takes the form of session key secrecy, where an adversary is not able to learn or gain unauthorized access of the session key agreed between the communicating parties. Broadly, there are two notions of secrecy - syntactic secrecy, where it is only required that the adversary is not able to learn the exact content of the protected data (by bits), and strong secrecy, which in fact ensures that the adversary is not able to detect the change in message content through the obtained ciphertext [Bla12]. Second important property, as defined earlier, is *authentication*. By and large, it implies that the communicating parties agree upon the same protocol parameters.

Today, *perfect forward secrecy* (PFS) has burgeoning importance and is a standard requirement in modern-day AKE protocols, accrediting to the ever-increasing adversarial storage and computational capabilities. PFS signifies that revelation

of long-term secrets should not divulge information exchanged in the past. This is attained by the renewal of key materials used to derive the shared session keys. A salient property, much of our interest, called *synchronization robustness* is proposed and defined in [BDdK+21]. The property proves to be cardinal in asserting correctness of AKE protocols based on only symmetric-key cryptography. We will delve more into synchronization robustness in Chapter 2. Another notable security property is *availability*, which concerns an adversary aiming to exhaust resources or disrupt the communication. For IoT, such exhaustion of constrained resources results in cryptographic *Denial-of-Service* (DoS). However, modeling availability for formal analysis is challenging and many tools lack support to represent the DoS [KHHJ17].

1.3 Contribution

The thesis seeks to demonstrate the advantage of and stimulate the use of formal analysis for the verification of AKE protocols. In this quest, we formally analyze a lightweight AKE protocol called Symmetric-key Authenticated Key Exchange (SAKE) [ACF20], using the automatic verification tool Tamarin. By doing so, we not only verify the claimed security properties of the protocol, *viz.*, session key secrecy, authentication and forward security, but also find an attack that breaks the property of synchronization robustness. The analysis, being one of the first works targeting AKE protocols that provide PFS with symmetric-key cryptography, also paves a way for future computer-aided verifications of protocols with similar security goals. In addition, with the help of an artificial example protocol, we present a cogent account of modeling in Tamarin, with appropriate focus on the operation and key features of the tool, as well as fundamental reasoning behind its analysis.

The remainder of this thesis is divided into four chapters. Chapter 2 presents a logical overview of the research area, along with delineation of the security properties used later in the work. Chapter 3 functions as a step-by-step guide to the usage and features of the formal analysis tool Tamarin. In Chapter 4, we use Tamarin to analyse the SAKE protocol [ACF20], and verify the claimed security guarantees. Finally, we briefly touch upon other symmetric-key schemes of interest providing PFS, and conclude with the future scope of our analysis in Chapter 5.

Chapter 2
Background

In this chapter, we first walk through the major milestones in the design of key exchange protocols (Section 2.1). Then, in Section 2.2, we introduce the two approaches in formal analysis to model security protocols, as well as the current extent of their usage. Finally, Section 2.3 illustrates the problem in achieving PFS with symmetric-key cryptography, and our focus on the verification of such protocols.

2.1 Key Exchange Protocols

Key agreement has been a subject of scientific research for over four decades after Diffie and Hellman presented a method [DH76] to securely transmit keying information over adversarially controlled network. In a Diffie-Hellman (DH) exchange, each participant knows or generates a private exponent, which is used for session key derivation. As shown in Figure 2.1, the adversary can only access the public components of the keys through the transcript of the protocol. Owing to the discrete logarithm problem, it is computationally infeasible to compute the private exponent $\mathtt{x}$ (respectively, $\mathtt{y}$) from the public key g^{x} (respectively, g^{y}). On the other hand, a simple exponentiation can be used to compute g^{xy} by either of the communicating parties. Note that the shown exchange is not authenticated and cannot be considered as an AKE protocol.

The earliest attempts to design an AKE protocol, both symmetric and asymmetric-key based, were demonstrated by Needham and Schroeder in 1978 [NS78]. The symmetric-key protocol availed a trusted third party that generates and establishes shared secrets between the communicating parties. However, a flaw in the freshness of the keying message resulted in impersonation of the initiator at a later point of time in case the previous session key is leaked [DS81].

Providing entity authentication is necessary to gain confidence in the derived session key. The introduction of *interleaving attacks* by [BGH+92] led to weaknesses in many the then existing AKE protocols. On similar lines, the concept of *matching protocol runs* is presented in [DvOW92], where the authors propose a form of

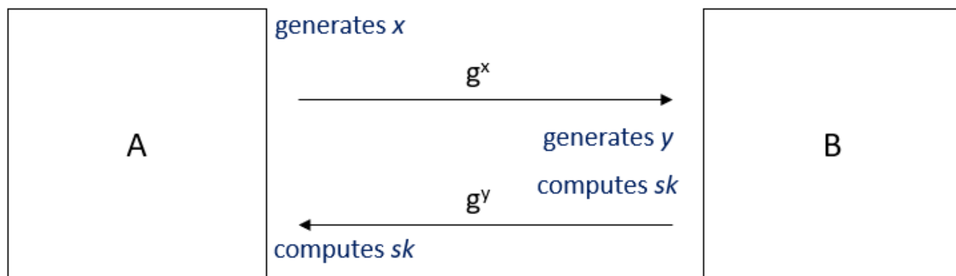


Figure 2.1: DH key exchange

authenticated Diffie-Hellman (DH) key exchange, concluding that authentication and key exchange must be linked to avoid impersonation attacks. Unfortunately, there was still an absence of formalism in the definition of these security notions.

By that time, the idea of *provable security* was already being used to standardize the security for cryptographic primitives such as probabilistic encryption and digital signatures. In the wake of substantially large number of flawed protocols, Bellare and Rogaway [BR94] were the first to apply this complexity-theoretic approach to model entity authentication and key distribution. In this formalization, not only did they consider a practical adversary controlling the network, but also incorporated the possibility of participants being engaged in concurrent sessions. The model laid the foundation for adversarial capabilities in modern-day formal analysis tools. While [BR94] focused on the symmetric-key case, extensions of this modeling were also proposed later for the public-key setting.

Subsequently, many other formalisms for the analysis of AKE protocols were suggested, such as [CK01], that intends to improve upon previously presented definitions, or [BAN90], that posits a whole new “logic-based” approach to prove correctness of the protocols. Amongst other standard security goals of session key secrecy and entity authentication, PFS gradually gained importance in many application scenarios. Nevertheless, until recently, fulfilling forward security was only modeled using asymmetric primitives, such as ephemeral public keys.

2.2 Modeling Protocols

The state-of-the-art formal analysis tools and their usage in the scientific community were reviewed in the project preceding this thesis [Kab21]. The findings from the project report are presented again in this section as follows.

In order to prove correctness, security protocols can be modeled in two ways. First, a *symbolic model*, also called Dolev-Yao model [DY83], which considers cryptographic primitives as black boxes. It realizes perfect cryptography in the sense that equations can model algebraic properties of the primitives but only these equalities hold [Bla12]. The verification aims to find contradiction to queries (desired security properties) and can be automated. The security properties must hold for all states in the state space of a model. Some of the notable tools operating in the symbolic model are AVISPA [ABB+05], ProVerif [Bla16], Tamarin [BCDS17] and Verifpal [KNT20].

Another way to approach the verification is using a *computational model*. Intuitively, it is used for manual proofs where a security parameter governs the key length, as well as runtime and success probability of a probabilistic polynomial-time (PPT) adversary. Furthermore, the primitives are functions of bitstrings [Bla12]. In practice, this model generates game-based proofs which might require human guidance [HS20]. Unlike the symbolic model, equalities other than the equations used to model the primitives may hold (with a negligible probability), and therefore, this model is more realistic. Although most tools verify protocols in the symbolic model, EasyCrypt [BDG+14] is a representative example of tools relying on the computational model. Unfortunately, no model can encompass all the possible attacker capabilities.

2.2.1 Formal Verification Today

Evidently, it is easier to analyze a protocol in the symbolic model than in the computational model. However, the state space of a model explodes to infinity due to the unbounded nature of message size and number of protocol runs (sessions) interleaved under an active attacker [Bla12]. Although limiting the state space to be explored mitigates this problem, it can only find attacks in the considered space but not disprove existence of the attacks. Constraint-Logic-based Attack Searcher (CL-AtSe) [CV01] and On-the-Fly Model Checker (OFMC) [BMV03] are some of the example tools using bounded number of sessions to verify protocols. Tamarin, on the other hand, considers verification under unbounded parallel sessions.

A substantial number of the recently proposed protocols have been analyzed mainly using two tools, namely Tamarin and ProVerif, which have proved to be efficacious verifiers in various facets of IoT. Novel 3-factor authentication schemes for Wireless Sensor Networks (WSNs) deployed in healthcare sector are verified for their security features under passive and active attackers [WLX+21; AA20; AIAA21]. The security of crucial protocols designed by Internet Engineering Task Force (IETF) for constrained devices has also been proven by multiple researchers. For instance, the lightweight AKE protocol Ephemeral Diffie-Hellman Over COSE (EDHOC) [SMP21] has been formally shown to provide session key secrecy, authentication and PFS using ProVerif [BSGS18]. Some vulnerabilities in older versions of EDHOC have also

been found through formal analysis with Tamarin [NSB21].

Furthermore, critical infrastructure has availed computer-aided verification to assert security of the protocols deployed in many of its sectors. ProVerif has been used to analyze lightweight authentication protocols for smart grids [MCN+18]. Amongst important low-power wireless standards, Li *et al.* [LPH20] analyzed ZigBee 1.0 and ZigBee 3.0 using Tamarin, showing vulnerabilities in the former as well as their absence in the newer specification. Likewise, an NFC-based mobile payment protocol is presented and verified for a secure end-to-end communication with Tamarin [BS19]. Kim *et al.* [KHHJ17] analyzed a handful of standard-based protocols like SigFox and Long Range alliance (LoRa) with Tamarin, but with a focus on the challenges to model DoS attacks and related countermeasures. However, formal verification and comparison of AKE protocols pertaining to IoT is not thoroughly investigated.

2.3 Achieving PFS with Symmetric Primitives

As emphasized previously, achieving PFS is a vital security goal in contemporary AKE protocols. This is aggravated by various deployment scenarios for IoT that make them susceptible to physical compromise. PFS is traditionally achieved using public key cryptography, for instance with Ephemeral Diffie-Hellman (DHE), which uses fresh private exponents generated by the communicating parties during every protocol run. Constrained devices, however, characterized by low memory and processing capacities, cannot support such computationally heavy schemes [GMS15]. Therefore, lightweight protocols that provide PFS with only symmetric primitives are designed.

One of the only approaches with symmetric key setting is the evolution of shared keys. More precisely, new long-term keys (LTKs) are derived using existing long-term keys, which are in turn used by the communicating parties to derive session keys. This updation occurs during every protocol run such that an adversary cannot use the new LTKs to efficiently compute prior keys. Unfortunately, this may lead to a situation where only one of the parties updated their keys, stemming from various reasons including unavailability and concurrency of sessions at an endpoint. This leads to yet another requirement of synchronization between the two parties.

Fundamentally, if two parties involved in a session end up with different LTKs after the run, their internal states are said to be desynchronized, and the derived session keys will not be symmetric. To break PFS trivially, an adversary may corrupt the party with older LTKs and derive session key corresponding to the information exchanged in past *wrt.* to the other party. Therefore, it is of utmost importance that LTKs are evolved and synchronized before either of the parties accepts the session and derive session keys. This is captured in the notion of synchronization robustness [BDdK+21]. Essentially, a protocol fulfilling this property ensures that

the participants are able to re-synchronize their internal states on completion of the protocol run, even in presence of an active adversary executing arbitrary number of parallel protocol sessions. Hence, synchronization robustness is necessary to achieve correctness of a key evolution protocol under stronger security notions.

One such protocol that uses a shrewd resynchronization algorithm to provide PFS with symmetric keys is Symmetric-Key Authenticated Key Exchange (SAKE) [ACF20]. While we are interested in verifying the claimed security properties, the protocol and its formal analysis is covered in detail in Chapter 4. Another set of protocols with similar security goals are provided in [BDdK+21]. In addition to defining the property of synchronization robustness, the authors set forth two types of protocols, linear and non-linear key evolving schemes. While the former follow the classic "derive-then-evolve" technique, much like their counterparts, they only achieve weaker form of synchronization robustness in which the target protocol run cannot be interfered by adversarial queries. In contrast, the latter set of schemes offer full synchronization robustness using puncturable pseudo-random functions (PPRFs), such that the participants are able to re-synchronize even with arbitrary adversarial queries during the target session [BDdK+21]. Henceforth, any mention of synchronization robustness only signifies the stronger definition, as the adversary in Tamarin considers all possible cases of manipulation using concurrent sessions.

Chapter 3
Tamarin Prover: The Essentials

In this chapter, we shall cover the rudiments of formal analysis with Tamarin Prover. First, Section 3.1 gives a brief account of the logic and reasoning behind Tamarin analysis. Then, Section 3.2 enlists and describes the basic building blocks that constitute a typical model in Tamarin. At last, Section 3.3 outlines how proofs are generated, as well as some advanced features used further in this thesis.

3.1 Tamarin Logic

Tamarin Prover is a symbolic model verification tool used to formally analyze security protocols. The overview of Tamarin's analysis logic was elicited in the project preceding this thesis [Kab21], as followed in the next two paragraphs. Tamarin, like other symbolic verifiers, takes as input the protocol model, which includes specifying an adversary and actions taken by the actors in a protocol, as well as targeted security properties. The tool then automatically verifies if the protocol fulfils these properties, even with an unbounded number of parallel sessions.

The execution model of the tool, however, resembles a labeled transition system [BCDS17]. Knowledge of the adversary, exchanged messages and the actor's internal states are represented as multi-sets of facts, constituting the state space of the model. The adversarial capabilities and protocol model are accordingly expressed by multi-set rewriting rules. Note that Tamarin considers a Dolev-Yao adversary by default that controls the network and can maliciously perform corresponding operations. Furthermore, the proofs to validate a security property employ the backward search accompanied with reasoning modulo equational theories [BCDS17]. These semantics enable Tamarin to handle complex control flows such as loops, stronger adversarial models for key exchange protocols such as the eCK model [LLM07].

The proof construction in Tamarin is a distinct feature that covers the best of two worlds, viz., automatic and manually-assisted verification.

– To begin with, the tool has an **automated mode**, combining deduction and equational reasoning to search for proofs. If terminated, the search yields either a proof that the security property in question holds for arbitrarily many protocol runs, or provides with a counterexample attack falsifying the property. The fully automated mode can be operated on the command-line and takes as argument a theory file typically containing three inputs: equational theory to model messages, multi-set rewriting rules to model protocols and the desired properties as lemmas. We will later look into each of these inputs in detail.

– Termination of proofs, however, is not guaranteed due to the undecidability of security properties [HS20]. In this case, users can avail Tamarin's **interactive mode** which is implemented as a web server. It aids the analysis with interactive proof guidance allowing inspection of attack graphs and proof states, through which a user can automate analysis for chosen parts of the state space [BCDS17]. We use the interactive mode in this work for presentation clarity.

3.2 Building Blocks

In this section, we will explain each of the inputs that together make up a protocol model to be analyzed in Tamarin, usually saved in a `.spthy` file. Throughout the rest of the chapter, we work on a simple two-message Diffie-Hellman (DH) exchange shown in Figure 3.1 to illustrate the Tamarin analysis in action. In this example protocol, participants A and B share a secret key `k` used to calculate a Message Authentication Code (MAC). Both the parties send their DH public keys over the network, while only party A appends with it a MAC over the public component and respective identities perceived in the exchange. B derives a secret session key $\mathtt{g}^{\mathtt{xy}}$ only after verifying the MAC, while A derives it on reception of the second message. We eventually intend to prove whether the derived key is secret, from the perspective of either parties. Recall that Tamarin is a symbolic model verifier, therefore, messages are modeled as terms and algebraic properties are modeled as equations.

3.2.1 Variables, functions and equations

In Tamarin, we generally encounter three types of variables. First is the *fresh* type, denoted by a tilde (`~`), used to signify freshly generated randomness, typically in secret keys or nonces. Secondly, there is the *public* type, denoted by a dollar (`$`), used to model publicly known values such as identities and labels. Finally, we have the *temporal* type, denoted by a hash (`#`), used to relate timestamps during a protocol execution. The temporal type variables are necessary to prove security properties as the protocol must fulfil these properties at desired stages of the run.

As shown in Snippet 3.1, a typical Tamarin file starts with a custom theory name, in our case, `example`, followed by the `begin` statement. Now, Tamarin supports

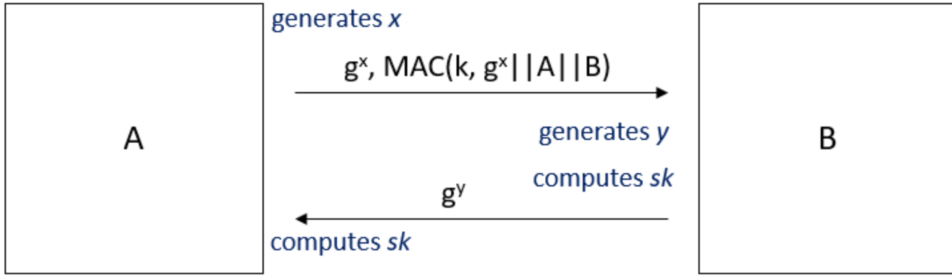


Figure 3.1: An artificial DH protocol

a number of built-in, as well as user-defined function symbols. The support for DH exponentiation and associated function symbols, like the generator constant `'g'`, comes with the built-in `diffie-hellman`. Additionally, we need functions `mac` and `vfy` to model the MAC generation and corresponding verification algorithms respectively. The arity of a function is defined by a number adjoining the function. Here, `mac` has an arity of 2, first input being the MAC key and the second being the authenticated message. Similarly, `vfy` has an arity of 3 to accommodate the key, the message as well as the MAC in question needed by the verification algorithm. Note that `<x1, x2, x3,..., xn>` denotes a tuple and can be used to signify any number of terms that constitute the message `m` in MAC generation. A function `true` (with 0-arity) is used to denote the Boolean TRUE output by the verification algorithm.

Snippet 3.1 Initialization, functions and equational theory

```
theory Example
begin

builtins: diffie-hellman
functions: vfy/3, mac/2, true/0
equations: vfy(k, m, mac(k, m)) = true
```

Further, we need equational theories to relate the properties of these functions. The relation between `mac` and `vfy` is shown under `equations` in Snippet 3.1. The built-ins come with a pre-defined set of equations necessary for the corresponding functions to work. For a full list of built-in theories, please refer to the Tamarin manual [Tea]. In an equation, both left-hand and right-hand sides can contain variables, and the variables on the right-hand side must be present on the left-hand side. Moreover, public constants are not allowed as a part of equations.

3.2.2 Rules and restrictions

We will now look into the key components of a Tamarin file used to model the exchanged messages and execution of the protocol, i.e., rules. We follow multiset rewriting system to specify rules, a common way to model concurrent systems [Tea], in our case being the protocol execution and actions of the adversary. This method defines a labeled transition system and each of the multiset rewriting rules are triplets of sequence of facts. A triplet has a left-hand side called premise, a label and a right-hand side called conclusion. Facts are the basic units of storing information in the protocol's state, and are of the form `Fact(t1, t2,..., tn)`, where `Fact` is the fact symbol and the arguments are terms. Each fact has a fixed arity and therefore, using same fact symbol with different arities results in an error.

Now, starting from an empty multiset as the initial state of the system, a rule defines how state transitions occur. A rule is executed if all the facts in the premise exist in the current state, resulting in a state that has the facts contained in the conclusion. Usually, the facts in the premise are consumed in the process unless marked persistent using an exclamation mark (`!`), in which case they are carried forward to the next state. This is useful while defining facts that do not change during the course of the protocol. The label, however, consists of action facts, which do not appear in the system state, rather in the traces on which security properties are specified. We will look more into action facts while defining such a property in the next sections. There are three special facts in Tamarin: `Fr`, `In` and `Out`.

- The `Fr` fact is used while dealing with fresh values, which are in turn its arguments. These facts can only appear in the premise and Tamarin ensures that each instance of such a fact produces a unique term.
- The `In` fact models the reception of a message by a party from the untrusted network, and like `Fr` fact, it can only appear in the premise.
- The `Out` fact, on the other hand, models the sending of a message by a party to the untrusted network, and can only appear in the conclusion.

It is a good practice to start with a rule that sets up knowledge of parties before the target protocol run. This can be achieved by writing a rule that models public key infrastructure, i.e., generating a fresh key and associating a party's identity with the respective private and public components persistently. This is shown in Snippet 3.2; note that the public key is also available to the untrusted network using `Out` fact.

We will, however, restrain ourselves to manually setting up the initial knowledge using a rule written in Snippet 3.3. This helps us to distribute symmetric MAC generation keys and also allows the parties to generate fresh private exponents during the protocol run, which is a safer practice as compared to storing long term keys.

Snippet 3.2 Public Key Infrastructure for DH-based protocols

```
rule pki:
[ Fr(~x) ]
--[]->
[ !Pk($A, 'g'^~x), !Ltk($A, ~x), Out('g'^~x) ]
```

Snippet 3.3 Setup initial key knowledge with participants

```
rule setup:
[ Fr(~k) ]
--[]->
[ AgStA($A, $B, ~k), AgStB($B, $A, ~k) ]
```

The `setup` rule can therefore be described as follows. First, generate a fresh MAC key `k` and then choose two public names A and B for which `k` acts as a symmetric key. As a result, generate the fact `AgSt` that associates the agent A with its partner B and the respective MAC key `k` (a similar fact is generated for the agent B). Now, a term may occur multiple times and might be too big to sustain readability of the rules. For such cases, Tamarin allows the use of `let...in` bindings. To illustrate, let us write our next rule where we model the sending of the first message by A as shown in Snippet 3.4. Here, the MAC generated by A is assigned as `maca` and the message sent is assigned as `m1` using the `let` statement. These assignments can then be used in our rule after the `in` statement. In this rule, A generates a fresh private exponent `x` and sends the corresponding public key to the network using the `Out` fact. The knowledge of the agent is updated and contained in the fact `AgStA1`.

Snippet 3.4 First message: A sends the public key and corresponding MAC

```
rule a1:
let
  maca = mac(~k, <'g'^~x, $A, $B>)
  m1 = <'g'^~x, maca>
in
[ AgStA($A, $B, ~k), Fr(~x) ]
--[]->
[ Out(m1), AgStA1($A, $B, ~k, ~x, m1) ]
```

Next, let us write a rule for the reception and processing of `m1` by B and sending its own public key. After receiving `m1` from the network using the `In` fact, B first verifies the MAC `maca`. This can be modeled using *restrictions*. As the name suggests, the purpose of a restriction is to restrict the traces considered for analysis. In our

case, we need to analyze only those traces for which the MAC verification succeeds. Like action facts, restrictions appear only on traces and therefore, are part of the labels. Snippet 3.5 shows the restriction `Equality`, and how it is used in the label of rule `b2`. It is a good practice to define restrictions just after the equations, for them to be used in any rule that follows. The `Equality` restriction roughly says that for any two values `x` and `y`, occurrence of `Eq(x,y)` at any given timepoint `i` implies that `x = y`. The restriction is used in rule `b2` such that only those traces of the protocol are considered for which the output of the verify function is equal to `true`. Please refer to the Tamarin manual [Tea] for more use cases of restrictions.

Once the MAC is verified, B generates its own fresh private exponent `y` and sends the corresponding public key to the network. It also derives the session key by calculating $\mathtt{X}^{\mathtt{y}}$, where `X` is the public key received from A. Similar to the rule `a1`, the knowledge of the agent is updated and contained in the fact `AgStB2`. We will look into the action fact `AcceptsB` in the following subsection.

Snippet 3.5 Second message: B processes `m1` and derives the session key

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

rule b2:
let
  m1 = <X, maca>
  m2 = 'g'^~y
  sk = X^~y
in
[ AgStB($B, $A, ~k), Fr(~y), In(m1) ]
--[ Eq(vfy(~k, <X, $A, $B>, maca), true),
    AcceptsB($B, sk)
  ]->
[ Out(m2), AgStB2($B, $A, ~k, ~y, X, m1, m2, sk) ]
```

As a last step in the protocol, the initiator party A receives `m2` and derives the session key by calculating $\mathtt{Y}^{\mathtt{x}}$, where `Y` is the public key received from B. This is depicted in Snippet 3.6. We will look into the action fact `AcceptsA` in the next subsection. Note that the message `m2` was not authenticated by a MAC, which will result in an attack as we prove later. This finishes our model for the protocol, and we move on prove the security properties.

Snippet 3.6 *A* receives `m2` and derives the session key

```
rule a3:
let
  m2 = Y
  sk = Y^~x
in
[ AgStA1($A, $B, ~k, ~x, m1), In(m2) ]
--[ AcceptsA($A, sk)
  ]->
[ AgStA3($A, $B, ~k, ~x, Y, m1, m2, sk) ]
```

3.2.3 Security properties and lemmas

The protocol properties can be specified either as trace or observational equivalence properties. Let us first explain trace properties. Whenever a rule is executed, the corresponding action facts in the label are appended to the trace [Tea]. All action facts in this labeled transition are said to have occurred at the same time. A trace property is simply a set of traces defined as guarded fragments of first-order logic formulas over action facts, sorted with timepoints. As opposed to rules, terms in such a formula cannot be built from function symbols, rather only quantified variables.

There are two kinds of quantified variables, universally quantified, starting with a keyword `All`, and existentially quantified, starting with a keyword `Ex`. While the former imposes the condition that the formula must hold for all instances of the variables, the latter only requires one such instance. Amongst the logical operators used, binding is tightest for negation, followed by conjunction (`&`), disjunction (`|`) and implication (`==>`). Note that the variables must be guarded, which means that they all appear in an action fact immediately after the quantifier. Additionally, inside the quantifier, it requires for universally quantified variables to have an implication, and for existentially quantified variables to have a conjunction as the outermost logical operator. Keeping this in mind, a property is defined using the keyword `lemma` followed by the appropriate formula. There are again two ways to define a property, either use the keyword `all-traces`, which signifies that the property must hold for all the traces of the protocol, or use the keyword `exists-trace`, which signifies that the property holds even if one satisfying trace is found.

To illustrate these notions, let us write a simple lemma to verify secrecy of the dervied session key with respect to either of the parties. Describing in words, we want that whenever a party accepts the session and derives a session key, it cannot be that the adversary knows the session key. This is written in the lemma `sessionKeySecrecyA` as shown in Snippet 3.7. Breaking it down, the lemma states

that for all the traces of the protocol, for all agent identities `a` and session key `sk`, if `a` accepts the session acting as party A and derives `sk` at a timepoint `i`, then there cannot exist a timepoint `j` such that the adversary knows `sk` at `j`. Note that in Tamarin, knowledge of the adversary is represented using the arguments in predicate `K`. This form of secrecy does not incorporate corruption of long term secrets or forward security. A similar lemma `sessionKeySecrecyB` is also written for party B. After the rules and the lemmas are specified, the model terminates with an `end` statement.

Snippet 3.7 Session key secrecy lemmas

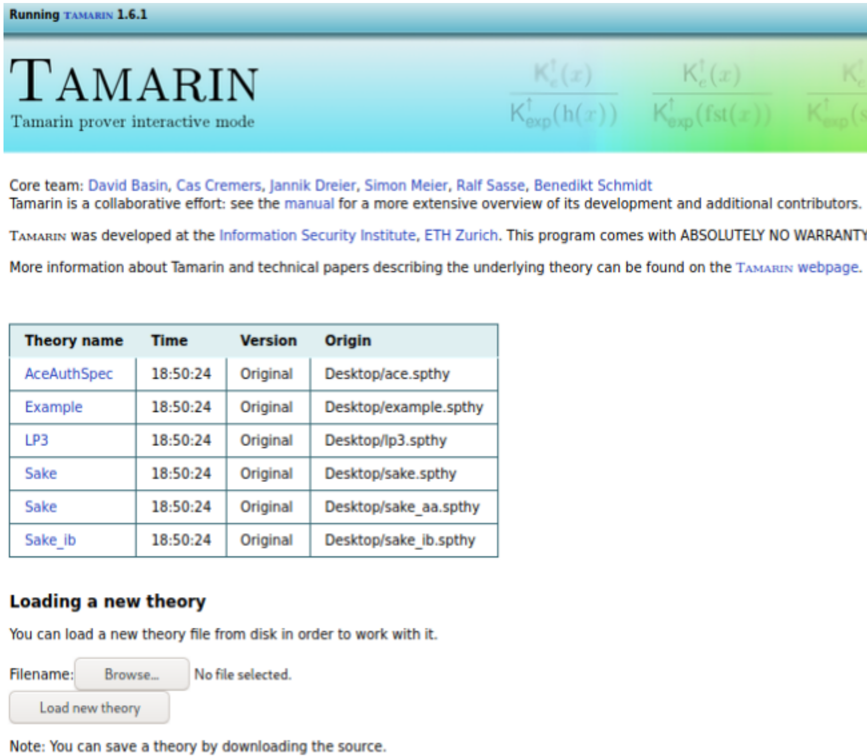
```
lemma sessionKeySecrecyA:
all-traces
"All a sk #i.
  AcceptsA(a, sk)@i ==>
  not (Ex #j. K(sk)@j)"

lemma sessionKeySecrecyB:
all-traces
"All b sk #i.
  AcceptsB(b, sk)@i ==>
  not (Ex #j. K(sk)@j)"
```

This sufficiently exemplifies the trace properties. Another method of specifying a security property is through observational equivalence. Unlike trace properties, these properties are not independently defined on each trace. Rather, they focus on two systems being virtually indistinguishable for an adversary. However, they are beyond our requirements and for further details, please refer to the manual [Tea].

3.3 Tamarin Analysis

Finally, let us analyse our model by running Tamarin and verify if the security properties are satisfied. To run the analysis in interactive mode, we use the command line `tamarin-prover interactive example.spthy`. Make sure there are no wellformedness errors with the model, which are usually shown in the terminal. If no errors are found, the theory file loads successfully, and a web server is started at **http://localhost:3001**, displaying all theory files in the same directory as shown in Figure 3.2. Click on the theory file named `Example`. This leads us to the theory page loaded with our model, with left half appearing as Figure 3.3.

Let us explain each of these pointers briefly. The *Message Theory* contains all the defined functions and equations. In addition, it lists all the functions that an adversary can use, the construction rules help the adversary to construct new terms from already available terms, while the deconstruction rules allow it extract terms



Theory name	Time	Version	Origin
AceAuthSpec	18:50:24	Original	Desktop/ace.spthy
Example	18:50:24	Original	Desktop/example.spthy
LP3	18:50:24	Original	Desktop/lp3.spthy
Sake	18:50:24	Original	Desktop/sake.spthy
Sake	18:50:24	Original	Desktop/sake_aa.spthy
Sake_ib	18:50:24	Original	Desktop/sake_ib.spthy

Figure 3.2: Tamarin interactive mode landing page

from larger terms. As an example, the rule written in Snippet 3.8 helps the adversary to calculate $\texttt{x.2}^{\texttt{x.1*x.3}}$ from already known values of $\texttt{x.2}^{\texttt{x.3}}$ and `x.1`.

Snippet 3.8 An example deconstruction rule

```
rule (modulo AC) d_exp:
   [ !KD( x.2^x.3 ), !KU( x.1 ) ] --> [ !KD( x.2^(x.1*x.3) ) ]
```

Second in line, the *Multiset Rewriting Rules* contains the rules written in the model, in addition to two default rules `isend` and `irecv`. While the former takes a value `x` from the adversary's knowledge `!KU(x)` and inputs it to the protocol `In(x)`, the latter allows the protocol output `Out(x)` to be passed to the adversary `!KD(x)`.

Next, we look into the *Raw and Refined sources*. As mentioned earlier in Section 3.1, Tamarin uses backward search to compute all possible sources of a fact for efficient analysis. These are called case distinctions, and basically represent all

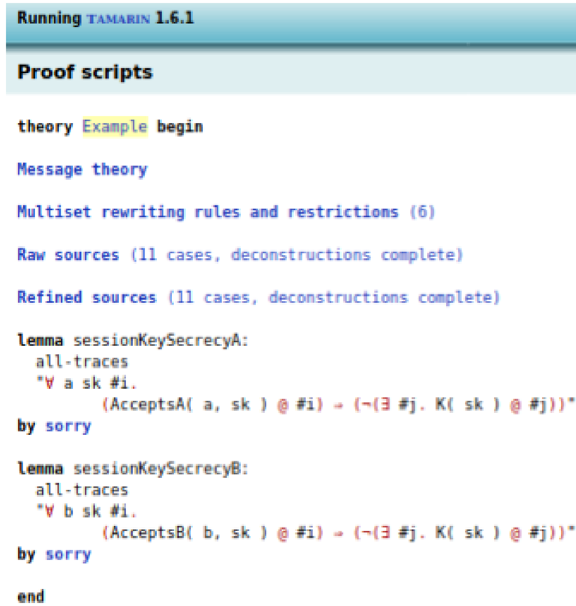


Figure 3.3: Model loaded from `Example` theory file

the rules that produce the fact in question. These sources are listed on the right half of the page. As an example, consider the fact `AgStA3`, the sources for which are depicted in Figure 3.4 (only one in this case). The boxes represent individual rules, ordered with respective premise, label and conclusion. The only possible source for `AgStA3` is the rule `a3`, the premise of which contains the fact `AgStA1`, that in turn sources from the rule `a1`, and the fact `In(.)`, that sources from the rule `isend`. This might hint at the attack we are going to cover in this section. The trapezoid at the bottom simply depicts the sink for the fact `AgStA3`.

The difference between Raw and Refined sources appear when Tamarin is not able to find sources for a fact, which is seen as a partial deconstruction is left. These halt the automatic proof generation and are required to be manually handled by techniques such as *source lemmas* [Tea]. These might be encountered while modeling very complex protocols but we will refrain from any further discussion in this work.

Let us now move on to the main task at hand, proving security properties. Before using Tamarin, we can roughly delineate our expectations from the proofs. The public key received from B is not authenticated by a MAC and therefore, can be replaced maliciously by a man-in-the-middle adversary. This leads to A calculating a session key that is no longer secret as the same session key can be computed by

Sources of "AgStA3(t.1, t.2, t.3, t.4, t.5, t.6, t.7, t.8) ▶₀ #i" (1 cases)

Source 1 of 1 / named "a3"

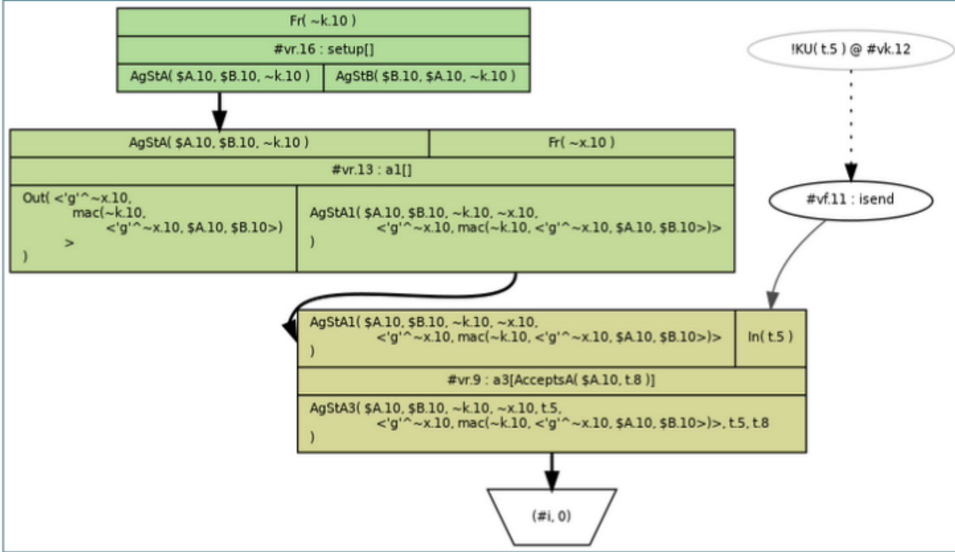


Figure 3.4: Computed sources of the fact `AgStA3`

the adversary using the public key sent by A over the network. This, however, is not the case with B. The associated MAC in message `m1` ensures that the public key is not altered and B computes the correct session key that cannot be accessed by the adversary. Consequently, the session key secrecy lemma should be proved only for party B. Now, the end of the list in Figure 3.3 contains the lemmas written in our model. For the first lemma `sessionKeySecrecyA`, click on `sorry`, and Tamarin will show the possible methods to start the proof (Figure 3.5).

The tool uses constraint solving to either prove that the property satisfies all possible cases or finds an attack. The `simplify` and `induction` are two approaches to the proof and can be used to manually guide through the proof in case of non-termination [Tea]. We are, however, interested in the end result, and by clicking the `autoprove` command, we use Tamarin heuristics to reach the final state of the proof. Other `autoprove` variants are used to customize the proof trajectory, for instance the one with a proof depth bound of 5 is used to analyze traces up-to 5 levels of hierarchy in the backward search for case distinctions.

Lemma: sessionKeySecrecyA

Applicable Proof Methods: Goals sorted according to the 'smart' heuristic (loop breakers delayed)

1. simplify

2. induction

a. autoprove (A. for all solutions)
b. autoprove (B. for all solutions) with proof-depth bound 5
s. autoprove (S. for all solutions) for all lemmas

Constraint system

last: none

formulas:
∃ a sk #i. (AcceptsA(a, sk) @ #i) ∧ ∃ #j. (K(sk) @ #j)

Figure 3.5: Methods to prove a lemma

On the left half of the page, the lemma is turned red as shown in Figure 3.6, and exhibits that a counter-example trace is found to disprove the secrecy property in case `a1`. Clicking on accompanying `solve` statements, this case first replaces the session key `sk` with $\mathtt{Y}^{\mathtt{x}}$, and then `Y` with generator constant `g`. The attack becomes clearer in the graph shown on the right half of the page as portrayed in Figure 3.7. As expected, the adversary modifies message `m2` and sends `g` instead of $\mathtt{g}^{\mathtt{y}}$ using the rule `isend`. As a result, the session key computed by A is $\mathtt{g}^{\mathtt{x}}$. Now, the adversary gets $\mathtt{g}^{\mathtt{x}}$ from the `Out` fact in rule `a1` (marked by a red arrow). Evidently, the session key secrecy does not hold with respect to A.

```
lemma sessionKeySecrecyA:
  all-traces
  "∀ a sk #i.
         (AcceptsA( a, sk ) @ #i) → (¬(∃ #j. K( sk ) @ #j))"
simplify
solve( AgStA1( $A, $B, ~k, ~x, m1 ) ▶₀ #i )
  case a1
  solve( splitEqs(0) )
    case split_case_1
    solve( !KU( Y^~x ) @ #vk.1 )
      case a1
      SOLVED // trace found
    qed
  qed
qed
```

Figure 3.6: Counter-example trace found

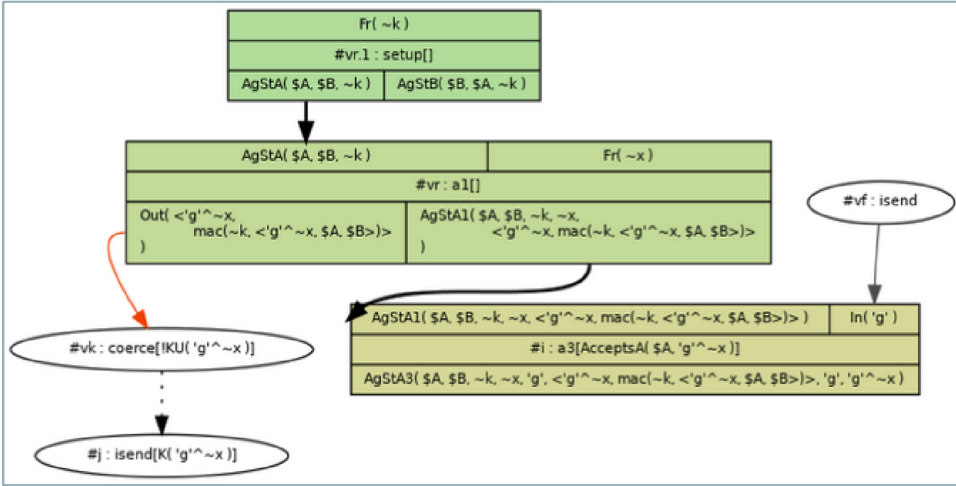


Figure 3.7: Counter-example attack diagram

At last, we will verify the second lemma by following similar steps and navigate to the final state of the proof by clicking on `sorry` followed by `autoprove`. This time, the lemma is turned green as shown in Figure 3.8, indicating that it was successfully proven. For each of the cases, by clicking on the `solve` statements and inspecting the corresponding graphs on the right, we can conclude that Tamarin was unable to find a counter-example. It is reasonable to ponder about the termination of proofs under unbounded parallel sessions assumption. This is where the backward search for case distinctions comes into picture, as the constraint solving system restricts the traces considered for analysis to all possible manipulations of the target protocol run by the adversary, which are in turn finite. In this way, Tamarin formally proves and confirms our expected results. We are now well-equipped with the essential functioning and features of Tamarin to analyze more complex protocols.

```
lemma sessionKeySecrecyB:
  all-traces
  "∀ b sk #i.
        (AcceptsB( b, sk ) @ #i) → (¬(∃ #j. K( sk ) @ #j))"
simplify
solve( AgStB( $B, $A, ~k ) ▶₀ #i )
  case setup
  solve( !KU( mac(~k, <X, $A, $B>) ) @ #vk.2 )
    case a1
    solve( !KU( 'g'^(~x*~y) ) @ #vk.3 )
      case a1
      by solve( !KU( ~y ) @ #vk.4 )
    next
      case b2
      by solve( !KU( ~x ) @ #vk.4 )
    next
      case c_exp
      by solve( !KU( ~x ) @ #vk.6 )
    qed
  next
    case c_mac
    by solve( !KU( ~k ) @ #vk.5 )
  qed
qed
```

Figure 3.8: Session key secrecy holds *wrt.* B

Chapter 4
Formal Analysis of SAKE

We shall now formally analyze the SAKE protocol [ACF20]. Section 4.1 reviews the protocol execution, underlying assumptions and the security properties fulfilled. In Section 4.2, we present an overview of the formal analysis of SAKE as modeled in Tamarin, also reasoning for the associated lemmas. Finally, in Section 4.2.3, we brief the weakness in their design and why it fails to provide synchronization robustness.

4.1 SAKE: Review

In this section, we outline the SAKE protocol and key assumptions used in its design. SAKE employs key evolution based on symmetric-key functions to provide mutual authentication, key agreement and forward secrecy [ACF20]. As opposed to counters or a clock, the scheme uses another chain of keys to track the internal state of a party. These keys are independent of the master keys used to derive session secrets and help in resynchronization of communicating parties.

4.1.1 Design Ideas

For its analysis, SAKE employs the security requirements for AKE protocols presented in [BJS16]. In practice, the adversary controls the network, and can arbitrarily modify, relay, replay, delete or insert messages sent over the network between communicating parties. A party can execute multiple runs of the protocol called sessions, each having their own local state variables and access to the long term keys [ACF20]. The model then mathematically defines security goals like entity authentication, key indistinguishability and security, along with the queries an adversary can make to the participants, however, we will omit the details in this work. Additionally, and more importantly, the authors of SAKE do not allow parallel executions of the sessions, i.e., for any party, the protocol only considers sequential runs for the resynchronization procedure to be effective.

The SAKE protocol uses two chains of keys, one called the derivation master key (K), used to derive session keys, and other called the authentication master key (K') used to synchronize internal states. K' and K are updated together, and is depicted in Figure 4.1 around an arbitrary epoch j. Here, `UPD` denotes a one-way function and `KDF` denotes the key derivation function keyed with K.

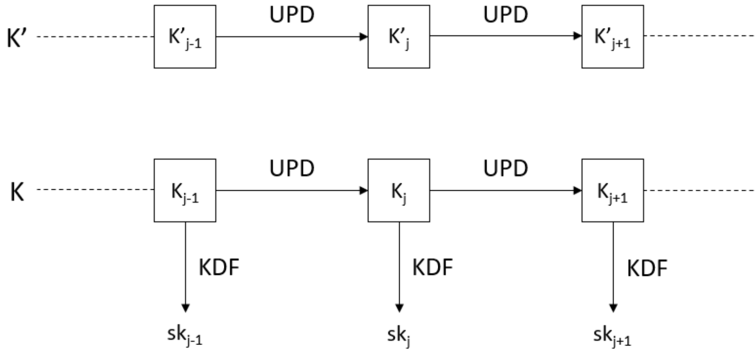


Figure 4.1: Key evolution and session key derivation in SAKE (adapted from [ACF20])

Now, in the case only sequential sessions are allowed, it can be shown that synchronization states of two parties in SAKE cannot differ by a number other than $\{-1, 0, 1\}$, if both of them are in sync at deployment [ACF20]. Then, consider an initiator-responder paradigm, for any epoch j as shown in Figure 4.1, the initiator stores four keys $(k_j, k'_{j-1}, k'_j, k'_{j+1})$. For the corresponding protocol sessions at the responder, it stores two keys (k, k'). Since the internal states of the two parties can only differ by a maximum of 1 step, $(k, k') \in \{(k_{j-1}, k'_{j-1}), (k_j, k'_j), (k_{j+1}, k'_{j+1})\}$. A party deems to 'accept' a session only after receiving confirmation of key updation from its partner, and therefore, considers the derived session keys to be fresh. Note that storing the authentication master key corresponding to a previous epoch does not trivially break PFS as session keys are derived from the derivation master key, which is only stored for the current epoch.

4.1.2 Protocol Flow

Let us consider two parties I and R, respectively denoting the initiator and responder. At deployment, responder keys k and k' are chosen uniformly at random. Also, the initiator keys are initialized as $k_j = k$, $k'_{j-1} = \perp$, $k'_j = k'$ and $k'_{j+1} = \mathrm{UPD}(k')$. The protocol flow for an arbitrary epoch j is shown in Figure 4.2. One round of updation at the initiator ($\texttt{Evol}_I$) entails the following operations:

- $k_j \leftarrow \texttt{UPD}(k_j)$

- $k'_{j-1} \leftarrow k'_j$
- $k'_j \leftarrow k'_{j+1}$
- $k'_{j+1} \leftarrow \texttt{UPD}(k'_{j+1})$

On the other hand, one round of updation at the responder ($\texttt{Evol}_R$) includes:

- $k \leftarrow \texttt{UPD}(k)$
- $k' \leftarrow \texttt{UPD}(k')$

$\texttt{Vfy}(k, m, \tau)$ is the MAC verification function that outputs `true` if τ is a valid MAC tag for message m and key k, otherwise outputs `false`. The key derivation function KDF uses a one-way function f that represents any operation between the nonces r_I and r_R generated by respective parties. The message m_R contains a MAC calculated with initial k', and therefore, helps I to know the state of R at the start of the protocol, and accordingly calculate the parameters δ and ϵ. Note that even though KDF is applied, a party 'accepts' only after reception of key update confirmations, which in turn depend on the respective value of ϵ [ACF20].

In essence, we have three cases for the difference in internal states of I and R at the start of the protocol. If I and R are in sync, $\delta = 0$, I evolves its keys and computes the session key, and R follows the same after reception of m_I. If I is one step behind R, $\delta = -1$, I first evolves once, and then same steps follow as when $\delta = 0$. If I is one step ahead of R, $\delta = 1$, R catches up first, evolves its keys again and computes the session key, followed by I performing similar operations. I always receives the key update confirmation with message τ_{RR} and hence, 'accepts' on reception. R, however, receives the confirmation with m_I if $\epsilon = 0$ and with τ_{II} if $\epsilon = 1$, and accordingly 'accepts' the session.

4.2 Modelling in Tamarin

Let us now model SAKE in Tamarin and try to prove the desired security properties. It is important to remember that Tamarin, by default, considers interleaving concurrent sessions while proving lemmas. Without loss of generality, we will only consider one of the three cases of internal state disparity between I and R, as others may follow similar proofs. Please refer to [sakemodel] for models covering other cases.

4.2.1 Rules

We will start with defining the key derivation function, the operation between nonces r_I and r_R, as well as the MAC and corresponding verification function. A simple equational theory defines the MAC verification as shown in Snippet 4.1. For simplicity, we will use hashing as a means to evolve our keys. Remember that it is cardinal to define the Boolean value `true` as a function for Tamarin to compile the model.

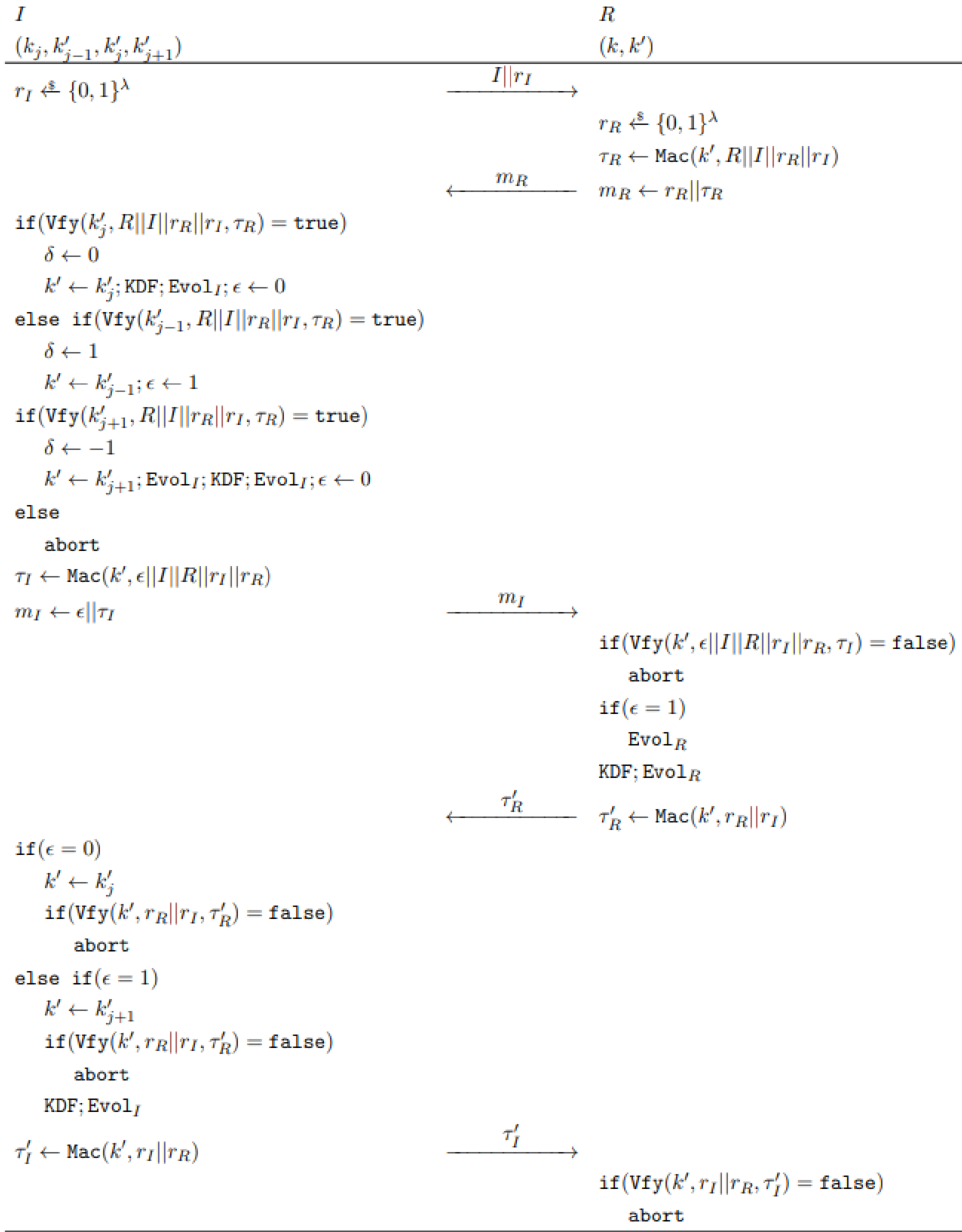


Figure 4.2: SAKE protocol flow for an arbitrary epoch j (adapted from [ACF20])

Snippet 4.1 Functions and equational theory

```
builtins: hashing
functions: kdf/2, opr/2, vfy/3, mac/2, true/0
equations: vfy(k, m, mac(k, m)) = true
```

Next, we initialize the protocol with a `setup` rule to define the knowledge of the parties prior to the start of the target run. In this example, we choose to consider the case when I is one step behind R (Figure 4.3). The fresh keys `k` and `l` in Snippet 4.2 represent root of the derivation and authentication master keys respectively. Additionally, we include the `Equality` restriction used for MAC verifications.

Snippet 4.2 Producing facts that represent knowledge of participants

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

rule setup:
[ Fr(~k), Fr(~l) ]
--[]->
[ AgStI($I, $R, <~k,~l,h(~l),h(h(~l))>), AgStR($R,<h(~k),h(h(~l))>) ]
```

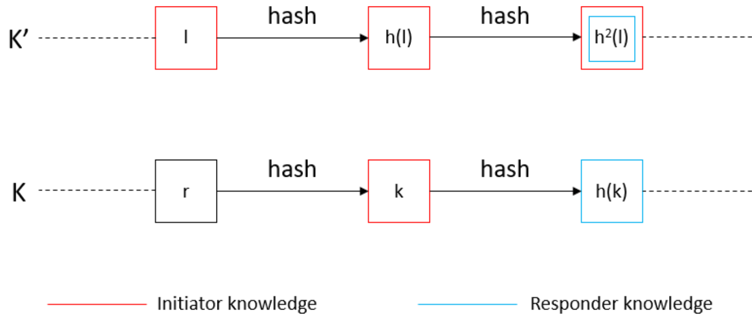


Figure 4.3: Knowledge of initiator and responder before the target protocol run

We will now write rules to model the messages exchanged between the parties, depicted in Figure 4.2. First, the initiator I chooses a fresh nonce `ri` and sends it to R over the network. Notice how facts representing the knowledge of the parties in all the following rules change from premise to that in conclusion. Also, the fact `KeyI1` shows which derivation master key is stored with I during the execution of the rule

I1. Finally, the action fact I1 shows the current stage of protocol execution in terms of messages exchanged with respect to I. Similar action facts are used further.

Snippet 4.3 First Message: initiator sends a nonce

```
rule I1:
let
  m1 = <$I, ~ri>
in
[ AgStI($I, $R, <~k,~l,h(~l),h(h(~l))>), Fr(~ri) ]
--[ I1($I, $R, m1)
  , KeyI1($I, ~k)
  ]->
[ AgStI1($I, $R, m1, ~ri, <~k,~l,h(~l),h(h(~l))>),
  Out(m1) ]
```

The responder R receives this first message, generates its own fresh nonce `rr`, computes the MAC `tr` using its current authentication master key `h(h(l))` and send them over the network to I. Similar to `KeyI1` and `I1`, the fact `KeyR2` shows the key stored with R and the action fact `R2` shows the current stage of protocol execution with respect to R after the rule `R2` is completed. It is important to observe that since the identity of the initiator is sent in plaintext, we use `i` instead of `$I` while generating `tr` to signify that the value can be manipulated.

Snippet 4.4 Second Message: responder sends a nonce and divulges its internal state

```
rule R2:
let
  m1 = <i, ri>
  tr = mac(h(h(~l)), <$R, i, ~rr, ri>)
  mr = <~rr, tr>
in
[ AgStR($R,<h(~k),h(h(~l))>), In(m1), Fr(~rr) ]
--[ R2(i, $R, m1, mr)
  , KeyR2($R, h(~k))
  ]->
[ AgStR2(i, $R, m1, mr, ri, ~rr, <h(~k),h(h(~l))>),
  Out(mr) ]
```

After receiving the MAC `tr`, I compares it with the MAC generated by the key `h(h(l))` in its knowledge, and eventually in this case, knows that $\epsilon(\texttt{ep}) = 0$. It also accordingly evolves its keys, derives the session key `sk` and generates the MAC `ti`. Finally, it sends `ti` and ϵ to R. Note that the MAC verification uses the restriction

`Equality` as one of the action facts to check if the function `vfy` outputs true. The protocol execution aborts if the equality check fails.

Snippet 4.5 Initiator evolves and derives a session key

```
rule I3:
let
  mr = <rr, tr>
  ep = 0
  sk = kdf(h(~k), opr(~ri, rr))
  ti = mac(h(h(~l)), <~ep, $I, $R, ~ri, rr>)
  mi = <~ep, ti>
in
[ AgStI1($I, $R, m1, ~ri, <~k,~l,h(~l),h(h(~l))>), In(mr), Fr(~ep) ]
--[ I3($I, $R, m1, mr, mi)
  , KeyI3($I, ~k)
  , Eq(vfy(h(h(~l)), <$R, $I, rr, ~ri>, tr), true)
  ]->
[ AgStI3($I, $R, m1, mr, mi, ~ri, rr, ~ep, <h(h(~k)),h(h(~l)),
  h(h(h(~l))),h(h(h(h(~l))))>, sk),
  Out(mi) ]
```

Next, R receives and verifies `ti`, and subsequently, evolves its keys and derives the session key `sk`. At this point, R may 'accept' as both the parties are evolved and share a session key with respect to the responder. However, for simplicity, we make the responder 'accept' only after the last message is processed. R also computes `trr` using the evolved authentication master key `h(h(h(l)))` and sends it to I.

Snippet 4.6 Responder evolves and derives a session key

```
rule R4:
let
  sk = kdf(h(~k), opr(ri, ~rr))
  mi = <ep, ti>
  trr = mac(h(h(h(~l))), <~rr, ri>)
in
[ AgStR2(i, $R, m1, mr, ri, ~rr, <h(~k),h(h(~l))>), In(mi) ]
--[ R4(i, $R, m1, mr, mi, trr)
  , KeyR4($R, h(~k))
  , Eq(vfy(h(h(~l)), <ep, i, $R, ri, ~rr>, ti), true)
  ]->
[ AgStR4(i, $R, m1, mr, mi, trr, ri, ~rr, <h(h(~k)),h(h(h(~l)))>, sk),
  Out(trr) ]
```

I, at this point, due to two rounds of update, has stored `h(h(l))`, `h(h(h(l)))` and `h(h(h(h(l))))` in its state. Therefore, it uses the middle key as per the protocol to verify the MAC `trr`, which also serves as the key update confirmation, making I accept. It then generates the MAC `tii` and sends it to R over the network. The action fact `EvolvedI` shows the knowledge of I at the end of the protocol.

Snippet 4.7 Initiator accepts the target run

```
rule I5:
let
  tii = mac(h(h(h(~l))), <~ri, rr>)
in
[ AgStI3($I, $R, m1, mr, mi, ~ri, rr, ~ep, <h(h(~k)),h(h(~l)),
  h(h(h(~l))),h(h(h(h(~l))))>, sk), In(trr) ]
--[ I5($I, $R, m1, mr, mi, trr, tii)
  , Eq(vfy(h(h(h(~l))), <rr, ~ri>, trr), true)
  , AcceptsI($I, $R, sk, <~ri, rr>)
  , EvolvedI($I, $R, <h(h(~k)),h(h(~l)),h(h(h(~l))),h(h(h(h(~l))))>)
  ]->
[ Out(tii) ]
```

Finally, R receives `tii`, verifies it using `h(h(h(l)))` and 'accepts' the target run, where `EvolvedR` shows the knowledge of R at the end of the protocol. The action facts `AcceptsI` and `AcceptsR` have `sk` as argument and help in writing lemmas for session key secrecy. After the protocol run, both the parties have evolved and synchronized their internal states as shown in Figure 4.4. Moreover, they share a session key derived using the derivation master key `h(k)`.

Snippet 4.8 Responder accepts the target run

```
rule R6:
[ AgStR4(i, $R, m1, mr, mi, trr, ri, ~rr, <h(h(~k)),h(h(h(~l)))>, sk),
  In(tii) ]
--[ R6(i, $R, m1, mr, mi, trr, tii)
  , Eq(vfy(h(h(h(~l))), <ri, ~rr>, tii), true)
  , AcceptsR(i, $R, sk, <ri, ~rr>)
  , EvolvedR($R, <h(h(~k)),h(h(h(~l)))>)
  ]->
[ ]
```

4.2.2 Lemmas

We shall now verify the security properties of SAKE by writing lemmas. When modeling real protocols, it is always a good practice to begin with an executability

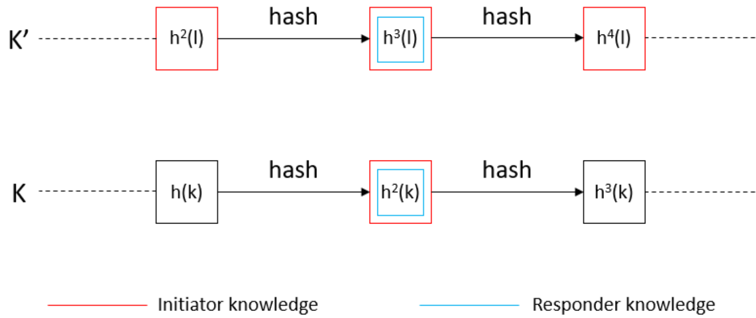


Figure 4.4: Knowledge of initiator and responder after the target protocol run

lemma to show that our protocol model is able to complete the target run. To this end, we define the lemma `executable` that uses the action facts `I1`, `R2`, `I3`... with a monotonically increasing timestamps. This confirms that there is at least one trace found which follows the logical order of the messages exchanged in SAKE such that `I1` occurs before `R2`, `R2` occurs before `I3`, and so on. Such existence proofs ensure that our lemmas are not vacuous truths and act as a sanity check that the model is actually executable. Following the analysis steps illustrated in Chapter 3, the existence of a trace is found as shown in Figure 4.5. Since our modeling is correct, by default, Tamarin shows a trace diagram on the right for which these action facts appear in the desired timepoint order. While Tamarin searches for the required trace, it cycles through other traces covered by each of the previous cases. When clicked, Tamarin simply displays a trace diagram for which the corresponding action facts appear in a rule, not necessarily in the desired order.

Snippet 4.9 Executablity check for the model

```
lemma executable:
exists-trace
"(Ex i r m1 mr mi trr tii #i #j #k #l #m #n.
   I1(i, r, m1)@i
 & R2(i, r, m1, mr)@j & #i < #j
 & I3(i, r, m1, mr, mi)@k & #j < #k
 & R4(i, r, m1, mr, mi, trr)@l & #k < #l
 & I5(i, r, m1, mr, mi, trr, tii)@m & #l < #m
 & R6(i, r, m1, mr, mi, trr, tii)@n & #m < #n
 )"
```

Next, we will verify the basic property of session key secrecy, with respect to both the initiator and the responder. In other words, whenever a party 'accepts'

```
lemma executable:
  exists-trace
  "∃ i r ml mr mi trr tii #i.1 #j #k #l #m #n.
         (((((((((I1( i, r, ml ) @ #i.1) ∧ (R2( i, r, ml, mr ) @ #j)) ∧
                 (#i.1 < #j)) ∧
                (I3( i, r, ml, mr, mi ) @ #k)) ∧
               (#j < #k)) ∧
              (R4( i, r, ml, mr, mi, trr ) @ #l)) ∧
             (#k < #l)) ∧
            (I5( i, r, ml, mr, mi, trr, tii ) @ #m)) ∧
           (#l < #m)) ∧
          (R6( i, r, ml, mr, mi, trr, tii ) @ #n)) ∧
         (#m < #n)"
simplify
solve( AgStI( $I, $R, <~k, ~l, h(~l), h(h(~l))> ) ▶₀ #i )
  case setup
  solve( AgStR( $R, <h(~k.1), h(h(~l.1))> ) ▶₀ #j )
    case setup
    solve( AgStI1( $I, $R, <$I, ~ri>, ~ri,
                   <~k.6, ~l.1, h(~l.1), h(h(~l.1))>
           ) ▶₀ #k.5 )
      case I1
      solve( AgStR2( $I, $R, <$I, ~ri>,
                     <~rr, mac(h(h(~l)), <$R, $I, ~rr, ~ri>)>, ~ri, ~rr,
                     <h(~k.3), h(h(~l))>
             ) ▶₀ #l.1 )
        case R2
        solve( AgStI3( $I, $R, <$I, ~ri>,
                       <~rr, mac(h(h(~l)), <$R, $I, ~rr, ~ri>)>,
                       <~ep, mac(h(h(~l)), <~ep, $I, $R, ~ri, ~rr>)>, ~ri,
                       ~rr, ~ep.1,
                       <h(h(~k.1)), h(h(~l)), h(h(h(~l))), h(h(h(h(~l))))>, sk
               ) ▶₀ #m )
          case I3
          solve( AgStR4( $I, $R, <$I, ~ri>,
                         <~rr, mac(h(h(~l)), <$R, $I, ~rr, ~ri>)>,
                         <~ep, mac(h(h(~l)), <~ep, $I, $R, ~ri, ~rr>)>,
                         mac(h(h(h(~l))), <~rr, ~ri>), ~ri, ~rr,
                         <h(h(~k.1)), h(h(h(~l)))>, sk
                 ) ▶₀ #n )
            case R4
            solve( !KU( ~ri ) @ #vk.2 )
              case I1
              solve( !KU( ~rr ) @ #vk.4 )
                case R2
                solve( !KU( ~ep ) @ #vk.7 )
                  case I3
                  solve( !KU( mac(h(h(~l)), <$R, $I, ~rr, ~ri>) ) @ #vk.6 )
                    case R2
                    solve( !KU( mac(h(h(~l)), <~ep, $I, $R, ~ri, ~rr>)
                           ) @ #vk.8 )
                      case I3
                      solve( !KU( mac(h(h(h(~l))), <~rr, ~ri>) ) @ #vk.9 )
                        case R4
                        solve( !KU( mac(h(h(h(~l))), <~ri, ~rr>) ) @ #vk.10 )
                          case I5
                          SOLVED // trace found
```

Figure 4.5: Tamarin proof: overview of the executability lemma

(represented with `AcceptsI` and `AcceptsR` action facts), it deems the session key derived in the corresponding protocol run to be safe from the adversary. Since we want this property to hold in all possible traces of the protocol, we use the '`all-traces`' keyword. The respective session key secrecy lemmas are written in Snippet 4.10, and the proofs are summarized in Figure 4.6 and Figure 4.7.

Let us dive further into one of the two proofs for the initiator. If we click on the `solve` statement corresponding to the fact `AgStI3`, Tamarin shows a graph in which it searches for a counter-example that has `AcceptsI` and `K(sk)` action facts, the only source of `AcceptsI` action being the rule `I5`. However, it is not able to find such a trace, as again evident in the graphs shown under each of the sub-cases. The other proof follows the same logic for the responder.

Snippet 4.10 Session key secrecy *wrt.* the initiator and the responder

```
lemma sessionKeySecrecyI:
all-traces
"All i r sk ri rr #i.
  AcceptsI(i, r, sk, <ri, rr>)@i ==>
  not (Ex #j. K(sk)@j)"

lemma sessionKeySecrecyR:
all-traces
"All i r sk ri rr #i.
  AcceptsR(i, r, sk, <ri, rr>)@i ==>
  not (Ex #j. K(sk)@j)"
```

However, our point of interest in such key evolving schemes is the provision of PFS for the derived session key. Specifically, we want to prove that the session key is safe from the adversary, even if it corrupts one of the parties after either accepts. Here, we make use of the action facts `KeyI1`, `KeyR2`... to represent key knowledge with the parties at different stages of the protocol execution. The PFS lemmas with respect to the initiator and the responder are written in Snippet 4.11. For instance, in lemma `skPFSI`, we intend to show that if I 'accepts', the session key is safe, or otherwise the derivation master keys stored with I or R have been previously compromised. In our case with I being one step behind, the facts `KeyI1` and `KeyI3` correspond to key `k`, while `KeyR2` and `KeyR4` correspond to key `h(k)`, both the keys enabling the adversary to derive the session key. The overview of PFS proofs are respectively shown in Figure 4.8 and Figure 4.9.

Similar to the session key secrecy proofs, let us look into the PFS proof for the initiator and the proof for the responder follows an analogous method. If we click on the `solve` statement corresponding to the fact `AgStI3`, Tamarin shows a graph

```
lemma sessionKeySecrecyI:
  all-traces
  "∀ i r sk ri rr #i.1.
         (AcceptsI( i, r, sk, <ri, rr> ) @ #i.1) →
         (¬(∃ #j. K( sk ) @ #j))"
simplify
solve( AgStI3( $I, $R, m1, mr, mi, ~ri, rr, ~ep,
               <h(h(~k)), h(h(~l)), h(h(h(~l))), h(h(h(h(~l))))>, sk
       ) ▶₀ #i )
  case I3
  solve( !KU( kdf(h(~k), opr(~ri, rr)) ) @ #vk.1 )
    case c_kdf
    solve( !KU( h(~k) ) @ #vk.5 )
      case c_h
      by solve( !KU( ~k ) @ #vk.7 )
    qed
  qed
qed
```

Figure 4.6: Tamarin proof: session key secrecy lemma *wrt*. the initiator

```
lemma sessionKeySecrecyR:
  all-traces
  "∀ i r sk ri rr #i.1.
         (AcceptsR( i, r, sk, <ri, rr> ) @ #i.1) →
         (¬(∃ #j. K( sk ) @ #j))"
simplify
solve( AgStR4( i.1, $R, m1, mr, mi, trr, ri, ~rr,
               <h(h(~k)), h(h(h(~l)))>, sk
       ) ▶₀ #i )
  case R4
  solve( !KU( kdf(h(~k), opr(ri, ~rr)) ) @ #vk.1 )
    case c_kdf
    solve( !KU( h(~k) ) @ #vk.8 )
      case c_h
      by solve( !KU( ~k ) @ #vk.10 )
    qed
  qed
qed
```

Figure 4.7: Tamarin proof: session key secrecy lemma *wrt*. the responder

in which it searches for a counter-example that has `AcceptsI` and `K(sk)` actions, without using any of `KeyI1`, `KeyR2`, `KeyI3` or `KeyR4` action facts. Additionally, below the graph, we see four formulas signifying that occurrence of any of these will lead to a contradiction. However, yet again, Tamarin is not able to find such a trace, as any of the constructed graphs under each of the sub-cases leads to the contradiction.

Snippet 4.11 Session key PFS *wrt.* the initiator and the responder

```
lemma skPFSI:
all-traces
"All i r sk ri rr #i.
  AcceptsI(i, r, sk, <ri, rr>)@i ==>
  (not (Ex #j. K(sk)@j))
      | (Ex k #z. KeyI1(i, k)@z & K(k)@z & z < i)
      | (Ex k #z. KeyR2(r, k)@z & K(k)@z & z < i)
      | (Ex k #z. KeyI3(i, k)@z & K(k)@z & z < i)
      | (Ex k #z. KeyR4(r, k)@z & K(k)@z & z < i)
 "
lemma skPFSR:
all-traces
"All i r sk ri rr #i.
  AcceptsR(i, r, sk, <ri, rr>)@i ==>
  (not (Ex #j. K(sk)@j))
      | (Ex k #z. KeyI1(i, k)@z & K(k)@z & z < i)
      | (Ex k #z. KeyR2(r, k)@z & K(k)@z & z < i)
      | (Ex k #z. KeyI3(i, k)@z & K(k)@z & z < i)
      | (Ex k #z. KeyR4(r, k)@z & K(k)@z & z < i)
 "
```

Finally, we will test the property of synchronization robustness. For SAKE, the property is fulfilled iff after the target run, the communicating parties end up with their internal states synchronized, similar to what is shown in Figure 4.4. In other words, after the protocol is complete, if there exists a trace in which the derivation master keys with both parties are not the same and the authentication master key with the responder is not same as the middle authentication master key with the initiator, then it signifies an attack against the desired property. The corresponding lemma is written in Snippet 4.12, where `EvolvedI` and `EvolvedR` are used to compare the keys stored with the parties after completion of the target run.

Figure 4.10 shows that, indeed, Tamarin is able to find such a trace, thereby indicating that SAKE is not synchronization robust. The whole trace diagram can be seen after the proof is completed, which roughly deconstructs as follows. Tamarin starts two parallel sessions between two different pairs of initiator and responder oracles. The first session follows the protocol steps until rule `I5` is executed, after

```
lemma skPFSI:
  all-traces
  "∀ i r sk ri rr #i.1.
         (AcceptsI( i, r, sk, <ri, rr> ) @ #i.1) ⇒
         (((((¬(∃ #j. K( sk ) @ #j)) ∨
             (∃ k #z.
               ((KeyI1( i, k ) @ #z) ∧ (K( k ) @ #z)) ∧ (#z < #i.1))) ∨
            (∃ k #z.
              ((KeyR2( r, k ) @ #z) ∧ (K( k ) @ #z)) ∧ (#z < #i.1))) ∨
           (∃ k #z.
             ((KeyI3( i, k ) @ #z) ∧ (K( k ) @ #z)) ∧ (#z < #i.1))) ∨
          (∃ k #z.
            ((KeyR4( r, k ) @ #z) ∧ (K( k ) @ #z)) ∧ (#z < #i.1)))"
simplify
solve( AgStI3( $I, $R, ml, mr, mi, ~ri, rr, ~ep,
               <h(h(~k)), h(h(~l)), h(h(h(~l))), h(h(h(h(~l))))>, sk
       ) ▶₀ #i )
  case I3
  solve( !KU( kdf(h(~k), opr(~ri, rr)) ) @ #vk.1 )
    case c_kdf
    solve( !KU( h(~k) ) @ #vk.5 )
      case c_h
      by solve( !KU( ~k ) @ #vk.7 )
    qed
  qed
qed
```

Figure 4.8: Tamarin proof: session key PFS lemma *wrt*. the initiator

which it is aborted, and the another session follows all the steps through to the end. As a result, both `EvolvedI` and `EvolvedR` action facts appear in the trace at some timepoints and the target run is completed. However, the derivation master key with the `EvolvedI` fact in the first session is not the same as the derivation master key with the `EvolvedR` fact in the second session. This stems from the fact that SAKE was not designed to handle concurrent protocol runs. One of the sessions will have its keys evolved first, and due to the key evolution in the other session, these action facts will be associated with different derivation master keys, thereby proving the lemma. Similar conclusion can be drawn for the authentication master keys.

Snippet 4.12 Synchronization robustness of the SAKE protocol

```
lemma SyncLossConcurrent:
exists-trace
"Ex i r k l m n o p #i #j.
  EvolvedI(i, r, <k, l, m, n>)@i
  & EvolvedR(r, <o, p>)@j
  & not((k = o) & (m = p))"
```

```
lemma skPFSR:
  all-traces
  "∀ i r sk ri rr #i.1.
        (AcceptsR( i, r, sk, <ri, rr> ) @ #i.1) ⇒
        (((((¬(∃ #j. K( sk ) @ #j)) ∨
            (∃ k #z.
              ((KeyI1( i, k ) @ #z) ∧ (K( k ) @ #z)) ∧ (#z < #i.1))) ∨
           (∃ k #z.
             ((KeyR2( r, k ) @ #z) ∧ (K( k ) @ #z)) ∧ (#z < #i.1))) ∨
          (∃ k #z.
            ((KeyI3( i, k ) @ #z) ∧ (K( k ) @ #z)) ∧ (#z < #i.1))) ∨
         (∃ k #z.
           ((KeyR4( r, k ) @ #z) ∧ (K( k ) @ #z)) ∧ (#z < #i.1)))"
simplify
solve( AgStR4( i.1, $R, ml, mr, mi, trr, ri, ~rr,
               <h(h(~k)), h(h(h(~l)))>, sk
       ) ▶₀ #i )
  case R4
  solve( !KU( kdf(h(~k), opr(ri, ~rr)) ) @ #vk.1 )
    case c_kdf
    solve( !KU( h(~k) ) @ #vk.8 )
      case c_h
      by solve( !KU( ~k ) @ #vk.10 )
    qed
  qed
qed
```

Figure 4.9: Tamarin proof: session key PFS lemma *wrt*. the responder

4.2.3 Discussion

Intuitively, for an initial state difference within one step, if multiple sessions run in parallel, each of them can be altered such that after processing of the message `mr` at the initiator, remaining messages are dropped. This might result in a state difference of more than two steps between the initiator and the responder. Consequently, the protocol logic breaks trivially as it is based on the assumption that the state difference is bounded by one step. Disallowing concurrent sessions, although required and deliberately chosen by the protocol designers, will weaken the security model and undermine adversarial capabilities. In essence, the synchronization robustness property incorporates concurrent sessions, and assures that all the traces end up with the parties being synchronized. As proven by Tamarin and shown in Figure 4.10, SAKE does not fulfil this property. For a complete diagram of the trace found as a counterexample, please run the protocol models [sakemodel] in Tamarin. Although the modeling and analysis presented in this chapter is pertaining to the case where the initiator is one step behind the responder at the start of the protocol, other cases where the initiator is in sync with or one step behind the responder follow the same proofs. The change in sequence or number of updation of keys does not affect the temporal logic of the action facts used to prove the desired security properties.

```
lemma SyncLossConcurrent:
  exists-trace
  "∃ i r k l m n o p #i.1 #j.
        ((EvolvedI( i, r, <k, l, m, n> ) @ #i.1) ∧
         (EvolvedR( r, <o, p> ) @ #j)) ∧
        (¬((k = o) ∧ (m = p)))"
simplify
solve( (¬(h(h(~k)) = h(h(~k.1)))) ∥
       (¬(h(h(h(~l))) = h(h(h(~l.1))))) )
  case case_1
  solve( AgStI3( $I, $R, ml, mr, mi, ~ri, rr, ~ep,
                 <h(h(~k)), h(h(~l)), h(h(h(~l))), h(h(h(h(~l))))>, sk
         ) ▶₀ #i )
    case I3
    solve( AgStR4( i.1, $R, ml, mr, mi, trr, ri.1, ~rr.1,
                   <h(h(~k.1)), h(h(h(~l.1)))>, sk
           ) ▶₀ #j )
      case R4
      solve( !KU( mac(h(h(h(~l))), <rr, ~ri>) ) @ #vk )
        case R4
        solve( !KU( mac(h(h(h(~l.1))), <ri.1, ~rr.1>) ) @ #vk.1 )
          case I5
          solve( !KU( mac(h(h(~l)), <$R, $I, ~rr, ~ri>) ) @ #vk.4 )
            case R2
            solve( !KU( mac(h(h(~l.1)), <ep.1, i.1, $R, ~ri.1, ~rr.1>)
                   ) @ #vk.7 )
              case I3
              solve( !KU( mac(h(h(~l)), <ep.2, $I, $R, ~ri, ~rr>)
                     ) @ #vk.13 )
                case I3
                solve( !KU( ~ri ) @ #vk.16 )
                  case I1
                  solve( !KU( ~rr ) @ #vk.9 )
                    case R2
                    solve( !KU( ~rr.1 ) @ #vk.19 )
                      case R2
                      solve( !KU( ~ri.1 ) @ #vk.15 )
                        case I1
                        solve( !KU( ~ep.1 ) @ #vk.14 )
                          case I3
                          solve( !KU( ~ep ) @ #vk.17 )
                            case I3
                            solve( !KU( mac(h(h(h(~l.1))), <~rr.1, ~ri.1>) ) @ #vk.19 )
                              case R4
                              solve( !KU( mac(h(h(~l.1)), <$R, $I.1, ~rr.1, ~ri.1>)
                                     ) @ #vk.20 )
                                case R2
                                SOLVED // trace found
```

Figure 4.10: Tamarin proof: existence of an attack against synchronization robustness

Chapter 5
Conclusion and Future Work

5.1 Remarks

Formal analysis has proven to be indispensable in verifying security of cryptographic protocols. It asserts soundness of the protocol in question and is increasingly becoming a standard requirement with the protocol design. Not only it serves as a universal language to scalably automate proofs, but also allows defining adversarial capabilities in accordance with the security model, thereby covering corner cases with multiple parallel protocol runs without the risk of manual errors.

In this thesis, we presented a rigorous formal analysis of the SAKE protocol using Tamarin Prover. Summarizing our findings, SAKE, as an AKE protocol, provides session key secrecy, as well as authentication using MACs in the exchanged messages. Additionally, using a simple key-evolving technique, it provides PFS for the derived session keys. However, the protocol is not synchronization robust, as Tamarin finds an attack resulting in the internal states of the participants being desynchronized after the target protocol run is completed. The participants of the protocol are able to synchronize their states only if parallel execution of sessions is prohibited. Furthermore, for the readers intending to work in and pick up formal analysis, we have delineated a concise and comprehensive guide to using Tamarin as a verification tool, besides reviewing the literature pertaining to key exchange protocols and present-day advancements in formal verification.

5.2 Achieving Full Synchronization Robustness

Only recently, AKE protocols providing forward security based on symmetric primitives have been proposed. Naturally, they are few in number and differ in efficiency in terms of computational and storage load on end devices. Several strategies are adopted to evolve the long-term keys. A protocol may update their keys based on time [DJ14], and therefore, require the clocks of the participants to be perfectly synchronized, which may incur additional costs. Other designs may use a separate

counter to keep track of the internal state of a participant [BP10]. In order to synchronize, these counters are exchanged during the run of the protocol. Unfortunately, few of these attempts at making a forward secure AKE protocol assumes one of the parties to be incorruptible [VBdM07; BP10]. This impractically weaken the security model, rendering the protocol secrets vulnerable to be compromised in a real setting.

[BDdK+21] presented a set of lightweight symmetric-key based AKE protocols that attain PFS without any such assumptions. The authors formally defined the property of synchronization robustness and proposed a set of protocols that achieve correctness even if concurrent sessions are allowed. While the linear key evolving schemes make use of counters to synchronize the communicating parties, the non-linear key evolving schemes use PPRFs to realize full synchronization robustness, and unlike SAKE, ensure that the internal states of the participants of the target session are efficiently re-synchronized, offering stronger security guarantees.

Formal analysis of the protocols demonstrated in [BDdK+21] will help in confirming the claimed security properties for yet another set of lightweight schemes providing PFS with symmetric-key cryptography, therefore, being able to be efficiently used by constrained devices. More importantly, it may help in formally defining and standardizing the property of synchronization robustness for future protocol designers, so that an adversary is not able to maliciously de-synchronize the internal states of the participants. With the use of counters, modeling these protocols in Tamarin may require multiple updates of the derivation master key at once. This, to the best of our knowledge, necessitates the construction of loops using Tamarin language. If at all possible, this is non-trivial, given the tool's features, and might serve as an interesting future extension of the analysis presented in this work.

References

[ABB+05] A. Armando, D. Basin, *et al.*, «The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications», in *Computer Aided Verification*, K. Etessami and S. K. Rajamani, Eds., Springer, 2005, pp. 281–285.

[ACF20] G. Avoine, S. Canard, and L. Ferreira, «Symmetric-Key Authenticated Key Exchange (SAKE) with Perfect Forward Secrecy», in *Topics in Cryptology – CT-RSA 2020, San Francisco, CA, USA*, S. Jarecki, Ed., Springer, 2020, pp. 199–224.

[AIM10] L. Atzori, A. Iera, and G. Morabito, «The Internet of Things: A survey», *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[AIAA21] B. Alzahrani, A. Irshad, *et al.*, «A Provably Secure and Lightweight Patient-Healthcare Authentication Protocol in Wireless Body Area Networks», *Wireless Personal Communications*, vol. 117, Mar. 2021.

[BAN90] M. Burrows, M. Abadi, and R. Needham, «A logic of authentication», *ACM Trans. Comput. Syst.*, vol. 8, no. 1, pp. 18–36, Feb. 1990. [Online]. Available: https://doi.org/10.1145/77648.77649.

[BCDS17] D. Basin, C. Cremers, *et al.*, «Symbolically Analyzing Security Protocols Using Tamarin», vol. 4, no. 4, pp. 19–30, Nov. 2017.

[BDdK+21] C. Boyd, G. T. Davies, *et al.*, *Symmetric key exchange with full forward security and robust synchronization*, Cryptology ePrint Archive, Report 2021/702, https://ia.cr/2021/702, 2021.

[BDG+14] G. Barthe, F. Dupressoir, *et al.*, *EasyCrypt: A Tutorial*, A. Aldini, J. Lopez, and F. Martinelli, Eds. Springer, 2014, pp. 146–166.

[BGH+92] R. Bird, I. Gopal, *et al.*, «Systematic design of two-party authentication protocols», in *Advances in Cryptology — CRYPTO '91*, J. Feigenbaum, Ed., Springer, 1992, pp. 44–61.

[BJS16] C. Brzuska, H. Jacobsen, and D. Stebila, «Safely exporting keys from secure channels», in *Advances in Cryptology – EUROCRYPT 2016*, M. Fischlin and J.-S. Coron, Eds., Springer, 2016, pp. 670–698.

[Bla12] B. Blanchet, «Security Protocol Verification: Symbolic and Computational Models», in *Principles of Security and Trust*, P. Degano and J. Guttman, Eds., Springer, 2012, pp. 3–29.

[Bla16] ——, «Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif», *Foundations and Trends in Privacy and Security*, vol. 1, pp. 1–135, Oct. 2016.

[BMV03] D. Basin, S. Mödersheim, and L. Viganò, «An On-the-Fly Model-Checker for Security Protocol Analysis», in *Computer Security – ESORICS 2003, Gjøvik, Norway*, E. Snekkenes and D. Gollmann, Eds., Springer, 2003, pp. 253–270.

[BP10] E. Brier and T. Peyrin, «A forward-secure symmetric-key derivation protocol», in *Advances in Cryptology - ASIACRYPT 2010*, M. Abe, Ed., Springer, 2010, pp. 250–267.

[BR94] M. Bellare and P. Rogaway, «Entity authentication and key distribution», in *Advances in Cryptology — CRYPTO' 93*, D. R. Stinson, Ed., Springer, 1994, pp. 232–249.

[BS19] S. Bojjagani and V. Sastry, «A secure end-to-end proximity NFC-based mobile payment protocol», *Computer Standards & Interfaces*, vol. 66, p. 103 348, 2019.

[BSGS18] A. Bruni, T. Sahl Jørgensen, *et al.*, «Formal Verification of Ephemeral Diffie-Hellman Over COSE (EDHOC)», in *Security Standardisation Research - 4th International Conference, 2018, Darmstadt, Germany*, C. Cremers and A. Lehmann, Eds., Springer, 2018, pp. 21–36.

[CHSvdM16] C. Cremers, M. Horvat, *et al.*, «Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication», in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 470–485.

[CK01] R. Canetti and H. Krawczyk, «Analysis of key-exchange protocols and their use for building secure channels», in *Advances in Cryptology — EUROCRYPT 2001*, B. Pfitzmann, Ed., Springer, 2001, pp. 453–474.

[CV01] Y. Chevalier and L. Vigneron, «A Tool for Lazy Verification of Security Protocols», Jan. 2001, pp. 373–376.

[DH76] W. Diffie and M. Hellman, «New directions in cryptography», *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[DJ14] M. S. Dousti and R. Jalili, «Forsakes: A forward-secure authenticated key exchange protocol based on symmetric key-evolving schemes», *IACR Cryptology ePrint Archive*, vol. 2014, p. 123, 2014. [Online]. Available: https://eprint.iacr.org/2014/123.

[DS81] D. E. Denning and G. M. Sacco, «Timestamps in key distribution protocols», *Commun. ACM*, vol. 24, no. 8, pp. 533–536, Aug. 1981. [Online]. Available: https://doi.org/10.1145/358722.358740.

[DvOW92] W. Diffie, P. van Oorschot, and M. Wiener, «Authentication and Authenticated Key Exchanges», in *Designs, Codes and Cryptography 2*, Springer, 1992, pp. 107–125.

[DY83] D. Dolev and A. C. Yao, «On the security of public key protocols», *IEEE Trans. Inf. Theory*, vol. 29, no. 2, pp. 198–207, 1983.

[GMS15] J. Granjal, E. Monteiro, and J. Sá Silva, «Security for the Internet of Things: A Survey of Existing Protocols and Open Research Issues», *IEEE Commun. Surv. Tutorials*, vol. 17, no. 3, pp. 1294–1312, 2015.

[HS20] K. Hofer-Schmitz and B. Stojanović, «Towards formal verification of IoT protocols: A Review», *Comp. Networks*, vol. 174, p. 107 233, 2020.

[Kab21] A. Kabra, *Automated Security Analysis of Lightweight Authentication and Key Exchange Protocols*, Dec. 2021.

[KHHJ17] J. Y. Kim, R. Holz, *et al.*, «Automated Analysis of Secure Internet of Things Protocols», in *Proceedings of the 33rd Annual Comp. Security Applications Conference, Orlando, FL, USA*, ACM, 2017, pp. 238–249.

[KNT20] N. Kobeissi, G. Nicolas, and M. Tiwari, «Verifpal: Cryptographic Protocol Analysis for the Real World», in *Progress in Cryptology – INDOCRYPT 2020*, K. Bhargavan, E. Oswald, and M. Prabhakaran, Eds., Springer, 2020, pp. 151–202.

[LLM07] B. LaMacchia, K. Lauter, and A. Mityagin, «Stronger Security of Authenticated Key Exchange», in *Provable Security*, W. Susilo, J. K. Liu, and Y. Mu, Eds., Springer, 2007, pp. 1–16.

[LPH20] L. Li, P. Podder, and E. Hoque, «A Formal Security Analysis of ZigBee (1.0 and 3.0)», in *Proceedings of the 7th Symposium on Hot Topics in the Science of Security*, Lawrence, Kansas: ACM, 2020.

[MCN+18] K. Mahmood, S. A. Chaudhry, *et al.*, «An elliptic curve cryptography based lightweight authentication scheme for smart grid communication», *Future Gener. Comput. Syst.*, vol. 81, pp. 557–565, 2018.

[NS78] R. M. Needham and M. D. Schroeder, «Using encryption for authentication in large networks of computers», *Commun. ACM*, vol. 21, no. 12, pp. 993–999, Dec. 1978. [Online]. Available: https://doi.org/10.1145/359657.359659.

[NSB21] K. Norrman., V. Sundararajan., and A. Bruni, «Formal Analysis of EDHOC Key Establishment for Constrained IoT Devices», in *Proceedings of the 18th International Conference on Security and Cryptography - SECRYPT,*, SciTePress, 2021, pp. 210–221.

[sakemodel] Tamarin models for possible initiator-responder initial state differences. [Online]. Available: https://github.com/adhirathkabra/sakemodels.git (last visited: Jun. 12, 2022).

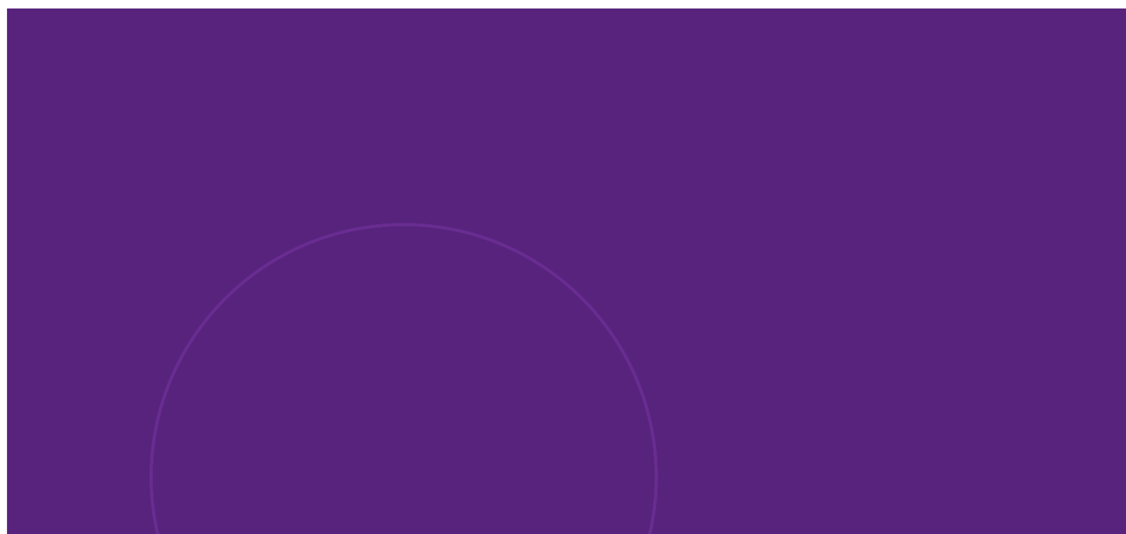
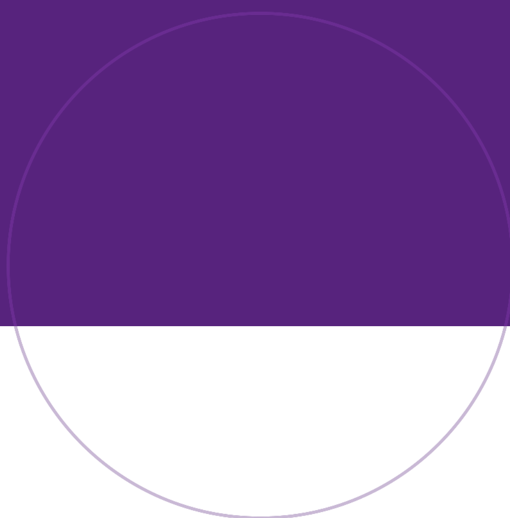
[SMP21] G. Selander, J. P. Mattsson, and F. Palombini, «Ephemeral Diffie-Hellman Over COSE (EDHOC)», IETF, Internet-Draft draft-ietf-lake-edhoc-12, Oct. 2021, Work in Progress, 80 pp.

[Tea] T. T. Team, Tamarin-Prover Manual: Security Protocol Analysis in the Symbolic Model, [Online]. Available: https://tamarin-prover.github.io/manual/index.html, (last visited: Nov. 8, 2021).

[VBdM07] T. Van Le, M. Burmester, and B. de Medeiros, «Universally composable and forward-secure rfid authentication and authenticated key exchange», in *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '07, Singapore: Association for Computing Machinery, 2007, pp. 242–252. [Online]. Available: https://doi.org/10.1145/1229285.1229319.

[WLX+21] F. Wu, X. Li, *et al.*, «A Novel Three-Factor Authentication Protocol for Wireless Sensor Networks With IoT Notion», *IEEE Systems Journal*, vol. 15, no. 1, pp. 1120–1129, 2021.

[AA20] A. M. Almuhaideb and K. S. Alqudaihi, «A Lightweight Three-Factor Authentication Scheme for WHSN Architecture», *Sensors*, vol. 20, no. 23, 2020.

NTNU

Norwegian University of
Science and Technology