BACHELOR'S THESIS COMPUTING SCIENCE

Automated Formal Proofs, Verification and Analysis of Security Protocols

WOUTER DOELAND
s1034816

June 1, 2023

Supervisor:
Security Engineer, Security Company

First assessor:
Dr. Bart Mennink, iCIS

Second assessor:
Dr. Simona Samardjiska, iCIS

Radboud University

Abstract

Computer-aided cryptography is a field that applies automated tools to the design and implementation of security protocols. Automated tools for the analysis of protocol designs have improved a lot over the years and have proven to be very valuable in the design and analysis of large protocols such as TLS 1.3 [19] and Signal [15]. In this thesis we discuss the available methods and tools for analyzing protocol design. We explain the notation of the tool Tamarin and we provide a method for formalizing, modeling and analysing protocols in this tool. Finally we apply this method to a certain proprietary protocol and find and fix a vulnerability in the protocol design.

This thesis was written at and supported by a company specializing in secure communication products. The research in this thesis was also applied to an in-development secure communication product from the company. Due to Non-Disclosure Agreements we further refer to them as Security Company and Secure Communication Product.

Contents

Chapter 1

Introduction

Security Company produces devices for secure communications. These communications are secured by protocols using cryptography. It is well known that developing secure products is hard. There can be faults in many areas such as protocol design, software implementation and hardware design. There are many cases of supposedly secure products being broken and usually these faults are only found after a thorough manual analysis. Security Company wants to know if it is possible to analyse the security of protocol designs through a more formal and automated analysis. A lot of the protocols running on the company's devices are developed in-house and are not an open standard. This is because the protocols have special use cases that are not covered by standard protocols. These protocols have to be secure, and to analyse the security of the protocol design an investigation into the design is conducted. Automated tools for the formal analysis of protocols are not yet used. These tools can help in finding vulnerabilities in protocol designs and make the designs even more secure. Formal analysis of protocols is also a requirement for high levels of security certification, for example for higher levels of Common Criteria certification [16]. It is therefore interesting to look at the possibilities of these automated security analysis tools.

The goal of this thesis is to provide a method for analysing security protocols through existing automated tools and to show that this method is useful by applying it to a protocol designed by Security Company. The protocol we analyse is the session key exchange of Secure Communication Product. We specifically analyse the secrecy of the keys that are exchanged.

We first start in Chapter 2 by investigating existing analysis methods and available tools. There we conclude that Tamarin [25] is the most suited tool for the analysis of the Secure Communication Product protocol as it is a mature tool that provides high levels of automation and allows us to model the protocol due to its extensive modeling capabilities and its support for the exclusive-or operator. In Chapter 3 we explain the notation of the Tamarin tool and we continue by developing a method for formalizing a protocol in

Chapter 4. This method explains how to draw a protocol diagram from a protocol design document and it explains how to transform this protocol diagram into a Tamarin model. After we explain how to create a Tamarin model, we go into how to analyse such a model in Chapter 5. We provide examples of properties to prove. In Chapter 6 we apply these methods to the Secure Communication Product protocol to develop a protocol diagram, Tamarin model and lemma's to prove. We analyse the protocol. The analysis found a vulnerability in the protocol. We apply a fix to the protocol and verify that this fix removes the vulnerability. We end by stating our conclusions and providing topics for future work in Chapter 7.

Chapter 2

Methods and Tools

There are several methods to analyse the security of communication protocol designs. Two methods are analysis of symbolic security and analysis of computational security [4]. In this Chapter we give an explanation of these methods and discuss some of the available tools. We explain why we select Tamarin [25] as the tool that we use in this thesis.

In this Chapter we only look at automated tools. We believe that automated tools instead of manual tools are the way forward because they allow modelling much larger protocols and enable rapid testing of redesigns. This is useful when an attack on a protocol is found and designers want to test a solution. This is also useful if protocol requirements have changed and a protocol redesign is necessary. It is then possible to analyse the security of the design again without having to redo the entire security proof. This allows a cycle of ‘breaking, fixing and verifying’ protocols [5, 6, 19, 23].

2.1 Symbolic Security Analysis

When applying symbolic security analysis, security protocols are analysed in an abstract way. Instead of defining messages as bit strings, they are defined as atomic terms. Since these terms are atomic they cannot be split into smaller parts. This means that an adversary cannot partially decrypt or guess some message, they can only know the whole message or not know the message at all. In the symbolic model, functions work on these atomic terms as black-box functions. This also means that, unlike the real world, cryptographic primitives are perfect. This makes reasoning about the security of a protocol design easier and enables (faster) automation of verification and proofs [4], allowing us to model large and complete protocols.

Functions are modelled using equational theory. An example of the definition of a function would be: $Dec(Enc(m, k), k) = m$. This definition states that to retrieve the message m from an encrypted message $Enc(m, k)$ the function $Dec(Enc(m, k), k)$ with key k must be applied. This is a very simple method of defining (cryptographic) functions and is implied to be perfectly secure. This is the case because there is no function that recovers the message m from the encrypted message $Enc(m, k)$ without the use of the key k [21]. Adversaries are restricted to using this knowledge of equational theories. It is important that all equations are modelled properly, as otherwise the adversary is significantly less capable than in the real world. The adversary in the symbolic model is already less capable than in the real world, because an adversary needs to know the full term (and not a partial term) before it can be used in a function. Again: only if the adversary knows the full key k, are they able to decrypt an encrypted message. These simplifications enable the use of symbolic logic to verify and analyse protocol designs in an automated way. The adversary in the symbolic model has full network capabilities and can read, halt, modify and insert messages on the network. This is also known as the Dolev-Yao adversary model [21].

There are two main types of security properties that we analyse: trace properties and equivalence properties. Analysing trace properties means analysing that an unwanted event never occurs. For example: we want there to be no occurrence of the adversary knowing the session key. Analysing equivalence properties means making sure that the adversary is unable to distinguish between two protocols. One of these protocols is the actual protocol design using 'real' values and equations and the other is the same protocol sending random data. This can be especially useful in proving privacy properties such as in e-voting systems [20].

Common tools that are used in symbolic security analysis are Tamarin [25] and ProVerif [10]. We explain how these tools work and we discuss their advantages, disadvantages and real-world usage. We also briefly discuss DEEPSEC and SAPIC+.

2.1.1 Tamarin

Tamarin [25] is a symbolic security analysis tool that allows modelling a wide range of security protocols [7]. It has been used to analyse (parts of) protocols such as TLS 1.3 [19], WireGuard [22], 5G-AKA [6, 18] and EMV [5]. A major advantage of Tamarin is that it is able to model and automatically analyse and verify large and complete protocols. This is due to its wide range of capabilities and the fact that it uses the symbolic model. An example of such a large and complete protocol is the EMV standard: the analysis of the EMV standard includes backwards compatibility, support for contactless payments, online and offline payments and difference in Visa and MasterCard schemes [5]. When analysing these protocols, researchers also tried to find and apply fixes to make the protocol more secure. This was done in the design phase of TLS 1.3 and it contributed to the security of the protocol [19].

Tamarin works using multiset rewriting rules. These rewriting rules work on the protocol participants' states, messages on the network, fresh terms and adversary knowledge [7]. For security properties (either as trace properties or as observational equivalence properties with the SAPIC extension) Tamarin can find a counter-example or construct a proof, but only if it terminates.

Tamarin also provides a web interface for interactive proofs and we consider the multi-set rewriting language fairly easy to use. A disadvantage of Tamarin is that it can sometimes take more effort to automatically prove certain properties and that solving proofs can be slower than alternatives such as ProVerif [13].

2.1.2 ProVerif

ProVerif is a symbolic security analysis tool that allows modelling protocols in an applied process-calculus. A process-calculus is a tool for algebraically describing systems, which is useful for studying distributed systems with algebra [3]. ProVerif uses the existing applied pi-calculus [1] which supports function symbols that can model cryptographic primitives [9]. Just like Tamarin, it uses equational theory to define cryptographic primitives. In this tool, protocols are modelled as processes. It can be used to analyse secrecy and authentication properties and process equivalences. ProVerif supports equational theory for symmetric and asymmetric encryption, but due to its design it cannot support exclusive-or operations or Diffie-Hellman groups without taking some shortcuts [10]. It also does not have a global mutual state. It has been used to analyse a large number of protocols. For example the Signal protocol [23], WireGuard [24] and privacy properties in e-voting systems [20] have been analysed with ProVerif.

2.1.3 DEEPSEC

Tamarin (through the SAPIC extension [2]) and ProVerif already allow proving diff-equivalence properties, however they do not allow proving trace-equivalence properties. For many protocols, diff-equivalence is too strong and it finds many false attacks [12], as the adversary can see the internal state of honest protocol runs [11]. DEEPSEC allows proving trace-equivalence properties, which can be applied to more protocols. Trace-equivalence states that an adversary cannot distinguish between two systems based on the messages sent after interacting with it [14]. This is useful for proving the indistinguishability of certain properties, for example in e-voting. It has been used to analyse a number of protocols including the Helios e-voting protocol [12]. It uses a process-calculus similar to that of ProVerif.

2.1.4 SAPIC+

SAPIC+ is a tool that translates models and lemmas written in its process-calculus to Tamarin, ProVerif and DEEPSEC [13]. It combines the strengths of these tools to automatically construct proofs. It can use Tamarin's extensive modeling capabilities to prove most trace properties and it can use ProVerif for fast and more automated analysis. ProVerif and DEEPSEC can be used to prove equivalence properties. Not all equational theory can be modeled in all languages, for example the exclusive-or operator and Diffie-Hellman groups cannot be modeled in ProVerif and then abstractions are used to approach the results of this equational theory. SAPIC+ uses a process-calculus language similar to that of DEEPSEC and ProVerif.

Since the translations of SAPIC+ are provably correct, lemmas that are proven correct in one tool can be used as axioms in another tool. It is also possible to run tools in parallel, terminating when one of the tools finds a proof or attack.

2.2 Computational Security Analysis

In computational security analysis we analyse security protocols in a more concrete way. We define messages as bit strings and cryptographic primitives as probabilistic algorithms on these messages [4]. Adversaries are probabilistic Turing machines (which can generate perfectly random values and execute many algorithms), and security goals are defined in terms of the adversaries' probability of successfully reaching some goal. This can be defined in a concrete or asymptotic way. Concrete goals describe a maximum attack probability after attacking the protocol for some time. Asymptotic security goals take the attack probability and time as a function of security parameters such as the key length and set requirements based on these values.

The two common methods of analysing protocols in the computational security model are game-based and simulation-based experiments. Game-based experiments state that an adversary must achieve some goal condition against a challenger and will try to guide the adversary to that goal. Simulation-based properties run two versions of a scheme, one with the simulated cryptographic primitives and one with the ideal functionality. A protocol is considered secure if for every attack on the real version an attack on the ideal function is also found [4].

A downside of the computational security model is that it is hard to model complete protocols. It is possible if the protocol is modelled and analysed in smaller portions and then 'glued together' using composition techniques, however these composition techniques do not guarantee all security properties [4].

There are few automated tools available that focus on analysing security protocol designs. We briefly go into CryptoVerif.

2.2.1 CryptoVerif

CryptoVerif is a computational security analysis tool that works on constructing game-based proofs of concrete security properties [8]. It has been used to analyse properties of the Signal protocol [23] and WireGuard [24]. It uses a language similar to that of ProVerif.

2.3 Tool Selection

Tamarin and ProVerif are the most mature and widely used modern tools for protocol design analysis. We decide to use Tamarin because of its extensive modelling abilities, interactive theorem-prover, high level of automation and general user-friendliness. Tamarin allows us to model the protocol from Chapter 6 because it has support for exclusive-or operations, which many tools do not have.

Chapter 3

Notation

This chapter describes the notation of Tamarin. Tamarin uses a term rewriting system in the symbolic model and its notation reflects that [25]. We define functions, equational theories, rules and lemmas to model our protocol and security properties. Tamarin reads these from a text file and applies its constraint solving algorithm to provide an analysis of the security properties. In this chapter we describe how all these parts are defined and how they work. We do not go into the SAPIC [2] extension of Tamarin. Most of this chapter is based on the Tamarin manual [27].

3.1 Functions and Equational Theories

Functions define the cryptographic primitives of Tamarin. They are defined at the start of the theorem file, in the functions definition. The definition of a function states the symbol, its arity and whether it is private or not. The symbol is the name of the function, and the arity of a function is the amount of arguments that it takes. If a function defines no arity it is a constant. Adversaries can access all functions except those marked with `[private]`. Because of the modelling freedom of these functions, it is simple to model schemes for (a)symmetric encryption, signing and hashing. We also have the possibility to model the exclusive-or operator, Diffie-Hellman groups and more. Below we define a few functions to show the different capabilities of the equational theory in Tamarin. We define a hash function, a symmetric encryption and decryption function, a private function and functions for a signature scheme.

```
functions: hash/1, senc/2, sdec/2, someprivatefunction/1 [private], sign/2,
    verify/3, pk/1, true
```

Because the `hash/1` function has an arity of one we can call it with one argument, like: `hash(`my string')`. The symmetric encryption function `senc/2` has an arity of two so we call it with two arguments, like: `senc(`my secret message', ~k)`. The `~` (fresh) prefix describes a fresh (random) value which can be generated with the built-in `Fr(~somevariable)` function. Other fact prefixes are discussed in Section 3.2.2.

The equational theory for these functions is then defined in the `equations` section, describing the relations between parameters. An equation states that we can swap the left side with the right side, if the parameters 'fit'. An equation could for example be used to replace the application of the decrypt function on an encrypted value with the original unencrypted value. If there is no equational theory there is no possibility to replace the function application with something else. This is useful for a hash function for example, which should provide no way of revealing the original value. Thus, in our example there is only an equation for symmetric decryption and signature verification. The hash function does not have any associated equations as it is a one-way function and there is no way to reverse it. The symmetric encryption function can be reversed through the symmetric decryption function, which we define below. We also define an equation for verifying a signed message, which uses the *true* constant. Because adversaries can only use the equations that are defined, it is important to provide all the relevant equational theory as otherwise attacks might be missed.

```
equations: sdec(senc(m, k), k) = m, verify(sign(m,sk),m,pk(sk)) = true
```

Note that only convergent equational theories with the finite variant property [17] are supported by the Tamarin prover. A theory falling under these assumptions means that it can be written in a normal form (meaning that it cannot be rewritten any further) in a fixed number of steps. This includes Abelian groups (for example the exclusive-or operator) and Diffie-Hellman groups, and these groups can therefore be modelled in Tamarin.

Tamarin comes with some built-in message theories that can be included with `builtins`. These are listed below.

hashing provides a one-way hashing function.

symmetric-encryption provides functions for symmetric encryption and decryption.

assymetric-encryption models a public key encryption scheme.

signing models a signature scheme.

reveal-signing models a message-revealing signature scheme.

diffie-hellman models Diffie-Hellman groups.

xor provides functionality for the exclusive-or operator.

bilinear-pairing models bilinear groups, extending the Diffie-Hellman theory.

multiset provides the + operator to model multisets.

reliable-channel models reliable channels for the SAPIC extension.

Full descriptions can be found in the Tamarin manual [27]. Tamarin by default includes the `pair/2` function, and the `fst/1` and `snd/1` functions to extract values from pairs. $< x, y >$ can also be used instead of `pair(x, y)` and $< v1, v2, ..., vn - 1, vn >$ is equivalent to $< v1, < v2, ..., < vn - 1, vn >$ ... $>$.

If we include the hashing, symmetric-encryption and signing built-ins, there is no need to define our own function definitions and equational theory anymore. We include these built-ins as follows:

```
builtins: hashing, symmetric-encryption, signing
```

3.2 Rules

After defining the primitives, we can work on modelling the protocol using rules. Tamarin works by using a term rewrite system. Such a system is well-suited for modelling parallel systems. We define rules that take some set of facts from the state and output another set of facts to the state. The state is a multiset of facts which we call the "bag of facts". To execute a rule, all facts that the rule takes must be present in the bag of facts. The rule removes these facts from the bag of facts and puts new facts back in the bag of facts. A rule can also specify action facts which are used to model trace properties. An example rule `ClientRule1` (listing 3.1) takes nothing from the bag, states the action fact `ExecuteClientRule1('a')` and adds `A('1', 'a')` and `B('2', 'b')` to the bag of facts. Another rule `ClientRule2` takes `A(n, c)` from the bag, states the action fact `ExecuteClientRule2(n, c)` and adds `C(n)` and `D('4', hash(c))` to the bag of facts. Because the second rule takes the fact `A(n, c)` from the bag of facts, it can only be executed when that fact is in the bag. In this example that fact can be added by the first rule. Execution of `ClientRule1` leads to a bag `[A('1', 'a'), B('2', 'b')]` and if we execute `ClientRule2` we end up with a bag `[C('1'), D('4', hash('a'))]`.

```
rule ClientRule1:
  [ ]
  --[ ExecuteClientRule1('a') ]->
  [ A('1', 'a'), B('2', 'b') ]

rule ClientRule2:
  [ A(n, c) ]
  --[ ExecuteClientRule2(n, c) ]->
  [ C(n), D('4', hash(c)), Out(hash(c)) ]
```

Listing 3.1: ClientRule1 and ClientRule2 in Tamarin

3.2.1 Local Macros

To make protocol rules more readable we use local macros with the `let .. in` block. The values defined in this block will be substituted in the protocol rule. This is useful when reusing terms. Global macros that are available outside of rules are not supported by default, but support can be added by using a preprocessor such as m4. The analysis of TLS 1.3 makes extensive use of this preprocessor. Listing 3.2 shows the definition of the `ClientSetup` rule, which uses four macros `a`, `dec_a`, `session_setup_answer` and `new_key`.

```
rule ClientSetup:
  let
    a = senc(enc_v, k)
    dec_a = sdec(enc_v, ~mykey)
    session_setup_answer = <'0x01', 'SESSION_SETUP_ANSWER', a>
    new_key = ~mykey XOR a
  in
  [ CState(~ID_C, 'SENT_SETUP', ~mykey)
  , In(session_setup_answer) ]
  --[
  C_Rcv_Setup(~ID_C, ~mykey, a, ~new_key)
  ]->
  [ CState(~ID_C, 'READY', ~new_key) ]
```

Listing 3.2: Usage of macros in Tamarin

3.2.2 Facts and Messages

Tamarin uses facts to carry messages. In the previous example we saw the facts A to D containing messages such as `'1'` and `hash('a')`. Facts need to keep the same arity.

Special Facts

There are three special facts that are used to model the network and to model the generation of fresh (random) values.

In(m) models receiving a message from the network.

Out(m) models sending a message on the network.

Fr(x) models generating a fresh (random) value. For example, a new private key can be modelled with `Fr(~PrK)`.

When we exchange a message from one party to another one, the `In` and `Out` rules are used. The sending party uses the `Out` rule, which then lets the other party use the `In` rule. Because adversaries can intercept, halt and modify the network traffic, the messages that are received via these rules can be intercepted, halted and modified by them.

A fact can be made persistent by adding the `!` prefix. When we make a fact persistent it will stay in the bag of facts after it is used. This can for example be useful when we want to model a long-term key: `!Ltk(~pubkey)`.

Variable Types

Tamarin provides four message types which are described below. Fresh and public types are always of type 'message' and a variable cannot be both of a fresh and public type. A string `'hello'` is automatically defined as a public constant.

~k denotes that the variable is fresh. This can be used when generating fresh keys for example.

$A denotes that the variable is public and constant. For example a username.

#t is used to denote a timestamp. This is used in lemmas, see Section 3.3.1.

3.3 Trace Properties

In Tamarin we verify properties for a protocol by writing lemmas with trace properties. We write these trace properties using first-order logic (often called predicate logic) on the action facts that are recorded when rules are executed at specific time points. Formulas are constructed using the quantifiers `All` (for all) and `Ex` (exists), logical connectives `==>` (implication), `&` (conjunction), `|` (disjunction), `not` (negation), action constraint `a @ t` (where action `a` occurs at time `t`), temporal ordering `i < j` and temporal equality `#i = \#j`, message equality `x = y` and predicates.

3.3.1 Lemmas

To write a lemma, we use the `lemma` construct. These are the properties we want to prove. We can reason about the adversary's knowledge using the `K(x)` function. Everything an adversary knows, can be queried using this rule. Listing 3.3 shows two lemma definitions.

```
lemma Rule2Executed:
  exists-trace
    "Ex a b #i.
      ExecuteClientRule2(a, b) @ i"

lemma CIsSecret:
  all-traces // stating all-traces is optional
    "All n c #i #j.
      ExecuteClientRule2(n, c) @ i
      ==> not K(c) @ j"
```
Listing 3.3: Lemmas in Tamarin

Lemma Annotations

To help the Tamarin prover automatically verify lemmas, we can annotate lemmas with the following options:

sources forces the lemma to be verified using induction, to use the raw sources (see Chapter 5) and to generate refined sources, which will be used by all non-sources lemmas.

use_induction forces the prover to use induction as the first step on this lemma instead of having to choose between simplification and induction.

reuse makes the verification of the lemmas following this lemma use this lemma.

hide_lemma=somelemma blocks some lemma from being used to verify this lemma.

Annotations are added to square brackets after the lemma name:

```
lemma reusable_induction_lemma [reuse, use_induction]:
  "..."
```

3.3.2 Restrictions

Often modelling a protocol requires certain restrictions. For example: rules may only run once, or two values must be equal. We can model this using the `restriction` construct. These restrictions are modelled as trace properties and are applied as an action fact in a rule, such as in the `Equality` restriction in Listing 3.4.

```
rule CSetup:
  [ CState(~ID_C, 'INIT', ~mykey)
  , In(h(s_key)) ]
  --[
  Eq(~mykey, s_key) // make sure the keys match!
  , C_Rcv_Setup(~ID_S, ~mykey, s_key)
  ]->
  [ CState(~ID_S, 'READY', ~mykey) ]


// Add Eq(x, y) statement
restriction Equality:
  "All x y #i. Eq(x,y) @ i ==> x = y"
```
Listing 3.4: Restrictions in Tamarin

Embedded Restrictions

It is also possible to embed a restriction into a rule itself. The restriction in Listing 3.5 is the same as the restriction in Listing 3.4, as embedded restrictions are just syntactic sugar.

```
rule CSetup:
  [ CState(~ID_C, 'INIT', ~mykey)
  , In(h(s_key)) ]
  --[
  _restrict(~mykey = s_key) // make sure the keys match!
  , C_Rcv_Setup(~ID_S, ~mykey, s_key)
  ]->
  [ CState(~ID_S, 'READY', ~mykey) ]
```

Listing 3.5: Embedded restrictions in Tamarin

3.3.3 Predicates

Predicates are defined with `predicates`. They are substituted when used in trace properties. This can make trace properties easier to read. In Listing 3.6 we use a predicate for the exclusive-or operator to make the `either_a_or_b` lemma more readable.

```
predicates:  ExlusiveOr(p, q) <=> (p | q) & Not (p & q)

...

lemma either_a_or_b:
  "All x y #i. A(x, y)@i ==> ExclusiveOr(K(x), K(y))"
```

Listing 3.6: Predicates in Tamarin

Chapter 4

Formalizing and Modeling

Now that we have an idea of what a model in Tamarin looks like, we can start formalizing and modeling protocols. To do so we developed a method which follows two steps:

1. Draw a diagram for an overview of the protocol.
2. Formalize this diagram into concrete steps in the Tamarin language.

In this Chapter we go into these steps, why we perform them and how we perform them properly. It is important that we closely follow the specification and only model things that are in the specification. If the protocol lacks details, we ask the protocol designers for a clarification and try to upstream these clarifications into the protocol specification so our model stays true to the specification. If the protocol designers cannot be contacted, assumptions have to be made, preferably based on existing implementations of the protocol.

4.1 Protocol Diagram

We assume that there is at least some description of the protocol in a natural language such as an RFC document. To get a better overview of the protocol, we first create a diagram (see figure 4.1) of it. Such a diagram contains the following: two or more communicating parties (e.g. Alice and Bob), private- and public pre-shared variables for each party and joint communication and computation steps of the protocol. We go into our recommended notation for each part.

4.1.1 Equational Theory

Before drawing the actual protocol, the equational theory that is used in the protocol is written down. This consists of three parts: function symbols,

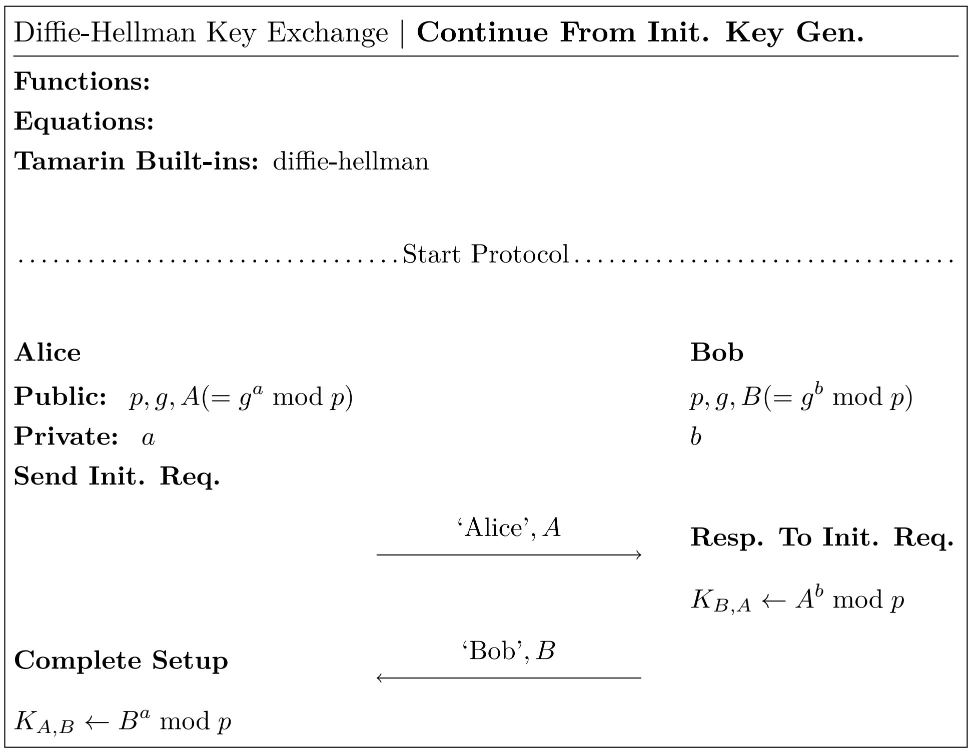
Diffie-Hellman Key Exchange | **Continue From Init. Key Gen.**
Functions:
Equations:
Tamarin Built-ins: diffie-hellman
Start Protocol
Alice
Bob
Public: $p, g, A(= g^a \bmod p)$
$p, g, B(= g^b \bmod p)$
Private: a
b
Send Init. Req.
'Alice', A
Resp. To Init. Req.
$K_{B,A} \leftarrow A^b \bmod p$
'Bob', B
Complete Setup
$K_{A,B} \leftarrow B^a \bmod p$

Figure 4.1: Diffie Hellman Key Exchange Protocol Diagram

function equations and built-ins imported from Tamarin. For functions their symbol and parameters are written such as `wrap(m, k)` and equations are defined as applications of the function and results separated by an equality sign like: `sdec(senc(m, k), k) = m`.

4.1.2 Parties

Now that we have defined the equational theory we start drawing the protocol. All communicating parties in the protocol are modelled. Most protocols have two communicating parties, but it can also be the case that there are more. For example in 5G Authenticated Key Exchange there is a scenario with three communicating parties (mobile device, home network and serving network) [6]. These parties get short and descriptive names which should be similar to the ones found in the original documentation. If the specification already uses names, then they are reused. The names of the parties are written at the top of the diagram with some horizontal space in between to allow for communication lines.

4.1.3 Sub Protocols

Sometimes there are sub protocols for steps such as initial key exchange, session key exchange and key renegotiation. Since Tamarin allows modelling large and complete protocols, we recommend including them in our model. If one protocol continues from another protocol, then that is specified at the top. To specify what values have already been exchanged we write down the public and private pre-exchanged variables below the party names.

4.1.4 Protocol Steps

We note a new step in the protocol every time data is sent or received. These steps receive a short descriptive name. If the protocol specification already gives names to steps, they are reused. In Tamarin we use these names for the rules of the model.

4.1.5 Communication

To note some communication between two parties, we draw an arrow and write the messages that are sent on top. The arrow points away from the sending party, to the receiving party. In Tamarin this is modeled with `Out` in the sending rule and `In` in the receiving rule. This communication is thus split over two protocol steps.

4.1.6 Work

Work is denoted as an assignment with a variable name, a left-pointing arrow and some work on the right side of the arrow. All variables that are used on the right hand side must be known to the specific party. All variables that have been calculated in some step can be reused in a later step.

Only functions that are defined at the start of the diagram, or that are imported as a Tamarin built-in can be used to execute work. A special type of work is initializing a fresh (random) value. In our diagram this is written as $v \leftarrow \mathbf{Fr}$, which translates to `Fr(~v)` in Tamarin.

4.2 Formalizing a Protocol Diagram

Now that we have an overview of the protocol, it can be transformed into a model in Tamarin. Protocols in Tamarin are modelled in a file containing functions, equational theory, imports from the standard library, rules and lemmas.

We start writing our theory file with naming the theory: `theory <name>` and stating that we want to `begin`. After those two lines we note all the functions, equational theory and imports from the standard library according to the description in Chapter 3.

4.2.1 Initialization Steps

We continue by transforming the diagram steps to rules. The first rules we write are the initialization rules (see Listing 4.1). These rules initialize public, private and pre-shared variables. First, the private variables are initialized with the `Fr(~var)` function in the input part of the rule. Each party also initializes an identifier using the same `Fr` function. We use these identifiers to keep the state attached to a specific protocol run. In the output part of the rule we create an initial state for each party in which we include the identifier and their private, pre-shared and public (denoted as `$var`) variables. In the trace part we include a message containing the identifier and possibly some public values.

```
theory dhexchange
begin

builtins: diffie-hellman

rule Init_A:
  let
    pkA = 'g'^~ltkA
  in
  [ Fr(~idA), Fr(~ltkA) ]
  --[ Create($A, ~idA) ]->
  [ A_Init(~idA, ~ltkA), !Ltk($A, pkA, ~ltkA) ]

rule Init_B:
  let
    pkB = 'g'^~ltkB
  in
  [ Fr(~idB), Fr(~ltkB) ]
  --[ Create($B, ~idB) ]->
  [ B_Init(~idB, ~ltkB), !Ltk($B, pkB, ~ltkB) ]
```

Listing 4.1: Start of the Tamarin Model

4.2.2 Protocol Steps

Having initialized the protocol, we work on modelling the protocol steps. These are implemented in order of appearance and start by creating a rule and retrieving the previous state in the input part of the rule. Any received message is retrieved through the `In` function. In the output we state the message(s) that we want to send through the `Out` function and we save a new state with the variables that are required in later steps. This new state is named after the name of its rule. In the trace a message is added containing the identifier and any messages that we want to prove trace properties over. This rule gets a name that corresponds to the description of the step in the diagram. Listing 4.2 contains some example rules. We recommend using macros to keep rules simple. Messages and (computed) variables that are used in multiple places of a rule can be 'stored' in a macro

```
rule SendInitialRequest:
  [ A_Init(~idA, ltkA), !Ltk(A, pkA, ltkA) ]
  --[ SendInitialRequest(A) ]->
  [ Out(<'Alice', pkA>)
  , A_SentInitialRequest(~idA, ltkA) ]

rule RespondToInitialRequest:
  let
    K_BA = pkA^ltkB
  in
  [ B_Init(~idB, ltkB), !Ltk(B, pkB, ltkB)
  , In(<'Alice', pkA>) ]
  --[ RespondToInitialRequest(B, K_BA) ]->
  [ Out(<'Bob', pkB>)
  , B_RespondedToInitialRequest(~idB, ltkB, K_BA) ]

rule CompleteSetup:
  let
    K_AB = pkB^ltkA
  in
  [ A_SentInitialRequest(~idA, ltkA), !Ltk(A, pkA, ltkA)
  , In(<'Bob', pkB>) ]
  --[ CompleteSetup(A, K_AB) ]->
  [ A_CompletedSetup(~idA, ltkA, K_AB) ]
```

Listing 4.2: Protocol Rules in Tamarin

4.2.3 Public Key Infrastructure

We can model public key infrastructure by adding a rule (see Listing 4.3) for key generation. The private key and public key are modelled as persistent facts and can be retrieved by client and server rules. Retrieving the private key (see Listing 4.4) should only be done in the server rules and not in any client rules. The public key is sent over the network so that it is available to the adversary. To prove forward secrecy properties we also add a long-term key reveal rule (see Listing 4.5) that allows an adversary to retrieve a private key for an agent X. This will be explained further in Chapter 5.

```
rule GenerateKeyPair:
  [ Fr(~ltk) ]
  -->
  [ !Pk($S, pk(~ltk))
  , Out(pk(~ltk))
  , !Ltk($S, ~ltk) ]
```

Listing 4.3: Key generation rule

```
rule SendEncryptedData:
  [ Fr(~data)
  , !Pk($S, pubkeyS) ]
  --[ SendEncryptedData(~data, pubkeyS) ]->
  [ Out(aenc(~data, pubkeyS)) ]
```

Listing 4.4: Retrieving the public key

```
rule LtkReveal:
  [ !Ltk(X, ltk) ]
  --[ LtkReveal(X) ]->
  [ Out(ltk) ]
```

Listing 4.5: Long-term key reveal

4.2.4 Next Steps

After modelling our protocol we write restriction, helper and security property lemmas. We go into how to write lemmas for the properties we want to prove in Chapter 5. The theorem file ends with `end`. In Chapter 5 we also describe how to make sure the model works properly.

Chapter 5

Proving

Having formalized our model in Tamarin, we can work on analysing it. To do so we formulate the properties we want to prove as lemmas and analyse them. As described in Chapter 3, lemmas are written using first-order logic. We provide examples for many types of security properties, however, it is not an inclusive list and care should be taken to think up all required security properties. It is also important to write these lemmas properly, as an incorrect lemma will ‘prove’ an incorrect property, thus not actually proving anything of value at all.

The basis for many of these properties comes from the analysis of TLS 1.3 [19] as this is a protocol that has the same security goals as many other secure communication protocols. We look into the same properties and explain the lemmas used by this analysis. For some properties we also provide examples for the Diffie-Hellman Key Exchange protocol from Chapter 4.

Tamarin analyses these properties automatically by solving a constraint system based on the protocol model and the property in ‘negation normal form’ [26]. An algorithm is applied to automatically and soundly transform the property to this special form and solve the constraint system. Because the property is in a negated form, we consider a property proven correct if for every system the algorithm finds a counter-example. If a solved system is found, a property is proven false. In such a case, Tamarin illustrates the counter-example (in the non-negated form) to the property.

5.1 Protocol Run

The first lemma (see Listing 5.1) we write is the lemma confirming that a successful interaction is possible. This will often take the form of an `exists-trace` lemma with an exists statement stating that there is indeed a trace where two or more clients have successfully exchanged keys. Without knowing for sure that the model allows for a normal trace, many security properties will not yield any useful result. For example: a property stating that a session key is never known by the adversary is not of much use if a session key is never exchanged in a normal protocol run either: the property might hold, even though there is no session key secrecy.

```
lemma executable: exists-trace
  "Ex A B k #i #j.
    RespondToInitialRequest(B, k)@i
    & CompleteSetup(A, k)@j
    & not ( Ex X #r. RevealLtk(X)@r )"
```

Listing 5.1: Successful protocol run lemma in Tamarin

5.2 Secrecy Properties

We analyse secrecy properties because we want to know if certain secret variables stay secret. We can study 'normal' secrecy and forward secrecy properties. For proving forward secrecy properties we add a long-term key reveal rule (see Listing 5.2) to our model. This rule allows an adversary to retrieve the long-term key, which is important to analyse forward secrecy properties. Without any constraints an adversary will always be able to retrieve the key through the long-term key reveal rule and break protocol security, thus usage of this rule needs to be limited. To limit the usage of this rule we add conditions that the key reveal rule has not been used at all in all 'normal' lemmas except for forward secrecy properties where we allow executing the rule after some actions have been executed. Of course, forward secrecy implies secrecy and thus if a forward secrecy property can be proven, then a secrecy property also holds.

```
rule RevealLtk:
  [ !Ltk(X, pk, ltk) ]--[ RevealLtk(X) ]->[ Out(ltk), Out(pk) ]
```

Listing 5.2: Long-term key reveal in Tamarin

5.2.1 Key Secrecy

Key secrecy is one of the most important properties we analyse. Lemmas to prove the secrecy of keys often take the following form: "for all variables it holds that: if we have successfully exchanged keys and the key reveal rules are not used, then the adversary does not know the secret key". A simple example is given in Listing 5.3 and a more extensive example from the analysis of TLS 1.3 is given in Listing 5.4.

```
lemma key_secrecy:
  "All A B k #i #j.
      RespondToInitialRequest(B, k)@i &
      CompleteSetup(A, k)@j &
      not (Ex X #r. RevealLtk(X)@r)
```

```
    ==> not Ex #g. K(k)@g"
```

Listing 5.3: Key secrecy lemma in Tamarin

```
lemma secret_session_keys [hide_lemma=sig_origin,hide_lemma=posths_rms]:
  "All tid actor peer kw kr pas #i.
      SessionKey(tid, actor, peer, <pas, 'auth'>, <kw, kr>)@i &
      not (Ex #r. RevLtk(peer)@r & #r < #i) &
      not (Ex tid3 x #r. RevDHExp(tid3, peer, x)@r & #r < #i) &
      not (Ex tid4 y #r. RevDHExp(tid4, actor, y)@r & #r < #i) &
      not (Ex rms #r. RevealPSK(actor, rms)@r) &
      not (Ex rms #r. RevealPSK(peer, rms)@r)
    ==> not Ex #j. K(kr)@j"
```

Listing 5.4: TLS 1.3 key secrecy lemma

5.2.2 Forward Secrecy

The forward secrecy lemma is similar to that of the normal key secrecy, but we allow leaking a long-term key after the session key was generated. This leakage is limited by using the temporal ordering operators `<` and `=`. The lemma in Listing 5.5 is similar to the normal key secrecy in Listing 5.3 except that it now only restricts the long-term key reveal to not be used before the session is completed.

```
lemma forward_key_secrecy:
  "All A B k #i #j.
      RespondToInitialRequest(B, k)@i &
      CompleteSetup(A, k)@j &
      not (Ex X #r. RevealLtk(X)@r & #r < #j)
    ==> not Ex #g. K(k)@g"
```

Listing 5.5: Forward secrecy lemma in Tamarin

5.3 Authentication Properties

Authentication properties often follow the following reasoning: "if a client believes they agreed on some property, then the server also agrees on this property". Thus we analyse if the client and server exchange the same values.

5.3.1 Establishing Same Session Keys

In a session key agreement lemma we write that the client and server always agree on the same session keys if they both believe that a successful key exchange has been completed. This can be seen in the example from the TLS 1.3 lemma in Listing 5.6.

```
lemma session_key_agreement [hide_lemma=sig_origin]:
  "All tid tid2 actor peer actor2 peer2 nonces keys keys2 cas as2 #i #j #k #
    l.
    SessionKey(tid, actor, peer2, <cas, 'auth'>, keys)@i &
    running(Nonces, actor, 'client', nonces)@j &
    SessionKey(tid2, peer, actor2, as2, keys2)@k &
    running2(Nonces, peer, 'server', nonces)@l &
     not (Ex #r. RevLtk(peer)@r & #r < #i & #r < #k) &
     not (Ex tid3 x #r. RevDHExp(tid3, peer, x)@r & #r < #i & #r < #k) &
     not (Ex tid4 y #r. RevDHExp(tid4, actor, y)@r & #r < #i & #r < #k) &
     not (Ex rms #r. RevealPSK(actor, rms)@r & #r < #i & #r < #k) &
     not (Ex rms #r. RevealPSK(peer, rms)@r & #r < #i & #r < #k)
     ==>
      keys = keys2"
```

Listing 5.6: TLS 1.3 session key agreement lemma

5.3.2 Uniqueness of Session Keys

The property of the session keys always being unique can be rewritten to the definition that if two session keys are the same, they must be from the same session. This is shown in the example from the TLS 1.3 lemma in Listing 5.7.

```
lemma unique_session_keys:
  "All tid tid2 actor peer peer2 kr kw as as2 #i #j.
     SessionKey(tid, actor, peer, as, <kr, kw>)@i &
     SessionKey(tid2, actor, peer2, as2, <kr, kw>)@j
      ==>
        #i = #j"
```

Listing 5.7: TLS 1.3 uniqueness of session keys lemma

5.3.3 Peer Authentication and Key Compromise Impersonation Resistance

TLS 1.3 provides unilateral and mutual authentication. Unilateral authentication (in Listing 5.8) states that if the client believes that it has completed a handshake with a server, that server has previously run the handshake with the client. Mutual authentication (in Listing 5.9) means that this property also holds both ways if both the client and server believe that they completed a handshake with each other. These two properties are also used to prove key compromise impersonation resistance.

```
lemma entity_authentication [reuse, use_induction]:
  "All tid actor peer nonces cas #i.
      commit(Nonces, actor, 'client', nonces)@i & commit(Identity, actor, '
    client', peer, <cas, 'auth'>)@i &
      not (Ex #r. RevLtk(peer)@r & #r < #i) &
      not (Ex tid3 x #r. RevDHExp(tid3, peer, x)@r & #r < #i) &
      not (Ex tid4 y #r. RevDHExp(tid4, actor, y)@r & #r < #i) &
      not (Ex rms #r. RevealPSK(actor, rms)@r) &
      not (Ex rms #r. RevealPSK(peer, rms)@r)
```

```
          ==> (Ex tid2 #j. running2(Nonces, peer, 'server', nonces)@j & #j <
     #i)"
```

Listing 5.8: TLS 1.3 unilateral entity authentication lemma

```
lemma mutual_entity_authentication [reuse, use_induction]:
  "All tid actor peer nonces sas #i.
      commit(Nonces, actor, 'server', nonces)@i & commit(Identity, actor, '
    server', peer, <sas, 'auth'>)@i &
      not (Ex #r. RevLtk(peer)@r & #r < #i) &
      not (Ex tid3 x #r. RevDHExp(tid3, peer, x)@r & #r < #i) &
      not (Ex tid4 y #r. RevDHExp(tid4, actor, y)@r & #r < #i) &
      not (Ex rms #r. RevealPSK(actor, rms)@r) &
      not (Ex rms #r. RevealPSK(peer, rms)@r)
          ==> (Ex tid2 #j. running2(Nonces, peer, 'client', nonces)@j & #j <
     #i)"
```

Listing 5.9: TLS 1.3 mutual entity authentication lemma

Chapter 6

Analysis of Secure Communication Product

To test and confirm the methods described in Chapter 4 and 5, we apply them to the Secure Communication Product protocol. The Secure Communication Product protocol uses symmetric key cryptography to establish a secure connection between a Client and a Server. Modelling the protocol and analysing security properties yielded some interesting results. We only analyse the session key exchange and not the full protocol. Recall that this is only a part of the protocol and that details such as names and parameters have been changed due to the non-disclosure agreement.

6.1 Diagram

We transform the existing documentation for the session key exchange into a protocol diagram (see figure 6.1). To explain what happens in the protocol: the Secure Communication Product protocol uses a pre-shared symmetric traffic key encryption key (TrKEK) to exchange four fresh values (two from the Client and two from the Server). The first pair (one value from the Client and one value from the Server) is used to create a session key by applying a bitwise XOR on the two fresh values. The second pair is used to ratchet the TrKEK by applying a bitwise XOR on the old TrKEK and the two values. Exchanging these session keys allows for sending encrypted data over an insecure network. Ratcheting the TrKEK makes sure that forward secrecy is kept.

6.2 Model

We model the protocol after the diagram we drew. All the cryptographic primitives and functions we use in the protocol diagram are also available in Tamarin. We used the built-in exclusive-or theory and wrote our own

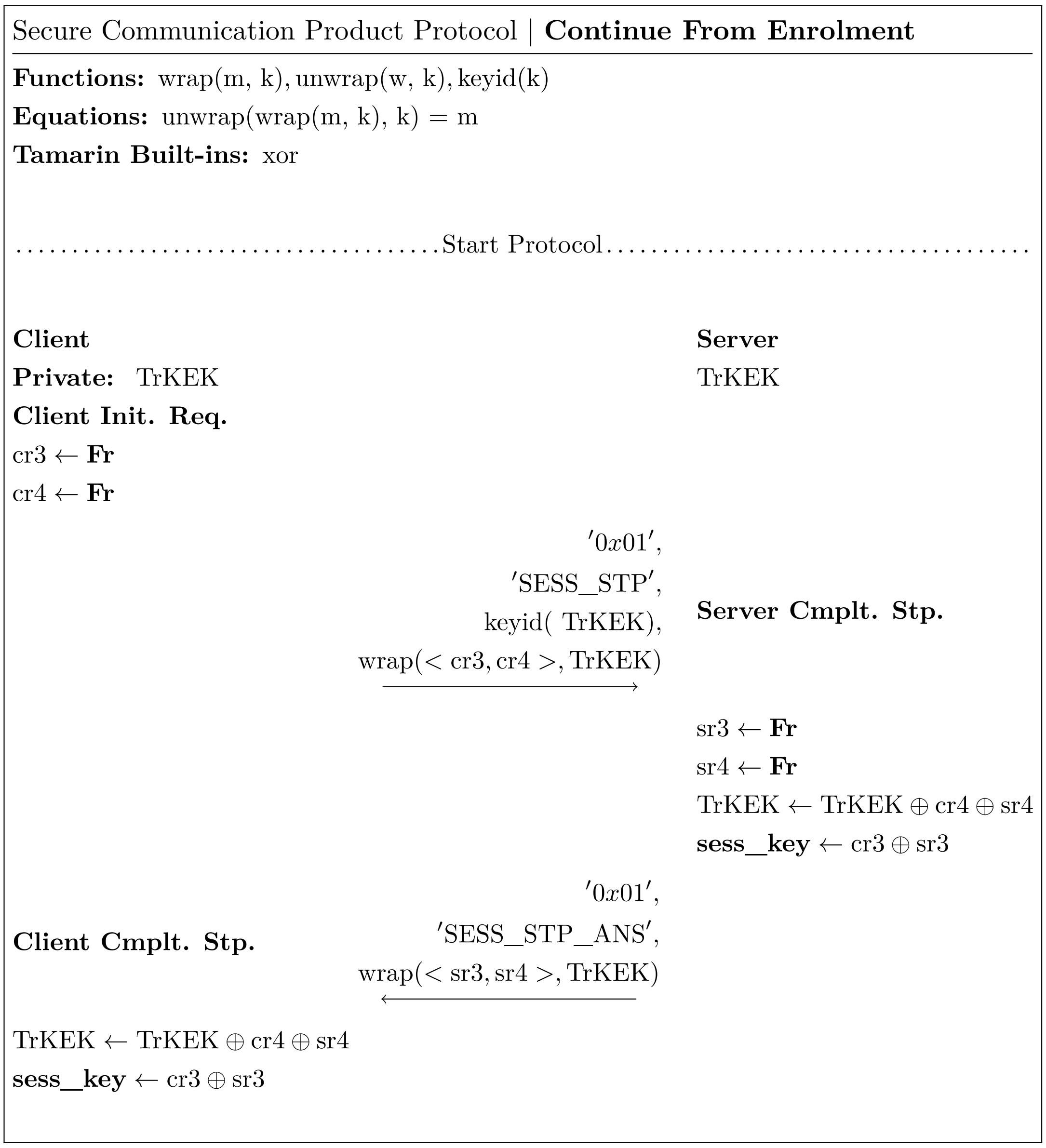
Secure Communication Product Protocol | **Continue From Enrolment**
Functions: wrap(m, k), unwrap(w, k), keyid(k)
Equations: unwrap(wrap(m, k), k) = m
Tamarin Built-ins: xor
Start Protocol
Client
Private: TrKEK
Client Init. Req.
cr3 ← **Fr**
cr4 ← **Fr**
Server
TrKEK
'0x01', 'SESS_STP', keyid(TrKEK), wrap(< cr3, cr4 >, TrKEK)
Server Cmplt. Stp.
sr3 ← **Fr**
sr4 ← **Fr**
TrKEK ← TrKEK ⊕ cr4 ⊕ sr4
sess_key ← cr3 ⊕ sr3
'0x01', 'SESS_STP_ANS', wrap(< sr3, sr4 >, TrKEK)
Client Cmplt. Stp.
TrKEK ← TrKEK ⊕ cr4 ⊕ sr4
sess_key ← cr3 ⊕ sr3

Figure 6.1: Secure Communication Product protocol diagram

wrapping and key id functions. These functions are the same as the standard symmetric cryptography and hashing functions, but use a different name to keep the documentation, protocol diagram and Tamarin model similar. In both the Client and Server we keep a state containing a thread id, a keyword for the 'state' they are in and the traffic key encryption key, possible session key, possible fresh values and possible packet counter. Macros are used to maintain readability of rules and keep calculations out of the input, output and trace sets of rules. This can for example be seen in the Client setup rule (Listing 6.1).

```
rule C_Setup:
  let
    session_setup_answer = <'0x01', 'SESSION_SETUP_ANSWER', wrap(<sr3, sr4>,
      msg_key)>
    wrapped_msg = unwrap(wrap(<sr3, sr4>, msg_key), ~TrKEK)
    sr3 = fst(wrapped_msg)
    sr4 = snd(wrapped_msg)
    sess_keys = ~cr3 XOR sr3
    new_TrKEK = ~TrKEK XOR ~cr4 XOR sr4
  in
  [ CState(~ID_C, 'SENT_SETUP', <~TrKEK, ~cr3, ~cr4>)
  , In(session_setup_answer) ]
  --[
    C_Rcv_Setup(~ID_C, ~cr3, ~cr4, sr3, sr4, ~TrKEK, sess_keys, new_TrKEK)
  ]->
  [ CState(~ID_C, 'READY', <new_TrKEK, sess_keys, '0'>) ]
```

Listing 6.1: Client setup rule

6.3 Validity of the Model

To check the validity of the model we want to know if its possible for the Client and Server to exchange the same session key and traffic key encryption key. To do so we add a lemma (see Listing 6.2) stating that there exists such a trace. Analysing this lemma results in Tamarin terminating and finding a solution, thus proving that it is possible to have a successful protocol run.

```
lemma successful_run: exists-trace
  "Ex ID_C ID_S TrKEK cr3 cr4 sr3 sr4 sess_keys new_TrKEK #i #j.
    C_Rcv_Setup(ID_C, cr3, cr4, sr3, sr4, TrKEK, sess_keys, new_TrKEK)@i &
    S_Rcv_Setup(ID_S, cr3, cr4, sr3, sr4, TrKEK, sess_keys, new_TrKEK)@j &
    not (Ex x #r. RevealLtk(x)@r)"
```

Listing 6.2: Successful run lemma

6.4 Security Properties

For this protocol we analyse the (forward) secrecy of the traffic key encryption key and session key. Problems that were found and solutions that were tested are discussed in Section 6.5.

6.4.1 Traffic Key Encryption Key Secrecy

To analyse the secrecy of the traffic key encryption key we want to prove that whenever the Client believes that they have executed a proper session setup, the adversary does not know their current or new traffic key encryption key.

The adversary is not allowed to execute a long-term key reveal in this case. Analysing this lemma (see Listing 6.3) results in Tamarin terminating and finding no counter-example, thus proving the property to be true.

```
lemma trkek_secrecy:
  "All ID_C cr3 cr4 sr3 sr4 TrKEK sess_keys new_TrKEK #i.
      C_Rcv_Setup(ID_C, cr3, cr4, sr3, sr4, TrKEK, sess_keys, new_TrKEK)@i &
      not (Ex x #r. RevealLtk(x)@r)
    ==> not ((Ex #r. K(TrKEK)@r) | (Ex #r. K(new_TrKEK)@r)) "
```

Listing 6.3: TrKEK secrecy lemma

6.4.2 Session Key Secrecy

To analyse the secrecy of the session key we want to prove that whenever the Client believes that they have executed a proper session setup, the adversary does not know the session key. Again, the adversary is not allowed to execute a long-term key reveal in this case. Analysing this lemma (see Listing 6.4) results in Tamarin terminating and finding a counter-example.

```
lemma sess_key_secrecy:
  "All ID_C cr3 cr4 sr3 sr4 TrKEK sess_keys new_TrKEK #i.
      C_Rcv_Setup(ID_C, cr3, cr4, sr3, sr4, TrKEK, sess_keys, new_TrKEK)@i &
      not (Ex x #r. RevealLtk(x)@r)
    ==> not Ex #r. K(sess_keys)@r"
```

Listing 6.4: Session key secrecy lemma

6.4.3 Forward Secrecy

To analyse the secrecy of the session key we want to prove that whenever the Client believes that they have executed a proper session setup, the adversary does not know the session key. This property should then hold even if the traffic key encryption key of a later session setup is leaked. Analysing this lemma (see Listing 6.5) results in Tamarin not terminating and running out of memory.

```
lemma sess_key_forward_secrecy:
  "All ID_C ID_S TrKEK cr3 cr4 sr3 sr4 sess_keys new_TrKEK #i #j.
      C_Rcv_Setup(ID_C, cr3, cr4, sr3, sr4, TrKEK, sess_keys, new_TrKEK)@i &
      S_Rcv_Setup(ID_S, cr3, cr4, sr3, sr4, TrKEK, sess_keys, new_TrKEK)@j &
      not (Ex x #r. RevealLtk(x)@r & #r < #i & #r < #j)
    ==> not Ex #r. K(sess_keys)@r"
```

Listing 6.5: Session key forward secrecy lemma

6.5 Problems and Solutions

6.5.1 Session Key Secrecy

Analysis of the secrecy of session keys resulted in Tamarin finding a counter-example. This means that there is an attack where an adversary knows the session key of the Client. Looking at the attack trace, we find that it is possible for the adversary to execute a replay attack: if the two fresh values that are sent by the Client in the Client init step are sent back to the Client in the Client setup step, the session key will be calculated as $\mathrm{cr3} \oplus \mathrm{cr3}$. This evaluates to zero, and zero is known by the adversary, thus breaking the secrecy of the session key.

To solve the vulnerability we apply domain separation so that it is no longer possible to replay the data sent by the Client to the Client itself. We do this by changing the sent message `wrap(<cr3, cr4>, trkek)` to `wrap(<cr3, cr4, ˋclient'>, trkek)`. We apply the same solution to the Server side, adding `ˋserver'` instead. On the receiving end we also change these messages. This works because all data in the `wrap` function is encrypted and thus an adversary is unable to change it. Tamarin's pattern matching makes sure that the Client accepts only the fresh values sent by the Server, and the Server accepts only those sent by the Client.

If we analyse the property again we find that Tamarin is now able to prove the session key secrecy property, showing that our solution works.

6.5.2 Forward Secrecy

Unfortunately we were unable to get a proper result from the forward secrecy properties, as Tamarin used too much memory for our machine. We think this is due to some looping because of the traffic key encryption key not really being a long-term key as it is ratcheted (and thus changed) on every protocol run. Time could be spent to investigate this property further so that forward secrecy properties can be properly evaluated.

6.6 Results Summary

In Table 6.2 we provide a summary of the results of the analysis. We state whether Tamarin found a positive result (✓, property is proved), found a negative result (↯, attack found) or timed out (⏲) and the time it took to analyse the property. The analysis was performed in a virtual machine running Fedora Linux with 10 processor threads and 10 gigabytes of RAM.

Property	Original Model	Improved Protocol Model
Successful Run	✓ 0:01.57	✓ 0:01.67
TrKEK Secrecy	✓ 0:15.54	✓ 0:16.05
Session Key Secrecy	↯ 0:02.86	✓ 0:14.14
Forward Secrecy	⏲ ...	⏲ ...

Figure 6.2: Results of Protocol Analysis

Chapter 7

Conclusions and Future Work

In this thesis we introduced the topic of automated protocol analysis. Through a literature study we compared methods and tools for automated protocol analysis and found that Tamarin was best suited to our needs. We described what a Tamarin theorem file looks like by explaining its notation. We developed a method for the formalization of a protocol specification into a semi-formal protocol diagram and transformation into a Tamarin model. We looked at common properties to prove and provided and explained examples from existing analyses. We also tested this method of formalisation and proving and applied it to a protocol developed by Security Company. Through the model and properties that we wrote, Tamarin found an attack on the protocol. We found a solution to this vulnerability and re-applied our methodology to prove the security of the improved protocol design.

We believe that analysing protocol designs through automated tools such as Tamarin is a valuable addition to the security of communication protocols. It is something that should be included in the toolbox of every protocol designer and time should be taken to carefully select and write security properties to analyse. Not only the analysis results are useful for the security of the protocol, formalizing a protocol is also a useful process in itself.

There is enough room for future work. Below is a list of possible topics for future work.

- Look at analysing equivalence properties. This could be done using a tool such as DEEPSEC or ProVerif. These properties are especially interesting to analyse privacy properties.

- Investigate the SAPIC+ extension of Tamarin, which translates into Tamarin, ProVerif and DEEPSEC. If this tool works well, it could provide more complete analysis results as it is capable of leveraging the analysis strengths of these different tools.

- Solve the analysis problems with forward secrecy properties in the analysis of the Secure Communication Product protocol. This could provide us with stronger security guarantees over the current secrecy properties.
- Look into the next layer of computer-aided cryptography: proving correctness of implementations according to a protocol design. Implementation correctness is important in guaranteeing that the protocol implementation keeps to the proven-secure protocol design.
- Expand an analysis tool so that a model of a protocol can be transformed into a protocol specification and diagram. This can then provide a 'single source of truth' for the analysis and implementation of the protocol.

Bibliography

[1] Martín Abadi and Cédric Fournet. "Mobile values, new names, and secure communication". In: *ACM Sigplan Notices* 36.3 (2001), pp. 104–115.

[2] Michael Backes et al. "A novel approach for reasoning about liveness in cryptographic protocols and its application to fair exchange". In: *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2017, pp. 76–91.

[3] Jos CM Baeten. "A brief history of process algebra". In: *Theoretical Computer Science* 335.2-3 (2005), pp. 131–146.

[4] Manuel Barbosa et al. "SoK: Computer-aided cryptography". In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2021, pp. 777–795.

[5] David Basin, Ralf Sasse, and Jorge Toro-Pozo. "The EMV Standard: Break, Fix, Verify". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 1766–1781.

[6] David Basin et al. "A Formal Analysis of 5G Authentication". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 1383–1396. ISBN: 9781450356930.

[7] David Basin et al. "Tamarin: Verification of Large-Scale, Real-World, Cryptographic Protocols". In: *IEEE Security & Privacy* 20.3 (2022), pp. 24–32.

[8] Bruno Blanchet. "A computationally sound mechanized prover for security protocols". In: *IEEE Transactions on Dependable and Secure Computing* 5.4 (2008), pp. 193–207.

[9] Bruno Blanchet, Martín Abadi, and Cédric Fournet. "Automated verification of selected equivalences for security protocols". In: *The Journal of Logic and Algebraic Programming* 75.1 (2008), pp. 3–51.

[10] Bruno Blanchet et al. *ProVerif 2.04: automatic cryptographic protocol verifier, user manual and tutorial.* 2021.

[11] Vincent Cheval. "Apte: an algorithm for proving trace equivalence". In: *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20.* Springer. 2014, pp. 587–592.

[12] Vincent Cheval, Steve Kremer, and Itsaka Rakotonirina. "DEEPSEC: deciding equivalence properties in security protocols theory and practice". In: *2018 IEEE symposium on security and privacy (SP).* IEEE. 2018, pp. 529–546.

[13] Vincent Cheval et al. "SAPIC+: protocol verifiers of the world, unite!" In: *USENIX Security Symposium (USENIX Security), 2022.* 2022.

[14] Rémy Chrétien, Véronique Cortier, and Stéphanie Delaune. "Decidability of trace equivalence for protocols with nonces". In: *2015 IEEE 28th Computer Security Foundations Symposium.* IEEE. 2015, pp. 170–184.

[15] Katriel Cohn-Gordon et al. "A formal security analysis of the signal messaging protocol". In: *Journal of Cryptology* 33 (2020), pp. 1914–1983.

[16] *Common Criteria for Information Technology Security Evaluation.* 2022.

[17] Hubert Comon-Lundh and Stéphanie Delaune. "The finite variant property: How to get rid of some algebraic properties". In: *Term Rewriting and Applications: 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005. Proceedings 16.* Springer. 2005, pp. 294–307.

[18] Cas Cremers and Martin Dehnel-Wild. "Component-based formal analysis of 5G-AKA: Channel assumptions and session confusion". In: *Internet Society* (2019).

[19] Cas Cremers et al. "A comprehensive symbolic analysis of TLS 1.3". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* 2017, pp. 1773–1788.

[20] Stéphanie Delaune, Steve Kremer, and Mark Ryan. "Verifying privacy-type properties of electronic voting protocols". In: *Journal of Computer Security* 17.4 (2009), pp. 435–487.

[21] Danny Dolev and Andrew Yao. "On the security of public key protocols". In: *IEEE Transactions on information theory* 29.2 (1983), pp. 198–208.

[22] Jason A Donenfeld and Kevin Milner. *Formal verification of the WireGuard protocol.* Tech. rep. 2017.

[23] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. "Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach". In: *2017 IEEE European symposium on security and privacy (EuroS&P)*. IEEE. 2017, pp. 435–450.

[24] Benjamin Lipp, Bruno Blanchet, and Karthikeyan Bhargavan. "A mechanised cryptographic proof of the WireGuard virtual private network protocol". In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2019, pp. 231–246.

[25] Simon Meier et al. "The TAMARIN Prover for the Symbolic Analysis of Security Protocols". In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 696–701. ISBN: 978-3-642-39799-8.

[26] Benedikt Schmidt et al. "Automated analysis of Diffie-Hellman protocols and advanced security properties". In: *2012 IEEE 25th Computer Security Foundations Symposium*. IEEE. 2012, pp. 78–94.

[27] The Tamarin Team. *Tamarin Prover Manual*. 2023.